INSTITUT FÜR INFORMATIK

Kiel Declarative Programming Days 2013

20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013)
22nd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2013)
27th Workshop on Logic Programming (WLP 2013)

Michael Hanus, Ricardo Rocha (Eds.)

Bericht Nr. 1306
September 2013
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Kiel Declarative Programming Days 2013

20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013)
22nd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2013)
27th Workshop on Logic Programming (WLP 2013)

Michael Hanus, Ricardo Rocha (Eds.)

Bericht Nr. 1306
September 2013
ISSN 2192-6247

e-mail: mh@informatik.uni-kiel.de, ricroc@dcc.fc.up.pt

This technical report contains the papers presented at the Kiel Declarative Programming Days held during September 11-13, 2013 in Kiel, Germany.

Preface

This report contains the papers presented at the *Kiel Declarative Programming Days 2013*, held in Kiel (Germany) during September 11-13, 2013. The Kiel Declarative Programming Days 2013 unified the following events:

- 20th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2013)
- 22nd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2013)
- 27th Workshop on Logic Programming (WLP 2013)

All these events are centered around *declarative programming*, an advanced paradigm for the modeling and solving of complex problems. These specification and implementation methods attracted increasing attention over the last decades, e.g., in the domains of databases and natural language processing, for modeling and processing combinatorial problems, and for high-level programming of complex, in particular, knowledge-based systems.

The INAP conferences provide a communicative forum for intensive discussion of applications of important technologies around logic programming, constraint problem solving, and closely related computing paradigms. It comprehensively covers the impact of programmable logic solvers in the internet society, its underlying technologies, and leading edge applications in industry, commerce, government, and societal services. Previous INAP editions have been held in Japan, Germany, Portugal, and Austria.

The international workshops on functional and logic programming (WFLP) aim at bringing together researchers interested in functional programming, logic programming, as well as the integration of these paradigms. Previous WFLP editions have been held in Germany, France, Spain, Italy, Estonia, Brazil, Denmark, and Japan.

The workshops on (constraint) logic programming (WLP) serve as the scientific forum of the annual meeting of the Society of Logic Programming (GLP e.V.) and bring together researchers interested in logic programming, constraint programming, and related areas like databases, artificial intelligence, and operations research. Previous WLP editions have been held in Germany, Austria, Switzerland, and Egypt.

In this year these events were jointly organized under the umbrella of the Kiel Declarative Programming Days in order to promote the cross-fertilizing exchange of ideas and experiences among researchers and students from the different communities interested in the foundations, applications, and combinations of high-level, declarative programming languages and related areas. The technical program of the event included presentations of refereed technical papers and system descriptions.

The program committees collected for each paper at least three reviews. The meetings of the program committees were conducted electronically during July 2013 with the help of the conference management system EasyChair. After careful discussions, the program committees selected 21 technical papers and two system descriptions for presentation. In addition to the selected papers, the scientific program included an invited lecture by Tom Schrijvers (University of Ghent, Belgium).

We would like to thank all authors who submitted papers to this event. We are grateful to the members of the program committees and all the additional reviewers for their careful and efficient work in the review process. Finally, we express our gratitude to all members of the local organizing committee for their help in organizing a successful event.

September 2013

Michael Hanus
Ricardo Rocha

Conference Organization

Conference Chairs

WFLP/WLP	**INAP**
Michael Hanus	Ricardo Rocha
Institut für Informatik	Departamento de Ciência de Computadores
Christian-Albrechts-Universität Kiel	Universidade do Porto
24098 Kiel, Germany	4169-007 Porto, Portugal
E-mail: mh@informatik.uni-kiel.de	E-mail: ricroc@dcc.fc.up.pt

WFLP/WLP Program Committee

Elvira Albert	Universidad Complutense de Madrid, Spain
Sergio Antoy	Portland State University, USA
François Bry	University of Munich, Germany
Jürgen Dix	Clausthal University of Technology, Germany
Rachid Echahed	CNRS, University of Grenoble, France
Moreno Falaschi	Università di Siena, Italy
Sebastian Fischer	Kiel, Germany
Thom Frühwirth	University of Ulm, Germany
Michael Hanus	University of Kiel, Germany (Chair)
Oleg Kiselyov	Monterey (CA), USA
Herbert Kuchen	University of Münster, Germany
Francisco Javier López-Fraguas	Universidad Complutense de Madrid, Spain
Torsten Schaub	University of Potsdam, Germany
Peter Schneider-Kamp	University of Southern Denmark, Denmark
Dietmar Seipel	University of Würzburg, Germany
Hans Tompits	Vienna University of Technology, Austria
German Vidal	Universidad Politécnica de Valencia, Spain
Janis Voigtländer	University of Bonn, Germany

INAP Committees

Track Chairs

Nonmonotonic Reasoning

Hans Tompits	Vienna University of Technology, Austria

Applications and System Implementations

Masanobu Umeda	Kyushu Institute of Technology, Japan

Extensions of Logic Programming

Salvador Abreu	University of Évora, Portugal

Databases, Deductive Databases and Data Mining

Dietmar Seipel	University of Würzburg, Germany

Program Committee

Salvador Abreu	University of Évora, Portugal
Sergio Alvarez	Boston College, USA
Christoph Beierle	FernUniversität in Hagen, Germany
Philippe Codognet	JFLI/CNRS at University of Tokyo, Japan
Daniel Diaz	University of Paris I, France
Ulrich Geske	University of Potsdam, Germany
Petra Hofstedt	Brandenburg University of Technology at Cottbus, Germany
Katsumi Inoue	National Institute of Informatics, Japan
Gabriele Kern-Isberner	University of Dortmund, Germany
Ulrich Neumerkel	Vienna University of Technology, Austria
Vitor Nogueira	University of Évora, Portugal
Enrico Pontelli	New Mexico State University, USA
Ricardo Rocha	University of Porto, Portugal (Chair)
Irene Rodrigues	University of Évora, Portugal
Carolina Ruiz	Worcester Polytechnic Institute, USA
Vítor Santos Costa	University of Porto, Portugal
Dietmar Seipel	University of Würzburg, Germany
Terrance Swift	Universidade Nova de Lisboa, Portugal
Hans Tompits	Vienna University of Technology, Austria
Masanobu Umeda	Kyushu Institute of Technology, Japan
Marina De Vos	University of Bath, United Kingdom
Armin Wolf	Fraunhofer FIRST, Berlin, Germany
Osamu Yoshie	Waseda University, Japan

Local Organization

Linda Haberland	University of Kiel, Germany
Michael Hanus	University of Kiel, Germany
Björn Peemöller	University of Kiel, Germany
Fabian Reck	University of Kiel, Germany
Jan Rasmus Tikovsky	University of Kiel, Germany

External Reviewers

Demis Ballis
Steffen Ernsting
Raúl Gutiérrez
Benny Höckner
Andy Jost
Arne König
Ludwig Ostermayer
Tony Ribeiro
José Miguel Rojas
Javier Romero
Miguel A. Salido
Peter Sauer
Luca Torella
Amira Zaki

Table of Contents

Delimited Continuations for Prolog: An Overview

Tom Schrijvers

Ghent University, Belgium
Tom.Schrijvers@UGent.be

Abstract

Delimited continuations are a famous control primitive that originates in the functional programming world. It allows the programmer to suspend and capture the remaining part of a computation in order to resume it later.

This invited talk puts a new Prolog-compatible face on this primitive: It specifies the semantics by means of a meta-interpreter, and illustrates the usefulness of the feature with many examples, such as DCGs, effect handlers and coroutines. Finally, the talk also covers how to easily and effectively add delimited continuations support to the WAM.

Acknowledgments

This inivited talk is based on joint work [1, 2] with Bart Demoen, Benoit Desouter and Jan Wielemaker that is accepted at ICLP 2013.

References

1. Schrijvers, T., Demoen, B., Desouter, B., Wielemaker, J.: Delimited continuations for prolog. Theory and Practice of Logic Programming (TPLP) (2013) Proceedings of the International Conference on Logic Programming (ICLP).
2. Demoen, B., Schrijvers, T., Desouter, B.: Delimited continuations in Prolog: semantics, use and implementation in the WAM. Report CW 631, Dept. of Computer Science, KU Leuven, Belgium (2013)

Extension of Gelfond-Lifschitz Reduction for Preferred Answer Sets : Preliminary Report

Alexander Šimko

Comenius University in Bratislava, Bratislava 842 48, Slovakia,
simko@fmph.uniba.sk,
WWW home page: http://dai.fmph.uniba.sk/~simko

Abstract. We consider the problem of extending the answer set semantics for logic programs with preferences on rules. Many interesting semantics have been proposed. In this paper we develop a descriptive semantics that ignores preferences between non-conflicting rules. It is based on the Gelfond-Lifschitz reduction extended by the condition: a rule cannot be removed because of a less preferred conflicting rule. It turns out that the semantics continues in the hierarchy of the approaches by Delgrande et. al., Wang et. al., and Brewka and Eiter, and guarantees existence of a preferred answer set for the class of call-consistent head-consistent extended logic programs. The semantics can be also characterized by a transformation from logic programs with preferences to logic programs without preferences such that the preferred answer sets of an original program correspond to the answer sets of the transformed program. We have also developed a prototypical solver for preferred answer sets using the meta-interpretation technique.

Keywords: knowledge representation, logic programming, preferred answer sets

1 Introduction

A knowledge base of a logic program possibly contains conflicting rules – rules that state mutually exclusive things. Having such rules, we often want to specify which of the rules to apply if both the rules can be applied.

Many interesting extensions of the answer set semantics for logic programs with preferences on rules have been proposed, e.g., [2, 4, 10, 13, 17, 18]. Among the ones that stay in the NP complexity class ([2, 4, 17]), there is none that guarantees existence of a preferred answer set for the subclass of stratified [1, 3] normal logic programs. This is the result of the fact, that the semantics do not ignore preferences between non-conflicting rules. Therefore these semantics are not usable in some situations, e.g., where the rules of a program are divided into modules, and the preferences on rules are inherited from the preferences on modules. Usefulness of such inheritance was shown, e.g., in [8, 19]. It is then important that the preferences between the non-conflicting rules do not cause side effects, e.g., non existence of preferred answer sets.

In this paper we propose a semantics, staying in the NP complexity class, with a very simple and elegant definition that ignores preferences between non-conflicting rules, and guarantees existence of a preferred answer set for the class of call-consistent [9, 11] head-consistent [14] extended logic programs. The semantics is defined using a modified version of the Gelfond-Lifschitz reduction. An additional principle is incorporated: a rule cannot be removed because of a less preferred conflicting rule. It turns out that the semantics continues in the hierarchy of the semantics [4, 17, 2] discovered in [12]. It preserves the preferred answer sets of these semantics and admits additional ones, which were rejected because of preferences between non-conflicting rules. We also present a simple and natural transformation from logic programs with preferences to logic programs without preferences such that the answer sets of the transformed program (modulo new special-purpose literals) are exactly the preferred answer sets of an original one.

The rest of the paper is organized as follows. Section 2 recapitulates preliminaries from logic programming and answer set semantics. Section 3 informally describes the approach. Section 4 develops an alternative definition of the answer set semantics that is extended to a preferred answer set semantics in Section 5. Section 6 presents a transformation from logic programs with preferences to logic programs without preferences. Section 7 analyses the properties of the semantics. In Section 8 we show the connection between the semantics and existing approaches. Section 9 summarizes the paper.

All the proofs not presented here can be found in the technical report [15].

2 Preliminaries

Let *At* be a set of all atoms. A *literal* is an atom or expression $\neg a$, where a is an atom. A *rule* is an expression of the form $l_0 \leftarrow l_1, \dots, l_m, not\ l_{m+1}, \dots, not\ l_n$, where $0 \le m \le n$, and each l_i $(0 \le i \le n)$ is a literal. Given a rule r of the above form we use $head(r) = l_0$ to denote the *head* of r, $body(r) = \{l_1 \dots, not\ l_n\}$ the *body* of r. Moreover, $body^+(r) = \{l_1, \dots, l_m\}$ denotes the *positive* body of r, and $body^-(r) = \{l_{m+1}, \dots, l_n\}$ the *negative* body of r. For a set of rules R, $head(R) = \{head(r) : r \in R\}$. A *logic program* is a finite set of rules.

A set of literals S is consistent iff $a \in S$ and $\neg a \in S$ holds for no atom a. A set of literals S *satisfies*: (i) the body of a rule r iff $body^+(r) \subseteq S$, and $body^-(r) \cap S = \emptyset$, (ii) a rule r iff $head(r) \in S$ whenever S satisfies $body(r)$, (iii) a logic program P iff S satisfies each rule of P.

A logic program P is *head-consistent* [14] iff $head(P)$ is consistent. A logic program without *not* is called *positive.*

For a positive logic program P, an *answer set* is defined as the least consistent set of literals satisfying P, and we denote it by $\mathcal{M}(P)$.

The Gelfond-Lifschitz *reduct* of a program P w.r.t. a set of literals S, denoted P^S, is the set $\{head(r) \leftarrow body^+(r) : r \in P \text{ and } body^-(r) \cap S = \emptyset\}$.

Definition 1 (Answer sets [7]). *A consistent set of literals S is an* answer set *of a logic program P iff S is an answer set of P^S.*

We will use $\mathcal{AS}(P)$ to denote the set of all the answer sets of a logic program P.

For a set of literals S, we also denote $\Gamma_P(S) = \{r \in P : \mathit{body}^+(r) \subseteq S \text{ and } \mathit{body}^-(r) \cap S \neq \emptyset\}$.

We will say that a set of literals S *defeats* a rule r iff $\mathit{body}^-(r) \cap S \neq \emptyset$. A set of rules R *defeats* a rule r iff $\mathit{head}(R)$ defeats r.

A *dependency graph* of a program P is an oriented labeled graph where (i) the literals are the vertices, (ii) there is an edge labeled $+$, called *positive*, from a vertex a to a vertex b iff there is rule $r \in P$ with $\mathit{head}(r) = a$ and $b \in \mathit{body}^+(r)$, (iii) there is an edge labeled $-$, called *negative*, from a vertex a to a vertex b iff there is rule $r \in P$ with $\mathit{head}(r) = a$ and $b \in \mathit{body}^-(r)$.

A program P is called *call-consistent* [9, 11] iff its dependency graph contains no cycle with an odd number of negative edges.

Definition 2. *A* logic program with preferences *is a pair $(P, <)$ where: (i) P is a logic program, and (ii) $<$ is a transitive and asymmetric relation on P. If $p < r$ for $p, r \in P$ we say that r is* preferred over *p.*

3 Informal Presentation

The logic programming way of encoding conflicting rules is adding guards to rules in the form of default negated literals. They prevent a program to have an answer set with conflicting literals, and cause the program to have multiple answer sets if both the conflicting rules are "applicable".

Example 1. Consider the well known penguin-fly program P:

r_1: $\mathit{flies} \leftarrow \mathit{bird}, \mathit{not}\ \neg \mathit{flies}$
r_2: $\neg \mathit{flies} \leftarrow \mathit{penguin}, \mathit{not}\ \mathit{flies}$
r_3: $\mathit{bird} \leftarrow$
r_4: $\mathit{penguin} \leftarrow$

The rules r_1 and r_2 have contrary heads, and contain guards $\mathit{not}\ \neg \mathit{flies}$ and $\mathit{not}\ \mathit{flies}$. Consequently, the program has the answer sets $A_1 = \{\mathit{bird}, \mathit{penguin}, \mathit{flies}\}$, and $A_2 = \{\mathit{bird}, \mathit{penguin}, \neg \mathit{flies}\}$ generated by the rules $R_1 = \Gamma_P(A_1) = \{r_1, r_3, r_4\}$ and $R_2 = \Gamma_P(A_2) = \{r_2, r_3, r_4\}$, respectively.

An answer set can be associated with a set of rules that generate it. We can alternatively see the answer set semantics as a guess and test whether a set of rules is a generating set of rules: (i) we guess a set R of all the generating rules of an answer set, (ii) we compute the reduct P^R – we remove each rule of P that is defeated by R, (iii) we compute the set $\mathcal{Q}(P^R)$ of all the rules that have the positive bodies supported in a non-cyclic way (iv) if our guess R coincides with $\mathcal{Q}(P^R)$, then R is a set of all the generating rules of an answer set $S = \mathit{head}(R)$.

The question is how the semantics changes in presence of preferences. Informally, our understanding of preferences on rules is as follows. *It cannot be the case that a rule is made "inapplicable" because of a less preferred conflicting rule.* This informal understanding of preferences is realized here as a condition that during construction of a reduct *a rule cannot be removed because of another conflicting less preferred rule.*

Example 2. Now consider the program from Example 1 with the rule r_2 being preferred over the rule r_1. We expect the program to have the unique preferred answer set A_2.

Assume the guess $R_1 = \{r_1, r_3, r_4\}$. We have that $R_1 \subseteq P^{R_1}$ as R_1 defeats no rule from R_1. We see that r_2 can be potentially removed as $r_1 \in R_1$ and $\mathit{head}(r_1) \in \mathit{body}^-(r_2)$. However, we keep it as r_2 and r_1 are conflicting, and r_1 is less preferred than r_2. Hence $P^{R_1} = \{r_1, r_2, r_3, r_4\}$. All the positive bodies of the rules from P^{R_1} are supported in a non-cyclic way, i.e., $\mathcal{Q}(P^{R_1}) = P^{R_1}$. Since $\mathcal{Q}(P^{R_1}) \neq R_1$, we have that $A_1 = \mathit{head}(R_1)$ is not a preferred answer set. Analogously we get that $A_2 = \mathit{head}(R_2)$ is a preferred answer set.

In the next section we make precise the alternative definition of answer sets, upon which the definition of preferred answer sets is built.

4 Alternative Definition of Answer Sets

When working with preferences on rules, we need to work on the level of rules rather than on the level of literals. We need to check which rules are used to generate an answer set, compare the rules w.r.t. preference relation, make a rule inapplicable, etc. Therefore in order to keep the definition of preferred answer sets as simple as possible, we reformulate answer set semantics in the terms of sets of rules rather than sets of literals.

First, we define when a set of rules positively satisfies the program. It is an alternative notion to the notion that a set of literals satisfies a positive program.

Definition 3 (Positive satisfaction). *Let P be a set of rules. A set of rules $R \subseteq P$* positively satisfies *P iff for each rule $r \in P$ we have that: If $\mathit{body}^+(r) \subseteq \mathit{head}(R)$, then $r \in R$. We will use $\mathcal{Q}(P)$ to denote the least (w.r.t. $\subseteq$) set of rules that positively satisfies P.*

Informally, $\mathcal{Q}(P)$ is the set of the rules that are applied during the bottom-up evaluation of a program P.

Example 3. Consider the following program P

r_1:	a	$\leftarrow$	b	r_3:	a	$\leftarrow$	
r_2:	b	$\leftarrow$		r_4:	d	$\leftarrow$	c

$\{a, b\}$ and $\{a, b, d\}$ satisfy the program. Alternatively $\{r_1, r_2, r_3\}$ and $\{r_1, r_2, r_3, r_4\}$ positively satisfy the program. We also have that $\mathcal{M}(P) = \{a, b\}$. Alternatively $\mathcal{Q}(P) = \{r_1, r_2, r_3\}$. Note that $\mathit{head}(\mathcal{Q}(P)) = \mathcal{M}(P)$.

Second, we define an alternative version of Gelfond-Lifschitz reduction.

Definition 4 (Reduct). *Let P be a logic program, and $R \subseteq P$ be a set of rules. The* reduct *P^R is obtained from P by removing each rule r with $head(R) \cap body^-(r) \neq \emptyset$.*

Now, we are ready to define answer sets.

Definition 5 (Generating set). *Let P be a logic program. A set of rules $R \subseteq P$ is a* generating set *of P iff $R = Q(P^R)$.*

Definition 6 (Answer set). *Let P be a logic program. A consistent set of literals S is an* answer set *of P iff there is a generating set R such that $head(R) = S$.*

The following example illustrates the alternative definition of answer sets alongside the original one.

Example 4. Consider the following program P:

r_1:	a	$\leftarrow$	*not* b
r_2:	c	$\leftarrow$	d, *not* b
r_3:	b	$\leftarrow$	*not* a

We will show that $S = \{a\}$ is an answer set.

	Gelfond-Lifschitz definition	alternative definition
guess	$S = \{a\}$	$R = \{r_1\}$
reduct	$P^S = \{a \leftarrow, c \leftarrow d\}$	$P^R = \{r_1, r_2\}$ r_3 is removed as $body^-(r_3) \cap head(R) \neq \emptyset$
"min model"	$\mathcal{M}(P^S) = \{a\}$	$\mathcal{Q}(P^R) = \{r_1\}$ r_2 is not included as $d \in body^+(r_2)$ cannot be derived
test	$\mathcal{M}(P^S) = S$	$\mathcal{Q}(P^R) = R$
conclusion	S is an answer set	R is a generating set and $head(R) = S$ is an answer set

The next propositions justify the name "generating set". It turns out that an answer set S of a program P is represented by a unique generating set, namely the set $\Gamma_P(S)$.

Proposition 1. *Let P be a logic program. Let R_1 and R_2 be generating sets such that $head(R_1) = head(R_2)$. Then $R_1 = R_2$.*

Proposition 2. *Let P be a logic program. Let R be a generating set, and S be a consistent set of literals such that $S = head(R)$.*
Then $\Gamma_P(S) = R$.

Theorem 1. *Let P be a logic program. A consistent set of literals S is an answer set of P (according to Definition 6) iff $\Gamma_P(S)$ is a generating set and $head(\Gamma_P(S)) = S$.*

The alternative definition of answer sets defines the same answer sets as the Gelfond-Lifschitz definition does.

Theorem 2. *Let P be a logic program. A consistent set of literals S is an answer set (according to Definition 1) iff S is an answer set (according to Definition 6).*

5 Preferred Answer Sets

In this section, we define preferred answer sets by formalizing the intuitions from Section 3.

In order to develop a definition of preferred answer sets we need to make clear what exactly a conflict is.

Definition 7 (Conflict). *Let r_1, r_2 be rules. We say that r_1 and r_2 are* conflicting *iff: (i) $head(r_1) \in body^-(r_2)$, and (ii) $head(r_2) \in body^-(r_1)$.*

We obtain the definition of preferred answer sets by requiring in Definition 4 that less preferred conflicting rules cannot cause a rule to be removed.

Definition 8 (Override). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. Let r_1 and r_2 be rules. We say that r_1 overrides r_2 iff (i) r_1 and r_2 are conflicting, and (ii) $r_2 < r_1$.*

Definition 9 (Reduct). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences, and $R \subseteq P$ be a set of rules. The* reduct *$\mathcal{P}^R$ is obtained from P by removing each rule $r \in P$ such that there is a rule $q \in R$ such that:*

– *$head(q) \in body^-(r)$, and*
– *r does not override q.*

Note that Definition 4 and Definition 9 differ only in the second condition. After removing it, the obtained definition is equivalent with Definition 4. Moreover, when a preference relation is empty, the reduct coincides with the reduct for logic programs without preferences as defined in Definition 4, which in turn corresponds to Gelfond-Lifschitz reduct.

Proposition 3. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. Let $R \subseteq P$ be a set of rules. If $<= \emptyset$, then $\mathcal{P}^R = P^R$.*

Definition 10 (Preferred generating set). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. A set of rules $R \subseteq P$ is a* preferred generating set *iff $R = Q(\mathcal{P}^R)$.*

Proposition 4. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences, and $R \subseteq P$ be a set of rules. If R is a preferred generating set, then R is a generating set.*

Definition 11 (Preferred answer set). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. A consistent set of literals S is a* preferred answer set *of $\mathcal{P}$ iff there is a preferred generating set R such that $head(R) = S$.*

We will use $\mathcal{PAS}(\mathcal{P})$ to denote the set of all the preferred answer sets of a logic program with preferences $\mathcal{P}$.

From Proposition 4 we have that analogous versions of Proposition 1 and Theorem 1 hold also for preferred answer sets.

Notice that the definition of preferred answer sets is almost identical to the definition of answer sets. The only difference is the additional simple condition in the definition of reduct. Consider we would adapt the original definition of reduct rather than the alternative one. It would be as follows: *Given a set of literals S we remove from a program P each rule r such that there exists a rule q where ...* Since the guess S is a set of literals, it is not straightforward where to get q from, i.e., we would have to introduce additional conditions that q must meet. We would then have to justify those conditions. On the other hand, the use of the alternative definition of reduct has completely freed us from this job as no such conditions are needed. Exactly this elegance was the main motivation for the alternative definition of answer sets.

6 Transformation to Logic Programs without Preferences

A logic program with preferences under the preferred answer set semantics as defined in Definition 11 can be transformed to a logic program without preferences such that the preferred answer sets of the original program are exactly the standard answer sets of the transformed program (modulo new special-purpose literals).

The basic idea of the transformation is to remove a default negated literal from the body of a rule if it is the head of a less preferred conflicting rule.

Example 5. Consider the program in the first column.

$$\begin{array}{llll c lll}
r_1: & a & \leftarrow & \mathit{not}\ b & & a & \leftarrow & \\
r_2: & b & \leftarrow & \mathit{not}\ a & \mapsto & b & \leftarrow & \mathit{not}\ a \\
 & r_2 < r_1 & & & & & &
\end{array}$$

We have that r_1 and r_2 are conflicting, and r_1 is preferred. Hence r_2 cannot defeat the rule r_1. Therefore we remove *not* b from r_1's body.

However, the situation is complicated if at least two rules have the same head. In general, we have to distinguish which rule can defeat a rule and which cannot. In order to do so, for each rule r we introduce a special-purpose literal n_r that denotes that r is applicable, and replace default negated literal *not* x by a collection of *not* n_r such that $\mathit{head}(r) = x$.

Example 6. Consider the program in the first column. First (the second column) we split a rule into two rules: (i) the first deriving n_r whenever r is applicable, and (ii) the second deriving $\mathit{head}(r)$ of the original rule. We also use n_t literals in the negative bodies. Next (the third column) we remove *not* n_q from the negative body of r if q is a less preferred conflicting rule.

$$\begin{array}{llll}
r_1\colon & a & \leftarrow & not\ b \\
r_2\colon & b & \leftarrow & not\ a \\
r_3\colon & b & \leftarrow & c \\
 & \multicolumn{3}{l}{r_2 < r_1}
\end{array}
\quad\mapsto\quad
\begin{array}{lll}
n_{r_1} & \leftarrow & not\ n_{r_2}, not\ n_{r_3} \\
a & \leftarrow & n_{r_1} \\
n_{r_2} & \leftarrow & not\ n_{r_1} \\
b & \leftarrow & n_{r_2} \\
n_{r_3} & \leftarrow & c \\
b & \leftarrow & n_{r_3}
\end{array}
\quad\mapsto\quad
\begin{array}{lll}
n_{r_1} & \leftarrow & not\ n_{r_3} \\
a & \leftarrow & n_{r_1} \\
n_{r_2} & \leftarrow & not\ n_{r_1} \\
b & \leftarrow & n_{r_2} \\
n_{r_3} & \leftarrow & c \\
b & \leftarrow & n_{r_3}
\end{array}$$

The next definition formalizes the transformation.

Definition 12 (Transformation). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences, and $r \in P$ be a rule. The* names of r's potential blockers *are*

$$B_{\mathcal{P}}(r) = \{n_q : q \in P, head(q) \in body^-(r), \text{ and } r \text{ does not override } q\}$$

The transformation $t_{\mathcal{P}}(r)$ *of a rule r is the set of the rules*

$$head(r) \leftarrow n_r \tag{1}$$

$$n_r \leftarrow body^+(r), not\ B_{\mathcal{P}}(r) \tag{2}$$

The transformation $t(\mathcal{P})$ *of a program $\mathcal{P}$ is given by $\bigcup_{r \in P} t_{\mathcal{P}}(r)$.*

It can be easily seen from the definition that the transformation can be computed in polynomial time, and the size of the transformed program is polynomial in the size of an original one.

The transformation captures semantics of preferred answer sets.

Theorem 3. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. Let S be a consistent set of literals (of the program $\mathcal{P}$). Let T be a consistent set of literals (of the program $t(\mathcal{P})$). Let $N_{\mathcal{P}}(S) = \{n_r : r \in \Gamma_P(S)\}$ and $N(\mathcal{P}) = \{n_r : r \in P\}$.*

If S is a preferred answer set of $\mathcal{P}$, then $S \cup N_{\mathcal{P}}(S)$ is an answer set of $t(\mathcal{P})$. If T is an answer set of $t(\mathcal{P})$, then $T \setminus N(\mathcal{P})$ is a preferred answer set of $\mathcal{P}$.

Next we show that the transformation does not introduce cycles with odd number of negative edges.

Proposition 5. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. If P is call-consistent, then $t(\mathcal{P})$ is call-consistent.*

Proof. In the following we use the notation: $t(l) = l$ if l is a literal from P, and $t(n_r) = head(r)$ for $r \in P$.

Assume there is an odd cycle in the $t(\mathcal{P})$'s dependency graph: there is a sequence $l_1, s_1, \ldots, s_{n-1}, l_n = l_1$ such that (l_i, l_{i+1}, s_i) is a labeled edge for each $i < n$, and for some $i < n$ we have $s_i = -$. Assume a literal l_i and an edge (l_i, l_{i+1}, s).

If l_i is a literal from P, then the edge came from a rule of the form (1). Hence $l_{i+1} = n_r$ for some $r \in P$, $s = +$, and $head(r) = l_i$.

If $l_i = n_r$ for some $r \in P$, then: (i) If $s = +$ then the edge came from a rule of the form (2). Hence l_{i+1} is a literal from P, $l_{i+1} \in body^+(r)$, and $(t(l_i), t(l_{i+1}), +)$

is an edge in P's dependency graph. (ii) If $s = -$ then the edge came from a rule of the form (2). Hence $l_{i+1} = n_p$ for some $p \in P$, $head(p) \in body^-(r)$, and $(t(l_i), t(l_{i+1}), -)$ is an edge in P's dependency graph.

Now, we create a new sequence by iterating over the sequence $l_1, \ldots, l_n$: (i) If $i < n$ and l_i is a literal of the program P skip l_i and s_i. From the above analysis we have $t(l_{i+1}) = l_i$, and $s_i = +$, hence the literal will be added in the next step, and no negative edge is lost, (ii) If $i < n$ and $l_i = n_r$ for some $r \in P$, add $t(l_i), s_i$ to the end of the resulting sequence. (iii) If $i = n$ add $t(l_i)$ to the end of the resulting sequence. From the above analysis we have that the sequence forms a cycle in P's dependency graph, and the number of negative edges is preserved. □

If P is not call-consistent, then $t(\mathcal{P})$ can be call-consistent for some $<$ and not call-consistent for other $<$.

Example 7. Consider the program $\mathcal{P} = (P, <)$:

r_1:	a	$\leftarrow$	$not\ b, not\ c$
r_2:	b	$\leftarrow$	$not\ a, not\ c$
r_3:	c	$\leftarrow$	$not\ b$

If $r_3 < r_2 < r_1$, then $t(\mathcal{P})$:

n_{r_1}	$\leftarrow$	$not\ c$	n_{r_2}	$\leftarrow$	$not\ a$	n_{r_3}	$\leftarrow$	$not\ b$
a	$\leftarrow$	n_{r_1}	b	$\leftarrow$	n_{r_2}	c	$\leftarrow$	n_{r_3}

is not call-consistent.

If $r_1 < r_2 < r_3$, then $t(\mathcal{P})$:

n_{r_1}	$\leftarrow$	$not\ b, not\ c$	n_{r_2}	$\leftarrow$	$not\ c$	n_{r_3}	$\leftarrow$	
a	$\leftarrow$	n_{r_1}	b	$\leftarrow$	n_{r_2}	c	$\leftarrow$	n_{r_3}

is call-consistent.

7 Properties of Preferred Answer Sets

In this section we show that the semantics enjoys several nice properties. First of all, the semantics is selective, i.e., each preferred answer set is an answer set.

Theorem 4. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. Then $\mathcal{PAS}(\mathcal{P}) \subseteq \mathcal{AS}(P)$.*

Proof. The theorem follows directly from Proposition 4. □

Second, for the two simple classes of programs: (i) programs with an empty preference relation, and (ii) stratified programs, the semantics is equivalent to the answer set semantics.

Theorem 5. *Let $\mathcal{P} = (P, \emptyset)$ be a logic program with preferences. Then $\mathcal{PAS}(\mathcal{P}) = \mathcal{AS}(P)$.*

Theorem 6. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences where P is stratified. Then $\mathcal{PAS}(\mathcal{P}) = \mathcal{AS}(P)$.*

Proof. If P is stratified, then there are no conflicting rules. Hence $P^R = \mathcal{P}^R$. $\square$

Next we show that our semantics satisfies both principles for preferential reasoning proposed in [2] by Brewka and Eiter.

Principle I tries to capture the meaning of preferences. If two answer sets are generated by the same rules except for two rules, the one generated by a less preferred rule is not preferred.

Principle I ([2]) *Let $\mathcal{P} = (P, <)$ be a logic program with preferences, A_1, A_2 be two answer sets of P. Let $\Gamma_P(A_1) = R \cup \{d_1\}$ and $\Gamma_P(A_2) = R \cup \{d_2\}$ for $R \subset P$. Let $d_2 < d_1$. Then $A_2 \notin \mathcal{PAS}(\mathcal{P})$.*

Theorem 7 (Principle I is satisfied). *Preferred answer sets as defined in Definition 11 satisfy Principle I.*

Proof. Assume that A_2 is a preferred answer set. Hence $\Gamma_P(A_2)$ is a preferred generating set, i.e., $\Gamma_P(A_2) = \mathcal{Q}(\mathcal{P}^{\Gamma_P(A_2)})$. We have that d_2 is the only rule $r \in \Gamma_P(A_2)$ with $head(r) \in body^-(d_1)$. We also have that d_1, and d_2 are conflicting, and $d_2 < d_1$. Hence $d_1 \in \mathcal{P}^{\Gamma_P(A_2)}$, and consequently $d_1 \in \Gamma_P(A_2)$. A contradiction. Therefore $\Gamma_P(A_2)$ is not a preferred generating set. $\square$

Principle II says that a particular type of preferences is irrelevant.

Principle II ([2]) *Let $\mathcal{P} = (P, <)$ be a logic program with preferences, $S \in \mathcal{PAS}(\mathcal{P})$ and r be a rule such that $body^+(r) \not\subseteq S$. Let $\mathcal{P}' = (P', <')$ be a logic program with preferences, where $P' = P \cup \{r\}$ and $<' \cap (P \times P) = <$. Then $S \in \mathcal{PAS}(\mathcal{P}')$.*

Theorem 8 (Principle II is satisfied). *Preferred answer sets as defined in Definition 11 satisfy Principle II.*

Proof. Let S be a preferred answer set of $\mathcal{P}$, i.e., there is a set of rules $R \subseteq P$ such that $R = \mathcal{Q}(\mathcal{P}^R)$ and $head(R) = S$. We show that $R = \mathcal{Q}(\mathcal{P}'^R)$.

First we show that R positively satisfies $\mathcal{P}'^R$. Assume $p \in \mathcal{P}'^R$ such that $body^+(p) \subseteq head(R)$. Hence $p \in \mathcal{P}'$ and $p \neq r$. Hence $p \in P$ and $p \in \mathcal{P}^R$. As R positively satisfies $\mathcal{P}^R$ we have that $p \in R$.

Second, we show that no proper subset of R positively satisfies $\mathcal{P}'^R$. Assume that $V \subset R$ positively satisfies $\mathcal{P}'^R$. Since $\mathcal{P}^R \subseteq \mathcal{P}'^R$, we have that V positively satisfies $\mathcal{P}^R$. A contradiction with $R = \mathcal{Q}(\mathcal{P}^R)$. Hence such V does not exist.

Therefore $\mathcal{Q}(\mathcal{P}'^R) = R$. $\square$

On the other hand, the semantics violates Principle III[1]. It requires that a program has a preferred answer set whenever a standard answer set exists. It follows the view that the addition of preferences should not cause a consistent program to be inconsistent.

[1] It is an idea from Proposition 6.1 from [2]. Brewka and Eiter did not consider it as a principle. On the other hand [13] did.

Principle III *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. If $\mathcal{AS}(P) \neq \emptyset$, then $\mathcal{PAS}(\mathcal{P}) \neq \emptyset$.*

Theorem 9 (Principle III is violated). *Preferred answer sets as defined in Definition 11 violate Principle III.*

Proof. Consider the following program $\mathcal{P} = (P, <)$:

r_1:	a	$\leftarrow$	$not\ b$	r_3:	inc	$\leftarrow$	$a, not\ inc$
r_2:	b	$\leftarrow$	$not\ a$		$r_2 < r_1$		

P has the unique answer set $\{b\}$. However it is not a preferred one. □

Even though the semantics does not guarantee existence of a preferred answer set when a standard answer set exists for the class of all programs (which is believed to rise computational complexity), it ensures existence of a preferred answer set for a subclass of programs.

Theorem 10. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences such that P is call-consistent and head-consistent. Then $\mathcal{PAS}(\mathcal{P}) \neq \emptyset$.*

Proof. Since P is head-consistent, we can assume explicitly negated literals to be new literals and view $t(\mathcal{P})$ as a normal logic program. From Proposition 5 we have that $t(P)$ is call-consistent. Then from Theorem 5.8 from [5] we get that $\mathcal{AS}(t(\mathcal{P})) \neq \emptyset$. Finally we get $\mathcal{PAS}(\mathcal{P}) \neq \emptyset$ using Theorem 3. □

Theorem 11. *Deciding whether $\mathcal{PAS}(\mathcal{P}) \neq \emptyset$ for a logic program with preferences $\mathcal{P}$ is NP-complete.*

Proof. Membership: $t(\mathcal{P})$ can be computed in polynomial time. Using Theorem 3 the decision problem whether $\mathcal{PAS}(\mathcal{P}) \neq \emptyset$ can be reduced to the decision whether $\mathcal{AS}(t(\mathcal{P})) \neq \emptyset$, which is in NP. *Hardness:* Deciding whether $\mathcal{AS}(P) \neq \emptyset$ for a logic program P is NP-complete. Using Theorem 5 we can reduce it to decision whether $\mathcal{PAS}((P, \emptyset)) \neq \emptyset$. □

8 Comparison with Existing Approaches

In this section we investigate the connection of preferred answer sets as defined in Definition 11 to existing approaches. We focus our attention to the selective approaches that stay in the NP complexity class.

In [12] Schaub and Wang have shown that the approaches $\mathcal{PAS}_{DST}$ [4], $\mathcal{PAS}_{WZL}$ [17] and $\mathcal{PAS}_{BE}$ [2] form a hierarchy. We will use $\mathcal{PAS}_{DST}(\mathcal{P})$, $\mathcal{PAS}_{WZL}(\mathcal{P})$ and $\mathcal{PAS}_{BE}(\mathcal{P})$ to denote the set of all the preferred answer sets of a program according to the respective semantics.

Theorem 12 ([12]). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. Then $\mathcal{PAS}_{DST}(\mathcal{P}) \subseteq \mathcal{PAS}_{WZL}(\mathcal{P}) \subseteq \mathcal{PAS}_{BE}(\mathcal{P}) \subseteq \mathcal{AS}(P)$.*

We show that our semantics continues in this hierarchy. We start by an alternative definition of our semantics.

Definition 13. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. An answer set X of P is called* <-satisfying *iff for each $p \in \Gamma_P(X)$ and $r \in P \setminus \Gamma_P(X)$ such that $p < r$ we have that:*

– *$body^+(r) \not\subseteq X$, or*
– *$body^-(r) \cap \{head(t) : t \in \Gamma_P(X)$ and r does not override $t\} \neq \emptyset$.*

Lemma 1. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. A consistent set of literals X is a preferred answer set iff X is a $<$-satisfying answer set of P.*

Proof. ($\Rightarrow$) There is a set of rules $R = \mathcal{Q}(\mathcal{P}^R)$ such that $X = head(R)$.

Assume there are $p \in \Gamma_P(X)$ and $r \in P \setminus \Gamma_P(X)$ such that $p < r$. Since R is a generating set, we have $R = \Gamma_P(X)$. Hence $r \notin R = \mathcal{Q}(\mathcal{P}^R)$. From that $body^+(r) \not\subseteq X = head(R)$ or $r \notin \mathcal{P}^R$. If $r \notin \mathcal{P}^R$, then there must be a rule $t \in R$ such that $head(t) \in body^-(r)$ and r does not override t.

($\Leftarrow$) Let $R = \Gamma_P(X)$ and consider $\mathcal{P}^R$. Since $R = \Gamma_P(X)$, we have $R \subseteq \mathcal{P}^R$ and $R \subseteq \mathcal{Q}(\mathcal{P}^R)$.

Assume that $\mathcal{Q}(\mathcal{P}^R) \not\subseteq R$, i.e., there is a rule $r \in \mathcal{Q}(\mathcal{P}^R)$ such that $r \notin R$.

Since $r \in \mathcal{Q}(\mathcal{P}^R)$, we have that $body^+(r) \subseteq head(\mathcal{Q}(\mathcal{P}^R))$, i.e., every literal in $body^+(r)$ is supported by a rule in $\mathcal{Q}(\mathcal{P}^R)$.

As $r \notin R$, we have that $body^+(r) \not\subseteq X = head(R)$ or $body^-(r) \cap head(R) \neq \emptyset$.

If $body^-(r) \cap head(R) \neq \emptyset$, there is a rule $p \in R$ with $head(p) \in body^-(r)$. Since $r \in \mathcal{P}^R$, we have that r overrides p. Hence there is $p \in \Gamma_P(X)$, $r \in P \setminus \Gamma_P(X)$, and $p < r$.

Assume there is $t \in R$ such that $head(t) \in body^-(r)$ and r does not override t. Then $r \notin \mathcal{P}^R$. A contradiction. Hence no such t exists. Therefore we get $body^+(r) \not\subseteq X$ as X is $<$-satisfying.

We have shown that $body^+(r) \not\subseteq X = head(R)$. Then there is a literal in $body^+(r)$ that is not supported by a rule from R. Hence there is a literal in $body^+(r)$ that is supported solely by a rule from $\mathcal{Q}(\mathcal{P}^R) \setminus R$. Hence each rule in $\mathcal{Q}(\mathcal{P}^R) \setminus R$ positively depends on a literal that can be derived only by a rule from $\mathcal{Q}(\mathcal{P}^R) \setminus R$. Then from minimality of $\mathcal{Q}(\mathcal{P}^R)$ we get that $r \notin \mathcal{Q}(\mathcal{P}^R)$. A contradiction. Therefore $\mathcal{Q}(\mathcal{P}^R) \subseteq R$.

Finally $\mathcal{Q}(\mathcal{P}^R) = R$, and $X = head(R)$ is a preferred answer set of $\mathcal{P}$. □

Definition 14 (Alternative definition of $\mathcal{PAS}_{BE}$ [12]). *Let $\mathcal{P} = (P, <)$ be a logic program with preferences. An answer set X of P is a* BE preferred answer set *of $\mathcal{P}$ iff there is an enumeration $\langle r_i \rangle$ of $\Gamma_P(X)$ such that for each i, j:*

– *if $r_i < r_j$, then $j < i$, and*
– *if $r_i < r$ and $r \in P \setminus \Gamma_P(X)$, then*
  • *$body^+(r) \not\subseteq X$ or*
  • *$body^-(r) \cap \{head(r_j) : j < i\} \neq \emptyset$ or*
  • *$head(r) \in X$*

The difference between our semantics and $\mathcal{PAS}_{BE}$ can be seen directly from Definitions 13 and 14. One of the main differences is that Definition 13 completely drops the condition for enumeration of the rules, i.e., *preferences are not interpreted as an order, in which the rules are applied.* As the result, the second condition requiring how a preferred rule is defeated changes. A preferred rule can be defeated only by a rule that is not less preferred and conflicting. Hence the second difference: an *explicit definition of conflict* is used, and *preferences between non-conflicting rules are ignored.* The condition $head(r) \in X$ is also completely dropped.

Very similar differences hold for $\mathcal{PAS}_{DST}$ and $\mathcal{PAS}_{WZL}$ (Definitions of $\mathcal{PAS}_{DST}$ and $\mathcal{PAS}_{WZL}$, similar to Definition 14, can be found in [12]).

Theorem 13. *Let $\mathcal{P}$ be a logic program with preferences. Then $\mathcal{PAS}_{BE}(\mathcal{P}) \subseteq \mathcal{PAS}(\mathcal{P})$.*

Proof. Let $\mathcal{P}$ be a logic program with preferences and $X \in \mathcal{PAS}_{BE}(\mathcal{P})$. Then there is an enumeration $\langle r_i \rangle$ of $\Gamma_P(X)$ satisfying the conditions from Definition 14.

Let $p \in \Gamma_P(X)$ and $r \in P \backslash \Gamma_P(X)$ such that $p < r$. Assume that (i) $body^+(r) \subseteq X$, and (ii) for each rule $t \in \Gamma_P(X)$ such that $head(t) \in body^-(r)$ it holds that r overrides t.

Since $body^+(r) \subseteq X$ and $r \notin \Gamma_P(X)$, we have that $body^-(r) \cap X \neq \emptyset$. There is a rule $t \in \Gamma_P(X)$ with $head(t) \in body^-(r)$. Then t and r are conflicting, i.e., $head(r) \in body^-(t)$. Since $t \in \Gamma_P(X)$ we have that $head(r) \notin X$.

Since $body^+(r) \subseteq X$ and $head(r) \notin X$, we have that there is a rule $r_k \in \Gamma_P(X)$ with $head(r_k) \in body^-(r)$ for $k < i$. From the conditions above, we have that r_k and r are conflicting and $r_k < r$. By the same argument as before, there is a rule $r_l \in \Gamma_P(X)$ with $head(r_l) \in body^-(r)$ and $r_l < r$ for some $l < k$, and so on, until we reach the beginning of the enumeration and no such rule can be found. A contradiction. Hence (i) $body^+(r) \not\subseteq X$, or (ii) $body^-(r) \cap \{head(t) : t \in R \text{ and } r \text{ does not override } t\} \neq \emptyset$.

Therefore X is $<$-satisfying, and $X \in \mathcal{PAS}(\mathcal{P})$. □

Theorem 14. *It does not hold $\mathcal{PAS}(\mathcal{P}) \subseteq \mathcal{PAS}_{BE}(\mathcal{P})$ for each logic program with preferences $\mathcal{P}$.*

Proof. Consider the program $\mathcal{P}$ from Example 5.5 from [2]

r_1:	c	$\leftarrow$	*not b*	
r_2:	b	$\leftarrow$	*not a*	$r_2 < r_1$

The program is stratified. $\mathcal{PAS}_{BE}(\mathcal{P}) = \emptyset$. On the other hand $\mathcal{PAS}(\mathcal{P}) = \{\{b\}\}$.

Theorem 15. *Let $\mathcal{P} = (P, <)$ be a logic program with preferences.*
Then $\mathcal{PAS}_{DST}(\mathcal{P}) \subseteq \mathcal{PAS}_{WZL}(\mathcal{P}) \subseteq \mathcal{PAS}_{BE}(\mathcal{P}) \subseteq \mathcal{PAS}(\mathcal{P}) \subseteq \mathcal{AS}(P)$.

Proof. It follows directly from Theorem 4, Theorem 12 and Theorem 13. □

Theorem 14 and Theorem 15 can be interpreted as follows. Our semantics continues in the hierarchy of approaches $\mathcal{PAS}_{DST}$, $\mathcal{PAS}_{WZL}$ and $\mathcal{PAS}_{BE}$. It preserves the preferred answer sets of these semantics and admits additional ones, which were rejected because of preferences between non-conflicting rules.

Theorem 6 is another distinguishing feature of our semantics. None of the approaches $\mathcal{PAS}_{DST}$, $\mathcal{PAS}_{WZL}$ and $\mathcal{PAS}_{BE}$ satisfies it. We consider Theorem 6 to be important as a stratified program contains no conflicting rules and its meaning is given by a unique answer set. Theorem 6 also allows us to resolve the problematic program shown in the proof of Theorem 14.

9 Conclusion and Future Work

In this paper we have developed a descriptive semantics for logic programs with preferences on rules. The main idea is to add an additional condition to the Gelfond-Lifschitz reduction: a rule cannot be removed because of a conflicting less preferred rule. As a result, the approach uses an explicit definition of conflicting rules and ignores preferences between non-conflicting rules. This feature, not present in other approaches, is important for scenarios where preferences between rules are automatically induced from preferences between modules, as we do not want such preferences to cause any side effects.

The semantics continues in the hierarchy of approaches [4, 17, 2]. It preserves the preferred answer sets of these semantics and admits additional ones, which were rejected because of preferences between non-conflicting rules. The semantics satisfies both principles for preferential reasoning proposed in [2]. In contrast to [4, 17, 2], it guarantees existence of a preferred answer set for the class of call-consistent head-consistent extended logic programs. The semantics can be also characterized by a transformation from logic programs with preferences to logic programs without preferences such that the preferred answer sets of an original program correspond to the answer sets of the transformed program. The transformation is based on a simple idea: we remove a default negated literal from a rule's body if it is derived by a conflicting less preferred rule. We have also developed a prototypical solver for preferred answer sets using meta-interpretation technique from [6]. A description of the solver can be found in the technical report [15] and an implementation can be downloaded from [16].

In this paper we have only considered the most common type of conflict – the heads of two conflicting rules are in each others negative bodies. In the future work we plan to consider indirect conflicts – literals in negative bodies are not derived directly by conflicting rules, but via other rules. Preliminary results in this direction can be found in technical report [15].

Acknowledgement We would like to thank the anonymous reviewers for detailed and useful comments. This work was supported by the grant UK/276/2013 of Comenius University in Bratislava and 1/1333/12 of VEGA.

References

1. Apt, K.R., Blair, H.A., Walker, A.: Towards a theory of declarative knowledge. In: Foundations of Deductive Databases and Logic Programming, pp. 89–148. Morgan Kaufmann (1988)
2. Brewka, G., Eiter, T.: Preferred answer sets for extended logic programs. Artificial Intelligence 109(1-2), 297–356 (1999)
3. Chandra, A.K., Harel, D.: Horn clauses queries and generalizations. Journal of Logic Programming 2(1), 1–15 (1985)
4. Delgrande, J.P., Schaub, T., Tompits, H.: A framework for compiling preferences in logic programs. Theory and Practice of Logic Programming 3(2), 129–187 (2003)
5. Dung, P.M.: On the relations between stable and well-founded semantics of logic programs. Theoretical Computer Science 105(1), 7–25 (1992)
6. Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Computing preferred answer sets by meta-interpretation in answer set programming. Theory and Practice of Logic Programming 3(4-5), 463–498 (2003)
7. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3/4), 365–386 (1991)
8. Illic, M., Leite, J., Slota, M.: ERASP – a system for enhancing recommendations using answer-set programming. International Journal of Reasoning-based Intelligent Systems (2009)
9. Kunen, K.: Signed data dependencies in logic programs. Journal of Logic Programming 7(3), 231–245 (1989)
10. Sakama, C., Inoue, K.: Prioritized logic programming and its application to commonsense reasoning. Artificial Intelligence 123(1-2), 185–222 (2000)
11. Sato, T.: On consistency of first-order logic programs. Tech. Rep. TR 87-12, ETL (1987)
12. Schaub, T., Wang, K.: A semantic framework for preference handling in answer set programming. Theory and Practice of Logic Programming 3(4-5), 569–607 (2003)
13. Šefránek, J.: Preferred answer sets supported by arguments. In: Proceedings of 12th International Workshop on Non-Monotonic Reasoning (NMR 2008). pp. 232–240 (2008)
14. Turner, H.: Signed logic programs. In: Logic Programming: Proceedngs of the 1994 International Symposium (ILPS'94). pp. 61–75 (1994)
15. Šimko, A.: Logic programming with preferences on rules. Tech. Rep. TR-2013-035, Comenius University in Bratislava (2013), `http://kedrigern.dcs.fmph.uniba.sk/reports/display.php?id=50`
16. Šimko, A.: Meta-interpreter for logic programs with preferences on rules (April 2013), `http://dai.fmph.uniba.sk/~simko/lpp/`
17. Wang, K., Zhou, L., Lin, F.: Alternating fixpoint theory for logic programs with priority. In: Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.K., Palamidessi, C., Pereira, L.M., Sagiv, Y., Stuckey, P.J. (eds.) Computational Logic. Lecture Notes in Computer Science, vol. 1861, pp. 164–178. Springer (2000)
18. Zhang, Y., Foo, N.Y.: Answer sets for prioritized logic programs. In: Proceedings of the 1997 International Logic Programming Symposium (ILPS'97). pp. 69–83 (1997)
19. Zhang, Y., Foo, N.Y.: Towards Generalized Rule-based Updates. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97 (1997)

Construction of Explanation Graphs from Extended Dependency Graphs for Answer Set Programs

Ella Albrecht, Patrick Krümpelmann, Gabriele Kern-Isberner

Technische Universität Dortmund

Abstract. Extended dependency graphs are an isomorphic representation form for Answer Set Programs, while explanation graphs give an explanation for the truth value of a literal contained in an answer set. We present a method and an algorithm to construct explanation graphs from a validly colored extended dependency graph. This method exploits the graph structure of the extended dependency graph for gradually build up explanation graphs. Moreover, show interesting properties and relations of the graph structures, such as loops and the answer set and the well-founded semantics. We also present two different approaches for the determination of assumptions in an extended dependency graph, an optimal but exponential and a sub-optimal but linear one.

1 Introduction

Graphs are an excellent tool for the illustration and understanding of non-monotonic reasoning formalisms and the determination and explanation of models and have a long history. For answer set programs two graph based representations have recently been proposed: extended dependency graphs (EDG) [2] and explanation graphs (EG) [1]. EDGs are an isomorphic representation of extended logic programs and use a coloring of the nodes to determine answer sets. explanation graphs on the other hand afford an explanation for the appearance of a single literal in an answer set. In [1] it was conjectured that there is a strong relation between a colored extended dependency graph and an explanation graph.

In this work we present a method to construct explanation graphs from a successfully colored extended dependency graph and prove its correctness. The way of proceeding exploits the structure of the EDG and the fact that explanation graphs can be built up gradually from smaller sub-explanation graphs. In [1] assumptions are introduced, which describe literals whose truth value has to be guessed during the determination process of answer sets, but there is actually no appropriate method given to determine assumptions. We present two approaches which extract assumptions from an EDG. This is the most difficult part of the construction of EGs, since intra-cyclic as well as inter-cyclic dependencies between nodes have to be considered. The first approach makes use of basic properties of assumptions and the Graph for reduce the size of

assumptions in linear runtime. The second approach exploits cycle structures and their interdependencies to determine the minimal assumptions, which comes with the cost of exponential runtime.
In Section 2 we give an introduction to answer set programming and in 3 we present extended dependency graphs and explanation graphs. Section 4 deals with the construction process of the EGs from a validly colored EDG. The fifth section deals with the different approaches of determining assumptions in an EDG.

2 Answer Set Programming

We consider extended logic programs under the answer set semantics [3]. An *extended logic program* P is a set of rules r of the form $r: \ h \leftarrow a_1, ..., a_n, not\ b_1, ..., not\ b_n$. where $h, a_1, ...a_m, b_1, ..., b_n$ are literals. A *literal* may be of the form x or $\neg x$ where x is a propositional symbol called *atom* and $\neg$ is the classical negation. $head(r) = \{h\}$ denotes the head, $pos(r) = \{a_1, ..., a_m\}$ denotes the positive, and $neg(r) = \{b_1, ..., b_n\}$ the negative body literals of a rule. The *herbrand basis* $\mathcal{H}(P)$ of a logic program P is the set of all grounded literals of P. A literal is *grounded*, if it does not contain a variable. In this work, we assume that the logic programs are grounded, i. e. every literal appearing in the program is grounded.
Let $M \subseteq \mathcal{H}(P)$ be a consistent set of literals, i. e. the M does not contain any complementary literals. M is *closed* under P if $head(r) \in M$ whenever $body(r) \subseteq M$ for every rule $r \in P$. M is an *answer set* for P if M is closed and M is minimal w.r.t. set inclusion.
Answer set semantics may yield multiple models resp. answer sets. Another semantics for logic programs is the well-founded semantics [6]. Its basic idea is that there exist literals which have to be true with certainty and literals which have to be false with certainty. Under the answer set semantics such information gets lost if no answer set exists.
For a logic program P and a logic program P^+ that we get if we remove all rules with negative body literals, the sequence $(K_i, U_i)_{i \geq 0}$ is defined as

$$K_0 = lfp(T_{P^+,\emptyset}),\ U_0 = lfp(T_{P,K_0}),\ K_i = lfp(T_{P,U_{i-1}}),\ U_i = lfp(T_{P,K_i})$$

where $T_{P,V}(S) = \{a \mid \exists r \in P: \ head(r) = a,\ pos(r) \subseteq S,\ neg(r) \cap V = \emptyset\}$. The well-founded model is then $WF_P = \langle W^+, W^- \rangle$ where $W^+ = K_j$ is the *well founded* set and $W^- = \mathcal{H}(P) \setminus U_j$ is the *unfounded* set for the first index j with $\langle K_j, U_j \rangle = \langle K_{j+1}, U_{j+1} \rangle$. Literals that are neither contained in W^+ nor in W^- are called *undefined*.

3 Graphs for Answer Set Programs

We introduce two types of graphs, the first graph type is the extended dependency graph [2]. A main feature of these is their representation of cycles and handles for those. A cycle consists of several literals that are dependent in a cyclic way, e. g. given the two rules $r_1 : a \leftarrow b.$ and $r_2 : b \leftarrow a.$ the literals a and b are interdependent in a cyclic way. Generally, cycles can be connected in two different ways to the rest of the program:

– *OR-handle*: Let the rule $r_1 :\ a \leftarrow \beta.$ be part of a cycle where β may be of the form b or *not* b. If there exists another rule $r_2 :\ a \leftarrow \delta.$ where δ may be of the form d or *not* d, then δ is an OR-handle for the cycle to which r_1 belongs. The OR-handle is called active if δ is true.
– *AND-handle*: If a rule r is part of a cycle and has a condition γ which is not part of the cycle, that means the rule is of the form $r :\ a \leftarrow \beta, \gamma$ where γ may be of the form c or *not* c, then γ is an AND-handle. The AND-handle is called active if γ is false.

An extended dependency graph extends a normal dependency graph in so far, that it distinguishes between AND- and OR-handles.

Definition 1 (Extended Dependency Graph). *The* extended dependency graph *for a logic program P with its Herbrand base $\mathcal{H}(P)$ is a directed graph $EDG(P) = (V, E)$ with nodes $V \subseteq \mathcal{H}(P)$ and edges $E \subseteq V \times V \times \{+, -\}$.*

The set of nodes and edges can be obtained by using the following rules:
V1 There exists a node a_i^k for every rule $r_k \in P$ where $head(r_k) = a_i$.
V2 There exists a node a_i^0 for every atom $a_i \in \mathcal{H}(P)$ which does not appear as the head of a rule.
E1 There exists an edge $(c_j^l, a_i^k, +)$ for every node $c_j^l \in V$ iff there is a rule $r_k \in P$ where $c_j \in pos(r_k)$ and $head(r_k) = a_i$.
E2 There exists an edge $(c_j^l, a_i^k, -)$ for every node $c_j^l \in V$ iff there is a rule $r_k \in P$ where $c_j \in neg(r_k)$ and $head(r_k) = a_i$

For every logic program a unique EDG can be constructed, this means a logic program is isomorphic to its representation as an EDG in the sense that the structure of the program is reflected one-to-one by the structure of the EDG. Properties of a logic program can be obtained from properties of the corresponding EDG and vice versa. One of these properties is the coloring. A valid coloring corresponds to an answer set of a logic program. A green colored node represents a rule where the head can be deduced and a red colored node represents a rule where the head cannot be deduced. A literal is contained in the answer set if there exists a node that represents the literal which is colored green. A literal is not contained in the answer set if all to nodes that correspond to the literal are colored red.

Definition 2 (Valid coloring of an EDG). *Let a program P be given. The coloring $\nu : V \rightarrow \{green, red\}$ of a graph $EDG(P) = (V, E)$ is* valid*, if the following conditions are met:*

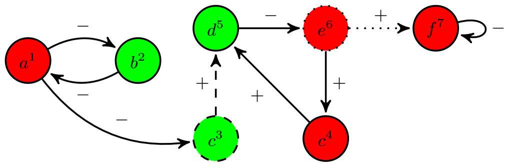
1. *$\forall i, k$ where $k \geq 1$ is $\nu(a_i^k) = green$, if a_i^k has no incoming edge.*
2. *$\forall i, k$ is $\nu(a_i^k) = green$, if the following two conditions are met*
   (a) $\forall j, m$ where $(a_j^m, a_i^k, +) \in E$ is: $\exists h$, such that $\nu(a_j^h) = green$
   (b) $\forall j, m$ where $(a_j^m, a_i^k, -) \in E$ is: $\nu(a_j^m) = red$
3. *$\forall i, k$ is $\nu(a_i^k) = red$, if at least one of the two following conditions is met*

(a) $\exists j, m$ *where* $(a_j^m, a_i^k, +) \in E$ *and* $\forall h$ *is* $\nu(a_j^h) = red$
(b) $\exists j, m$ *where* $(a_j^m, a_i^k, -) \in E$ *and* $\nu(a_j^m) = green$

4. *For every positive cycle* C *the following condition is met: If* $\forall i, k$ *where* $a_i^k \in C$ *is* $\nu(a_i^k) = green$*, then* $\exists i$ *and* $r \neq k$ *where* $\nu(a_i^r) = green$*.*

$$
\begin{aligned}
P_1 := \{ & a \leftarrow not\ b.\\
& b \leftarrow not\ a.\\
& c \leftarrow \{not\ a\}.\\
& c \leftarrow e.\\
& d \leftarrow c.\\
& e \leftarrow not\ d.\\
& f \leftarrow [e],\ \ not\ f.\}
\end{aligned}
$$

(a) Logic program P_1



(b) Successfully colored EDG for program P_1

Fig. 1. A logic program and the corresponding EDG

Example 1 *Figure 1 shows a logic program (a) and the corresponding EDG (b). The EDG has a valid coloring which represents the answer set* $\{b, c, d\}$*. The AND-handle is marked in the program with* [ ] *and is dotted in the EDG. The OR-handle is marked in the program with* { } *and is dashed in the EDG.*

The second graph type is the explanation graph (EG). In contrast to the EDGs, which visualize the structure of a whole logic program, EGs provide an explanation why a single literal appears or does not appear in an answer set and is always constructed with regard to an answer set and a set of assumptions. Assumptions are literals for which no explanation is needed since their value is assumed. All literals that are qualified for being used as assumptions are called *tentative assumptions* and are formally defined as follows:

Definition 3 (Tentative Assumptions). *Let* P *be a logic program,* M *an answer set of* P *and* $WF = \langle WF^+, WF^- \rangle$ *the well-founded model of* P*. Then the set of* tentative assumptions *of* P *w.r.t.* M *is*

$$\mathcal{TA}_P(M) = \{a \mid a \in NANT(P) \text{ and } a \notin M \text{ and } a \notin WF^+ \cup WF^-\}$$

where $NANT(P)$ *is the set of all literals appearing in* P *as a negative body literal:* $NANT(P) = \{a \mid \exists r \in P: \ a \in neg(r)\}$

Given a logic program P and a subset $U \subseteq \mathcal{TA}_P(M)$ of tentative assumptions, one can obtain the negative reduct $NR(P, U)$ of a program P w.r.t. U by removing all rules where $head(r) \in U$.

Definition 4 (Assumption). *An* assumption *of a program P regarding an answer set M is a set $U \subseteq \mathcal{TA}_p(M)$ where the well-founded model of the negative reduct corresponds to the answer set M, i. e. $WF_{NR(P,U)} = \langle M, \mathcal{H}(P) \setminus M \rangle$*

This means that setting all literals of the assumption U to *false* leads to all literals being defined in the well-founded model.
Explanation graphs base on local consistent explanations (LCE). These are sets of literals which directly influence the truth value of a literal a. For a literal a that is contained in the answer set and a rule where a is the head and all conditions of the rule are fulfilled, i. e. all body literals are true, the LCE consists of all body literals of the rule. Since there may exist several fulfilled rules with a as head, a can also have various LCEs. For a literal a that is not contained in the answer set, an LCE is a minimal set of literals that together falsify all rules that define a. For this purpose the LCE has to contain one falsified condition from each rule.

Definition 5 (Local Consistent Explanation). *Let a program P be given, let a be a literal, let M be an answer set of P, let U be an assumption and let $S \subseteq \mathcal{H}(P) \cup \{not\ a \mid a \in \mathcal{H}(P)\} \cup \{assume, \top, \bot\}$ be a set of literals.*

1. *S is an LCE for a^+ w.r.t. (M, U), if $a \in M$ and*
   - *$S = \{assume\}$ or*
   - *$S \cap \mathcal{H}(P) \subseteq M$, $\{c \mid not\ c \in S\} \subseteq (\mathcal{H}(P) \setminus M) \cup U$ and there exists a rule $r \in P$ where $head(r) = a$ and $S = body(r)$. For the case that $body(r) = \emptyset$ one writes $S = \{\top\}$ instead of $S = \emptyset$.*
2. *S is an LCE for a^- w.r.t. (M, U), if $a \in (\mathcal{H}(P) \setminus M) \cup U$ and*
   - *$S = \{assume\}$ or*
   - *$S \cap \mathcal{H}(P) \subseteq (\mathcal{H}(P) \setminus M) \cup U$, $\{c \mid not\ c \in S\} \subseteq M$ and S is a minimal set of literals, such that for every $r \in P$: if $head(r) = a$ then $pos(r) \cap S \neq \emptyset$ or $neg(r) \cap \{c \mid not\ c \in S\} \neq \emptyset$. For the case S being the empty set one writes $S = \{\bot\}$.*

In an EDG an edge (a_i^k, a_j^l, s) with $s \in \{+, -\}$ means that the truth value of literal a_j depends on the truth value of literal a_i. In an explanation graph the edges are defined the other way round, so that an edge (a_i, a_j, s) with $s \in \{+, -\}$ means that a_j explains or supports the truth value of a_i. The *support* of a node a_i in an EG is the set of all direct successors of a_i in the EG and is formally defined as follows:

Definition 6 (Support). *Let $G = (V, E)$ be a graph with nodes $V \subseteq \mathcal{H}^p \cup \mathcal{H}^n \cup \{assume, \top, \bot\}$ and edges $E \subseteq V \times V \times \{+, -\}$. Then the* support *of a node $a \in V$ is defined as:*
- *$support(a, G) = \{atom(c) \mid (a, c, +) \in E\} \cup \{not\ atom(c) \mid (a, c, -) \in E\}$,*
- *$support(a, G) = \{\top\}$ if $(a, \top, +) \in E$,*
- *$support(a, G) = \{\bot\}$ if $(a, \bot, -) \in E$ or*
- *$support(a, G) = \{assume\}$ if $(a, assume, s) \in E$ where $s \in \{+, -\}$.*

Definition 7 (Explanation Graph). *An* explanation graph *for a literal $a \in \mathcal{H}^p \cup \mathcal{H}^n$ in a program P w.r.t. an answer set M and an assumption $U \in Assumptions(P, M)$ is a directed graph $G = (V, E)$ with nodes $V \subseteq \mathcal{H}^p \cup \mathcal{H}^n \cup \{assume, \top, \bot\}$ and edges $E \subseteq V \times V \times \{+, -\}$ where $\mathcal{H}^p = \{a^+ \mid a \in \mathcal{H}(P)\}$ and $\mathcal{H}^n = \{a^- \mid a \in \mathcal{H}(P)\}$.*

The graph for a literal a has to meet the following conditions:

1. The only sinks in the graph are *assume*, $\top$ and $\bot$. $\top$ is used to explain facts of the program P, $\bot$ is used to explain literals which do not appear as a head of any rule and *assume* is used to explain literals for which no explanations are needed since their value is assumed to be false.
2. If $(c, l, s) \in E$ where $l \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$, then (c, l, s) is the only outgoing edge for every $c \in V$.
3. Every node $c \in V$ is reachable from a.
4. For every node $c \in V \setminus \{assume, \top, \bot\}$ the support $support(c, G)$ is an LCE for c regarding M and U.
5. There exists no $c^+ \in V$, such that $(c^+, assume, s) \in E$ where $s \in \{+, -\}$.
6. There exists no $c^- \in V$, such that $(c^-, assume, +) \in E$.
7. $(c^-, assume, -) \in E$ iff $c \in U$.

$$\begin{tikzcd} b^+ \arrow[r, "-"] & a^- \arrow[r, "-"] & \boxed{assume} \end{tikzcd}$$

(a) EG for b

$$\begin{tikzcd} f^- \arrow[r, "+"] & e^- \arrow[r, "-"] & d^+ \arrow[r, "+"] & c^+ \arrow[r, "-"] & a^- \arrow[r, "-"] & \boxed{assume} \end{tikzcd}$$

(b) EG for f

Fig. 2. EG b and f in P_1 w.r.t. $M = \{b, d, c\}$ and $U = \{a\}$

4 Construction of Explanation Graphs

In this section we introduce an approach for the construction of explanation graphs by extracting the required information from a validly colored extended dependency graph. Suppose we are given an extended dependency graph $G = (V, E)$ with a valid coloring $\nu : V \to \{green, red\}$. In the first step, we clean up the EDG by removing irrelevant edges and nodes. Irrelevant edges and nodes are those edges and nodes that do not have influence on the appearance or non-appearance of a literal in the answer set. This means they do not provide an explanation for a literal and hence are not needed for any explanation graph.

Definition 8 (Irrelevant edge, irrelevant node). *An edge (a_i^k, a_j^l, s) is* irrelevant *if*

- *$\nu(a_i^k) = green$, $\nu(a_j^l) = green$ and $s = -$,*
- *$\nu(a_i^k) = green$, $\nu(a_j^l) = red$ and $s = +$,*
- *$\nu(a_i^k) = red$, $\nu(a_j^l) = green$ and $s = +$ or*
- *$\nu(a_i^k) = red$, $\nu(a_j^l) = red$ and $s = -$.*

A node a_i^k is irrelevant *if $\nu(a_i^k) = red$ and there exists $l > 0$ where $\nu(a_i^l) = green$.*

If an irrelevant node is removed all its incoming and outgoing edges are also removed. After removing irrelevant edges and nodes we get an EDG $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. In the second step, nodes are gradually marked in the EDG. The marking process starts at nodes which have no incoming edges, because the explanation graphs for these nodes do not depend on other nodes. Every time a node is marked, the explanation graphs for the marked node are built. For this purpose five types of transformations are defined. The both first transformations describe the construction of explanation graphs for simple nodes, i. e. nodes which have no incoming edges in the EDG. The third and fourth transformation describe the construction of nodes which are dependent on other nodes, i. e. have incoming edges, distinguished by the color of the nodes. The last transformation is used for the construction of EGs for literals that are used as assumptions.

Transformation 1 (Transformation of fact nodes) *The EG for a node a_i^k which has no incoming edges and satifies $\nu(a_i^k) = green$ consists of a node a_i^+, a node $\top$ and an edge $(a_i^+, \top, +)$ (Fig. 3a), because such a node corresponds to a fact of the logic program.*

Transformation 2 (Transformation of unfounded nodes) *The EG to a node a_i^k which has no incoming edges and satisfies $\nu(a_i^k) = red$ consists of a node a_i^-, a node $\bot$ and an edge $(a_i^-, \bot, -)$(Fig. 3b).*

After marking nodes without incoming edges, we can mark nodes in positive cycles (cycles that contain only positive edges), that do not have an active handle since the corresponding literals do not have a supportive justification and are unfounded in the well-founded model. Since there exists no active handle for the cycle there is no other explanation for the nodes of the cycle than the one consisting of the cycle itself (with reversed edges). Now we continue marking nodes using the following rules until no more nodes can be marked:
A green node a_i^k can be marked if

- for all a_j^l where $(a_j^l, a_i^k, +) \in E'$, $\nu(a_j^l) = green$ and there exists $n \neq l$ with $a_j^n \in V'$, such that a_j^n is marked, and
- for all other nodes a_j^l where $(a_j^l, a_i^k, s) \in E'$ with $s \in \{+, -\}$, a_j^l is marked.

That means that a green node can be marked if all its predecessor nodes are marked. For literals which are represented by multiple green nodes, it is sufficient if one of these nodes is marked.
A red node a_i^k can be marked if

- $\exists (a_j^l, a_i^k, -) \in E'$ where $\nu(a_j^l) = green$ and a_j^l is marked, or
- $\exists (a_j^l, a_i^k, +) \in E'$ where $\nu(a_j^l) = red$ and all $a_j^n \in V'$ are marked.

That means that a red node can be marked, if at least one of its predecessor nodes is marked. In case that a predecessor literal is represented by multiple red nodes, all these nodes have to be marked.

Lemma 1. *The well-founded set W^+ corresponds to the set of all marked green nodes.*

Lemma 2. *The unfounded set W^- corresponds to the set of all marked red nodes.*

Proof sketch. It has to be shown that K_i always contains marked green nodes and $X_i = \mathcal{H}(P) \setminus U_i$ always contains marked red nodes. For this purpose the fixpoint operator $T_{P,V}$ for the generation of K_i and X_i has to be adjusted to X_i instead of U_i, especially in the adjustment of the operator for X_i positive cycles have to be considered. Then it can be seen, that the resulting operators exactly describe the process of marking green resp. red nodes.

From Lemma 1 and Lemma 2 we directly get the following proposition:

Proposition 1. *All unmarked nodes are undefined in the well-founded model.*

Green nodes represent literals that are contained in the answer set. So the local consistent explanation for such a node consists of all direct predecessor nodes (resp. the literals they represent).

Transformation 3 (Transformation of dependent green nodes) *Let $ie(a_i^k)$ be the set of incoming edges of node a_i^k. An EG for a node a_i^k where $\nu(a_i^k) = green$ and $ie(a_i^k) \neq \emptyset$ consists of a node a_i^+ and edges $E_{EG} = \{(a_i^+, EG(a_j), s) \mid (a_j^l, a_i^k, s) \in E'\}$, where $EG(a_j)$ is an explanation graph for a_j (Fig. 3d).*

Red nodes represent literals that are not contained in the answer set. In most cases red nodes have only active edges. The only exception is a predecessor literal a_j of a red node a_i^k is represented by multiple red nodes, formally $|\{a_j^l \mid (a_j^l, a_i^k, -) \in E'\}| \geq 2$. To get the LCEs for literals represented by a red node, all nodes representing this literal have to be considered. Each node represents a rule where the incoming edges represent the conditions of the rule. An LCE has to contain exactly one violated condition from each rule. Since we have removed all irrelevant edges, every edge represents a violated condition. That means that an LCE contains exactly one incoming edge for every node representing the literal.

Definition 9 (Local consistent explanation in an EDG). *Let $ie(a_i^k)$ be the set of incoming edges of node a_i^k and $\{a_i^1 \dots a_i^n\}$ the nodes representing a literal a_i. We set $L(a_i) = \{\{b_1, ..., b_n\} \mid b_1 \in ie(a_i^1), ..., b_n \in (a_i^n)\}$. $L(a_i)$ is an LCE for a_i if $L(a_i)$ is minimal w.r.t. set inclusion.*

Transformation 4 (Transformation of dependent red nodes) *The explanation graph for a node a_i^k where $\nu(a_i^k) = red$ w.r.t. an LCE $L(a_i)$ consists of a node a_i^- and edges*
$E_{EG} = \{(a_i^-, EG(a_j), s \mid a_j \in L, (a_j^l, a_i^k, s) \in E' \ for\ any\ l, k\}$ *(Fig. 3e).*

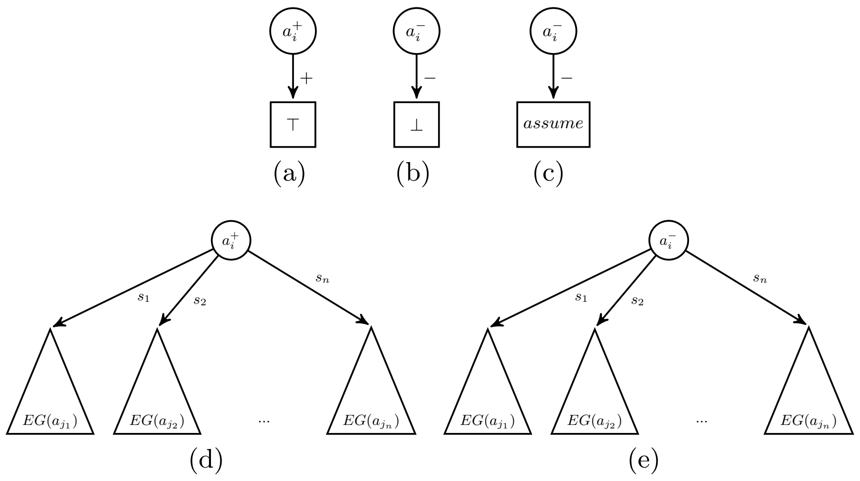
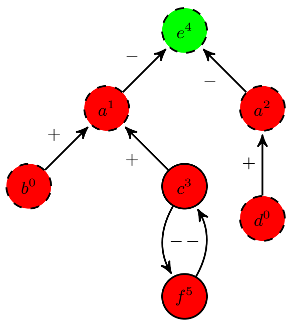


Fig. 3. Templates for constructing an explanation graph for a literal a_i

As mentioned before, every time a node is marked, the corresponding explanation graphs for this node are constructed. It should be remarked, that not all explanation graphs for a literal can be created when a node is reached the first time. This follows from the fact that there might exist multiple nodes for one literal and that a red node can be already marked if one of its predecessors is marked.

$$
\begin{aligned}
P_3 := \{ & a \leftarrow b, c. \\
& a \leftarrow d. \\
& c \leftarrow not\ f. \\
& e \leftarrow not\ a. \\
& f \leftarrow not\ c. \}
\end{aligned}
$$

(a) Logic program P_3



(b) Successfully colored EDG for program P_3

Fig. 4. A logic program and the corresponding EDG

Example 2 *In the graph from Figure 4b for the logic program P_3 (Fig. 4a) we can see that although both nodes are marked, we cannot construct all explanation graphs for the literal a and also e, since e depends on a. The explanation graph for the LCE $\{c, d\}$ of a is missing, because c depends on a cycle where an assumption has to be determined. So, if* *the node a^1 is reached again by the other edge $(c^3, a^1, +)$ during marking process, the set of its explanation graphs has to be updated and the information propagated to all successor nodes.*

The next step is to determine assumptions. Different approaches for choosing assumptions are proposed in Section 4. After choosing the assumption U, the incoming edges of all nodes representing a literal from U are removed, because no other explanations for these literals are allowed. So we get an extended dependency graph $G'' = (V', E'')$ with $E'' \subseteq E'$. The nodes of these literals are marked and the explanation graph can be constructed.

Transformation 5 (Transformation of assumption nodes)
The explanation graph for an assumption node a_i^k consists of a node a_i^-, a node assume and an edge $(a_i^-, assume, -)$ (Fig. 3c).

Then we proceed as before, marking nodes in G'' and simultaneously constructing the explanation graphs.

5 Choosing Assumptions

In most cases it is desirable to choose as few literals as possible as assumption. Assumptions where no literal can be removed without the set being no assumption anymore are called minimal assumptions. Finding them in an EDG can be very complex since all dependencies between the unmarked cycles have to be considered. For this purpose two different approaches are presented in this section. The first approach does not consider dependencies between cycles so that assumptions can be computed in $O(|V| + |E|)$. The disadvantage of this approach is that the determined assumptions are not minimal in most cases. The second approach determines all minimal assumptions of an EDG at the price of exponential complexity.
For the determination of assumptions we first have to determine all tentative assumptions. From the definition of tentative assumptions we know that a tentative assumption has to meet three conditions: a) it must not be contained in the answer set, b) it has to appear as negative body literal in a rule and c) it has to be undefined by the well-founded model. Transferred to a node in G', the first condition is fulfilled exactly by red nodes, the second condition is fulfilled exactly by nodes which have outgoing negative edges in G and the third condition is exactly fulfilled by unmarked nodes:

$$\mathcal{TA}(G') = \{a_i \mid a_i^k \in V', \nu(a_i^k) = red, a_i^k \text{ } not\ marked, (a_i^k, a_j^l, -) \in E\}$$

The set of tentative assumptions is always an assumption as shown in [1]. Since the set of tentative assumptions is often large we are looking for an approach to reduce the set. The aim is that all other nodes can be marked after defining the assumption.

Lemma 3. *In a graph without any cycles all nodes can be marked without making assumptions.*

Proof sketch. Since there exist no cycles the nodes can be ordered into different levels on the graph. Every level contains all nodes from the lower level and nodes whose predecessors are contained in a lower level. Then it can be shown via induction that all nodes can be marked.

When we choose an assumption, the truth value of the literal is set to $false$. Since no further explanations than the one that the literal is an assumption are allowed, all incoming edges of the assumption nodes are removed. We treat cycles here as sets of nodes. A cycle is then minimal w.r.t. set inclusion. Choosing one assumption node in each minimal cycle breaks up the minimal cycles. Since all bigger cycles contain a minimal cycle they are also broken up; hence there exist no more unmarked cycles. Since it is better to choose as few literals as possible as assumption we only choose those possible assumptions that are minimal with regard to set inclusion. Then a possible assumption consists of one tentative assumption from each minimal cycle.

Approach 1 *Let $C_1, C_2, ..., C_n$ be all minimal unmarked cycles in G'.*

$$\begin{aligned} Assumptions(G') = &\{\{a_1, a_2, ..., a_n\} \mid a_1 \in C_1, a_2 \in C_2, ..., a_n \in C_n, \\ &\{a_1, a_2, ...a_n\} \textit{ is minimal w.r.t. set inclusion}\} \end{aligned}$$

Of course there still may exist cycles which are marked. But since we know that they can be marked we can replace a marked cycle $C_m = (V_m, E_m)$ by a dummy node c_m with incoming edges $ie(c_m) = \bigcup_{v \in V_m} ie(v)$ and outgoing edges $oe(c_m) = \bigcup_{v \in V_m} oe(v)$. C
One very simple possibility to determine minimal assumptions is to try all combinations of tentative assumptions. Such a combination is an assumption, if the whole graph can be marked after choosing the assumption. A minimal assumption is then the combination that is successful and minimal with regard to set inclusion. But with an increasing number of tentative assumptions this approach will not be very efficient. For this reason, we introduce an approach that tries to reduce the number of combinations that have to be checked. Its basis is not to check all tentative assumptions and combinations, but only those literals that are important to determine the value of a so called *critical node*. It is obvious that this approach is only more efficient if the number of such literals is smaller than the number of tentative assumptions. For the sake of simplicity the approach is limited to graphs where each literal in a cycle is represented only by one node, i. e. there are no OR-handles. When OR-handles have to be considered a similar approach can be used.
Since marked nodes are irrelevant for the determination of assumptions we remove all marked nodes and their outgoing and incoming edges from the graph and obtain a sub-graph $G_{unmarked}$. In the next step, we are looking for strongly connected components of $G_{unmarked}$. A strongly connected component is a maximal sub-graph where each node is reachable

from each other node. This means that every node has an influence on every other node in the same strongly connected component, so that a strongly connected component behaves like one big cycle. For this reason we call the strongly connected components *linked cycles*.
If a linked cycle consists of several smaller cycles, there exist nodes belonging to multiple cycles. Such a node is *critical* if its value depends on more than one cycle. Since we have removed all irrelevant edges and have no OR-handles a red node has only active AND-handles. This means that the truth value of just one predecessor node is sufficient to determine the truth value of the red node. So a red node does not depend on more than one cycle, which means that only green nodes can be critical.

Definition 10 (Critical Node). *Let $LC = (V_{LC}, E_{LC})$ be a linked cycle. A node a_i^k is* critical, *if $\nu(a_i^k) = green$ and it has at least two incoming edges $(a_j^l, a_i^k, +) \in E_{LC}$ where $s \in \{+, -\}$. The set of all critical nodes of a linked cycle LC will be denoted as $\mathcal{CN}(LC)$.*

If a linked cycle has no critical nodes a node can be deduced from any other node. Then a minimal assumption consists of a single literal which is a tentative assumption and is represented by a node of the linked cycle. The set of all minimal assumptions of the linked cycle $LC(V_{LC}, E_{LC})$ in G' is: $\mu Assumptions(LC) = \{\{a_i\} \mid a_i^k \in V_{LC}, a_i \in \mathcal{TA}(G')\}$.
The value of every critical node depends on the value of its predecessor nodes. We call these nodes pre-conditions.

Definition 11 (Pre-condition). *Let $LC = (V_{LC}, E_{LC})$ be a linked cycle. The* pre-conditions *for a green critical node a_i^k are:*

$$pre(a_i^k) = \{a_j^l \mid (a_j^l, a_i^k, s) \in E_{LC}, s \in \{+, -\}\}$$

$Preconditions(LC) = \bigcup_{a_i^k \in \mathcal{CN}(LC)} pre(a_i^k)$ is the set of all pre-conditions in the linked cycle.

Lemma 4. *Nodes that are not critical can be deduced from at least one critical one node.*

Proof sketch. It can be shown by induction that the truth value of a node a_n on a path $c, a_1, a_2, ..., a_{n-1}, a_n$ from a critical node c can be deduced, if $a_1, ...a_n$ are not critical.

If we can deduce all critical nodes with an assumption, we also can deduce all other literals of the linked cycle with the assumption. $lfp(Succ(\{cn\}))$ calculates the nodes that can be deduced from a critical node $cn \in \mathcal{CN}(LC)$ with $Succ(S) = \{a_j^l \mid (a_i^k, a_j^l, s) \in E_{LC}, s \in \{+, -\}, a_i^k \in S\}$. Then the set of all nodes that can be deduced from a set of nodes S can be calculated with $lfp(T(S))$ with

$$T(S) = \{a_i^k \mid pre(a_i^k) \subseteq S\} \cup \{a_i^k \mid a_i^k \in Succ(a_j^l), a_j^l \in S \cap \mathcal{CN}(LC)\}.$$

Now combinations c of pre-conditions have to be tested for success. A combination c is successful if all critical nodes can be deduced from them, i. e. $\mathcal{CN}(LC) \subseteq lfp(T(c))$. We know that:

1. each combination c has to contain at least one complete pre-condition set $pre(cn) \subset c,\ cn \in \mathcal{CN}(LC)$. Otherwise the fix-point operator could not deduce any critical node.
2. Since we are looking for minimal assumption sets we do not have to check combinations c_1 where we already have found a smaller successful combination c_2, i. e. $c_2 \subseteq c_1$ and $\mathcal{CN}(LC) \subseteq lfp(T(c_2))$.

The way of proceeding is to first test single pre-condition sets $pre(cn) \subset c,\ cn \in \mathcal{CN}(LC)$ for success (exploits Fact 1). If a set is successful we add it to the set of successful combinations C and otherwise put it to NC. Then we test sets $n \cup \{a_i^k\}$, where $n \in NC$ and $a_i^k \in Preconditions(LC)$. This means we test different combinations of adding one more pre-condition to all sets that have not been successful in the step before (exploits fact 2). Again we add successful combinations to C and set NC to the combination that were not successful. This is repeated till $NC = \emptyset$ or the set to be tested consists of all pre-conditions. Then C contains all combinations of pre-conditions that suffice to deduce all critical nodes and therefore to deduce also all other nodes in the linked cycle, since they are not critical. For the purpose of determining assumptions we determine all nodes from which a pre-condition p can be deduced. These nodes lie on paths from critical nodes to the pre-condition.

Definition 12 (Pre-condition paths). *The* pre-condition path *to a pre-condition p from a linked cycle $LC = (V_{LC}, E_{LC})$ can be obtained by $path(p) = lfp(T_{path}(\{p\}))$ where*

$$T_{path}(S) = \{a_i^k \mid (a_i^k, a_j^l, s) \in E_{LC}, s \in \{+, -\}, a_i^k \notin \mathcal{CN}(LC)\}$$

A pre-condition path contains the nodes from which a pre-condition can be deduced. For deducing all nodes of a linked cycle we have to deduce all pre-conditions of a successful combination $c = \{p_1, ..., p_n\}$. This means that we need exactly one node from the path of each pre-condition. Since we want to determine assumptions the nodes also have to fulfill the other conditions of an assumption.

Definition 13 (Path assumptions). *The set of path assumptions for a path p in a validly colored EDG $G = (V, E)$ is defined by*

$$\mathcal{PA}(p) = \{a_i \mid \exists k, a_i^k \in p, a_i^k \in \mathcal{TA}(G')\}$$

Let C be the set of all successful pre-condition combinations $c = \{p_1, ..., p_n\}$. Then the set

$$Assumptions(LC) = \bigcup_{c \in C} \{\{a_1, ..., a_n\} \mid a_1 \in \mathcal{PA}(p_1), ..., a_n \in \mathcal{PA}(p_n)\}$$

is the set of possible minimal assumptions.

Proposition 2. *Minimal assumptions $\mu Assumptions(LC)$ of a linked cycle LC are those sets of $Assumptions(LC)$ that are minimal with regard to set inclusion.*

Proof sketch. It has to be shown that each set $S \in \mu Assumptions(LC)$ is a minimal assumption for the linked cycle LC, i. e. S is an assumption and there exists no set $S' \subset S$ such that S' is an assumption for LC. To show that S is an assumption it has to be checked if S meets the conditions of an assumptions. To show that S is minimal one looks at the successful combinations from which S and S' are created and differentiate between all cases. For every case it can be shown by contradiction that neither S nor S' can be in $\mu Assumptions(LC)$.

Approach 2 *In the first step all marked nodes are removed so we get a graph G''. After that all linked cycles $LC_1, LC_2, ..., LC_m$ without incoming edges are determined since they are independent of other linked cycles and need to contain an assumption. A linked cycle $LC = (V_{LC}, E_{LC})$ has no incoming edges if for all $a_i^k \in V_{LC}$ there is no $a_j^l \in V_{unmarked} \setminus V_{LC}, s \in \{+, -\}$ such that $(a_j^l, a_i^k, s) \in E_{unmarked}$. In each linked cycle LC_i the minimal assumptions for the cycle have to be determined. For that purpose the critical nodes of the linked cycle and their pre-conditions have to be specified and the pre-condition paths have to be calculated. In the next step it is looked for successful combinations of pre-conditions. The path assumptions are determined and used to calculate minimal assumptions*

$$\mu Assumptions(LC_1), \mu Assumptions(LC_2), ..., \mu Assumptions(LC_n)$$

of the linked cycles. To determine dependencies between linked cycles, the independent linked cycles $LC_1, LC_2, ...LC_m$ are replaced by dummy nodes (see Page 5). As result we get a new graph G'''. We continue marking nodes in G''' using the marking rules from Section 4. With the marking process we can see which nodes or which other linked cycles depend on the independent linked cycles determined in the second step. Now we start again with step one and remove all marked nodes so that we can determine which linked cycles are still independent. The whole process is repeated untill there are no more unmarked nodes and we got a set of independent linked cycles $LC_1, LC_2, ..., LC_m, ..., LC_n$ and their minimal assumptions $\mu Assumptions(LC_1)$, $\mu Assumptions(LC_2)$, ..., $\mu Assumptions(LC_m)$, ..., $\mu Assumptions(LC_n)$.

Proposition 3. *We obtain the minimal assumptions of the EDG by taking one minimal assumption of each independent linked cycle:*

$$\mu Assumptions(G') = \{a_1 \cup \cdots \cup a_n \mid a_1 \in \mu Assumptions(LC_1), \ldots \\ a_n \in \mu Assumptions(LC_n)\}$$

Proof sketch. For every assumption $a \in \mu Assumptions(G')$ two things have to be shown: a) a is an assumption for the EDG and b) a is minimal. a) can be directly shown by the termination condition of the algorithm. For the proof of b) a literal is removed from a and it can be shown that not all nodes in the EDG can be marked, because of the definition of the minimal assumption in an EDG and the definition of independent linked cycles.

6 Conclusion

We presented an approach to construct explanation graphs from validly colored extended dependency graphs. We exploited that the logic program is already present in graph form. In EDGs the nodes may differ in their number of incoming edges and the coloring. We defined different types of transformations to build the EGs for a node from an EDG. For the determination of assumptions it was necessary to determine the well-founded model. A strong relationship between well-founded models and the marking process of an EDG was observed; an unmarked node represents a literal which is undefined in the well-founded model. We presented two different approaches for the determination of assumptions. While first approach determines non-minimal assumptions in $O(|V| + |E|)$, the determination of minimal assumptions has an exponential complexity.

Acknowledgement: This work has been supported by the German Research Foundation DFG, Collaborative Research Center SFB876, Project A5. (http://sfb876.tu-dortmund.de)

References

1. Enrico Pontelli, Tran Cao Son and Omar Elkhatib: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming. 9 (2009)
2. Stefania Constantini and Alessandro Provetti: Graph representations of consistency and truth-dependencies in logic programs with answer set semantics. The 2nd Int'l IJCAI Workshop on Graph Structures for Knowledge Representation and Reasoning (GKR 2011). 1 (2011)
3. Michael Gelfond and Vladimir Lifschitz: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing. 9 (1991)
4. John W. Lloyd: Foundations of Logic Programming, 2nd Edition. Springer. (1987)
5. G. Brignoli, S. Costantini and A. Provetti: Characterizing and computing stable models of logic programs: the non-stratified case. Proceedings of the Conference on Information Technology, Bhubaneswar, India. AAAI Press, (1999).
6. Allen Van Gelder, Kenneth A. Ross and John S. Schlipf: The Well-Founded Semantics for General Logic Programs. Journal of the ACM. 38 (1991)
7. Yannis Dimopoulos and Alberto Torres: Graph theoretical structures in logic programs and default theories. Theoretical Computer Science, 170, (1996)

Some Experiments on Light-Weight Object-Functional-Logic Programming in Java with Paisley

Baltasar Trancón y Widemann[1 2] and Markus Lepper[2]

[1] Programming Languages and Compilers, Technische Universität Ilmenau
[2] `<semantics/>` GmbH

Abstract. The Paisley library and embedded domain-specific language provides light-weight nondeterministic pattern matching on the Java platform. It fully respects the imperative semantics and data abstraction of the object-oriented paradigm, while leveraging the declarative styles of pattern-based querying and searching of complex object models. Previous papers on Paisley have focused on the functional paradigm and data flow issues. Here, we illustrate its use under the logic paradigm. We discuss the expressiveness and evaluate the performance of Paisley in terms of the well-known combinatorial search problem "send more money" and its generalizations.

1 Introduction

We describe one link in a chain of efforts to bring the object-oriented programming paradigm closer to the more declarative functional and logic paradigms. Historically, there have been many attempts to reconstruct or reinvent objects on top of a logic platform (for instance [4,7]); however, our basic approach is exactly opposite. Our starting point is a full commitment to mainstream object-orientation, undoubtedly the dominant paradigm of our times, with unparalleled tool and library support for real-world programming. We develop "prosthetic" tools and programming techniques that amend well-known weaknesses in the expressiveness of plain object-orientation, without sacrificing broadness of scope or forcing programmers to leave their comfort zone. See the homepage at [8].

The present paper presents first results on the use of our Paisley library and language, designed for object-oriented pattern matching, as a toolkit for logic programming on an object-oriented platform.

1.1 Outline

The remainder of this paper is structured as follows: Section 2 summarizes and discusses the design of Paisley and its practical consequences under the various paradigms, as far as needed for the understanding of the following case studies. Technical details, further usage examples and comparison to related work can be found in [12,13]. Section 3 demonstrates logic programming in the Paisley style by means of the problem domain of cryptarithmetic puzzles and their most famous instance, “send more money”. Section 4 presents comparative performance measurements. Section 5 summarizes the experiences gained so far, and gives some outlook into future work.

2 Paisley

2.1 Design Considerations

The Paisley library and programming style [12,13] provide sorely missed pattern matching capabilities to the Java platform. For both theoretical and practical reasons, it does so in the form of a light-weight embedded domain-specific language (DSL). The following paragraphs discuss the conceptual implications of the approach concerning style and software engineering. The philosophically unconcerned reader is welcome to skip ahead for more technical matters.

The qualifier “embedded” means that absolutely no extension of the language or associated tools such as compilers or virtual machines is required. Extending an evolving language such as Java, although academically attractive, is fraught with great practical problems, mostly of maintenance and support: History shows that language extensions either get adopted into the main branch of development quickly, or die as academic prototypes. Instead, an embedded language shares the syntax, type system and first-class citizens of its host language, that is in the Java case, objects. Ideally, it is also “reified”, meaning that elements expose their DSL-level properties at host-level expressive public interfaces, and can be constructed, queried and manipulated freely and compositionally by the user.

The qualifier “light-weight” means that there is no technical distinction between “source” and “executable” forms of the DSL. No global pre-processing or compilation procedure is required, and no central interpreter engine exists. The capabilities of the DSL are distributed modularly over the implementation of DSL elements in the host language.

The two qualifiers together have wide-ranging implications for programming: They ensure that the embedded language is open and can easily be extended and customized by the user. They also guarantee tight integration and “impedance match” of interfaces, with fast and precise transfer of control and without data marshaling, between domain-specific and host-level computations. The price for this freedom is that a compositional structure precludes some global optimizations and refactorings of the DSL implementation.

For illustration purposes, consider parser combinators in a functional language as a prime example of reified light-weight embedded DSLs. Subparsers are ordinary (monadic) functions an can be defined and used directly as such, including definitions of whole multi-level context-free grammars as recursive functional programs. On the downside, global syntax analysis such as performed routinely by monolithic parser generators is poorly supported in a combinatorial setting.

By contrast, consider regular expression notations for string matching as a prime example of DSLs that are neither reified, embedded nor light-weight.

Typical implementations involve compilation to nondeterministic finite automata (NFA), and hide their implementation behind terse global interfaces. User control over control features, most importantly nondeterminism, is indirect and awkward (in the form of a plethora of analogous combinators with subtly different amounts of greediness). On the upside, the NFA implementation is well-known to improve global performance greatly in appropriate cases. The XPath language for XML document navigation is another prominent example of the same kind.

Non-reified DSLs are typically used in a monolithic fashion: whole DSL programs are passed textually at the platform interface, and the computational means for *programming* the DSL are conceptually and technically separate from the means for *implementing* it. Conversely, reified DSLs lend themselves to compositional programming: In the simplest case a DSL program is a statically nested constructor expression, that is the abstract syntax analog of a literal non-reified DSL program. But the real power comes from more complex uses, where the structure of the program under construction is either *abstracted* into host-level functions, or *dynamized* by host-level control flow. (Contrast the construction code depicted in Figs. 1–6 with the resulting DSL programs depicted in Fig. 9.)

2.2 Patterns, Object-Orientedly

Several theoretically well-founded paradigms for pattern matching exist: regular expressions, inverse algebraic semantics (in functional programming), term unification (in logic programming). However, for a tool to be practically useful in an object-oriented environment, it is of crucial importance not to impose any of the axioms of such theories, since they are typically not warranted for realistic object data models, and pretending otherwise causes impedance mismatch and is a source of much trouble and subtle bugs.

What, then, is object-oriented pattern matching proper? Starting from the rough approximation that (mainstream) object-orientation is imperative programming with data abstraction (encapsulation), patterns are a declarative specialist notation for data *queries*: They are applicable to object data models solely in terms of their public interface, which may not safely be assumed to have sound mathematical properties, such as statelessness, invertibility, completeness or extensionality. Object-oriented pattern matching organizes actual (getter) method calls, not meta-level semantic case distinctions.

Four generic aspects of querying can be discerned: data are subjected to *tests* for acceptability; matching may proceed to other accessible data by *projections*; information may flow back to the user in the form of side-effect variable *bindings*; control flow of matching links different patterns according to the outcome of tests with logical *combinators*. The absence of compositional programming constructs for these aspects in Java leads to awkward idioms, discussed in detail in [13].

In Paisley, a pattern that can process data of some type A is an object of type Pattern⟨A⟩. A match is attempted by invoking method **boolean** match(A target), with the return value indicating success (determined by the test aspect of the pattern). Nondeterminism is generally allowed, in the sense that a pattern may match the same target data in more than one way. These multiple solutions can be explored by repeatedly invoking method **boolean** matchAgain() until it fails. See Fig. 8 for a typical loop-based usage example.

Extracted information (determined by the projection and binding aspects of the pattern) is not available from the pattern root object, nor reflected in its type. Instead, references to the variables occurring in the pattern must be retained by the user. Variables are objects of type Variable⟨A⟩ **extends** Pattern⟨A⟩. They match any target data deterministically and store a reference by side-effect to represent the binding, which can then be extracted with method A getValue(). Again, see Fig. 8 for the extraction of binding values in the loop body. Variables are imperative, in the sense that they have no discernible unbound state, and may be reused (sequentially) at no cost.

Elementary patterns performing particular test and projection duties are predefined in the Paisley library and may be extended freely by the user. They are combined by two universal control flow combinators for *conjunction* (all) and *disjunction* (some). These are fully aware of nondeterminism (analogous to the Prolog operators , and ;), and also guarantee strict sequentiality of side-effects and have very efficient implementations for deterministic operands (analogous to the C-family operators && and ||). As an immediate consequence, subpatterns may observe bindings of variables effected in earlier branches of a conjunction.

2.3 Patterns, Functionally

The core concept of functional pattern matching, namely that initial algebra semantics can be imposed on data and inverted for querying, is valid only for degenerate cases of object-oriented programming. Real-world interface contracts are more subtle; while effective query strategies can be devised for particular problems, automatic "optimizations" such as transparent pattern restructuring for compilation of pattern-based definitions [2,9] are generally out of the question. Nevertheless, several functional principles can be used to good effect in the design of a powerfully abstract pattern object library:

Pattern building blocks effecting projections often correspond directly to a getter method of the object data model. Getters of class C with result type D can be conceived as functions from C to D; patterns of type Pattern⟨A⟩ can be conceived as functions from A to some complex solution/effect type. Hence each getter induces a contravariant lifting from Pattern⟨D⟩ to Pattern⟨C⟩ by mere function composition (the Hom-functor for the categorically-minded).

Functions between pattern types, both patterns as ad-hoc functions of a distinguished variable, and encapsulated pattern factories, are a very powerful abstraction, and a prerequisite for higher-order pattern operations. In Paisley they are represented by the interface Motif⟨A, B⟩ with a method Pattern⟨B⟩ apply(Pattern⟨A⟩).

2.4 Patterns, Logically

The main contribution of the logic paradigm to the design of Paisley is ubiquitous and transparent nondeterminism. It integrates with the operational semantics of patterns by having a fixed and precise resolution strategy, namely backtracking with strictly ordered choices. The other key idea of patterns in logic programming, namely unification, does not carry over soundly to the object-oriented paradigm, because of the lack of a stable global notion of equality for objects.

Nondeterminism can be introduced ad-hoc using explicit disjunctive combinators. But a natural kind of more abstract and useful sources of nondeterminism is the *imprecise* lifting of parameterized getter functions, abstracting from their qualifying parameters. For instance, the method A get(**int** index) of the Java collection interface List⟨A⟩ gives rise not only to a deterministic lifting from (Pattern⟨A⟩, **int**) to Pattern⟨List⟨A⟩⟩, but also to a nondeterministic variant from Pattern⟨A⟩ to Pattern⟨List⟨A⟩⟩ that tries each element of the target list in order. It is implemented in Paisley as the factory method CollectionPatterns.anyElement.

Nondeterminism, once introduced, is operationalized by the logical pattern combinators. A highly portable backtracking implementation is realized by eliminating the choice stack from the call stack (to which the programmer has limited access on the Java platform), and storing choice points on the heap, directly in the objects that instantiate pattern conjunction and disjunction. This has the notable effect that dynamic backtracking state is reified alongside static pattern structure, and can be deferred indefinitely, canceled abruptly, cloned and reused, committed to persistent storage etc., without interfering with normal control flow. The price for this flexibility is that the call stack needs to be reconstructed for backtracking (by iterated recursive descent), and that some caveats regarding pattern sharing and reentrance apply.

In the functional-logic spirit, matches of a pattern p as nondeterministic function of a distinguished variable x applied to target data t can be exhaustively explored (encapsulated search) by writing x.eagerBindings(p, t) or x.lazyBindings(p, t), with immediate or on-demand backtracking, respectively.

Combinatorial search problems can be encoded in the Paisley style as follows: Nondeterministic *generator* patterns for the involved variables are combined conjunctively (spanning the Cartesian product of solution candidates) and combined with *constraints*. Constraints are represented as patterns that take no target data, may observe previously bound variables (by earlier branches of a conjunction) and succeed at most once. They are implemented in Paisley by the class Constraint **extends** Pattern⟨Object⟩ with the method **boolean** test().

There is a strong trade-off between the effort to determine that a constraint is safe to test because all concerned variables have been bound, and the associated gain due to early pruning of the search tree. The following case study discusses a prominent combinatorial search problem, its generic object-oriented implementation in the Paisley style, and various strategies spread along the axis of the trade-off, from brute force to complex scheduling.

3 Case Study

The arithmetical puzzle “send more money” [5] is a well-known combinatorial problem that has been used ubiquitously to exemplify notations and implementations of logic programming. It specifies an assignment of decimal digits to variables $\{D, E, M, N, O, R, S, Y\}$, such that $SEND + MORE = MONEY$ in usual decimal notation. This equation, together with the implicit assumptions that the assignment is *injective* (the variable values are all different) and the numbers are *normal* (leading digits are nonzero), has a unique solution, namely $\{D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2\}$.

This particular problem easily suggests a number of generalizations, and has indeed not been the first of its kind. As a fairly broad class of similar problems, we consider the *cryptarithmetic puzzles* with arbitrary number of digits in each term, arbitrary number of terms in the sum, and arbitrary choice of base.

The following sections present different solution strategies with increasing performance and implementation complexity. The meta-level discussion is complemented with corresponding fragments of the actual application code, written in Java 7 using the Paisley libraries and style. Only basic knowledge of the Java syntax and collection framework is necessary to fully appreciate the code fragments; no particularly advanced or obscure coding techniques are employed. As a truly embedded language, Paisley can be interspersed finely with host code, and consequently hard to spot. For the reader's convenience, all Paisley-specific types and operations are underlined. Note that the purpose of most of the Java code is to *construct* an embedded Paisley program for combinatorial search; for some possible results see Fig. 9 below.

3.1 Basic Model

The basic model of cryptarithmetic puzzles is depicted in Fig. 1. It is parameterized at construction time with the chosen base, and terms encoded as strings. For instance, the original puzzle can be specified concisely as:

```
new CryptArith(10, "SEND", "MORE", "MONEY")
```

Also at construction time, Paisley objects for computations independent of a particular solution strategy are allocated internally: A variable is assigned to each character occurring in the terms; computed by a method `vars(String...)` not shown. A local non-zero constraint is assigned to each character occurring in leading position; computed by method `noLeadingZeroes(String...)`. A global constraint expressing the sum equation in terms of the created variables is formed from the terms; computed by method `sum(String...)`.

The constraint `sum` created by the latter adds the values of all terms but the last, and compares the sum to the value of the last term. For term evaluation it resorts to the auxiliary method `number` that computes a number from its b-adic representation by the currently bound values of the sequence of digit variables. Hence the constraints `sum`, as well as the constraints `noLeadingZeroes` depend on all or one variable, respectively, and must be tested only after the concerned variables have been bound successfully.

Any solution strategy for the cryptarithmetic puzzles consists of nondeterministic matches of all variables against valid digit values (ranging from zero, inclusive, to the given base, exclusive), and constraints equivalent to injectiveness (pairwise difference of all variables), absence of leading zeroes and the sum

```
public class CryptArith {
  private final int base;
  private final Map⟨Character, Variable⟨Integer⟩⟩ vars;
  private final Map⟨Variable⟨Integer⟩, Constraint⟩ noLeadingZeroes;
  private final Constraint sum;

  public CryptArith(int base, String... args) {
    if (base < 2 || args.length < 1)
      throw new IllegalArgumentException();
    this.base = base;
    this.vars = vars(args);
    this.noLeadingZeroes = noLeadingZeroes(args);
    this.sum = sum(args);
  }

  private Constraint sum(String... args) {
    final int n = args.length;
    final List⟨List⟨Variable⟨Integer⟩⟩⟩ rows = new ArrayList⟨⟩(n);
    for (String s : args) {
      // add characterwise list of variables to rows
    }
    return new Constraint() {
      public boolean test() {
        int s = 0;
        for (List⟨Variable⟨Integer⟩⟩ r : rows.subList(0, n − 1))
          s += number(r);
        return s == number(rows.get(n − 1));
      }
    };
  }
  private Map⟨Variable⟨Integer⟩, Constraint⟩ noLeadingZeroes(String... args) {
    final Map⟨Variable⟨Integer⟩, Constraint⟩ result = new HashMap⟨⟩();
    for (String s : args) {
      final Variable⟨Integer⟩ v = vars.get(s.charAt(0));
      result.put(v, Constraints.neq(v, 0));
    }
    return result;
  }
  private int number(List⟨? extends Variable⟨Integer⟩⟩ vs) {
    int n = 0;
    for (Variable⟨Integer⟩ v : vs)
      n = n ∗ base + v.getValue();
    return n;
  }
}
```

Fig. 1. Basic model of cryptarithmetic puzzles

equation. Strategies differ in, and draw their varying efficiency from, the early use of constraints to prune the search tree.

3.2 Brute-Force Generate and Test

The programmatically simplest, least efficient strategy is to defer all constraints until after all variables have been bound. This is of course the infamous *generate and test* pattern for combinatorial search. The implementation is depicted in Fig. 2. It uses a generic auxiliary method generate to produce generator pattern my mapping a nondeterministic Motif pattern function over the collection of variables, and another generic auxiliary method allDifferent to constraint them. The latter traverses a triangle matrix of all variables in order to produce pairwise inequality constraints.

```
public Pattern⟨Iterable⟨? extends Integer⟩⟩ strategy1() {
  return Pattern.all(generate(domain(), vars.values()),       // generate
                     allDifferent(vars.values()),             // and test,
                     Pattern.all(noLeadingZeroes.values()),   // test,
                     sum);                                    // test.
}

private ⟨A, B⟩ Pattern⟨B⟩ generate(Motif⟨A, B⟩ m,
                                   Collection⟨Variable⟨A⟩⟩ vars) {
  final List⟨Pattern⟨B⟩⟩ ps = new ArrayList⟨⟩();
  for (Variable⟨A⟩ v : vars)
    ps.add(m.apply(v));
  return Pattern.all(ps);
}

private ⟨A⟩ Constraint allDifferent(Collection⟨Variable⟨A⟩⟩ vars) {
  final List⟨Variable⟨A⟩⟩ done = new ArrayList⟨⟩(vars.size());
  final List⟨Constraint⟩ neqs = new ArrayList⟨⟩();
  for (Variable⟨A⟩ v : vars) {
    for (Variable⟨A⟩ u : done)
      neqs.add(neq(u, v));
    done.add(v);
  }
  return Constraint.all(neqs);
}
```

Fig. 2. Strategy 1: naïve generate and test

This strategy refers to auxiliary methods, shared by the other strategies, depicted in Fig. 3. The method domain() produces the nondeterministic motif used to generate candidate values for variables, by simply instantiating a generic motif for nondeterministic element selection from the Paisley collection framework.

The method neq(Variable, Variable) produces a single inequality constraint between the current values of two variables.

```
private Motif<Integer, Iterable<? extends Integer>> domain() {
  return CollectionPatterns.anyElement();
}
private <A> Constraint neq(final Variable<A> v, final Variable<A> w) {
  return new Constraint() {
    public boolean test() {
      return !v.getValue().equals(w.getValue());
    }
  };
}
```

Fig. 3. All strategies: generic utilities

3.3 Early Checking of Simple Constraints

The preceding brute-force strategy 1 has the disadvantage of actually generating all possible variable assignments, that is b^n combinations for n variables over base b. But most of the constraints that prune the search tree (in fact all of them except the sum equation proper) concern at most two variables. Hence it is easy to predict the earliest point in the search plan where they can be checked.

This observation gives rise to an improved strategy 2 depicted in Fig. 4. It works by splicing together the first three phases of strategy 1, each of which has a loop over the variables, into a single loop. Only the global constraint of the sum equation, which concerns all variables and must necessarily come last, is left behind. This straightforward refactoring reduces the number of generated assignments greatly, to less than $n! \cdot \binom{b}{n}$.

3.4 Exploiting Partial Sums

It is known that early pruning of the search tree can be improved further by approximations to the sum using modular arithmetics. Each partial sum of the k least significant digits must be satisfied up to carry, which can be expressed as a congruence modulo b^k. While these partial sum relations are implied by the exact sum equation (a congruence modulo infinity), and hence logically redundant, they have the practically advantageous property of concerning fewer variables. Hence they can be checked earlier in the search plan. Fig. 5 depicts the partial sum congruences for "send more money" with $k = 1, \ldots, 5$, each together with the set of variables concerned for that k at the earliest.

The strategic information discussed above is reified in the model extension depicted in Fig. 7 below. The inner class PartialSum encapsulates both the set of

```
public Pattern⟨Iterable⟨? extends Integer⟩⟩ strategy2() {
  return Pattern.all(generateAndTestEarly(domain(), vars.values()), sum); // ...  test.
}

private ⟨A, B⟩ Pattern⟨B⟩ generateAndTestEarly(Motif⟨A, B⟩ m,
                                              Collection⟨Variable⟨A⟩⟩ vars) {
  final List⟨Variable⟨A⟩⟩ done = new ArrayList⟨⟩(vars.size());
  final List⟨Pattern⟨? super B⟩⟩ pats = new ArrayList⟨⟩();
  for (Variable⟨A⟩ v : vars) {
    pats.add(m.apply(v));                       // generate
    if (noLeadingZeroes.containsKey(v))
      pats.add(noLeadingZeroes.get(v));         // and test,
    for (Variable⟨A⟩ u : done)
      pats.add(neq(u, v));                      // test,  ...
    done.add(v);
  }
  return Pattern.all(pats);
}
```

Fig. 4. Strategy 2: generate with early checks

$D+$	$E\equiv$	Y	mod	10	$\{D,E,Y\}$
$ND+$	$RE\equiv$	EY	mod	100	$\{N,R\}$
$END+$	$ORE\equiv$	NEY	mod	1 000	$\{O\}$
$SEND+$	$MORE\equiv$	$ONEY$	mod	10 000	$\{M,S\}$
$SEND+$	$MORE\equiv$	$MONEY$	mod	∞	$\{\}$

Fig. 5. Partial sum modular congruences for "send more money"

concerned variables and the associated congruence as a constraint. It refers to an extended version of the auxiliary method sum (compare Fig. 1), parameterized with the number of digits under consideration. The sequence of partial sums is precomputed at model construction time. The full strategy 3 can then be generated by a loop over this sequence, issuing generators for newly introduced variables, inequality and nonzero checks, and partial sum congruences in turn, as depicted in Fig. 6.

3.5 Dynamic Constraint Scheduling

If precise constraint scheduling, as in the preceding strategy, is deemed unfeasible, one can still resort to dynamic scheduling techniques. Constraints can be implemented such that their evaluation is suspended if some concerned variable is not yet bound, and resumed when that condition changes. Then, a trivial strategy simply places the generators last. However, tracking the state of variables has a significant run-time overhead, especially in Paisley where all components are as

```
public Pattern⟨Iterable⟨? extends Integer⟩⟩ strategy3() {
  return generatePartialSums(domain());
}

⟨B⟩ Pattern⟨B⟩ generatePartialSums(Motif⟨Integer, B⟩ m) {
  final List⟨Variable⟨Integer⟩⟩ done = new ArrayList⟨⟩();
  final List⟨Pattern⟨? super B⟩⟩ pats = new ArrayList⟨⟩();
  for (PartialSum s : partialSums) {
    for (Variable⟨Integer⟩ v : s.getDependencies())
      if (!done.contains(v)) {
        pats.add(m.apply(v));                      // generate
        if (noLeadingZeroes.containsKey(v))
          pats.add(noLeadingZeroes.get(v));        // and test,
        for (Variable⟨Integer⟩ u : done)
          pats.add(neq(v, u));                     // test,
        done.add(v);
      }
    pats.add(s.getConstraint());                   // test.
  }
  return Pattern.all(pats);
}
```

Fig. 6. Strategy: generate with partial sums (operation)

light-weight as possible, and nondeterminism is expected to incur no appreciable cost if not actually used.

We have added a prototype implementation of suspendable constraints to our case study, implemented using the well-known *observer* pattern of object-oriented programming. Strategies 2’ and 3’ are the analogs of 2 and 3, respectively, but with suspendable constraints preceding the generators they depend on.

4 Evaluation

All experiments have been performed on a single MacBook Air containing a Intel Core i5-3317U CPU with 4 cores at 1.7 GHz and 8 GiB of RAM running the OpenJDK 7-21 Java environment on Ubuntu 12.10. Reported times are wall-clock time intervals, measured with System.nanoTime() to the highest available precision. All experiments have been repeated 10 times, without restarting the Java machine or interfering with automatic memory management.

Table 1 summarizes the findings for all strategies, obtained by exhaustive search in a loop, as depicted in Fig. 8. Measurements are fairly consistent, with little random variation. Memory management appears to have negligible impact, as expected for a combinatorial computation with tiny data footprint.

```
private final List⟨PartialSum⟩ partialSums = new ArrayList⟨⟩() ;

public CryptArith(int base, String... args) {
  // ...
  this.partialSums = partialSums(args);           // analogous to noLeadingZeroes
}

class PartialSum {

  private final int length;
  private final Set⟨Variable⟨Integer⟩⟩ dependencies ;         // cf. Fig. 5 right
  private final String[] args;

  PartialSum(int length,
             Set⟨? extends Variable⟨Integer⟩⟩ dependencies,
             String... args) {
    // initialize fields
  }

  public Set⟨Variable⟨Integer⟩⟩ getDependencies() {
    return dependencies ;
  }

  public Constraint getConstraint() {
    return sum(length, args);
  }
}

private Constraint sum(int length, String... args) {
  final int n = args.length;
  final List⟨List⟨Variable⟨Integer⟩⟩⟩ rows = new ArrayList⟨⟩(n);
  for (String s : args) {
    // add characterwise list of last length variables to rows
  }
  final int m = power(base, length);
  return new Constraint() {
    public boolean test() {
      int s = 0;
      for (List⟨Variable⟨Integer⟩⟩ r : rows.subList(0, n − 1))
        s += number(r);
      return s % m == number(rows.get(n − 1)) % m ;             // congruence
    }
  };
}
```

Fig. 7. Strategy 3: generate with partial sums (model extension)

```
public void run(final Pattern⟨Iterable⟨? extends Integer⟩⟩ p) {
  if (p.match(digits())) do {
    for (Map.Entry⟨Character, Variable⟨Integer⟩⟩ e : vars.entrySet())
      System.out.println(e.getKey() + "␣=␣" + e.getValue().getValue());
    System.out.println();
  } while (p.matchAgain());
}
private Collection⟨Integer⟩ digits() {
  final List⟨Integer⟩ result = new ArrayList⟨Integer⟩(base);
  for (int i = 0; i < base; i++)
    result.add(i);
  return result;
}
```

Fig. 8. Running a strategy

5 Conclusion

Fig. 9 gives a synopsis of the inner structure of patterns produced by the strategies 1–3 for the "send more money" example. In each case, elementary generator and constraint patterns are composed associatively into a global conjunction. For the more advanced strategies, more powerful constraints appear earlier in the sequence. The patterns can be used immediately as depicted in Fig. 8, or used in every other conceivable way as ordinary Java objects. As such, our implementation of the problem domain on top of Paisley acts technically as a domain-specific compiler to a threaded code back-end, given by the Paisley operations.

The Paisley approach leads to a style that has some of the best of both worlds: The object-oriented paradigm has excellent support for data abstraction and encapsulation. Object-oriented models of the problem domain have expressive interfaces close to the programmer's intentions and intuitions, and high documentation value.

On the other hand, the declarative style of the logic paradigm allows for abstraction from the complex control flow of searching by composition of simple nondeterministic fragments. Note that all explicit control flow in the given code

Table 1. Experimental evaluation: send more money, $N = 10$

Strategy	**Time (ms)**		
	min	**median**	**max**
1	5 396.93	5 470.24	5 775.04
2	737.56	770.25	809.48
3	2.34	2.37	3.60
2'	761.37	771.93	797.53
3'	850.30	863.21	881.72

$$D < b, E < b, S < b, R < b, N < b, O < b, M < b, Y < b,$$
$$D \neq E, D \neq S, E \neq S, D \neq R, E \neq R, S \neq R, D \neq N, E \neq N, S \neq N, R \neq N,$$
$$D \neq O, E \neq O, S \neq O, R \neq O, N \neq O, D \neq M, E \neq M, S \neq M, R \neq M, N \neq M,$$
$$O \neq M, D \neq Y, E \neq Y, S \neq Y, R \neq Y, N \neq Y, O \neq Y, M \neq Y,$$
$$M \neq 0, S \neq 0,$$
$$SEND + MORE \equiv MONEY \bmod \infty$$

$$D < b, E < b, D \neq E, S < b, S \neq 0, D \neq S, E \neq S, R < b, D \neq R, E \neq R, S \neq R,$$
$$N < b, D \neq N, E \neq N, S \neq N, R \neq N, O < b, D \neq O, E \neq O, S \neq O, R \neq O, N \neq O,$$
$$M < b, M \neq 0, D \neq M, E \neq M, S \neq M, R \neq M, N \neq M, O \neq M,$$
$$Y < b, D \neq Y, E \neq Y, S \neq Y, R \neq Y, N \neq Y, O \neq Y, M \neq Y,$$
$$SEND + MORE \equiv MONEY \bmod \infty$$

$$Y < b, D < b, D \neq Y, E < b, E \neq Y, E \neq D, D + E \equiv Y \bmod 10, R < b, R \neq Y,$$
$$R \neq D, R \neq E, N < b, N \neq Y, N \neq D, N \neq E, N \neq R, ND + RE \equiv EY \bmod 100,$$
$$O < b, O \neq Y, O \neq D, O \neq E, O \neq R, O \neq N, END + ORE \equiv NEY \bmod 1\,000,$$
$$S < b, S \neq 0, S \neq Y, S \neq D, S \neq E, S \neq R, S \neq N, S \neq O,$$
$$M < b, M \neq 0, M \neq Y, M \neq D, M \neq E, M \neq R, M \neq N, M \neq O, M \neq S,$$
$$SEND + MORE \equiv ONEY \bmod 10\,000, SEND + MORE \equiv MONEY \bmod \infty$$

Fig. 9. Unfolded search plans generated by strategies 1, 2, 3, respectively

samples is exclusively for the *construction* of a particular instance of the generic puzzle model. The actual control flow of searching is hidden entirely in the invocations of pattern combinators, most notably Pattern.all, consequently reified in a complex Pattern object that both represents and implements the search, and finally effected using Pattern.match and Pattern.matchAgain, as depicted in Fig. 8.

The influence of the functional paradigm is evidently the weakest in the examples discussed here: They make a single use of the Motif class. But there is considerable, unfulfilled potential: Virtually all of the loops in the example code express *comprehensions*, and could be rephrased in terms of the higher-order functions *map*, *reduce* and friends. These are conspicuously absent from the traditional Java collection framework; but there is hope that the rise of anonymous functions in Java 8 [6] will improve the situation. The Paisley approach is expected to profit greatly from equally high expressiveness in all three paradigms. Alternatively, Paisley could be ported to Scala, in order to reap the benefits of decent functional programming immediately.

The relative performance of more intelligent strategies within the Paisley framework is encouraging: A measured speedup of over three orders of magnitude by means of a moderately complex model extension that captures only well-understood heuristics about the problem domain is certainly worth the effort.

The absolute performance of Paisley implementations is of course no match for low-level optimized solver code. For instance, the C program obtainable from [11] takes approximately 0.17 ms to solve "send more money" on our test machine (compiled with `gcc -O`), about an order of magnitude less than our best effort with strategy 3. On the other hand, we have tested a simple, portable implementation [10] in the functional-logic programming language Curry, contributed by a developer of the KiCS2 compiler [3]. It follows a similar strategy as our strategy 2 and uses no dedicated solver modules, takes about 7.49 s on the same machine (compiled with KiCS2 0.2.4 `+optimize`), an order of magnitude more than its most direct Paisley competitor, and even more than our brute-force strategy 1. Note that this result is not representative of the language at large; more sophisticated Curry implementations of cryptarithmetic puzzles such as the one described in [1] can be fairly competitive.

Considering the costs of portable backtracking and object-oriented data abstraction, Paisley appears to be well on the way. The only disappointment so far is the performance of the dynamically scheduled constraints in strategies 2' and 3', although the current implementation is merely a proof-of-concept prototype. Here the scheduling overhead clearly dominates the actual computation. More research into efficient implementations is needed.

References

1. Antoy, S., Hanus, M.: Concurrent distinct choices. Journal of Functional Programming 14, 657–668 (2004)
2. Augustsson, L.: Compiling pattern-matching. In: Proc. Functional Programming Languages and Computer Architecture. pp. 368–381. No. 201 in LNCS, Springer-Verlag (1985)
3. Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A new compiler from curry to haskell. In: Proc. 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011). pp. 1–18. No. 6816 in LNCS, Springer-Verlag (2011)
4. Conery, J.S.: Logical objects. In: Kowalski, R., Bowen, K. (eds.) Proc. Logic Programming, pp. 420–434. MIT Press (1988)
5. Dudeney, H.E.: Strand Magazine 68, 97 (July 1924)
6. Goetz, B.: Lambda expressions for the java programming language (draft review 3). Java Specification Request 335, Oracle (2013)
7. Kahn, K., Tribble, E.D., Miller, M.S., Bobrow, D.G.: Vulcan: logical concurrent objects, chap. 30, pp. 274–303. MIT Press (1987)
8. Lepper, M., Trancón y Widemann, B.: metatools homepage (2013), http://bandm.eu/metatools/
9. Pettersson, M.: A term pattern-match compiler inspired by finite automata theory. In: Proc. Compiler Construction (CC 1992), pp. 258–270. No. 641 in LNCS, Springer-Verlag (1992)
10. Reck, F.: smm.curry (2013), personal communication
11. Tamura, N.: Cryptarithmetic puzzle solver (2004), http://bach.istc.kobe-u.ac.jp/llp/crypt.html, accessed 2013-06-01
12. Trancón y Widemann, B., Lepper, M.: Paisley: pattern matching à la carte. In: Proc. 5th International Conference on Model Transformation (ICMT 2012). pp. 240–247. No. 7307 in LNCS, Springer-Verlag (2012)
13. Trancón y Widemann, B., Lepper, M.: Paisley: A pattern matching library for arbitrary object models. In: Proc. 6. Arbeitstagung Programmiersprachen (ATPS 2013). pp. 171–186. No. 215 in LNI, Gesellschaft für Informatik (2013), http://www.se2013.rwth-aachen.de/downloads/proceedings/SE2013WS.pdf

On an Approach to Implementing Exact Real Arithmetic in Curry

Christoph Beierle, Udo Lelitko

Dept. of Computer Science
FernUniversität in Hagen
58084 Hagen
Germany

Abstract. While many functions on the real numbers are not exactly computable, the theory of exact real arithmetic investigates the computation of such functions with respect to any given precision. In this paper, we present an approach to implementing exact real arithmetic in the functional logic language Curry. It is demonstrated how the specific features of Curry can be used to obtain a high-level realization that is close to the underlying theoretical concepts. The new Curry data type Real and its corresponding functions can easily be used in other function definitions.

And... action! – Monoid Acts and (Pre)orders

Nikita Danilenko

Institut für Informatik, Christian-Albrechts-Universität Kiel
Olshausenstraße 40, D-24098 Kiel
nda@informatik.uni-kiel.de

Abstract Orders occur naturally in many areas of computer science and mathematics. In several cases it is very simple do describe an order mathematically, but may be cumbersome to implement in some programming language. On the other hand many order relations are defined in terms of an existential quantification. We provide a simple abstraction of these definitions using the well-known concept of monoid acts and furthermore show that in fact every order relation can be obtained from a specific monoid act.

1 Introduction

In beginners' courses on mathematics (for computer scientists) several ordering relations are usually given as examples. Consider for instance the natural order on the natural numbers $\leq_{\mathbb{N}}$ defined in terms of

$$x \leq_{\mathbb{N}} y :\Longleftrightarrow \exists\, z \in \mathbb{N} : x + z = y$$

for all $x, y \in \mathbb{N}$. It is a basic and straightforward task to verify the three laws of an order relation for $\leq_{\mathbb{N}}$. A short time after this definition we might find ourselves confronted with the concept of lists and prefix lists. Now suppose that "$+\!\!+$" denotes the list concatenation and consider the definition of a prefix list: for any set M and any two lists $x, y \in M^*$ we define

$$x \trianglelefteq y :\Longleftrightarrow \exists\, z \in M^* : x +\!\!+ z = y$$

and call $\trianglelefteq$ the "is-prefix-of"-relation. Observe how the definition itself is basically the one of $\leq_{\mathbb{N}}$ – we merely exchanged $\mathbb{N}$ and $+$ by M^* and $+\!\!+$ respectively. We know that $(\mathbb{N}, +, 0)$ and $(M^*, +\!\!+, \varepsilon)$ are monoids and this facts seems to be encoded in the definition of the orders somehow. Let us have a look at a final example, namely the order on $\mathbb{Z}$, which is defined for all $x, y \in \mathbb{Z}$ by:

$$x \leq_{\mathbb{Z}} y :\Longleftrightarrow \exists\, z \in \mathbb{N} : x + z = y.$$

Again we notice the resemblance to the previous definitions, but in this case the addition is a "skew" one, since its functionality is $+ : \mathbb{Z} \times \mathbb{N} \to \mathbb{Z}$.

We observe that all of these orders are structurally defined as

$$x \sqsubseteq y :\Longleftrightarrow \exists\, z \in M : x \otimes z = y,$$

for all $x, y \in A$, where A, M are some sets and $\otimes : A \times M \to A$.

From these observations we can derive a simple concept for defining order relations and study order properties in terms of this concept. While the basic "ingredients" (as monoid actions) are well-known and have been studied well, to the best of our knowledge our approach to orders has not.

In the following we will use "functions"[1] called "swap", "curry" and "fix". For sets A, B, C and a function $f : A \times B \to C$ we have that

$$\begin{aligned}
\mathrm{swap}(f) &: B \times A \to C,\ (b, a) \mapsto f(a, b)\,,\\
\mathrm{curry}(f) &: A \to C^B,\ a \mapsto \Big(b \mapsto f(a, b)\Big)\,,\\
\mathrm{fix} &: A \to A^A,\ g \mapsto \{a \in A \,|\, g(a) = a\}\,.
\end{aligned}$$

We use $f(a, -) = (x \mapsto f(a, x))$ and $f(-, b) = (x \mapsto f(x, b))$ to denote partial applications and f^{-1} for the preimage of f.

The paper is structured as follows.

- We present the abstraction behind the orders we have just discussed.
- A characterisation of orders in terms of monoid actions is given.
- It is shown how to obtain an action that creates a given order.
- We provide an implementation of functions that can be used to obtain orders in the functional logic language Curry.

2 Monoid acts

The concept of a structure (e.g. group, algebra) acting on some set (or other structure) is well-known in mathematics, particularly in algebraic contexts.

Let us begin with a simple abstraction of the observation concerning the function type of the addition in the last example of the introduction[2]. Recall that a monoid is an algebraic structure $(M, \cdot, e)$ where $\cdot$ is a binary, associative operation and e is a neutral element with respect to $\cdot$.

Definition 1 (Monoid action).
Let $(M, \cdot, e)$ be a monoid and A a non-empty set. A mapping $\varphi : M \times A \to A$ is called ***monoid action of*** *M* ***on*** *A, if and only if the following conditions hold:*

(1) $\varphi(e, -) = id_A$, *(preservation of unity)*
(2) $\forall\, x, y \in M : \forall\, a \in A : \varphi(x, \varphi(y, a)) = \varphi(x \cdot y, a)$ *(associativity).*

Thus a monoid act gives us an outer operation on A (cf. inner operations). The terms "action" and "act" are used synonymously. If $\varphi : M \times A \to A$ is a monoid action, we will abbreviate $\varphi(m, a) =: m \cdot_\varphi a$ for all $m \in M$ and $a \in A$.

[1] Both can be considered functional classes.

[2] Actually we should have written $+ : \mathbb{N} \times \mathbb{Z} \to \mathbb{Z}$ for congruence with the following definition. The reason we did not is that for the three examples the monoid is better placed in the second component, whereas from a mathematical point of view it is more convenient to place it in the first one.

Considering a monoid act in its curried version $\mathrm{curry}(\varphi) : M \to A^A$ gives us that $\mathrm{curry}(\varphi)$ is a monoid homomorphism. Conversely every monoid homomorphism from $f : M \to A^A$ can be converted into a monoid act[3]. In fact these two operations are inverse to each other. These properties are known so well that they constitute typical exercises for students.

We proceed to provide some examples of monoid acts.

Example (Monoid acts)

1. Let $(M, \cdot, e)$ be a monoid. Then $\cdot$ is a monoid act of M on M.
2. The mapping $+ : \mathbb{N} \times \mathbb{Z} \to \mathbb{Z}$ is a monoid act of $(\mathbb{N}, +, 0)$ on $\mathbb{Z}$.
3. Let (Q, Σ, δ) be a transition system. Then $\mathrm{swap}(\delta^*) : \Sigma^* \times Q \to Q$ is a monoid act of $(\Sigma^*, \mathrm{swap}(+\!\!+), \varepsilon)$ on Q.
4. Let A be a set and $\varphi : A^A \times A \to A, (f, x) \mapsto f(x)$. Then φ is a monoid act of $(A^A, \circ, \mathrm{id}_A)$ on A.

These properties are easily checked: the first one is trivially true, the second one can be shown in a large variety of simple ways, the third one relies on the fact that δ^* is the homomorphic continuation of δ on Σ^* and the fourth one merely rephrases elementary properties of function composition and application. $\square$

It is little surprising that monoid acts have certain *permanence properties* e.g. direct products of monoid acts form monoid acts. Categorically speaking these properties state that the category of monoid acts is closed under certain operations. We will not deal with these properties since they are mostly well known. Instead we use the concept of monoid acts to define the (ordering) relations we have seen in the introduction.

Definition 2 (Act preorder).
Let $(M, \cdot, e)$ be a monoid, A a set and $\varphi : M \times A \to A$ a monoid act. We then define for all $a, b \in A$:

$$a \sqsubseteq_\varphi b :\Longleftrightarrow \exists\, m \in M : m \cdot_\varphi a = b.$$

The relation $\sqsubseteq_\varphi$ is called ***act(ion) preorder****.*

Note that the definition captures the essence of all orders we have presented in the beginning of the paper. To justify the anticipatory name of the relation we need to show a simple lemma. The proof is very simple and we include it only for the purpose of demonstration.

Lemma 1 (Act preorder).
Let $(M, \cdot, e)$ be a monoid, A a set and $\varphi : M \times A \to A$ a monoid act. Then the following hold:

(1) The act preorder $\sqsubseteq_\varphi$ is in fact a preorder on A.
(2) For all $m \in M$ the mapping $\varphi(m, -)$ is expanding, i.e. $\forall\, a \in A : a \sqsubseteq_\varphi m \cdot_\varphi a$.

[3] Simply set $\varphi(m, a) := f(m)(a)$ for all $m \in M$ and $a \in A$.

Proof. (1) Let $x \in A$. By the preservation of units we get $x = e \cdot_\varphi x$, thus $x \sqsubseteq_\varphi x$. Now let $a, b, c \in A$ such that $a \sqsubseteq_\varphi b$ and $b \sqsubseteq_\varphi c$. Then there are $m, n \in M$ such that $m \cdot_\varphi a = b$ and $n \cdot_\varphi b = c$. The associativity of the action gives us

$$c = n \cdot_\varphi b = n \cdot_\varphi (m \cdot_\varphi a) = (n \cdot m) \cdot_\varphi a.$$

Since $n \cdot m \in M$ we get $a \sqsubseteq_\varphi c$.
(2) Left as an exercise to the reader. □

Let us make two observations concerning this lemma. First – showing the reflexivity and transitivity of actual relations (like $\leq_\mathbb{N}$ or $\trianglelefteq$) will always result in essentially the very proof of this lemma. Second – the two properties "preservation of units" and "associativity" of monoid acts supply the sufficient conditions for "reflexivity" and "transitivity" respectively.

So far we have seen some examples of act preorders (that incidentally were orders as well). In such a setting two questions suggest themselves:

1. When is an act preorder an order?
2. Is every order an act preorder?

Ideally the answer to the first question should be some kind of characterisation and the answer to the second should be a Boolean value followed by a construction in the positive case.

Before we turn to the use of these definitions and properties for implementation we would like to provide answers to both questions. The applications will follow in Section 4.

Finally let us note that the *relation* defined by the act preorder is very well known in group theory in the context of group actions. In this context the relation above is always an equivalence relation (again this is known well and often used as an exercise) and is commonly used to investigate act properties (cf. the orbit-stabiliser-theorem [1].). To the best of our knowledge little effort has been invested in the study of this relation in the presence of monoid acts.

3 Act Preorders

First of all let us deal with the question when an act preorder is an order. When we start to prove the antisymmetry of an ordering relation like $\leq_\mathbb{N}$ we take $a, b \in \mathbb{N}$ s.t. $a \leq_\mathbb{N} b$ and $b \leq_\mathbb{N} a$. Then we find that there are $c, d \in \mathbb{N}$ satisfying $c + a = b$ and $d + b = a$. Thus we get

$$a = d + b = d + (c + a) = (d + c) + a.$$

So far we have used the associativity of $+$, but from the equation above we need to find that $a = b$. In case of the naturals we would probably proceed as follows: since $a = (d + c) + a$, we find that $0 = d + c$ and then $d = 0 = c$. We used injectivity of adding a number in the first step and some kind of "non-invertability property" in the second one. Clearly requiring these properties in an abstracted fashion immediately results in the proof of antisymmetry. Since we used a single proof layout for this abstraction it is not surprising that these two properties turn out to be sufficient, but not necessary conditions for the antisymmetry of the action preorder. Fortunately they can be abstracted into a single property that is applicable in the general case.

Proposition 1 (Characterisation of antisymmetry I).
Let $(M, \cdot, e)$ be a monoid, A a set and $\varphi : M \times A \to A$ a monoid act. Then the following statements are equivalent:

(1) The act preorder $\sqsubseteq_\varphi$ is antisymmetric (i.e. an order).
(2) $\forall\, x, y \in M : \forall\, a \in A : (x \cdot y) \cdot_\varphi a = a \;\Rightarrow\; y \cdot_\varphi a = a$.

Proof. (1) $\Longrightarrow$ (2): We invite the reader to verify this on his or her own.
(2) $\Longrightarrow$ (1): Assume that (2) holds. Let $a, b \in A$ such that $a \sqsubseteq_\varphi b$ and $b \sqsubseteq_\varphi a$. Then there are $x, y \in M$ such that $x \cdot_\varphi a = b$ and $y \cdot_\varphi b = a$, hence

$$b = x \cdot_\varphi a = x \cdot_\varphi (y \cdot_\varphi b) = (x \cdot y) \cdot_\varphi b.$$

By (2) this yields $y \cdot_\varphi b = b$, but on the other hand $a = y \cdot_\varphi b$, so $a = b$. $\square$

Note how the proof of (2) $\Longrightarrow$ (1) resembles our exemplary proof from the beginning of this section. As we have mentioned before, $\text{curry}(\varphi)$ is a monoid homomorphism and thus its image $S := \text{curry}(\varphi)(M)$ is a submonoid of A^A. Observe that if there is a function $f \in S$ such that f is invertible in S and $f \neq \text{id}_A$, there are $x, y \in M$ such that $f = \text{curry}(\varphi)(y)$ and $\text{curry}(\varphi)(x) \circ f = \text{id}_A$. We then find that there is an $a \in A$ such that $y \cdot_\varphi a = f(a) \neq a$, but $(x \cdot y) \cdot_\varphi a = a$, so the above proposition states that $\sqsubseteq_\varphi$ is *not* antisymmetric. In other words: if S has non-trivial invertible elements, the corresponding preorder is not an order. In particular, if M is a group with more than one element and $\text{curry}(\varphi)$ is not the trivial homomorphism (i.e. $m \mapsto \text{id}_A$) then S is a group, too and thus $\sqsubseteq_\varphi$ is not an order.

The property that is equivalent to the antisymmetry can be viewed as a kind of fixpoint property: for all $x, y \in M$ and $a \in A$ we have that if a is a fixpoint of $b \mapsto (x \cdot y) \cdot_\varphi b$ it is also a fixpoint of $b \mapsto y \cdot_\varphi b$ (which also implies that it is a fixpoint of $b \mapsto x \cdot_\varphi b$). This fact can be expressed as follows.

Proposition 2 (Characterisation of antisymmetry II).
Let $(M, \cdot, e)$ be a monoid, A a set and $\varphi : M \times A \to A$ a monoid act. Then the following statements are equivalent:

(1) The act preorder $\sqsubseteq_\varphi$ is antisymmetric (i.e. an order).
(2) $\text{fix} \circ \text{curry}(\varphi) : (M, \cdot, e) \to (2^A, \cap, A)$ is a monoid homomorphism.

Proof. Let $\psi := \text{fix} \circ \text{curry}(\varphi)$. First of all we find that

$$(\text{fix} \circ \text{curry}(\varphi))(e) = \text{fix}(\text{curry}(\varphi)(e)) = \text{fix}(\text{id}_A) = A\,.$$

Now let $x, y \in M$. Then we can reason as follows:

$$\begin{aligned}
& \forall\, a \in A : (x \cdot y) \cdot_\varphi a = a \;\Rightarrow\; y \cdot_\varphi a = a \\
\Longleftrightarrow\; & \{ \text{ note above this proposition } \} \\
& \forall\, a \in A : (x \cdot y) \cdot_\varphi a = a \;\Rightarrow\; y \cdot_\varphi a = a \wedge x \cdot_\varphi a = a \\
\Longleftrightarrow\; & \{ \text{ fixpoint rephrasing } \} \\
& \forall\, a \in A : a \in \mathrm{fix}(\mathrm{curry}(\varphi)(x \cdot y)) \Rightarrow a \in \mathrm{fix}(\mathrm{curry}(\varphi)(x)) \cap \mathrm{fix}(\mathrm{curry}(\varphi)(y)) \\
\Longleftrightarrow\; & \{ \text{ definition of } \subseteq \} \\
& \mathrm{fix}(\mathrm{curry}(\varphi)(x \cdot y)) \subseteq \mathrm{fix}(\mathrm{curry}(\varphi)(x)) \cap \mathrm{fix}(\mathrm{curry}(\varphi)(y)) \\
\Longleftrightarrow\; & \{ \ (*) \ \} \\
& \mathrm{fix}(\mathrm{curry}(\varphi)(x \cdot y)) = \mathrm{fix}(\mathrm{curry}(\varphi)(x)) \cap \mathrm{fix}(\mathrm{curry}(\varphi)(y)) \\
\Longleftrightarrow\; & \{ \text{ definition of composition and application } \} \\
& \psi(x \cdot y) = \psi(x) \cap \psi(y)\,.
\end{aligned}$$

The equivalence denoted by $(*)$ is simple, since for any functions $f, g : A \to A$ we have that if $x \in \mathrm{fix}(f) \cap \mathrm{fix}(g)$ then $f(g(x)) = f(x) = x$ and thus $x \in \mathrm{fix}(f \circ g)$. Now we get

$$\begin{aligned}
& \sqsubseteq_\varphi \ \text{ is antisymmetric} \\
\Longleftrightarrow\; & \{ \text{ by Lemma 1 } \} \\
& \forall\, m, n \in M : \forall\, a \in A : (m \cdot n) \cdot_\varphi a = a \;\Rightarrow\; n \cdot_\varphi a = a \\
\Longleftrightarrow\; & \{ \text{ equivalence above } \} \\
& \forall\, m, n \in M : \psi(m \cdot n) = \psi(m) \cap \psi(n) \\
\Longleftrightarrow\; & \{ \text{ see above } \} \\
& \psi \text{ is a monoid homomorphism.}
\end{aligned}$$

□

Let us now show how to create a fitting monoid and a monoid act for a given preorder. The idea is quite simple – we want to define a transition system, such that its transition function is the act. To do that, we observe that orders and preorders are quite often drawn as their Hasse diagrams. These diagrams are designed specifically to omit reflexivity and transitivity since they can be restored in a trivial fashion as demonstrated in Figure 1. The last image bears a striking resemblance with a transition system except that there is no alphabet that can be used to act upon the states. Still, we can introduce an alphabet as indicated in Figure 2.

The sketched idea behind this alphabet can be formalised as follows. Let A be a non-empty set and $\preccurlyeq \subseteq A \times A$ a preorder, set $S := A$, $\Sigma := A$ and

$$\delta : S \times \Sigma \to S, \quad (s, \sigma) \mapsto \begin{cases} \sigma & : s \preccurlyeq \sigma \\ s & : \text{otherwise}\,. \end{cases}$$

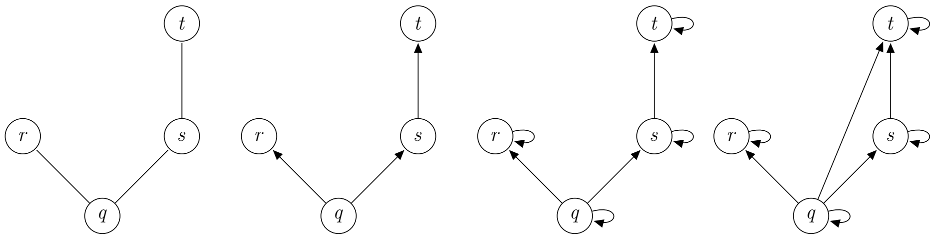


Fig. 1. A Hasse diagram, added directions, added reflexivity, added transitivity

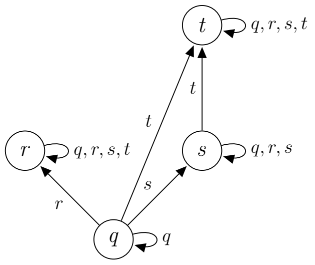


Fig. 2. An order transformed to a transition system

Obviously δ is well-defined. Let $\mathfrak{A} := (S, \Sigma, \delta)$. Then $\mathfrak{A}$ is a transition system. Let $x, y \in A$ and consider x a state and y a letter. This consideration yields the equivalence:

$$\delta(x, y) = y \iff x \preccurlyeq y.$$

When we rewrite δ as an infix operation (i.e. $x \,\delta\, y$ instead of $\delta(x, y)$), the above equivalence is very similar to the following well-known property of sets:

$$A \cup B = B \iff A \subseteq B.$$

In any lattice $(L, \sqcup, \sqcap)$ (which generalises the powerset of a set) we have

$$a \sqcup b = b \iff a \sqsubseteq b$$

by definition and in every idempotent, commutative semigroup $(S, +)$ we have

$$a + b = b \iff a \leq b.$$

Now let $M := \Sigma^*$ and

$$\varphi_\delta : \Sigma^* \times S \to S, \quad (w, s) \mapsto \begin{cases} s & : w = \varepsilon \\ \delta(s, w) & : w \in \Sigma \\ \varphi_\delta\left(\mathsf{tail}(w), \delta(s, \mathsf{head}(w))\right) & : \text{otherwise}\,. \end{cases}$$

Note that $\varphi_\delta = \mathrm{swap}(\delta^*)$. As mentioned in Example 2 the mapping φ_δ is a monoid action, which yields that $\sqsubseteq_{\varphi_\delta}$ as defined in Definition 2 is a preorder on $S = A$. It turns out that this new preorder is the same relation as the original one.

Proposition 3 (Generation of preorders).
We have $\sqsubseteq_{\varphi_\delta} = \preccurlyeq$, where φ_δ is the action and $\preccurlyeq$ is the preorder introduced above.

Proof. First of all we have:

$$\sqsubseteq_\varphi = \preccurlyeq \iff \forall\, a, b \in A : a \preccurlyeq b \Leftrightarrow \Big(\exists\, w \in A^* : \varphi(w, a) = b\Big).$$

by definition of the equality of relations. We now prove the second statement.
"$\Rightarrow$": Let $a, b \in A$ such that $a \preccurlyeq b$ and $w := b$. Then $w \in \Sigma \subseteq \Sigma^*$ and we have:

$$\varphi(w, a) = \delta(a, w) = \delta(a, b) = b\,.$$

"$\Leftarrow$": To simplify the proof we observe that the following holds:

$$\begin{aligned}
& \forall\, a, b \in A : \Big(\exists\, w \in A^* : \varphi(w, a) = b\Big) \Rightarrow a \preccurlyeq b \\
\iff \quad & \forall\, a, b \in A : \forall\, w \in A^* : \varphi(w, a) = b \Rightarrow a \preccurlyeq b \qquad (*) \\
\iff \quad & \forall\, w \in A^* : \forall\, a, b \in A : \varphi(w, a) = b \Rightarrow a \preccurlyeq b\,.
\end{aligned}$$

The equivalence marked with $(*)$ holds since the conclusion is independent of w. The latter statement will be proved by induction on the word length.
Ind. beginning: Let $w \in A^*$ such that $|w| = 0$. Then we have that $w = \varepsilon$. Now let $a, b \in A$ such that $\varphi(w, a) = b$. This gives us $b = \varphi(w, a) = \varphi(\varepsilon, a) = a$ and the reflexivity of $\preccurlyeq$ yields $a \preccurlyeq b$.
Ind. hypothesis: Let $n \in \mathbb{N}$ and assume that the following holds:

$$\forall\, w \in A^* : |w| = n \Rightarrow \Big(\forall\, a, b \in A : \varphi(w, a) = b \Rightarrow a \preccurlyeq b\Big).$$

Ind. step: Let $v \in A^*$ such that $|v| = n + 1$. Then there are $x \in A$ and $w \in A^*$ such that $v = xw$ and $|w| = n$. Let $a, b \in A$ such that $\varphi(v, a) = b$. Then we have:

$$b = \varphi(v, a) = \varphi(xw, a) = \varphi(w, \delta(a, x)) = \begin{cases} \varphi(w, x) & : a \preccurlyeq x \\ \varphi(w, a) & : \text{otherwise}\,. \end{cases}$$

If $a \preccurlyeq x$, then by the induction hypothesis we have $x \preccurlyeq b$, which, by the transitivity of $\preccurlyeq$, results in $a \preccurlyeq b$. If on the other hand $a \not\preccurlyeq x$, then $b = \varphi(w, a)$ and by the induction hypothesis we immediately obtain $a \preccurlyeq b$, since $|w| = n$. This concludes the induction and the proof as well. ☐

We now rephrase the result of this section for the purpose of legibility.

Corollary 1 (Preorders are monoidally generated).
Let A be a non-empty set and $\preccurlyeq \subseteq A \times A$ a preorder.
Then there is a monoid M and a monoid act $\varphi : M \times A \to A$ such that $\preccurlyeq = \sqsubseteq_\varphi$.

For any non-empty set A the set A^* is infinite (and countable iff A is countable). This monoid is somewhat large, since it is the free monoid generated by the set A. However we can use the well-known quotient construction (cf. [6]) to obtain an action in which different monoid elements act differently[4](i.e. curry(φ)

[4] The cited source deals with finite state sets only, but the technique easily carries over to infinite sets as well.

is injective; such an action is called *faithful*). To form a quotient one defines for all $m, n \in M$:

$$m \sim n :\Longleftrightarrow \operatorname{curry}(\varphi)(m) = \operatorname{curry}(\varphi)(n).$$

Clearly, $\sim$ is an equivalence relation, moreover it is a monoid congruence such that $M/_\sim$ is a monoid as well. The new action

$$\varphi_{\text{quotient}} : M/_\sim \times A \to A, ([m]_\sim, a) \mapsto \varphi(m, a)$$

is then faithful. The quotient monoid is usually far smaller than the free monoid. Also in many cases the quotient monoid can be described in a less generic (and more comprehensible) way than as the quotient modulo a congruence.

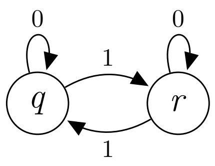


Fig. 3. An act that doesn't induce an order

Let us revisit the antisymmetry of the action preorder one more time in the context of transition systems. Consider the system in Figure 3. The preorder induced by this act is not an order, because we have $\delta(1, q) = r$ and $\delta(1, r) = q$, which gives us $q \sqsubseteq_\delta r$ and $r \sqsubseteq_\delta q$ respectively, but $q \neq r$. What is the key ingredient to break antisymmetry in this example? It is the existence of a non-trivial cycle. With our previous results we can then prove the following lemma.

Proposition 4 (Characterisation of antisymmetry III).
Let $\mathfrak{A} = (S, \Sigma, \delta)$ be a transition system. Then the following statements are equivalent:

(1) $\sqsubseteq_{\text{swap}(\delta^*)}$ is antisymmetric.
(2) $\mathfrak{A}$ contains no non-trivial cycles, i.e. if a word does not change a state, so does every prefix of this word.

This can be proved with Lemma 1, since the second condition is easily translated into the second statement of the cited lemma.

There is an interesting analogue of this lemma. Consider a graph $G = (V, E)$, where V is a set and $E \subseteq V \times V$. The reachability relation $\rightsquigarrow_G$ of G is given by

$$x \rightsquigarrow_G y :\Longleftrightarrow \text{ there is a path from } x \text{ to } y$$

for all $x, y \in V$. It is easy to see that $\rightsquigarrow_G$ is reflexive and transitive[5]. Also it is well-known that if G contains no non-trivial cycles (i.e. loops are allowed), then $\rightsquigarrow_G$ is an order relation. One can also show that the antisymmetry of $\rightsquigarrow_G$ results in no non-trivial cycles. These facts demonstrate that antisymmetry and cycle-freeness are closely related and can be considered in the very same light.

[5] In fact $\rightsquigarrow_G$ is the reflexive-transitive closure of E, cf. [7].

4 Implementation in Curry

In this section we use monoid actions to implement orders in the functional logical programming language Curry (cf. [5]). We provide a simple prototypical implementation of monoid acts and resulting preorders and discuss its shortcomings as well. To run our code we use the KiCS2 compiler (see [4]).

The components of a monoidally generated order (a monoid and a monoid action) can be expressed more generally for simplicity. We can use that to implement a very simple version of the general preorder.[6]

```
type MonoidAct μ α = μ → α → α
type OrderS α = α → α → Success
preOrder :: MonoidAct μ α → OrderS α
preOrder (⊗) x y = z ⊗ x =:= y
  where z free
```

Let us consider an example. To that end we define the naturals as Peano numbers.

```
data ℕ = 𝕆 | S ℕ
(⊕) :: ℕ → ℕ → ℕ
𝕆   ⊕ y = y
S x ⊕ y = S (x ⊕ y)
```

This addition yields a notion of comparison[7]:

```
(⊑⊕) :: OrderS ℕ
(⊑⊕) = preOrder (⊕)
```

Loading these definitions in KiCS2 we get

```
kics2> S (S 𝕆) ⊑⊕ 𝕆
No more values
kics2> S (S 𝕆) ⊑⊕ S (S 𝕆)
Success
No more values
kics2> x ⊑⊕ S 𝕆 where x free
{x = (S 𝕆)} Success
{x = 𝕆} Success
No more values
```

Note that the very concept of an act preorder uses both concepts integrated in Curry – a functional component "⊗" and a logical one "*z* **where** *z* **free**".

[6] Clearly this is a greatly simplified approach, since not every function $f :: \mu \to \alpha \to \alpha$ is a monoid act. A user has to verify that μ is a monoid and f is a monoid act to ensure that *preOrder f* is in fact a preorder. Alternatively a proof assistant (e.g. Coq [3]) can be used to guarantee that *preOrder* is applicable only once the necessary conditions have been proved.

[7] It is simple (but lengthy) to define the integers based on the naturals and to extend the definitions of $\oplus$ and $\sqsubseteq_{\oplus}$ to allow the implementation of $\leq_{\mathbb{Z}}$.

While this implementation is very close to its mathematical basis, the reader may have noticed that our order type does not have a relational look-and-feel, since orders are more likely to have the type $\alpha \to \alpha \to \mathit{Bool}$. Such a relational version can be obtained by using negation as failure which can be expressed elegantly in Curry using set functions [2]. Since our implementation is intended to be prototypical and the above type is rather natural in light of logic programming we omit the presentation of such a version.

Let us illustrate these "logical orders" with an example function. In total orders[8] each two elements are comparable, which allows the notion of a maximum.

$$
\begin{array}{l}
\mathit{maximumBy} :: \mathit{OrderS}\ \alpha \to \alpha \to \alpha \to \alpha \\
\mathit{maximumBy}\ (\leqslant)\ x\ y \mid x \leqslant y = y \\
\mathit{maximumBy}\ (\leqslant)\ x\ y \mid y \leqslant x = x
\end{array}
$$

The rules we gave are overlapping – in case of equality both rules are applicable, resulting in multiple results. Fortunately, when $\leqslant$ is an order all results are equal, since the constraints $x \leqslant y$ and $y \leqslant x$ imply that $x = y$.

Clearly the previous example of an order generated by a monoid act was trivial, since the action was the monoid operation itself. Let us consider a non-trivial example next – the "has-suffix-order" $\trianglerighteq$ on A^* for some given set A. The order is defined by

$$xs \trianglerighteq ys :\Longleftrightarrow \exists\, zs \in A^* : xs = zs +\!\!+\, ys\,.$$

Note that the definition does not hint at a possible monoid act that generates this order, because such an act needs to apply a function to the *first* element of the comparison and not the second one. Still we can calculate the following:

$$
\begin{aligned}
xs \trianglerighteq ys &\iff \exists\, zs \in A^* : xs = zs +\!\!+\, ys \\
&\iff \exists\, zs \in A^* : zs +\!\!+\, ys = xs \\
&\iff ys \sqsubseteq_{(+\!\!+)} xs\,,
\end{aligned}
$$

which can be rephrased as $(\trianglerighteq) = \mathit{flip}\ (\mathit{preOrder}\ (+\!\!+))$. How is this relation more interesting? We were able to define this order as a flipped version of another order. Clearly this is interesting in its own right, but Corollary 1 states that there is a monoid and a monoid act yielding precisely the order we need. Let us define an auxiliary function.[9]

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathit{drop}_{\mathbb{N}} :: \mathbb{N} \to [\alpha] \to [\alpha]} \\
\mathit{drop}_{\mathbb{N}}\ \mathbb{O} & x@(\_ : \_) & = x \\
\mathit{drop}_{\mathbb{N}}\ \_ & [\,] & = [\,] \\
\mathit{drop}_{\mathbb{N}}\ (S\ x) & (\_ : ys) & = \mathit{drop}_{\mathbb{N}}\ x\ ys
\end{array}
$$

We observe that the following holds for all $xs, ys :: [\alpha]$:

$$xs \trianglerighteq ys \iff \exists\, n :: \mathbb{N} : \mathit{drop}_{\mathbb{N}}\ n\ xs = ys\,.$$

This does look like an act preorder. But is $\mathit{drop}_{\mathbb{N}}$ an action? It is indeed.

[8] An order $\leq \subseteq A \times A$ is called **total** iff for all $x, y \in A$ it is true that $x \leq y$ or $y \leq x$.
[9] The pattern matching in the first rule makes the rules non-overlapping.

Lemma 2 (Properties of $drop_{\mathbb{N}}$).
Let A be a non-empty set and

$$\delta : \mathbb{N} \times A^* \to A^*,\ (n, l) \mapsto \begin{cases} l & : n = 0 \\ [\,] & : l = [\,] \\ \delta(n', l') & : \exists n' \in \mathbb{N} : n = 1 + n' \wedge \exists x \in A : l = x : l'. \end{cases}$$

Then δ is (well-defined and) a monoid action of $\mathbb{N}$ on A^* and $\trianglerighteq = \sqsubseteq_\delta$ is an order.

Using properties of natural numbers the proof is basically a straightforward induction. We omit it for two reasons – avoiding unnecessary clutter and the fact that δ from the above lemma is only a version of $drop_{\mathbb{N}}$ that operates on finite and deterministic arguments, while $drop_{\mathbb{N}}$ can be used on infinite or non-deterministic arguments as well.

We can implement the order $\trianglerighteq$ in terms of $drop_{\mathbb{N}}$.

```
(⊵) :: OrderS [α]
(⊵) = preOrder drop_ℕ
```

For comparison we also define the version discussed above.

```
(⊵₂) :: OrderS [α]
(⊵₂) = flip (preOrder (++))
```

The difference between these implementation is that $(\trianglerighteq_2)$ searches in an upward fashion, while $(\trianglerighteq)$ does the same in an downward direction. This is to say that for $(\trianglerighteq_2)$ we perform the unification $z +\!\!+\, y =:= x$ with a free variable z. This creates a list structure omitting unnecessary components, which essentially searches for a possible number of cons cells that can be ignored in x to obtain y. Using $(\trianglerighteq)$ does precisely that explicitly, because we search for a natural number of elements to explicitly drop from x to obtain y.

Typically functionally similar parts of programs are abstracted as far as possible to be applicable in different situations. In functional languages such abstractions usually include higher-order functions, that take necessary operations as additional arguments. Our implementation of *preOrder* is such a higher order function, that allows us to define *every* order in terms of a specific act.

A drawback of the general abstraction is its possible complexity or even indecidability. Concerning the complexity consider the comparison of $one = S\ \mathbb{O}$ to some value $(S\ x) :: \mathbb{N}$. Using $\sqsubseteq_\oplus$ this operation is quadratic in the size of $S\ x$. Clearly an implementation of the same order "by hand" requires precisely two comparisons – first match the outer S and then compare $\mathbb{O}$ to x, which is automatically true. As for the indecidability we consider integer lists. Now consider the list $ones = 1 : ones$. We can then define $x = 0 : 1 : ones$ and $y = 1 : 0 : ones$. When equipped with the lexicographical order $\leq_{\text{lex}}$, which we implement by hand, we can easily determine that $x \leq_{\text{lex}} y$. Now suppose we have found the monoid act φ that generates the order $\leq_{\text{lex}}$. Then there is a value z such that $\varphi\ z\ x = y$ (mathematically!), but this equality is not decidable.

In general the above implementation is applicable to orders on finite terms without any occurrences of $\bot$ (the latter is due to the semantics of $=:=$, see [5]).

5 An Alternative Abstraction

The reader may have noticed that until now we have presented orders on countable sets only. But what about, say, the order on $\mathbb{R}$? The usual definition states that for $x, y \in \mathbb{R}$ we have

$$x \leq_{\mathbb{R}} y :\Longleftrightarrow y - x \in \mathbb{R}_{\geq 0} \iff \exists\, z \in \mathbb{R}_{\geq 0} : z + x = y.$$

Structurally the latter definition looks exactly like the order on $\mathbb{Z}$. While it seems odd that we use $\mathbb{R}_{\geq 0}$ in the definition, we recall that the order on $\mathbb{R}$ requires the existence of a so-called *positive cone* that is named $\mathbb{R}_{>0}$ in this example and $\mathbb{R}_{\geq 0} = \mathbb{R}_{>0} \cup \{0\}$. A positive cone of a group $(G, +, 0)$ is a $P \subseteq G$ that satisfies

$$P + P \subseteq P \quad \wedge \quad 0 \notin P \quad \wedge \quad G = -P \cup \{0\} \cup P,$$

where $P + P$ and $-P$ are understood element-wise. Then $P_0 := P \cup \{0\}$ is a submonoid of G and $+|_{P_0 \times G}$ is a monoid act. This construction covers the act that creates the order on $\mathbb{Z}$ as well as the action that induces the order on $\mathbb{R}$.

Observe that the notion of a positive cone requires an inversion operation of the group operation. In case of monoids the operation of the monoid usually cannot be inverted. Replacing the monoid with a group transforms the monoid act into a group act (without any additional requirements). Then the action preorder becomes an equivalence relation and non-trivial orders are never equivalences. In fact the concept of cycle-freeness requires the monoids in our examples to be "anti-groups", which is to say that the image of the monoid under the curried act does not contain non-trivial invertible elements. What is more is that every order that is defined in terms of a positive cone is total, which is easily shown using the above definition. Since not every order is total, we cannot expect to find a "positive-cone representation" for every order.

As we have mentioned above the use of a positive cone to define an order is a special case of a monoid act. The latter requires less structure and is thus easily defined. The former automatically comes with more properties and is generally more natural in the context of ordering groups or fields.

6 Related Work and Discussion

Monoid acts and corresponding preorders appear in different contexts naturally e.g. transition systems [6] and algebra [1]. The examples from the introduction constitute well-known orders that are defined as act preorders, while the "has-suffix"-order is slightly less common. Orders are also ubiquitous in computer science and mathematics, but they are mostly treated in as tools, rather than objects of research.

From our results we know that every order can be defined in terms of a monoid action and we have given an exemplary implementation in Curry. In our implementation we merely checked whether there is a variable, that acts on the first argument in a way that results in the second one. Clearly, we could also ask for this variable as well and obtain the following function.

```
$cofactor :: MonoidAct\ \mu\ \alpha \to \alpha \to \alpha \to \mu$
$cofactor\ (\otimes)\ a\ b \mid z \otimes a =:= b = z$
  **where** $z$ **free**
```

The above function bears a striking similarity with the following definition of an "inverse function".

```
$inverse :: (a \to b) \to b \to a$
$inverse\ f\ y \mid f\ x =:= y = x$
  **where** $x$ **free**
```

In fact we can redefine *cofactor* in terms of *inverse*, namely:

```
$cofactor\ (\otimes)\ a\ b = inverse\ (\otimes a)\ b$
```

When translated back into mathematical notation the above states that *cofactor* yields some $z \in \varphi(-,a)^{-1}(\{b\})$, where φ is $\otimes$ uncurried. Searching for preimage values (at least implicitly) is a common task in logic programming that appears naturally in a variety of definitions, e.g.:

```
$predecessor :: \mathbb{N} \to \mathbb{N}$
$predecessor\ n \mid S\ m =:= n = m$
  **where** $m$ **free**
$predecessorAsPreimage = inverse\ S$
```

As we stated earlier, the action preorder is a well-known relation and we have used very little of existing knowledge. If $\varphi : M \times A \to A$ is a monoid act, we can consider the orbit of some element $a \in A$ that is defined as

$$\begin{aligned}\mathrm{orbit}_M(a) &:= \{m \cdot_\varphi a \mid m \in M\} = \{b \in A \mid \exists\, m \in M : m \cdot_\varphi a = b\}\\ &= \{b \in A \mid a \sqsubseteq_\varphi b\}.\end{aligned}$$

The latter set is simply the majorant of a (sometimes denoted $\{a\}^{\uparrow}$ or (a)). This simple connection allows to study the notion of majorants in the context of monoid actions (and vice versa) thus combining two well-known and well-studied concepts. An additional similarity occurs when comparing our application of the free monoid with the construction of a free group of a set [1]. We omit the details here due to lack of space and immediate applicability.

In Curry every type is already ordered lexicographically w.r.t. the constructors (see definition of *compare* in [5]). Clearly this gives a total order on every type, but the actual comparison results depend on the order of constructors, which requires careful choice of this order. Defining integers as

```
**data** $\mathbb{Z} = Pos\ \mathbb{N} \mid Neg\ \mathbb{N}$
```

leads to positive numbers being smaller than negative ones. Actions provide a simple way to define (additional) orders in terms of functions that do not depend on the order of the constructors. In the above example a library that defines the data type $\mathbb{Z}$ is likely to define an addition on integers $add :: \mathbb{Z} \to \mathbb{Z} \to \mathbb{Z}$ and an embedding $toInteger :: \mathbb{N} \to \mathbb{Z}$. With these two functions one can easily define

$act :: \mathbb{N} \to \mathbb{Z} \to \mathbb{Z}$
$act\ n\ z = add\ (toInteger\ n)\ z$

which yields the usual order on $\mathbb{Z}$ as *preOrder act*. An additional gain in using actions is that one can define orders in a more declarative way – instead of thinking about bit representations or constructors one simply states what is necessary for one number to be smaller than another.

The prototypical stencil of the above implementation can be varied in different ways. One variation concerns the adjustment of the type to the relational version $\alpha \to \alpha \to Bool$ that we have mentioned before. Often an order on a set A is naturally expressed in terms of another order $(B, \leq_B)$ and an injective function $f : A \to B$ such that for all $a, a' \in A$ we have

$$a \leq_A a' :\Longleftrightarrow f(a) \leq_B f(a').$$

Additionally if M is a monoid and $\varphi : M \times B \to B$ is a monoid act that generates $\leq_B$ we can incorporate f into the existential quantification:

$$a \leq_A a' \iff \exists\, z \in M : z \cdot_\varphi f(a) = f(a').$$

While the codomain of f has at least the cardinality of A (because of the injectivity of f), the comparison of values in B may be less complex.

On a more theoretical note it is interesting to study properties of orders in terms of properties of actions and vice versa. For instance one can show that a faithful act of a monoid that has no invertible elements except for its unit always yields an infinite order, which is not obvious at first glance. We suspect that there are quite a few connections between these seemingly different concepts.

Hats off to: Fabian Reck for detailed explanation of Curry, Rudolf Berghammer for encouraging this work and additional examples and Insa Stucke for general discussions.

References

1. Aluffi, P. *Algebra: Chapter 0.* AMS, 2009.
2. Antoy, S., and Hanus, M. Set Functions for Functional Logic Programming. In *Proc. of the 11th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'09)* (2009), ACM Press, pp. 73–82.
3. Bertot, Y., and Castéran, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
4. Brassel, B., Hanus, M., Peemöller, B., and Reck, F. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)* (2011), Springer LNCS 6816, pp. 1–18.
5. Hanus (ed.), M. Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at http://www.curry-language.org, 2012.
6. Holcombe, W. M. L. *Algebraic Automata Theory.* Cambridge Univ. Press, 1982.
7. Schmidt, G., and Ströhlein, T. *Relations and Graphs.* Springer-Verlag, 1993.

Sharing and Exchanging Data

Rana Awada[1], Pablo Barceló[2], and Iluju Kiringa[1]

[1] University of Ottawa, EECS
[2] University of Chile, Department of Computer Science

Abstract. Exchanging and integrating data that belong to worlds of different vocabularies are two prominent problems in the database literature. These problems have been, so far, solved separately in data exchange and data coordination settings, respectively, but never been studied in a unified setting. In this paper, we propose a class of mappings – called *DSE*, for *data sharing and exchange* – that represents such a unified setting. We introduce a particular DSE setting with a particular interpretation of related data where an ordinary data exchange or data integration solution cannot be applied. We define the class of *DSE* solutions for such a DSE setting, that allow to store a part of explicit data and a set of inference rules used to generate the complete set of exchanged data. We prove that a particular DSE solution with good properties – namely, one that contains a minimal amount of explicit data – can be computed in LOGSPACE. Finally, we define the set of certain answers to conjunctive queries and we show how to compute those efficiently.

1 Introduction

Different problems of integrating and accessing data in independent data sources have been introduced in the literature, and different settings have been proposed to solve those problems. Data exchange [10, 5] and data coordination [1, 2, 13, 4] are two among these introduced settings that have received wide attention. A data exchange setting considers the problem of moving data residing in independent applications and accessing it through a new target schema. This process of exchange only allows to move data from a source into a target that uses the same set of vocabularies, and thus, transformation occurs to the structure of the data, and not to the data itself. On the other hand, a data coordination setting allows the access of data residing in independent sources without having to exchange it and while maintaining autonomy.

We show in the following examples that a collaborative process – including coordination tools for managing different vocabularies of different sources and exchange tools – would yield interoperability capabilities that are beyond the ones that can be offered today by any of the two tasks separately.

Recall that a data exchange (DE) setting [10] $\mathfrak{S}$ consists of a source schema $\mathbf{S}$, a target schema $\mathbf{T}$, and a set Σ_{st} of database dependencies – the so-called source-to-target dependencies – describing structural changes made to data as we move it from the source to the target. This exchange solution supports exchanging information between two applications that refer to the same object using the same instance value. We present in Example 1 a DE example.

Example 1. Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \Sigma_{st})$ be a DE setting. Let $\mathbf{S}$ be a source schema for the University of Ottawa and $\mathbf{T}$ be a target schema. Suppose $\mathbf{S}$ has the relations: $Student(Sname, Sage)$, $Course(Cid, Cname, Pname)$, and $Enroll(Sname, Cid, Cgrade)$. Also, let $\mathbf{T}$ consist of the relation symbols $St(Sname, Sage, Saddress)$, $Cr(Cid, Cname, Pname)$ and $Take(Sname, Cid, Cgrade)$.

Relation *Student* (*St*) stores students name and age (and address) information. Relation *Course* (*Cr*) stores courses ids and names information, in addition to the program name which provides each course. Finally, relation *Enroll* (*Take*) stores the set of courses that each student completed. Further, assume that Σ_{st} consists of the following st dependencies:
$(a) \forall x \forall y \big(Student(x, y) \rightarrow \exists z St(x, y, z)\big)$
$(b) \forall x \forall y \forall z \big(Enroll(x, y, z) \rightarrow Take(x, y, z)\big)$.

Let $I = \{Student(Alex, 18)$, $Course(CSI1390$, *Introduction to Computers*, $CS)$, $Enroll(Alex, CSI1390, 80)\}$ be a source instance. Then, $J = \{St(Alex, 18, \perp_1)$, $Take(Alex, CSI1390, 80)\}$, where $\perp_1$ is fresh unknown (or *null*) value, is a target instance that satisfies Σ_{st} when considered with the source instance I. Following usual DE terminology [10], we say that J is a solution for I under $\mathfrak{S}$. □

In Example 1, source values are referred to in both the source and the target instances using the same names. However, there exist cases where objects in the source are named and referred to differently in the target. A motivating example of such an exchange scenario is exchanging information for students applying for program transfers from one university to a different one. Indeed, different universities can offer different courses and a course in one university can possess one or more equivalent courses in the new university.

Unlike data exchange, data coordination (DC) settings [2, 4, 13] solve the problem of integrating information of different sources that possess different yet related domain of constants by using the *mapping table* construct [13]. A mapping table specifies for each source value the set of *related* (or *corresponding*) target values. DC settings have been studied mainly in peer-to-peer networks, where sources – called peers – possess equal capabilities and responsibilities in terms of propagating changes and retrieving related information. A DC setting $\mathfrak{S}$ consists of two schemas $\mathbf{S}_1$ and $\mathbf{S}_2$, and a set of mapping tables $\{\mathcal{M}\}$. We give in the following example a data coordination instance that allows integrating information from two different universities.

Example 2. Let $\mathfrak{S}$ be a DC setting. Suppose that $\mathbf{S}_1$ in $\mathfrak{S}$ is a schema for the University of Ottawa (UOO) and $\mathbf{S}_2$ in $\mathfrak{S}$ is the schema of the University of Carleton (UOC). In reference to Example 1, assume that $\mathbf{S}_1$ is the same schema as $\mathbf{S}$ and $\mathbf{S}_2$ is the same schema as $\mathbf{T}$. Further, assume that $\mathbf{S}_1$ and $\mathbf{S}_2$ are connected by a mapping table $\mathcal{M}$ where $\mathcal{M}$ consists of the following pairs $\{(CSI1390, ECOR1606)$, $(CSI1390, COMP1005)$, (CS, CS), $(ENG, ENG)\}$.

Let I be an instance of $\mathbf{S}_1$ and $J = \{St(Alex, 18, Ottawa)$, $Cr(ECOR1606$, *Problem Solving and Computers*, $ENG)$, $Cr(COMP1005$, *Introduction to Computer Science I*, $CS)$, $Take(Alex$, $ECOR1606$, $80)\}$ be an instance of $\mathbf{S}_2$.

According to [4], posing a query q to I that computes the list of students considered to have finished CS courses in UOO, will re-write q to a query q' to retrieve a similar list from UOC following the semantics of $\mathcal{M}$. A query q' can be the following: q': Select $Sname$ From Cr, $Take$ Where $Cr.Cid = Take.Cid$ And $Cr.Pname = 'CS'$. In this case, the answer of posing q' to J is $\emptyset$. $\square$

Assume that UOO accredits a 'CS' course to a student doing program transfer from UOC only if this student finishes an equivalent 'CS' course, according to $\mathcal{M}$, in UOC. In Example 2, *Alex* is not considered as finished a 'CS' course at UOC. Therefore, if *Alex* does a transfer to the CS program in UOO, he will not be credited the *Introduction to Computers* course with code $CSI1390$. However, if the semantics of the mapping table $\mathcal{M}$ in this example specifies that course $CSI1390$ in UOO is equivalent to the ENG course $ECOR1606$ in UOC, and course $CSI1390$ in UOO is equivalent to the CS course $COMP1005$ in UOC, then it can be deduced that courses $ECOR1606$ and $COMP1005$ are considered equivalent with respect to UOO according to $\mathcal{M}$. Therefore, given the fact that $Take(Alex$, $ECOR1606$, $80) \in J$ in Example 2 and according to the semantics in $\mathcal{M}$, *Alex* is considered to have finished the equivalent CS course $COMP1005$ in UOC and he should be credited the 'CS' course $CSI1390$ with a grade 80 if he did a transfer to UOO.

To solve such a problem, we introduce a new class of settings, called *data sharing and exchange* (DSE) settings, where exchanges occur between a source and a target that use different sets of vocabularies. Despite the importance of the topic, the fundamentals of this process have not been laid out to date. In this paper, we embark on the theoretical foundations of such problem, that is, exchanging data between two independent applications with a different set of domain of constants. DSE settings extend DE settings with a *mapping table* $\mathcal{M}$, introduced in [13], to allow collaboration at the instance level. In addition, the set of source to target dependencies Σ_{st} in DSE refers to such mapping table so that coordination of distinct vocabularies between applications takes place together with the exchange.

From what we have mentioned so far about DSE, one would think that all DSE instances can be reduced to a usual DE instance where the source schema is extended with the st-mapping table $\mathcal{M}$. However, we argue in this paper that there exist DSE settings with particular interpretation of related data in mapping tables where DSE can not be reduced to a DE setting (like the case in Example 2). We consider in this paper a particular interpretation of related data in a mapping table; that is, a source element is mapped to a target element only if both are considered to be equivalent (i.e denote the same object). In this DSE scenario, DSE and DE are different because source and target data can be incomplete with respect to the "implicit" information provided by the semantics of mapping tables. So, in Example 2, we can say that J is incomplete with respect to the semantics of $\mathcal{M}$, and *Alex* is considered by UOO to have

finished the CS course *Introduction to Computers*. To formalize this idea we use techniques developed by Arenas et al. in [8], where authors introduced a *knowledge exchange* framework for exchanging knowledge bases. It turns out that this framework suits our requirements, and, in particular, allows us to define the exchange of both explicit and implicit data from source to target. Our main contributions in this work are the following:
(1) Universal DSE solutions We formally define the semantics of a DSE setting and introduce the class of universal DSE solutions, that can be seen as a natural generalization of the class of universal data exchange solutions [10] to the DSE scenario, and thus, as "good" solutions. A universal DSE solution consists of a subset of explicit data that is necessary to infer the remaining implicit information using a given set Σ_t of rules in the target.
(2) Minimal universal DSE solutions We define the class of minimal universal DSE solutions which are considered as "best" solutions. A minimal universal DSE solution contains the minimal amount of explicit data required to compute the complete set of explicit and implicit data using a set of target rules Σ_t. We show that there exists an algorithm to generate a *canonical minimal universal* DSE solution, with a well-behaved set Σ_t of target rules, in LOGSPACE.
(3) Query answering We formally define the set of DSE certain answers for conjunctive queries. We also show how to compute those efficiently using canonical minimal universal DSE solutions.

2 Preliminaries

A *schema* $\mathbf{R}$ is a finite set $\{R_1, \ldots, R_k\}$ of relation symbols, with each R_i having a fixed arity $n_i > 0$. Let D be a countably infinite domain. An *instance* I of $\mathbf{R}$ assigns to each relation symbol R_i of $\mathbf{R}$ a finite n_i-ary relation $R_i^I \subseteq \mathsf{D}^{n_i}$. Sometimes we write $R_i(\bar{t}) \in I$ instead of $\bar{t} \in R_i^I$, and call $R_i(\bar{t})$ a *fact* of I. The *domain* $dom(I)$ of instance I is the set of all elements that occur in any of the relations R_i^I. We often define instances by simply listing the facts that belong to them. Further, every time that we have two disjoint schemas $\mathbf{R}$ and $\mathbf{S}$, an instance I of $\mathbf{R}$ and an instance J of $\mathbf{S}$, we define (I, J) as the instance K of schema $\mathbf{R} \cup \mathbf{S}$ such that $R^K = R^I$, for each $R \in \mathbf{R}$, and $S^K = S^J$, for each $S \in \mathbf{S}$.

Data exchange settings As is customary in the data exchange literature [10, 5], we consider instances with two types of values: constants and nulls.[3] More precisely, let Const and Var be infinite and disjoint sets of constants and nulls, respectively, and assume that $\mathsf{D} = \mathsf{Const} \cup \mathsf{Var}$. If we refer to a schema $\mathbf{S}$ as a *source* schema, then we assume that for an instance I of $\mathbf{S}$, it holds that $dom(I) \subseteq \mathsf{Const}$; that is, source instances are assumed to be "complete", as they do not contain missing data in the form of nulls. On the other hand, if we refer to a schema $\mathbf{T}$ as a *target* schema, then for every instance J of $\mathbf{T}$, it holds that $dom(J) \subseteq \mathsf{Const} \cup \mathsf{Var}$; that is, target instances are allowed to contain null values.

[3] We usually denote constants by lowercase letters $a, b, c, \ldots$, and nulls by symbols $\perp, \perp', \perp_1, \ldots$

A *data exchange* (DE) *setting* is a tuple $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \Sigma_{st})$, where $\mathbf{S}$ is a source schema, $\mathbf{T}$ is a target schema, $\mathbf{S}$ and $\mathbf{T}$ do not have predicate symbols in common, and Σ_{st} consists of a set of *source-to-target tuple-generating* dependencies (st-tgds) that establish the relationship between source and target schemas. An st-tgd is a FO-sentence of the form: $\forall \bar{x} \forall \bar{y}\, (\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z}))$, where $\phi(\bar{x}, \bar{y})$ is a conjunction of relational atoms over $\mathbf{S}$ and $\psi(\bar{x}, \bar{z})$ is a conjunction of relational atoms over $\mathbf{T}$.[4] A *source* (resp. *target*) instance K for $\mathfrak{S}$ is an instance of $\mathbf{S}$ (resp. $\mathbf{T}$). We usually denote source instances by $I, I', I_1, \ldots$, and target instances by $J, J', J_1, \ldots$.

An instance J of $\mathbf{T}$ is a *solution* for an instance I under $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \Sigma_{st})$, if the instance (I, J) of $\mathbf{S} \cup \mathbf{T}$ satisfies every st-tgd in Σ_{st}. If $\mathfrak{S}$ is clear from the context, we say that J is a solution for I.

The data exchange literature has identified a class of preferred solutions, called the *universal* solutions, that in a precise way represents all other solutions. In order to define these solutions, we need to introduce the notion of homomorphism between instances. Let K_1 and K_2 be instances of the same schema $\mathbf{R}$. A *homomorphism* h from K_1 to K_2 is a function $h : dom(K_1) \to dom(K_2)$ such that: (1) $h(c) = c$ for every $c \in \mathsf{Const} \cap dom(K_1)$, and (2) for every $R \in \mathbf{R}$ and tuple $\bar{a} = (a_1, \ldots, a_k) \in R^{K_1}$, it holds that $h(\bar{a}) = (h(a_1), \ldots, h(a_k)) \in R^{K_2}$. Let $\mathfrak{S}$ be a DE setting, I a source instance and J a solution for I under $\mathfrak{S}$. Then J is a *universal* solution for I under $\mathfrak{S}$, if for every solution J' for I under $\mathfrak{S}$, there exists a homomorphism from J to J'. For instance, consider again the DE setting $\mathfrak{S}$ shown in Example 1 in the Introduction. It is not hard to prove that $J = \{St(Alex, 18, \perp_1), Take(Alex, CSI1390, 80)\}$, where $\perp_1$ is a fresh null value, is a universal solution for the source instance I under $\mathfrak{S}$.

For the class of data exchange settings that we referred to in this paper, every source instance has a universal solution [10]. Further, given a DE setting $\mathfrak{S}$, there is a procedure (based on the *chase* [9]) that computes a universal solution for each source instance I under $\mathfrak{S}$. In the case when $\mathfrak{S}$ is fixed such procedure works in LOGSPACE. Assuming $\mathfrak{S}$ to be fixed is a usual and reasonable assumption in data exchange [10], as mappings are often much smaller than instances. We stick to this assumption for the rest of the paper.

Mapping tables Coordination can be incorporated at the *data* level, through the use of *mapping* tables [13]. These mechanisms were introduced in data coordination settings [4] to establish the correspondence of related information in different domains. In its simplest form, mapping tables are just binary tables containing pairs of corresponding identifiers from two different sources. Formally, given two domains D_1 and D_2, not necessarily disjoint, a mapping table over $(\mathsf{D}_1, \mathsf{D}_2)$ is nothing else than a subset of $\mathsf{D}_1 \times \mathsf{D}_2$. Intuitively, the fact that a pair (d_1, d_2) belongs to the mapping table implies that value $d_1 \in \mathsf{D}_1$ *corresponds* to value $d_2 \in \mathsf{D}_2$. Notice that the exact meaning of "correspondence" between values is unspecified and depends on the application.

[4] We usually omit universal quantification in front of st-tgds and express them simply as $\phi(\bar{x}, \bar{y}) \to \exists \bar{z}\, \psi(\bar{x}, \bar{z})$.

In this paper we deal with a very particular interpretation of the notion of correspondence in mapping tables. We assume that the fact that a pair (a, b) is in a mapping table implies that a and b are equivalent objects. We are aware of the fact that generally mapping tables do not interpret related data in this way. However, we argue that this particular case is, at the same time, practically relevant (e.g. in peer-to-peer settings [13]) and theoretically interesting (as we will see along the paper).

This particular interpretation of mapping tables implies that they may contain implicit information that is not explicitly listed in their extension. For instance, assume that $\mathcal{M}$ is a mapping table that consists of the pairs $\{(a, c), (b, c)\}$. Since a and c are equivalent, and the same is true about b and c, we can *infer* that a and b are equivalent. Such implicit information is, of course, valuable, and cannot be discarded at the moment of using the mapping table as a coordination tool. In particular, we will use this view of mapping tables as being incomplete with respect to its implicit data when defining the semantics of DSE settings.

3 Data Sharing and Exchange Settings

We formally define in this section DSE settings that extend DE settings to allow collaboration via mapping tables.

Definition 1 (DSE setting). *A* data sharing and exchange *(DSE) setting is a tuple $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$, where:* (1) $\mathbf{S}$ *and* $\mathbf{T}$ *are a source and a target schema, respectively;* (2) $\mathcal{M}$ *is a binary relation symbol that appears neither in* $\mathbf{S}$ *nor in* $\mathbf{T}$*, and that is called a* source-to-target mapping (*we call the first attribute of* $\mathcal{M}$ *the source attribute and the second one the target attribute*)*; and* (3) Σ_{st} *consists of a set of* mapping *st-tgds, which are* FO *sentences of the form*

$$\forall \bar{x} \forall \bar{y} \forall \bar{z}\, (\phi(\bar{x}, \bar{y}) \wedge \mu(\bar{x}, \bar{z}) \;\rightarrow\; \exists \bar{w}\, \psi(\bar{z}, \bar{w})),$$

where (*i*) $\phi(\bar{x}, \bar{y})$ *and* $\psi(\bar{z}, \bar{w})$ *are conjunctions of relational atoms over* $\mathbf{S}$ *and* $\mathbf{T}$*, resp.,* (*ii*) $\mu(\bar{x}, \bar{z})$ *is a conjunction of atomic formulas that only use the relation symbol* $\mathcal{M}$*,* (*iii*) $\bar{x}$ *is the tuple of variables that appear in* $\mu(\bar{x}, \bar{z})$ *in the positions of source attributes of* $\mathcal{M}$*, and* (*iv*) $\bar{z}$ *is the tuple of variables that appear in* $\mu(\bar{x}, \bar{z})$ *in the positions of target attributes of* $\mathcal{M}$*.*

We provide some terminology and notations before explaining the intuition behind the different components of a DSE setting. As before, instances of $\mathbf{S}$ (resp. $\mathbf{T}$) are called source (resp. target) instances, and we denote source instances by $I, I', I_1, \ldots$ and target instances by $J, J', J_1, \ldots$. Instances of $\mathcal{M}$ are called *source-to-target* mapping tables (st-mapping tables). By slightly abusing notation, we denote st-mapping tables also by $\mathcal{M}$.

Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a DSE setting. We distinguish between the set of source constants, denoted by $\mathsf{Const}^{\mathbf{S}}$, and the set of target constants, denoted by $\mathsf{Const}^{\mathbf{T}}$, since applications that collaborate on data usually have different data domains. As in the case of usual data exchange, we also assume the existence of a countably infinite set Var of labelled nulls (that is disjoint from both $\mathsf{Const}^{\mathbf{S}}$ and $\mathsf{Const}^{\mathbf{T}}$). Also, in a DSE the domain of a source instance I is contained in $\mathsf{Const}^{\mathbf{S}}$, while the domain of a target instance J belongs to $\mathsf{Const}^{\mathbf{T}} \cup \mathsf{Var}$. On the other hand, the domain of the st-mapping table $\mathcal{M}$ is a subset of $\mathsf{Const}^{\mathbf{S}} \times \mathsf{Const}^{\mathbf{T}}$. Thus, coordination between the source and the target at the data level occurs when $\mathcal{M}$ identifies which source and target constants denote the same object. The intuition behind st-tgds is that they specify how source data has to be transformed to conform to the target schema (that is, coordination at the schema level). However, since in the DSE scenario we are interested in transferring data based on the source instance as well as on the correspondence between source and target constants given by the st-mapping table that interprets $\mathcal{M}$, the mapping st-tgds extend usual st-tgds with a conjunction μ that filters the target data that is related via $\mathcal{M}$ with the corresponding source data.

More formally, given a source instance I and an st-mapping table $\mathcal{M}$, the mapping st-tgd $\phi(\bar{x}, \bar{y}) \wedge \mu(\bar{x}, \bar{z}) \rightarrow \exists \bar{w} \psi(\bar{z}, \bar{w})$ enforces the following: whenever $I \models \phi(\bar{a}, \bar{b})$, for a tuple $(\bar{a}, \bar{b})$ of constants in $\mathsf{Const}^{\mathbf{S}} \cap dom(I)$, and the tuple $\bar{c}$ of constants in $\mathsf{Const}^{\mathbf{T}}$ is related to $\bar{a}$ via μ (that is, $\mathcal{M} \models \mu(\bar{a}, \bar{c})$), then it must be the case that $J \models \psi(\bar{c}, \bar{d})$, for some tuple $\bar{d}$ of elements in $dom(J) \cap (\mathsf{Const}^{\mathbf{T}} \cup \mathsf{Var})$, where J is the materialized target instance. In usual terms, we should say that J is a solution for I and $\mathcal{M}$ under $\mathfrak{S}$, i.e $(((I \cup \{\mathcal{M}\}), J) \models \Sigma_{st})$. However, as we see in the next section, solutions have to be defined differently in DSE. Therefore, in order to avoid confusions, we say J is a *pre-solution* for I and $\mathcal{M}$ under $\mathfrak{S}$.

Example 3. (Example 2 cont.). Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a DSE setting. Suppose that $\mathbf{S}$ in $\mathfrak{S}$ is the schema of UOC and $\mathbf{T}$ in $\mathfrak{S}$ is the schema of UOO.

Suppose that $\mathcal{M}$ in $\mathfrak{S}$ consists of the following pairs $\{(ECOR1606, CSI1390)$, $(COMP1005, CSI1390)$, $(COMP1005, CSI1790)$, (CS, CS), $(ENG, ENG)\}$. Finally, let Σ_{st} consist of the following st-mapping dependencies: (a) $St(x, y, z) \wedge Take(x, w, u) \wedge Cr(w, v, \text{‘}CS\text{’}) \wedge \mathcal{M}(x, x') \wedge \mathcal{M}(y, y')$
$$\rightarrow Student(x', y').$$
(b) $St(x, y, z) \wedge Take(x, w, u) \wedge Cr(w, v, \text{‘}CS\text{’}) \wedge \mathcal{M}(x, x') \wedge \mathcal{M}(w, w') \wedge \mathcal{M}(u, u')$
$$\rightarrow Enroll(x', w', u').$$

It is clear that this DSE instance is exchanging information of UOC students that have taken ‘CS’ courses with the list of courses they have finished. Also, $\mathcal{M}$ specifies that the *Introduction to Computers* course with $Cid = CSI1390$ in UOO has a French version course *Introduction aux Ordinateurs* with $Cid = CSI1790$ provided at UOO. Let $I = \{St(Alex, 18, Ottawa)$, $Cr(ECOR1606$, *Problem Solving and Computers*, $ENG)$, $Cr(COMP1005$, *Introduction to Computer Science I*, $CS)$, $Take$ $(Alex$, $ECOR1606$, $80)\}$ be an instance of $\mathbf{S}$. Then, $J = \emptyset$ is a pre-solution for I and $\mathcal{M}$ under $\mathfrak{S}$. □

We can see in Example 3 that in the pre-solution J, *Alex* is not considered as have finished a ‘CS’ course. However, if the st-mapping table $\mathcal{M}$ follows the semantics we adopt in this paper, then *Alex* should be considered to have completed the ‘CS’ course *Introduction to Computer Science I.* Therefore, we can easily deduce that in a DSE setting $\mathfrak{S}$, we cannot identify solutions with pre-solutions. One reason is that a source instance I in $\mathfrak{S}$ can be *incomplete* with respect to the semantics of $\mathcal{M}$ as the case in Example 3 above. A second reason is that data mappings in an st-mapping table $\mathcal{M}$ in $\mathfrak{S}$ can also be *incomplete* with respect to the semantics of $\mathcal{M}$ as we shall show in Section 4. Data mappings in an st-mapping table $\mathcal{M}$ are usually specified by domain specialists. However, $\mathcal{M}$ should record not only the associations suggested by the domain specialists, but also the ones inferred by its semantics. Therefore, to capture the real semantics of the DSE problem, we came up with a more sophisticated notion of solution that we introduce in the following section.

4 DSE and Knowledge Exchange

From now on we use the equivalence relation $\sim$ as $a \sim b$ to intuitively denote that a and b, where $\{a, b\} \subseteq \mathsf{Const}^{\mathbf{S}}$ (or $\{a, b\} \subseteq \mathsf{Const}^{\mathbf{T}}$) are *inferred* by the semantics of an st-mapping table $\mathcal{M}$ as equivalent objects. Let us revisit Example 3. There are two ways in which the data in $\mathfrak{S}$ is incomplete: First of all, since $\mathcal{M}(ECOR1606, CSI1390)$ holds in $\mathfrak{S}$, then UOC course $ECOR1606$ is equivalent to the UOO course $CSI1390$. Also, since $\mathcal{M}(COMP1005, CSI1390)$ holds, then UOC course $COMP1005$ is equivalent to the UOO course $CSI1390$. Therefore, we can deduce that $ECOR1606 \sim COMP1005$ with respect to the target UOO. This means, according to semantics of $\sim$, the source instance I is incomplete, since I should include the tuple $Take(Alex$, $COMP1005$, 80) in order to be complete with respect to $\mathcal{M}$.

Second, since $\mathcal{M}(COMP1005, CSI1390)$ holds in $\mathfrak{S}$, then the UOC course $COMP1005$ is equivalent to the UOO course $CSI1390$ according to the semantics of $\mathcal{M}$. Also, since $\mathcal{M}(COMP1005$, $CSI1790)$ holds in $\mathfrak{S}$, then course $COMP1005$ is equivalent to the UOO course $CSI1790$. Therefore, we can deduce that $CSI1390 \sim CSI1790$, according to the semantics of $\mathcal{M}$. This implies that $\mathcal{M}$ is incomplete, since the fact that $\{(ECOR1606$, $CSI1390)$, $(COMP1005$, $CSI1390)$, $(COMP1005$, $CSI1790)\} \subseteq \mathcal{M}$ entails from the semantics of $\sim$ the fact that $(ECOR1606$, $CSI1790) \in \mathcal{M}$. Therefore, we say I and $\mathcal{M}$ are incomplete in the sense that they do not contain all the data that is implied by the semantics of $\mathcal{M}$. Further, it is not hard to see that the completion process we just sketched can become recursive in more complex DSE instances.

From what we explained so far, we conclude that the real semantics of a DSE setting is based on the explicit data contained in I and $\mathcal{M}$, in addition to the implicit data obtained by following a completion process for the source, the target, and $\mathcal{M}$. We define below a set of FO sentences, of type full tgds [5], over a schema $\mathbf{S} \cup \mathcal{M}$ $(\mathbf{T} \cup \mathcal{M})$ extended with a fresh binary relation symbol EQUAL that appears neither in $\mathbf{S}$ nor in $\mathbf{T}$ and that captures the semantics of $\sim$ in a recursive scenario, which formally defines this completion process:

Definition 2 (Source and Target completion). *Let* $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ *be a DSE setting. The* source completion *of* $\mathfrak{S}$*, denoted by* Σ_s^c*, is the conjunction of the following* FO *sentences over the schema* $\mathbf{S} \cup \{\mathcal{M}, \text{EQUAL}\}$*:*

[5] Full tgds are tgds that do not use existential quantication.

1. *For each* $S \in \mathbf{S} \cup \{\mathcal{M}\}$ *of arity* n *and* $1 \leq i \leq n$*:*
   $\forall x_1 \cdots \forall x_n (S(x_1, \ldots, x_i, \ldots, x_n) \rightarrow \textsc{Equal}(x_i, x_i))$.
2. $\forall x \forall y (\textsc{Equal}(y, x) \rightarrow \textsc{Equal}(x, y))$.
3. $\forall x \forall y \forall z (\textsc{Equal}(x, z) \wedge \textsc{Equal}(z, y) \rightarrow \textsc{Equal}(x, y))$.
4. $\forall x \forall y \forall z (\mathcal{M}(x, z) \wedge \mathcal{M}(y, z) \rightarrow \textsc{Equal}(x, y))$.
5. $\forall x \forall y \forall z \forall w (\mathcal{M}(x, z) \wedge \textsc{Equal}(x, y) \wedge \textsc{Equal}(z, w) \rightarrow \mathcal{M}(y, w))$.
6. *For each* $S \in \mathbf{S}$ *of arity* n*:*
   $\forall x_1, y_1 \cdots \forall x_n, y_n \, (S(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{n} \textsc{Equal}(x_i, y_i) \rightarrow S(y_1, \ldots, y_n))$.

The target completion *of* $\mathfrak{S}$*, denoted* Σ_t^{c}*, is defined analogously by simply replacing the role of* $\mathbf{S}$ *by* $\mathbf{T}$ *in* Σ_s^{c}*, and then adding the rule* 7. $\forall x \forall y \forall z (\mathcal{M}(z, x) \wedge \mathcal{M}(z, y) \rightarrow \textsc{Equal}(x, y))$ *that defines the completion of* $\mathcal{M}$ *over the target.*

Notice that the first 3 rules of Σ_s^{c} make sure that Equal is an equivalence relation on the domain of the source instance. The fourth rule detects which source elements have to be declared equal by the implicit knowledge contained in the st-mapping table. The last two rules allow to complete the interpretation of $\mathcal{M}$ and the symbols of $\mathbf{S}$, by adding elements declared to be equal in Equal. The intuition for Σ_t^{c} is analogous.

Summing up, data in a DSE scenario always consists of two modules: (1) The explicit data stored in the source instance I and the st-mapping table $\mathcal{M}$, and (2) the implicit data formalized in Σ_s^{c} and Σ_t^{c}. This naturally calls for a definition in terms of *knowledge exchange* [8], as defined next. A knowledge base (KB) over schema $\mathbf{R}$ is a pair (K, Σ), where K is an instance of $\mathbf{R}$ (the explicit data) and Σ is a set of logical sentences over $\mathbf{R}$ (the implicit data). The knowledge base representation has been used to represent various types of data including ontologies in the semantic web, which are expressed using different types of formalisms including *Description Logic* (DL) [16].

The set of *models* of (K, Σ) [8], denoted by $\mathsf{Mod}(K, \Sigma)$, is defined as the set of instances of $\mathbf{R}$ that contain the explicit data in K and satisfy the implicit data in Σ; that is, $\mathsf{Mod}(K, \Sigma)$ corresponds to the set $\{K' \mid K'$ is an instance of $\mathbf{R}$, $K \subseteq K'$ and $K' \models \Sigma$ $\}$. In DSE, we consider source KBs of the form $((I \cup \{\mathcal{M}\}), \Sigma_s^{\mathrm{c}})$, which intuitively correspond to completions of the source instance I with respect to the implicit data in $\mathcal{M}$, and, analogously, target KBs of the form $((J \cup \{\mathcal{M}\}), \Sigma_t^{\mathrm{c}})$.

A good bulk of work has recently tackled the problem of exchange of KBs that are defined using different DL languages [6, 7]. we formalize the notion of (universal) DSE solution to extend the KB (universal) solution introduced in [8]. The main difference is that in DSE solutions we need to coordinate the source and target information provided by $\mathcal{M}$, as opposed to KB solutions that require no data coordination at all. This is done by establishing precise relationships in a (universal) DSE solution between the interpretation of $\mathcal{M}$ in $\mathbf{S}$ and $\mathbf{T}$, respectively. KB exchange in DL showed that target KB (universal) solutions [8] present several limitations since these can miss some semantics of the source KB [6, 7]. Universal DSE solutions, on the other hand, do not possess those limitations and they reflect the semantics in the source KB and the st-mapping table accurately.

From now on, $K_{\mathbf{R}'}$ denotes the restriction of instance K to a subset $\mathbf{R}'$ of its schema $\mathbf{R}$. Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a DSE setting, I a source instance, $\mathcal{M}$ an st-mapping table, J a target instance. Recall that $\Sigma_s^{\mathrm{c}}, \Sigma_t^{\mathrm{c}}$ are the source and target completions of $\mathfrak{S}$, respectively. Then:

1. J is a *DSE solution* for I and $\mathcal{M}$ under $\mathfrak{S}$, if for every $K \in \mathsf{Mod}((J \cup \{\mathcal{M}\}), \Sigma_t^{\mathrm{c}})$ there is $K' \in \mathsf{Mod}((I \cup \{\mathcal{M}\}), \Sigma_s^{\mathrm{c}})$ such that the following hold: (a) $K'_{\mathcal{M}} \subseteq K_{\mathcal{M}}$, and (b) $K_{\mathbf{T}}$ is a pre-solution for $K'_{\mathbf{S}}$ and $K'_{\mathcal{M}}$ under $\mathfrak{S}$.
2. In addition, J is a *universal* DSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$, if J is a DSE solution, and for every $K' \in \mathsf{Mod}((I \cup \{\mathcal{M}\}, \Sigma_s^{\mathrm{c}})$ there is $K \in \mathsf{Mod}((J \cup \{\mathcal{M}\}, \Sigma_t^{\mathrm{c}})$ such that (a) $K_{\mathcal{M}} \subseteq K'_{\mathcal{M}}$, and (b) $K_{\mathbf{T}}$ is a pre-solution for $K'_{\mathbf{S}}$ and $K'_{\mathcal{M}}$ under $\mathfrak{S}$.

In Example 3, $J = \{Student(Alex, 18)$, $Enroll\ (Alex, CSI1390, 80)$, $Enroll\ (Alex, CSI1790, 80)\}$ is a universal DSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$. We define below a simple procedure $\texttt{CompUnivDSESol}_{\mathfrak{S}}$ that, given a DSE setting $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ and a source instance I and an st-mapping table $\mathcal{M}$, it generates a universal DSE solution J for I and $\mathcal{M}$ under $\mathfrak{S}$.
$\texttt{CompUnivDSESol}_{\mathfrak{S}}$:
Input: A source instance I, an st-mapping table $\mathcal{M}$, and a set Σ_{st} of st-tgds.
Output: A Canonical Universal DSE solution J for I and $\mathcal{M}$ under $\mathfrak{S}$.

1. Apply the source completion process, Σ_s^{c}, to I and $\mathcal{M}$, and generate $\hat{I}$ and $\hat{\mathcal{M}}$ respectively.
2. Apply a procedure (based on the *chase* [9]) to the instance $(\hat{I} \cup \{\hat{\mathcal{M}}\})$, and generate a canonical universal pre-solution J for $\hat{I}$ and $\hat{\mathcal{M}}$.

The procedure $\texttt{CompUnivDSESol}_{\mathfrak{S}}$ works as follows: step 1 applies the source completion process Σ_s^{c}, given in Definition 2, to I and $\mathcal{M}$, and returns as outcome the source instance $\hat{I}$ and the st-mapping table $\hat{\mathcal{M}}$ that are complete with respect to the implicit data in $\mathcal{M}$. Next, step 2 generates a canonical universal pre-solution J for $\hat{I}$ and $\hat{\mathcal{M}}$ such that $((\hat{I} \cup \hat{\mathcal{M}}), J) \models \Sigma_{st}$.

We can combine the fact that universal solutions in fixed data exchange settings $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \Sigma_{st})$ can be computed in LOGSPACE with some deep results in the computation of symmetrical binary relations [14], to show that universal DSE solutions can be computed in LOGSPACE:

Proposition 1. *Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a fixed DSE setting. Then computing a universal DSE solution J for a source instance I and an st-mapping table $\mathcal{M}$ is in* LOGSPACE.

5 Minimal Universal DSE Solutions

In the context of ordinary data exchange, "best" solutions – called cores – are universal solutions with minimal size. In knowledge exchange, on the other hand, "best" solutions are cores that materialize a minimal amount of explicit data. Intuitively, a *minimal* universal DSE (MUDSE) solution is a core universal DSE solution J that contains a minimal amount of explicit data in J with respect to Σ_t^c, and such that no universal DSE solution with strictly less constants is also a universal DSE solution with respect to Σ_t^c.

We define this formally: Let $\mathfrak{S}$ be a DSE setting, I be a source instance, $\mathcal{M}$ an st-mapping table, and J a universal DSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$. Then J is a MUDSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$, if: (1) There is no proper subset J' of J such that J' is a universal DSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$, and; (2) There is no universal DSE solution J' such that $dom(J') \cap \mathsf{Const}^{\mathbf{T}}$ is properly contained in $dom(J) \cap \mathsf{Const}^{T}$.

So, in Example 3, $J = \{Student(Alex, 18),\ Enroll\ (Alex,\ CSI1390,\ 80)\}$ is a MUDSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$. Note that the DSE solution $J' = \{Student(Alex, 18),\ Enroll\ (Alex,\ CSI1390,\ 80), Enroll\ (Alex,\ CSI1790,\ 80)\}$ is a core universal DSE solution, however it is not the most compact one. Condition (2) in the definition of MUDSE solutions is not part of the original definition of minimal solutions in knowledge exchange [8]. However, this condition is necessary as we see below.

Assume that the universal DSE solution in Example 3 includes the following two facts $\{Teach(Anna, CSI1390), Teach(Anna, CSI1790)\}$, where $\mathbf{T}$ is extended with the relation $Teach(Tid, Cid)$ which specifies the teachers and the list of courses they teach. Then, the DSE solution $J = \{Student(Alex, 18),\ Enroll\ (Alex,\ CSI1390,\ 80),\ Teach(Anna, CSI1790)\}$ does not satisfy condition (2) and provides us with redundant information with respect to I and $\mathcal{M}$, since we can conclude that $CSI1390$ and $CSI1790$ are equivalent courses. A MUDSE solution however would be $J = \{Student(Alex, 18),\ Enroll(Alex,\ CSI1390,\ 80),\ Teach(Anna, CSI1390)\}$.

We define below a procedure $\texttt{CompMUDSEsol}_{\mathfrak{S}}$, that given a DSE setting $\mathfrak{S}$, a source instance I, and an st-mapping table $\mathcal{M}$, it computes a MUDSE solution J^* for I and $\mathcal{M}$ under $\mathfrak{S}$ in LOGSPACE. This procedure works as follows:

$\texttt{CompMUDSEsol}_{\mathfrak{S}}$:
Input: A source instance I, an st-mapping table $\mathcal{M}$, and a set Σ_{st} of st-tgds.
Output: A Minimal Universal DSE solution J^* for I and $\mathcal{M}$ under $\mathfrak{S}$.

1. Apply the source completion process, Σ_s^c, to I and $\mathcal{M}$, and generate $\hat{I}$ and $\hat{\mathcal{M}}$ respectively.
2. Define an equivalence relation $\sim$ on $dom(\hat{\mathcal{M}}) \cap \mathsf{Const}^{\mathbf{T}}$ as follows: $c_1 \sim c_2$ iff there exists a source constant a such that $\hat{\mathcal{M}}(a, c_1)$ and $\hat{\mathcal{M}}(a, c_2)$ hold.
3. Compute equivalence classes $\{C_1,\ \ldots,\ C_m\}$ for $\sim$ over $dom(\hat{\mathcal{M}}) \cap \mathsf{Const}^{\mathbf{T}}$ such that c_1 and c_2 exist in C_i only if $c_1 \sim c_2$.
4. Choose a set of witnesses $\{w_1, \ldots, w_m\}$ where $w_i \in C_i$, for each $1 \leq i \leq m$.
5. Compute from $\hat{\mathcal{M}}$ the instance $\mathcal{M}_1 := \texttt{replace}(\hat{\mathcal{M}}, w_1, \ldots, w_m)$ by replacing each target constant $c \in C_i \cap dom(\hat{\mathcal{M}})$ $(1 \leq i \leq m)$ with $w_i \in C_i$.
6. Apply a procedure (based on the *chase* [9]) to the instance $(\hat{I} \cup \{\mathcal{M}_1\})$, and generate a canonical universal pre-solution J for $\hat{I}$ and $\mathcal{M}_1$.
7. Apply a procedure (based on the core [11]) to the target instance J and generate the target instance J^* that is the core of J.

We prove the correctness of $\texttt{CompMUDSEsol}_{\mathfrak{S}}$ in the following Theorem.

Theorem 1. *Let $\mathfrak{S}$ be a DSE setting, I a source instance, and $\mathcal{M}$ an st-mapping table. Suppose that J^* is an arbitrary result for* $\texttt{CompMUDSEsol}_{\mathfrak{S}}(I, \mathcal{M})$*. Then, J^* is a minimal universal DSE solution for I and $\mathcal{M}$ under $\mathfrak{S}$.*

In data exchange, the smallest universal solutions are known as *cores* and can be computed in LOGSPACE [11]. With the help of such result we can prove that MUDSE solutions can be computed in LOGSPACE too. Also, in this context MUDSE solutions are unique up to isomorphism:

Theorem 2. *Let $\mathfrak{S}$ be a fixed DSE setting. There is a* LOGSPACE *procedure that computes, for a source instance I and an st-mapping table $\mathcal{M}$, a MUDSE solution J for I and $\mathcal{M}$ under $\mathfrak{S}$. Also, for any two MUDSE solutions J_1 and J_2 for I and $\mathcal{M}$ under $\mathfrak{S}$, it is the case that J_1 and J_2 are isomorphic.*

6 Query Answering

In data exchange, one is typically interested in the *certain answers* of a query Q, that is, the answers of Q that hold in each possible solution [10]. For the case of DSE we need to adapt this definition to solutions that are knowledge bases. Formally, let $\mathfrak{S}$ be a DSE setting, I a source instance, $\mathcal{M}$ an st-mapping table, and Q a FO conjunctive query over $\mathbf{T}$. The set of certain answers of Q over I and $\mathcal{M}$ and under $\mathfrak{S}$, denoted $\mathsf{certain}_{\mathfrak{S}}((I \cup \{\mathcal{M}\}), Q)$, corresponds to the set of tuples that belong to the evaluation of Q over $K_{\mathbf{T}}$, for each DSE solution J for I and $\mathcal{M}$ and $K \in \mathsf{Mod}((J \cup \{\mathcal{M}\}), \Sigma_t^c)$.

Example 4. We refer to the DSE setting given in Example 3. Let $Q(x, y, z) = Enroll(x, y, z)$. Then, $\mathsf{certain}_{\mathfrak{S}}((I \cup \{\mathcal{M}\}), Q) = \{Enroll(Alex,\ CSI1390,\ 80),\ Enroll(Alex,\ CSI1790,\ 80)\}$. □

In DE, certain answers of unions of CQs can be evaluated in LOGSPACE by directly posing them over a universal solution [12, 10], and then discarding tuples with null values. The same complexity bound holds in DSE by applying a slightly different algorithm. In fact, certain answers cannot be simply obtained by posing Q on a universal DSE solution J, since J might be incomplete with respect to the implicit data in Σ_t^c.

One possible solution would be to apply the target completion program Σ_t^c to a universal DSE solution J (denoted as $\Sigma_t^c(J)$) as a first step, then apply Q to $\Sigma_t^c(J)$. A second method is to compute certain answers of Q using a MUDSE solution. A MUDSE solution J in DSE possesses an interesting property, that is, applying Q to J returns a set of certain answers U that minimally represents the set of certain answers U' returned when Q is applied to $\Sigma_t^c(J)$. We can compute $\mathsf{certain}_{\mathfrak{S}}((I \cup \{\mathcal{M}\}), Q)$ directly using J, by first applying rules in Σ_t^c, excluding rule 6, to generate the binary table EQUAL. Then complete the evaluation of Q on J, $Q(x_1, \ldots, x_n)$, and return $\hat{Q}(y_1, \ldots, y_n) = Q(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{n} \text{EQUAL}(x_i, y_i)$. Thus, we obtain the following result:

Proposition 2. *Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a fixed DSE setting, I a source instance, $\mathcal{M}$ an st-mapping table, J a MUDSE solution, and Q a fixed CQ over $\mathbf{T}$. Then,* $\mathsf{certain}_{\mathfrak{S}}((I \cup \{\mathcal{M}\}), Q) = \hat{Q}(J)$ *where* $\hat{Q}(y_1, \ldots, y_n) = Q(x_1, \ldots, x_n) \wedge \bigwedge_{i=1}^{n} \textsc{Equal}(x_i, y_i)$

In addition, we prove in the following proposition that we can still compute the set of certain answers of a conjunctive query Q in Logspace.

Proposition 3. *Let $\mathfrak{S} = (\mathbf{S}, \mathbf{T}, \mathcal{M}, \Sigma_{st})$ be a fixed DSE setting and Q a fixed union of CQs. There is a* Logspace *procedure that computes* $\mathsf{certain}_{\mathfrak{S}}((I \cup \{\mathcal{M}\}), Q)$*, given a source instance I and an st-mapping table $\mathcal{M}$.*

7 Experiments

We implement the knowledge exchange semantics we introduced in this paper in a DSE prototype system. This system effectively generates universal DSE and MUDSE solutions that can be used to compute certain answers for CQs using the two methods introduced in Section 6. We used the DSE scenario of Example 3 extended with the st-tgd: $Cr(x, y, z) \wedge \mathcal{M}(x, x') \wedge \mathcal{M}(y, y') \wedge \mathcal{M}(z, z') \rightarrow Course(x', y', z')$. Due to the lack of a benchmark that enforces recursion of the $\sim$ equivalence relation in the st-mapping table $\mathcal{M}$ and due to size restrictions, we synthesized the data in our experiments.

We show in our experiments that as the percentage of recursion increases in an st-mapping table, the run time to generate a universal DSE solution exceeds the time to generate a MUDSE solution. We also show that computing certain answers using a MUDSE solution is more effective than using a universal DSE solution. The experiments were conducted on a Lenovo workstation with a Dual-Core Intel(R) 1.80GHz processor running Windows 7, and equipped with 4GB of memory and a 297 GB hard disk. We used Python (v2.7) to write the code and PostgreSQL (v9.2) database system.

DSE and MUDSE Solutions Computing Times We used in this experiment a source instance I of 4,500 tuples, and 500 of those were courses information. The DSE system leveraged the work done in the state of the art ++Spicy system [17] to generate MUDSE solutions. We mapped courses data in the source to common target courses in $\mathcal{M}$, with different $\sim$ equivalence percentages (to enforce a recursive $\sim$ relation). The remaining set of source data was mapped to itself in $\mathcal{M}$. Figure 1 shows that as the percentage of recursion in $\sim$ equivalence relation over $\mathcal{M}$ increases, the run times to generate universal DSE and MUDSE solutions increase. The reason is, as the $\sim$ percentage increases, the number of source values (and target values) inferred to be $\sim$ increases, and thus the size of Equal created in Σ_s^c and Σ_t^c increases. Also, since target instances are usually larger than $\mathcal{M}$, the run time of completing the former to generate DSE solutions exceeds the time of completing the later when generating MUDSE solutions.
Conjunctive Queries Computing Times We have selected a set of 8 queries to compare the performance of computing certain answers using a universal

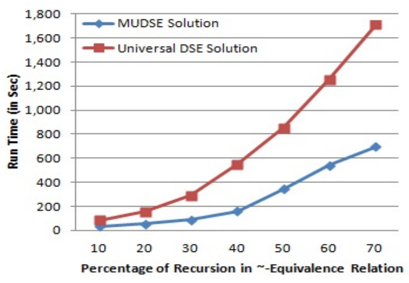


Fig. 1. MUDSE and Universal DSE solutions Generation Times

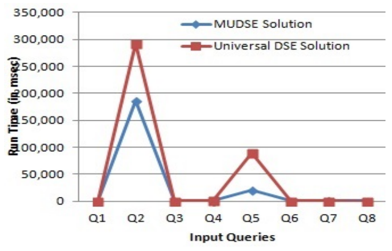


Fig. 2. Queries run times against a Core of a universal DSE solution and a MUDSE solution

DSE solution (following the first method in Section 6) versus a MUDSE solution (following the second method in Section 6). We provide the list of queries in Table 1.

Table 1. List of Queries

$Q1$	*Fetch all the students names and the name of courses they have taken*
$Q2$	*Fetch the list of pairs of students ids and names that took the same course*
$Q3$	*Fetch all the students names and the grades they have received*
$Q4$	*Fetch the list of pairs of courses names that belong to the same program*
$Q5$	*Fetch for each student id the pair of courses that he has finished with the same grade*
$Q6$	*Fetch all the courses ids and their names*
$Q7$	*Fetch all the students ids and their names*
$Q8$	*Fetch the list of pairs of students ids that possess the same address*

We applied the list of input queries to a DSE instance where the $\sim$ percentage is 40% and a course in the source is mapped to a maximum of two courses in the target. We chose a universal DSE solution, with a property of being a core of itself, that had around 18,000 records, and a MUDSE solution that contained around 4,900 records. Figure 2 shows that computing the sets of certain answers for the input conjunctive queries using a MUDSE solution take less run times than when computing these using a DSE solution. In addition, the deterioration in performance of query execution against the DSE solution appeared more in queries $Q2$ and $Q5$ than the remaining queries, is because both queries apply join operations to the *Enroll* table that involves a lot of elements which are inferred to be *equivalent* by $\mathcal{M}$.

8 Concluding Remarks

We introduced a Data Sharing and Exchange setting which exchanges data between two applications that have distinct schemas and distinct yet related sets of vocabularies. To capture the semantics of this setting, we defined DSE as a knowledge exchange system with a set of source and target rules that infer the implicit data should be in the target. We formally defined DSE solutions and identified the minimal among those. Also, we studied certain answers for CQs. Finally, we presented a prototype DSE system that generates universal DSE solutions and minimal ones, and it computes certain answers of CQs from both. Our implementation did not aim at optimality in performance, but was intended to be a valuable direction of future work. In future work, we will investigate a more general DSE setting were mapped elements are not necessarily equal.

References

1. M. Lawrence, R. Pottinger, and S. Staub-French: Data coordination: supporting contingent updates. In *Proc. of the VLDB Endowment*, 2011.
2. P. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serani, and I. Zaihrayeu: Data Management for Peer-to-Peer Computing: A Vision. In *ACM SIGMOD WebDB Workshop 2002*, 2002.
3. V. Savenkov, and R. Pichler: Towards practical feasibility of core computation in data exchange. In *Proc. of LPAR*, pp. 62–78, 2008.
4. M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos: The hyperion project: from data integration to data coordination. In *SIGMOD Record*, pp. 53–58, 2003.
5. M. Arenas, P. Barceló, L. Libkin, and F. Murlak: Relational and XML data exchange. *Morgan & Claypool Publishers*, 2010.
6. M. Arenas, E. Botoeva, and D. Calvanese: Knowledge base exchange. In *Proc. of DL*, 2011.
7. M. Arenas, E. Botoeva, D. Calvanese, V. Ryzhikov, and E. Sherkhonov: Exchanging description logic knowledge bases. In *Proc. of KR*, 2012.
8. M. Arenas, J. Perez, and J. L. Reutter: Data exchange beyond complete data. In *Proc. of PODS*, pp. 83–94, 2011.
9. C. Beeri, and M. Y. Vardi: A proof procedure for data dependencies. In *Journal of the ACM*, pp.718–741, 1984.
10. R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa: Data exchange: semantics and query answering. In *Proc. of ICDT*, pp. 207–224, 2003.
11. R. Fagin, P. G. Kolaitis, and L. Popa: Data exchange: getting to the core. In *Proc. of PODS*, pp. 90–101, 2003.
12. T. Imielinski, and W. Lipski: Incomplete information in relational databases. In *Journal of the ACM*, pp. 761–791, 1984.
13. A. Kementsietsidis, M. Arenas, and R. J. Miller: Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proc. of SIGMOD*, pp. 325–336, 2003.
14. O. Reinghold: Undirected connectivity in log-space. In *Journal of ACM*, 2008.
15. M. Arenas, J. Reutter, and P. Barceló: Query Languages for Data exchange: beyond unions of conjunctive queries. In *Proc. of ICDT*,pp. 73–83, 2009.
16. A. Borgida: On the relative expressiveness of description logics and predicate logics. In *Artificial Intelligence*, pp. 353–367, 1996.
17. B. Marnette, G. Mecca, and P. Papotti: ++Spicy: an Open-source tool for second-generation schema mapping and data exchange. In *VLDB*, 2011.

On Axiomatic Rejection for the Description Logic $\mathcal{ALC}$

Gerald Berger and Hans Tompits

Institut für Informationssysteme 184/3,
Technische Universität Wien,
Favoritenstraße 9-11, A-1040 Vienna, Austria
{berger,tompits}@kr.tuwien.ac.at

Abstract. Traditional proof calculi are mainly studied for formalising the notion of *valid inference*, i.e., they axiomatise the valid sentences of a logic. In contrast, the notion of *invalid inference* received less attention. Logical calculi which axiomatise invalid sentences are commonly referred to as *complementary calculi* or *rejection systems*. Such calculi provide a proof-theoretic account for deriving non-theorems from other non-theorems and are applied, in particular, for specifying proof systems for nonmonotonic logics. In this paper, we present a sound and complete sequent-type rejection system which axiomatises *concept non-subsumption* for the description logic $\mathcal{ALC}$. Description logics are well-known knowledge-representation languages formalising ontological reasoning and provide the logical underpinning for semantic-web reasoning. We also discuss the relation of our calculus to a well-known tableau procedure for $\mathcal{ALC}$. Although usually tableau calculi are syntactic variants of standard sequent-type systems, for $\mathcal{ALC}$ it turns out that tableaux are rather syntactic counterparts of complementary sequent-type systems. As a consequence, counter models for witnessing concept non-subsumption can easily be obtained from a rejection proof. Finally, by the well-known relationship between $\mathcal{ALC}$ and multi-modal logic $\mathbf{K}$, we also obtain a complementary sequent-type system for the latter logic, generalising a similar calculus for standard $\mathbf{K}$ as introduced by Goranko.

1 Introduction and Overview

Research on proof theory is usually guided by the semantic concept of *validity*, finding appropriate (i.e., sound and complete) proof calculi for various types of logics. This is reasonable insofar as logical methods have been devised since their very beginning for characterising the valid sentences by virtue of their form rather than their semantic denotations. However, the complementary notion of validity, that is, *invalidity*, has rarely been studied by syntactic means. From a proof-theoretic point of view, the invalidity of sentences is largely established by the exhaustive search for counter models.

Proof systems which axiomatise the invalid sentences of a logic are commonly coined under the terms *complementary calculi* or *rejection systems*. Such calculi formalise *proofs for invalidity*, i.e., with the existence of a *sound* and *complete* rejection system of a logic under consideration, one is able to check for the invalidity of a sentence by syntactic deduction. Another way to characterise this notion is that a proof in such a complementary calculus witnesses the non-existence of a proof in a corresponding (sound and complete) positive proof system.

To the best of our knowledge, a first systematic theory of rejection was established by Jan Łukasiewicz in the course of research on the Aristotelian syllogistic [1]. Indeed, even the forefather of modern logic, Aristotle, recognised that showing the invalidity of a syllogistic form is not only possible by providing a counterexample, but also by employing some form of axiomatic reasoning. This notion was put into formal, axiomatic terms by Łukasiewicz.

Up to now, rejection systems have been studied for different families of logics including classical logic [2, 3], intuitionistic logic [4, 5], modal logics [6, 7], and many-valued logics [8] (for an overview, cf., e.g., Wybraniec-Skardowska [9] and Caferra and Peltier [10]). Many of them are analytic sequent-type systems, which proved fruitful in axiomatising invalidity without explicitly referring to validity. In contrast, the fundamental rule of rejection in the system of Łukasiewicz makes reference to a positive proof system too.

Besides a general proof-theoretic interest in such calculi, they received also attention in research on proof theory for nonmonotonic logics. In particular, Bonatti and Olivetti [11] employed complementary sequent-type systems when devising proof systems for default logic [12], autoepistemic logic [13], and propositional circumscription [14]. Furthermore, in logics in which the validity of a formula A is tantamount to checking unsatisfiability of the negation of A, a complementary calculus provides a proof-theoretic account of satisfiability checking.

In this paper, we deal with in the issue of complementary calculi in the context of description logics. More specifically, we consider the description logic $\mathcal{ALC}$ and present a sound and complete sequent-type rejection system for axiomatising *concept non-subsumption* for this logic. Note that, informally speaking, $\mathcal{ALC}$ is the least expressive of the so-called *expressive* description logics (for more details on the vast topic of description logics, we refer the reader to the overview article by Baader *et al.* [15]).

Concerning previous work on sequent-type calculi for description logics, we mention an axiomatisation of concept subsumption for different description logics, including $\mathcal{ALC}$, by Rademaker [16] and an earlier calculus for $\mathcal{ALC}$ by Borgida *et al.* [17].

As pointed out above, in our approach, we study an axiomatisation of concept *non-subsumption* for $\mathcal{ALC}$. We view this as a starting point for further investigations into complementary calculi for description logics as the more general case of dealing with reasoning from *knowledge bases*, which are usually the principal structures of description logics where reasoning operates on, would be a natural next step. In fact, our calculus is devised to axiomatise the invalidity of single *general concept inclusions* (GCIs) without reference to any knowledge base.

We also discuss the relation of our calculus to a well-known tableau procedure for $\mathcal{ALC}$ [18]. In general, as well known, sequent-type systems and tableau calculi are closely related—indeed, traditionally, tableau calculi are merely syntactic variants of standard sequent-type systems. However, popular tableau algorithms for description logics are formalised in order to axiomatise satisfiability rather than validity (cf. Baader and Sattler [18] for an overview). Hence, tableaux correspond in the case of description logics to *complementary* sequent systems. As a consequence, counter models for witnessing concept non-subsumption can easily be obtained from a rejection proof. We describe the relation of our calculus to the tableau algorithm for $\mathcal{ALC}$ as described by

Baader and Sattler [18] in detail, and show how to construct a counter model from a proof in the complementary sequent system.

Finally, as also well-known, $\mathcal{ALC}$ can be translated into the multi-modal logic $\mathbf{K}_m$, which extends standard modal logic $\mathbf{K}$ by providing countably infinite modal operators of form $[\alpha]$, where α is a modality. In view of this correspondence, we obtain from our complementary calculus for $\mathcal{ALC}$ also a complementary sequent-type calculus for $\mathbf{K}_m$. This calculus generalises a similar one for modal logic $\mathbf{K}$ as introduced by Goranko [6]. In fact, Goranko's calculus served as a starting point for the development of our calculus for $\mathcal{ALC}$. We briefly discuss the complementary calculus for $\mathbf{K}_m$, thereby showing the relation of our calculus for $\mathcal{ALC}$ to Goranko's one for $\mathbf{K}$.

2 Notation and Basic Concepts

With respect to terminology and notation, we mainly follow Baader *et al.* [15].

Syntactically, $\mathcal{ALC}$ is formulated over countably infinite sets N_C, N_R, and N_O of *concept names*, *role names*, and *individual names*, respectively. The syntactic artefacts of $\mathcal{ALC}$ are *concepts*, which are inductively defined using the *concept constructors* $\sqcap$ ("concept intersection"), $\sqcup$ ("concept union"), $\neg$ ("concept negation"), $\forall$ ("value restriction"), $\exists$ ("existential restriction"), as well as the concepts $\top$ and $\bot$ as usual. For the sake of brevity we agree upon omitting parentheses whenever possible and assign $\neg$, $\forall$, and $\exists$ the highest rank, and $\sqcap$ and $\sqcup$ the least binding priority. We use $C, D, \ldots$ as metavariables for concepts and $p, q, r, \ldots$ as metavariables for role names. When we consider concrete examples, we assume different metavariables to stand for distinct syntactic objects.

By an *interpretation* we understand an ordered pair $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$, where $\Delta^{\mathcal{I}}$ is a non-empty set called *domain* and $\cdot^{\mathcal{I}}$ is a function assigning each concept name $C \in \mathsf{N}_C$ a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each role name $r \in \mathsf{N}_R$ a set $r^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual name $a \in \mathsf{N}_O$ an element $a^{\mathcal{I}} \in \Delta^{\mathcal{I}}$. The function $\cdot^{\mathcal{I}}$ is furthermore required to obey the semantics of the concept constructors in the usual way. For a concept C and an interpretation $\mathcal{I}$, $C^{\mathcal{I}}$ is the *extension* of C under $\mathcal{I}$. A concept C is *satisfiable* if there exists an interpretation $\mathcal{I}$ such that $C^{\mathcal{I}} \neq \emptyset$, and *unsatisfiable* otherwise.

A *general concept inclusion* (GCI) is an expression of form $C \sqsubseteq D$, where C and D are arbitrary concepts. An interpretation $\mathcal{I}$ *satisfies* a GCI iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and *falsifies* it otherwise. In the former case, $\mathcal{I}$ is a *model* of $C \sqsubseteq D$, while in the latter case, $\mathcal{I}$ is a *counter model* of $C \sqsubseteq D$. The GCI $C \sqsubseteq D$ is *valid* if every interpretation satisfies it. In this case, we say that D *subsumes* C.

When considering proof systems, it is convenient and necessary to focus on interpretations of a special form when deciding semantic properties of language constructs. A *tree-shaped interpretation* is an interpretation $\mathcal{I}$ such that the set $\delta(\mathcal{I}) := \{(v, w) \mid (v, w) \in \bigcup_{r \in \mathsf{N}_R} r^{\mathcal{I}}\}$ of ordered tuples forms a tree. If the tree is finite, then $\mathcal{I}$ is a *finite* tree-shaped interpretation. The *root* of a tree-shaped interpretation $\mathcal{I}$ is defined to be the root of $\delta(\mathcal{I})$ and the *length* of $\mathcal{I}$ is the length of the longest path in $\delta(\mathcal{I})$. The following property of $\mathcal{ALC}$ is well-known [18].

Proposition 1. *A concept C is satisfiable iff there exists a finite tree-shaped interpretation $\mathcal{T}$ such that $v_0 \in C^{\mathcal{T}}$, where v_0 is the root of $\mathcal{T}$.*

When we say that a tree-shaped interpretation $\mathcal{I}$ *satisfies* a concept, then we mean that the root of $\mathcal{I}$ is in the extension of the concept. Let $\mathcal{T}$ and $\mathcal{T}'$ be tree-shaped interpretations with roots v and v', respectively. Then, $\mathcal{T}'$ is an *r-subtree* of $\mathcal{T}$ (symbolically, $\mathcal{T}' \lhd_r \mathcal{T}$) if $(v, v') \in r^{\mathcal{T}}$ and $\delta(\mathcal{T}')$ is a subtree of $\delta(\mathcal{T})$ in the usual sense. $\mathcal{T}'$ is a *subtree* of $\mathcal{T}$ if there exists an $r \in \mathsf{N}_R$ such that $\mathcal{T}'$ is an r-subtree of $\mathcal{T}$.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be tree-shaped interpretations such that for $i \neq j$ it holds that $\delta(\mathcal{T}_i) \cap \delta(\mathcal{T}_j) = \emptyset$ $(1 \leq i, j \leq n)$. Then, $\mathcal{T} = \langle v_0; r_1, \mathcal{T}_1; \ldots; r_n, \mathcal{T}_n \rangle$ expresses the fact that $\mathcal{T}$ is a tree-shaped interpretation with root v_0 and, for every $i = 1, \ldots, n$, the interpretation $\mathcal{T}_i$ is an r_i-subtree of $\mathcal{T}$. Furthermore, $\mathcal{T}_1, \ldots, \mathcal{T}_n$ are the only subtrees of $\mathcal{T}$.

3 A Rejection Calculus for $\mathcal{ALC}$

We now proceed defining our rejection calculus, which we denote by $\mathbf{SC}^c_{\mathcal{ALC}}$. The calculus will be devised to refute GCIs of form $C \sqsubseteq D$, where C and D are arbitrary concepts. Thereby, we define new syntactic artefacts, viz. *anti-sequents*.

Definition 2. *An* anti-sequent *is an ordered pair of form $\Gamma \dashv \Delta$, where Γ and Δ are finite multi-sets of concepts. Γ is the* antecedent *and Δ is the* succedent *of $\Gamma \dashv \Delta$. An anti-sequent $\Gamma \dashv \Delta$ is* propositional *if neither a concept of form $\forall r.C$ nor a concept of form $\exists r.C$ occurs as subconcept in any $D \in \Gamma \cup \Delta$.*

As usual, given a concept C or a set Σ of concepts, "$\Gamma, C \dashv \Delta$" denotes "$\Gamma \cup \{C\} \dashv \Delta$", and "$\Gamma, \Sigma \dashv \Delta$" denotes "$\Gamma \cup \Sigma \dashv \Delta$". Moreover, "$\dashv \Delta$" stands for "$\emptyset \dashv \Delta$" and "$\Gamma \dashv$" means "$\Gamma \dashv \emptyset$".

A *proof* in $\mathbf{SC}^c_{\mathcal{ALC}}$ is defined as usual in sequential systems. Furthermore, we will use terms which are common in sequent-type systems, like *end-sequent*, etc., without defining them explicitly (we refer the reader to Takeuti [19] for respective formal definitions of such concepts).

Definition 3. *An interpretation $\mathcal{I}$* refutes *an anti-sequent $\Gamma \dashv \Delta$ if $\mathcal{I}$ is a counter model of the GCI $\bigsqcap_{\gamma \in \Gamma} \gamma \sqsubseteq \bigsqcup_{\delta \in \Delta} \delta$, where the empty concept intersection is defined to be $\top$ and the empty concept union is defined to be $\bot$. If there is an interpretation which refutes $\Gamma \dashv \Delta$, then we say that $\Gamma \dashv \Delta$ is* refutable. *Furthermore, $\iota(\Gamma \dashv \Delta)$ stands for $\bigsqcap_{\gamma \in \Gamma} \gamma \sqcap \neg(\bigsqcup_{\delta \in \Delta} \delta)$.*

In the following, we denote finite multi-sets of concepts by capital Greek letters $\Gamma, \Delta, \ldots$, while capital Latin letters $C, D, \ldots$ denote concepts.

It is easy to see that the problem of deciding whether an anti-sequent is refutable can be reduced to the problem of deciding whether a concept is satisfiable.

Theorem 4. *The anti-sequent $s = \Gamma \dashv \Delta$ is refutable iff $\iota(s)$ is satisfiable.*

An immediate consequence of this observation and Proposition 1 is that an anti-sequent is refutable iff it is refuted by some finite tree-shaped interpretation. Note also that a concept C is satisfiable iff the anti-sequent $C \dashv$ is refutable. Furthermore, a concept D does *not* subsume a concept C iff the anti-sequent $C \dashv D$ is refutable.

Now we turn to the postulates of $\mathbf{SC}^c_{\mathcal{ALC}}$. Roughly speaking, the axioms and rules of $\mathbf{SC}^c_{\mathcal{ALC}}$ are generalisations of a sequential rejection system for modal logic **K** [6], by exploiting the well-known property that $\mathcal{ALC}$ is a syntactic variant of the multi-modal version of **K** [20] and by incorporating multiple modalities into Goranko's system. We discuss the relationship to Goranko's system, in terms of a multi-modal generalisation of his system, in Section 5. Besides that, the rules for the propositional connectives $\sqcap$, $\sqcup$, and $\neg$ correspond directly to those of the rejection system for propositional logic [6, 2, 3]. Note that these rules exhibit non-determinism as opposed to exhaustive search in standard proof systems.

Let us fix some notation. For a set Σ of concepts and a role name r, we define $\neg\Sigma := \{\neg C \mid C \in \Sigma\}$, $\forall r.\Sigma := \{\forall r.C \mid C \in \Sigma\}$, and $\exists r.\Sigma := \{\exists r.C \mid C \in \Sigma\}$. Moreover, for any role name r, Γ^r and Δ^r stand for multi-sets of concepts where every concept is either of form $\forall r.C$ or $\exists r.C$.

Definition 5. *The axioms of* $\mathbf{SC}^c_{\mathcal{ALC}}$ *are anti-sequents of form*

$$\Gamma_0 \dashv \Delta_0 \text{ and} \tag{1}$$

$$\forall r_1.\Gamma_1, \ldots, \forall r_n.\Gamma_n \dashv \exists r_1.\Delta_1, \ldots, \exists r_n.\Delta_n, \tag{2}$$

where Γ_0 and Δ_0 are disjoint multi-sets of concept names, $\Gamma_1, \ldots, \Gamma_n$ and $\Delta_1, \ldots, \Delta_n$ are multi-sets of concepts, and $r_1, \ldots, r_n$ are role names. Furthermore, the rules of $\mathbf{SC}^c_{\mathcal{ALC}}$ *are depicted in Figure 1, where $r_1, \ldots, r_n$ are assumed to be distinct role names.*

Note that each axiom of form (1) is a propositional anti-sequent and, accordingly, we refer to an axiom of such a form as a *propositional axiom*.

Intuitively, in order to derive an anti-sequent s, our calculus tries to build a model which satisfies $\iota(s)$. When we speak in terms of modal logic, the mix rules guarantee that the resulting model contains "enough" worlds to satisfy $\iota(s)$. For example, a world which satisfies the anti-sequent $\exists r.C \dashv \exists r.(C \sqcap D)$ has to be connected to another world which is contained in the extension of C, but not in the extension of $C \sqcap D$. This is exactly what is achieved by the rules $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$.

Example 6. A proof of $\exists r.C \sqcap \exists r.D \dashv \exists r.(C \sqcap D)$ in $\mathbf{SC}^c_{\mathcal{ALC}}$ is depicted by the following tree (with C and D being distinct concept names):

$$\dfrac{\dfrac{\dfrac{D \dashv C}{D \dashv C \sqcap D}\,(\sqcap, r)_1 \qquad \dfrac{\dfrac{C \dashv D}{C \dashv C \sqcap D}\,(\sqcap, r)_2 \quad \dashv \exists r.(C \sqcap D)}{\exists r.C \dashv \exists r.(C \sqcap D)}\,(\textsc{Mix}, \exists)}{\exists r.C, \exists r.D \dashv \exists r.(C \sqcap D)}\,(\textsc{Mix}, \exists)}{\exists r.C \sqcap \exists r.D \dashv \exists r.(C \sqcap D)}\,(\sqcap, l)$$

□

We informally describe how a counter model can be obtained from a rejection proof. For simplicity, we consider the case where each rule application of $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$ has $k = l = 1$. A tree-shaped counter model can be obtained from a proof by reading the proof from bottom to top and assigning each anti-sequent a node of the tree. Thereby, one starts by assigning the end-sequent of the proof the root node of the

LOGICAL RULES OF $\mathbf{SC}^c_{\mathcal{ALC}}$

$$\frac{\Gamma, C, D \dashv \Delta}{\Gamma, C \sqcap D \dashv \Delta}\ (\sqcap, l) \qquad \frac{\Gamma \dashv C, \Delta}{\Gamma \dashv C \sqcap D, \Delta}\ (\sqcap, r)_1 \qquad \frac{\Gamma \dashv D, \Delta}{\Gamma \dashv C \sqcap D, \Delta}\ (\sqcap, r)_2$$

$$\frac{\Gamma \dashv C, D, \Delta}{\Gamma \dashv C \sqcup D, \Delta}\ (\sqcup, r) \qquad \frac{\Gamma, C \dashv \Delta}{\Gamma, C \sqcup D \dashv \Delta}\ (\sqcup, l)_1 \qquad \frac{\Gamma, D \dashv \Delta}{\Gamma, C \sqcup D \dashv \Delta}\ (\sqcup, l)_2$$

$$\frac{\Gamma \dashv C, \Delta}{\Gamma, \neg C \dashv \Delta}\ (\neg, l) \qquad \frac{\Gamma, C \dashv \Delta}{\Gamma \dashv \neg C, \Delta}\ (\neg, r) \qquad \frac{\Gamma \dashv \Delta}{\Gamma, \top \dashv \Delta}\ (\top) \qquad \frac{\Gamma \dashv \Delta}{\Gamma \dashv \bot, \Delta}\ (\bot)$$

$$\frac{\Gamma_0 \dashv \Delta_0 \qquad \Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}}{\Gamma_0, \Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta_0, \Delta^{r_1}, \ldots, \Delta^{r_n}}\ (\textsc{Mix}),$$

where $\Gamma_0 \dashv \Delta_0$ is a propositional axiom.

$$\frac{\Sigma_k \dashv \Lambda_k, C_k \ \cdots \ \Sigma_l \dashv \Lambda_l, C_l \qquad \Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}}{\Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}, \forall r_k.C_k, \ldots, \forall r_l.C_l}\ (\textsc{Mix}, \forall),$$

where, for $i = k, \ldots, l$, $\Sigma_i = \{C \mid \forall r_i.C \in \Gamma^{r_i}\}$, $\Lambda_i = \{C \mid \exists r_i.C \in \Delta^{r_i}\}$, and $1 \leq k \leq l \leq n$.

$$\frac{\Lambda_k, C_k \dashv \Sigma_k \ \cdots \ \Lambda_l, C_l \dashv \Sigma_l \qquad \Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}}{\Gamma^{r_1}, \ldots, \Gamma^{r_n}, \exists r_k.C_k, \ldots, \exists r_l.C_l \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}}\ (\textsc{Mix}, \exists),$$

where, for $i = k, \ldots, l$, $\Sigma_i = \{C \mid \exists r_i.C \in \Delta^{r_i}\}$, $\Lambda_i = \{C \mid \forall r_i.C \in \Gamma^{r_i}\}$, and $1 \leq k \leq l \leq n$.

STRUCTURAL RULES OF $\mathbf{SC}^c_{\mathcal{ALC}}$

$$\frac{\Gamma, C \dashv \Delta}{\Gamma \dashv \Delta}\ (w^{-1}, l) \quad \frac{\Gamma \dashv \Delta, C}{\Gamma \dashv \Delta}\ (w^{-1}, r) \quad \frac{\Gamma, C \dashv \Delta}{\Gamma, C, C \dashv \Delta}\ (c^{-1}, l) \quad \frac{\Gamma \dashv C, \Delta}{\Gamma \dashv C, C, \Delta}\ (c^{-1}, r)$$

Fig. 1: Rules of $\mathbf{SC}^c_{\mathcal{ALC}}$.

model, say v_0. In the proceeding steps, in case of an application of a rule $(\textsc{Mix}, \forall)$ or $(\textsc{Mix}, \exists)$, a new child is created, where the parent of the new node is the node assigned to the conclusion of the rule application. The left premiss is then assigned the new node, while the right premiss is assigned the node of the conclusion of the rule application. The resulting arc is labelled with the role name r_1, as represented in the exposition of the rules $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$ (cf. Figure 1), indicating that the arc represents a tuple in the interpretation of r_1. In case of a rule application different from $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$, the node assigned to the conclusion is also assigned to the premiss(es). In order to complete the specification of the counter model, it remains to define the extensions of the atomic concepts. If $\Gamma_0 \dashv \Delta_0$ is an axiom in our proof and v' its assigned node, then we simply ensure that (i) v' is in the extension of each $C \in \Gamma_0$ and (ii) v' does not occur in the extension of any $D \in \Delta_0$.

Example 7. A counter model to $\exists r.C \sqcap \exists r.D \sqsubseteq \exists r.(C \sqcap D)$ is given by the interpretation $\mathcal{I} = \langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$ defined by $\Delta^{\mathcal{I}} = \{v_0, v_1, v_2\}$, $C^{\mathcal{I}} = \{v_2\}$, $D^{\mathcal{I}} = \{v_1\}$, and $r^{\mathcal{I}} = \{(v_0, v_1), (v_0, v_2)\}$. The reader may easily verify that this counter model can be read off the proof given in Example 6 by the informal method just described. □

The next statements justify the soundness of particular rules and axioms. The knowledge about tree-shaped interpretations, i.e., the shape of models satisfying a concept, provide a semantic justification of the mix rules.

Lemma 8. *Let $\mathcal{T}$ be a tree-shaped interpretation with root v_0 which refutes an anti-sequent of form*

$$\Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}. \tag{3}$$

Then, the anti-sequent (3) is refuted by every tree-shaped interpretation $\mathcal{T}'$ such that

1. *for every role name $r \in \mathsf{N}_R$, we have $r^{\mathcal{T}} = r^{\mathcal{T}'}$, and*
2. *for every concept name C, either $C^{\mathcal{T}'} = C^{\mathcal{T}} \cup \{v_0\}$, $C^{\mathcal{T}'} = C^{\mathcal{T}} \setminus \{v_0\}$, or $C^{\mathcal{T}'} = C^{\mathcal{T}}$ holds.*

Lemma 9. *Let $\Gamma_1, \ldots, \Gamma_n, \Delta_1, \ldots, \Delta_n$ be non-empty multi-sets of concepts and C a concept. Then,*

1. *every axiom of form*

$$\forall r_1.\Gamma_1, \ldots, \forall r_n.\Gamma_n \dashv \exists r_1.\Delta_1, \ldots, \exists r_n.\Delta_n$$

is refuted by some tree-shaped interpretation, and

2. *for every $i = 1, \ldots, n$, if $\Gamma_i \dashv C$ is refuted by some tree-shaped interpretation $\mathcal{T}_0$ and*

$$\forall r_1.\Gamma_1, \ldots, \forall r_n.\Gamma_n \dashv \forall r_1.\Delta_1, \ldots, \forall r_n.\Delta_n$$

is refuted by some tree-shaped interpretation $\mathcal{T}$ such that there exist disjoint subtrees $\mathcal{T}_1, \ldots, \mathcal{T}_n$, where $\mathcal{T}_j \triangleleft_{r_j} \mathcal{T}$ ($j = 1, \ldots, n$), then the tree-shaped interpretation $\mathcal{T}' = \langle v'; r_1, \mathcal{T}_1; \ldots; r_n, \mathcal{T}_n; r_i, \mathcal{T}_0 \rangle$ refutes the anti-sequent $\forall r_1.\Gamma_1, \ldots, \forall r_n.\Gamma_n \dashv \forall r_1.\Delta_1, \ldots, \forall r_n.\Delta_n, \forall r_i.C$, where v' does not occur in the domain of any T_j ($j = 1, \ldots, n$).

Theorem 10. $\mathbf{SC}^c_{\mathcal{ALC}}$ *is sound, i.e., only the refutable anti-sequents are provable.*

Proof (Sketch). The proof proceeds by induction on proof length. This amounts to showing the refutability of the axioms and the soundness of each rule separately. For the induction base, the refutability of a propositional axiom is obvious, while the refutability of an axiom of form $\forall r_1.\Gamma_1, \ldots, \forall r_n.\Gamma_n \dashv \exists r_1.\Delta_1, \ldots, \exists r_n.\Delta_n$ is exactly the statement of Item 1 of Lemma 9.

For the inductive step, we have to distinguish several cases depending on the last applied rule. It is a straightforward argument to show the soundness of the rules dealing with the propositional connectives $\sqcap$, $\sqcup$, and $\neg$. Hence, we just consider the mix rules briefly. For the rule (Mix), let $\Gamma_0 \dashv \Delta_0$ be a propositional axiom of $\mathbf{SC}^c_{\mathcal{ALC}}$ and $\Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta^{r_1}, \ldots, \Delta^{r_n}$ be refuted by a tree-shaped interpretation $\mathcal{T}$ with root

v_0. By Lemma 8, we define an interpretation $\mathcal{T}'$ such that $C^{\mathcal{T}'} = C^{\mathcal{T}} \cup \{v_0\}$, for every $C \in \Gamma_0$, and $C^{\mathcal{T}'} = C^{\mathcal{T}} \backslash \{v_0\}$, for every $C \in \Delta_0$. Since $\Gamma_0 \cap \Delta_0 = \emptyset$, the interpretation $\mathcal{T}'$ is well-defined and refutes the anti-sequent $\Gamma_0, \Gamma^{r_1}, \ldots, \Gamma^{r_n} \dashv \Delta_0, \Delta^{r_1}, \ldots, \Delta^{r_n}$. For multi-sets Γ, Δ, and Π, it is easy to see that an anti-sequent of form $\Gamma \dashv \neg\Pi, \Delta$ is refutable iff $\Gamma, \Pi \dashv \Delta$ is. Considering this fact and that value restriction is dual to existential restriction (i.e., $(\forall r.C)^{\mathcal{I}} = (\neg\exists r.\neg C)^{\mathcal{I}}$, for every interpretation $\mathcal{I}$), the soundness of the rules $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$ can easily be shown by repeated application of Item 2 of Lemma 9. □

Following Goranko [6], the completeness argument for $\mathbf{SC}^c_{\mathcal{ALC}}$ is divided into two steps: first, one proves completeness of the propositional fragment of our calculus. This approach is also made by Bonatti [2] for the rejection system for classical propositional logic and is thus left out for the sake of brevity. The second step consists of showing completeness by induction on the length of the refuting tree-shaped interpretation.

In what follows, the *logical complexity*, $||C||$, of a concept C is defined to be the number of connectives occurring in C. The logical complexity $||\Gamma||$ of a multi-set Γ of concepts is the sum of the logical complexities of the concept occurring in Γ. For an anti-sequent $\Gamma \dashv \Delta$, we define the logical complexity $||\Gamma \dashv \Delta||$ to be $||\Gamma|| + ||\Delta||$. Given an anti-sequent $s = \Gamma \dashv \Delta$, we denote by $con(s)$ the set of distinct concept names occurring in s.

Lemma 11. *Every refutable propositional anti-sequent is provable in* $\mathbf{SC}^c_{\mathcal{ALC}}$.

Lemma 12. *Let s be a refutable anti-sequent which is refuted by an interpretation I such that $C_1, \ldots, C_n$ are exactly those distinct concept names among s whose extensions are non-empty under I and $D_1, \ldots, D_m$ exactly those whose extensions are empty under I, respectively. Then, there exists a proof of s in* $\mathbf{SC}^c_{\mathcal{ALC}}$ *which has as its only axiom $C_1, \ldots, C_n \dashv D_1, \ldots, D_m$.*

In what follows, an *atom* of an anti-sequent $\Gamma \dashv \Delta$ is either a concept name from $\Gamma \cup \Delta$ or some concept of form $\mathsf{Q}r.C$ ($\mathsf{Q} \in \{\forall, \exists\}$) which occurs in a concept from $\Gamma \cup \Delta$ and is not in the scope of some $\mathsf{Q} \in \{\forall, \exists\}$.

Theorem 13. $\mathbf{SC}^c_{\mathcal{ALC}}$ *is complete, i.e., every refutable anti-sequent is provable.*

Proof. Let $\Gamma \dashv \Delta$ be an arbitrary refutable anti-sequent and $\mathcal{T}$ a tree-shaped interpretation of length $\ell(\mathcal{T})$ with domain $\Delta^{\mathcal{T}}$ which refutes $\Gamma \dashv \Delta$. Since $\Gamma \dashv \Delta$ is refuted by $\mathcal{T}$, there must be some $c \in \Delta^{\mathcal{T}}$ such that (i) $c \in C^{\mathcal{T}}$, for all $C \in \Gamma$, and (ii) $c \notin D^{\mathcal{T}}$, for all $D \in \Delta$. In the following, we say that a concept C is *falsified by $\mathcal{T}$* if $c \notin C^{\mathcal{T}}$ and *satisfied by $\mathcal{T}$* otherwise. Now let $C_1, \ldots, C_k$,$\forall r_1.D_1, \ldots, \forall r_l.D_l$,$\exists s_1.F_1, \ldots, \exists s_\lambda.F_\lambda$ be the atoms of $\Gamma \cup \Delta$ which are satisfied by $\mathcal{T}$, and $E_1, \ldots, E_{k'}$,$\forall p_1.G_1, \ldots, \forall p_\mu.G_\mu$, $\exists q_1.H_1, \ldots, \exists q_m.H_m$ those which are falsified by $\mathcal{T}$, where $C_1, \ldots, C_k, E_1, \ldots, E_{k'}$ are atomic concepts and $r_1, \ldots, r_l, s_1, \ldots, s_\lambda, p_1, \ldots, p_\mu, q_1, \ldots, q_m$ are not necessarily distinct role names. We define the following sets:

$$\begin{aligned} \Gamma_0 &= \{C_1, \ldots, C_k\}, & \Delta_0 &= \{E_1, \ldots, E_{k'}\}, \\ \Sigma^{\forall} &= \{\forall r_1.D_1, \ldots, \forall r_l.D_l\}, & \Lambda^{\exists} &= \{\exists s_1.F_1, \ldots, \exists s_\lambda.F_\lambda\}, \\ \Pi^{\forall} &= \{\forall p_1.G_1, \ldots, \forall p_\mu.G_\mu\}, & \Phi^{\exists} &= \{\exists q_1.H_1, \ldots, \exists q_m.H_m\}. \end{aligned}$$

It suffices to infer

$$\varGamma_0, \varSigma^{\forall}, \varLambda^{\exists} \dashv \varDelta_0, \varPi^{\forall}, \varPhi^{\exists} \tag{4}$$

since Lemma 11 and Lemma 12 allow us to infer $\varGamma \dashv \varDelta$ from (4). This is accomplished by replacing all atoms of (4) by new distinct concept names which then becomes a propositional axiom. Then, we can infer a propositional anti-sequent $\varGamma' \dashv \varDelta'$ by Lemma 11 and Lemma 12 which is obtained from $\varGamma \dashv \varDelta$ by exactly the same replacement as mentioned before. Hence, there exists a proof of $\varGamma' \dashv \varDelta'$—substituting the new concept names back we obtain a proof of $\varGamma \dashv \varDelta$.

Obviously, we have that $\varGamma_0 \cap \varDelta_0 = \emptyset$, hence, $\varGamma_0 \dashv \varDelta_0$ is a propositional axiom. Furthermore, the anti-sequent

$$\varSigma^{\forall} \dashv \varPhi^{\exists} \tag{5}$$

constitutes an axiom of $\mathbf{SC}^c_{\mathcal{ALC}}$. We now proceed by induction on $\ell(\mathcal{T})$. For the base case, if $\ell(\mathcal{T}) = 0$, (4) must be of form $\varGamma_0, \varSigma^{\forall} \dashv \varDelta_0, \varPhi^{\exists}$, by the semantics of $\mathcal{ALC}$. This anti-sequent is inferred by the following application of rules:

$$\frac{\varGamma_0 \dashv \varDelta_0 \qquad \varSigma^{\forall} \dashv \varPhi^{\exists}}{\varGamma_0, \varSigma^{\forall} \dashv \varDelta_0, \varPhi^{\exists}}\ (\textsc{Mix})$$

This completes the base case. Now assume that every anti-sequent which is refuted by some tree $\mathcal{T}$ with length $\ell(\mathcal{T}) \leq n$ is provable in $\mathbf{SC}^c_{\mathcal{ALC}}$. Furthermore, for every role name r, define $\varTheta(r) = \{C \,|\, \forall r.C \in \varSigma^{\forall}\}$ and $\varXi(r) = \{C \,|\, \exists r.C \in \varPhi^{\exists}\}$. If (4) is refuted by some tree $\mathcal{T}$ with length $\ell(\mathcal{T}) = n + 1$, then the anti-sequents

$$\varTheta(p_i) \dashv G_i, \varXi(p_i), \text{ for } i = 1, \ldots, \mu, \text{ and} \tag{6}$$
$$\varTheta(s_j), F_j \dashv \varXi(s_j), \text{ for } j = 1, \ldots, \lambda, \tag{7}$$

are refuted by immediate subtrees of $\mathcal{T}$ with length $\ell(\mathcal{T}) = n$ and are therefore, by induction hypothesis, provable in $\mathbf{SC}^c_{\mathcal{ALC}}$. We first consider (6) and start for $i = 1$ applying $(\textsc{Mix}, \forall)$ to Axiom (5):

$$\frac{\varTheta(p_1) \dashv G_1, \varXi(p_1) \qquad \varSigma^{\forall} \dashv \varPhi^{\exists}}{\varSigma^{\forall} \dashv \forall p_1.G_1, \varPhi^{\exists}}\ (\textsc{Mix}, \forall)$$

Now, for every $i = 2, \ldots, \mu$, we proceed constructing a proof of the anti-sequent $\varSigma^{\forall} \dashv \forall p_1.G_1, \ldots, \forall p_i.G_i, \varPhi^{\exists}$ from $\varSigma^{\forall} \dashv \forall p_1.G_1, \ldots, \forall p_{i-1}.G_{i-1}, \varPhi^{\exists}$ in the following way:

$$\frac{\varTheta(p_i) \dashv G_i, \varXi(p_i) \qquad \varSigma^{\forall} \dashv \forall p_1.G_1, \ldots, \forall p_{i-1}.G_{i-1}, \varPhi^{\exists}}{\varSigma^{\forall} \dashv \forall p_1.G_1, \ldots, \forall p_i.G_i, \varPhi^{\exists}}\ (\textsc{Mix}, \forall)$$

For $i = \mu$, we obtain a proof of the anti-sequent $\varSigma^{\forall} \dashv \varPi^{\forall}, \varPhi^{\exists}$. Building upon this anti-sequent, we proceed in a similar way by constructing the following proof:

$$\frac{\varTheta(s_1), F_1 \dashv \varXi(s_1) \qquad \varSigma^{\forall} \dashv \varPi^{\forall}, \varPhi^{\exists}}{\varSigma^{\forall}, \exists s_1.F_1 \dashv \varPi^{\forall}, \varPhi^{\exists}}\ (\textsc{Mix}, \exists)$$

Again, for every $i = 2, \ldots, \lambda$, we proceed constructing a proof of the anti-sequent $\varSigma^{\forall}, \exists s_1.F_1, \ldots, \exists s_i.F_i \dashv \varPi^{\forall}, \varPhi^{\exists}$ from $\varSigma^{\forall}, \exists s_1.F_1, \ldots, \exists s_{i-1}.F_{i-1} \dashv \varPi^{\forall}, \varPhi^{\exists}$ in the following manner:

$$\frac{\varTheta(s_i), F_i \dashv \varXi(s_i) \qquad \varSigma^{\forall}, \exists s_1.F_1, \ldots, \exists s_{i-1}.F_{i-1} \dashv \varPi^{\forall}, \varPhi^{\exists}}{\varSigma^{\forall}, \exists s_1.F_1, \ldots, \exists s_i.F_i \dashv \varPi^{\forall}, \varPhi^{\exists}} \;(\textsc{Mix}, \exists)$$

For $i = \lambda$ we obtain a proof of the anti-sequent $\varSigma^{\forall}, \varLambda^{\exists} \dashv \varPi^{\forall}, \varPhi^{\exists}$. Finally, we apply the rule $(\textsc{Mix})$ in order to obtain a proof of the desired anti-sequent:

$$\frac{\varGamma_0 \dashv \varDelta_0 \qquad \varSigma^{\forall}, \varLambda^{\exists} \dashv \varPi^{\forall}, \varPhi^{\exists}}{\varGamma_0, \varSigma^{\forall}, \varLambda^{\exists} \dashv \varDelta_0, \varPi^{\forall}, \varPhi^{\exists}} \;(\textsc{Mix})$$

Hence, (4) is inferred and the induction step is completed. Since every refutable anti-sequent is refuted by some tree-shaped interpretation, every refutable anti-sequent is provable and $\mathbf{SC}^{c}_{\mathcal{ALC}}$ is complete as desired. □

4 Comparing $\mathrm{SC}^{c}_{\mathcal{ALC}}$ with an $\mathcal{ALC}$ Tableau Algorithm

The most common reasoning procedures which have been studied for description logics are tableau algorithms. They are well known for $\mathcal{ALC}$ and its extensions and have been implemented in state-of-the-art reasoners (like, e.g., in the FaCT system [21]). Tableau algorithms rely on the construction of a *canonical model* which witnesses the satisfiability of a concept or a knowledge base. We now briefly discuss the relationship of our calculus and the tableau procedure for concept satisfiability as discussed by Baader and Sattler [18].

The basic structure the algorithm works on is the so-called *completion graph*. A completion graph is an ordered triple $\langle V, E, \mathcal{L} \rangle$, where V is a set of *nodes*, $E \subseteq V \times V$ a set of *edges*, and $\mathcal{L}$ a *labelling function* which assigns a set of concepts to each node and a role name to each edge. Given a concept C in negation normal form (i.e., where negation occurs in C only in front of concept names), the *initial completion graph of* C is a completion graph $\langle V, E, \mathcal{L} \rangle$ where $V = \{v_0\}$, $E = \emptyset$, and $\mathcal{L}(v_0) = \{C\}$. A completion graph $G = \langle V, E, \mathcal{L} \rangle$ contains a *clash* if $\{D, \neg D\} \subseteq \mathcal{L}(v)$ for some node v and some concept D. G is *complete* if no rules are applicable any more. The algorithm operates at each instant on a set $\mathbf{G}$ of completion graphs. The *completion rules* specify the rules which may be applied to infer a new set of completion graphs $\mathbf{G}'$ from some set of completion graphs $\mathbf{G}$. Given a concept C, the algorithm starts with the set $\mathbf{G}_0 = \{G_0\}$, where G_0 is the initial completion graph of C, and successively computes a new set $\mathbf{G}_{i+1}$ of completion graphs from the set $\mathbf{G}_i$. Thereby, every completion graph which has a clash is immediately dropped. The algorithm halts if for some $j \geq 0$, $\mathbf{G}_j$ contains a complete completion graph or $\mathbf{G}_j = \emptyset$. In the former case, the algorithm answers that the concept C is satisfiable, in the latter case it answers that C is unsatisfiable. It is well-known that a model of the concept under consideration can be extracted from a complete completion graph and that (in the case of concept satisfiability) a complete completion graph represents a tree [18].

$$\dfrac{[C \dashv]_{v_0} \quad \dfrac{\dfrac{\dfrac{[D \dashv C]_{v_3}}{[C \sqcup D \dashv C]_{v_3}}\,(\sqcup, l)_2 \quad [\dashv \exists p.C]_{v_1}}{[\exists p.(C \sqcup D) \dashv \exists p.C]_{v_1}}\,(\textsc{Mix}, \exists) \quad \alpha}{[\forall r.\exists p.(C \sqcup D) \dashv \forall r.\exists p.C, \forall r.D]_{v_0}}\,(\textsc{Mix}, \forall)}{\dfrac{[\forall r.\exists p.(C \sqcup D), C \dashv \forall r.\exists p.C, \forall r.D]_{v_0}}{\dfrac{[\forall r.\exists p.(C \sqcup D), C \dashv \forall r.\exists p.C \sqcup \forall r.D]_{v_0}}{[\forall r.\exists p.(C \sqcup D) \sqcap C \dashv \forall r.\exists p.C \sqcup \forall r.D]_{v_0}}\,(\sqcap, l)}\,(\sqcup, r)}\,(\textsc{Mix})$$

where α is the following proof:

$$\dfrac{\dfrac{[\dashv D]_{v_2} \quad \dfrac{\dfrac{[C \dashv]_{v_4}}{[C \sqcup D \dashv]_{v_4}}\,(\sqcup, l)_1 \quad [\dashv]_{v_2}}{[\exists p.(C \sqcup D) \dashv]_{v_2}}\,(\textsc{Mix}, \exists)}{[\exists p.(C \sqcup D) \dashv D]_{v_2}}\,(\textsc{Mix}) \quad [\forall r.\exists p(C \sqcup D) \dashv]_{v_0}}{[\forall r.\exists p.(C \sqcup D) \dashv \forall r.D]_{v_0}}\,(\textsc{Mix}, \forall)$$

$$\begin{tikzcd}
 & v_0 \ \mathcal{L}(v_0) \arrow[dl, "r"'] \arrow[dr, "r"] & \\
\{\exists p.(C \sqcup D), \forall p.\neg C\} \ v_1 \arrow[d, "p"] & & v_2 \ \{\neg D, \exists p.(C \sqcup D)\} \arrow[d, "p"] \\
\{C \sqcup D, \neg C, D\} \ v_3 & & v_4 \ \{C \sqcup D, C\}
\end{tikzcd}$$

where $\mathcal{L}(v_0) = \{\forall r.\exists p.(C \sqcup D) \sqcap C, \exists r.\forall p.\neg C \sqcap \exists r.\neg D\} \cup$
$\{\forall r.\exists p.(C \sqcup D), C, \exists r.\forall p.\neg C, \exists r.\neg D\}$.

Fig. 2: A proof in $\mathbf{SC}^c_{\mathcal{ALC}}$ and a corresponding completion graph.

We now describe how to obtain a complete completion graph from a proof of $\mathbf{SC}^c_{\mathcal{ALC}}$ such that the root of the completion graph is labelled with the end-sequent of the proof (for the sake of readability, we consider without loss of generality the case where $k = l = 1$ in the rules $(\textsc{Mix}, \forall)$ and $(\textsc{Mix}, \exists)$). Let $\mathit{nnf}(C)$ denote the negation normal form of a concept C. For a set of concepts $\mathit{\Gamma}$, define $\mathit{nnf}(\mathit{\Gamma}) = \{\mathit{nnf}(C) \mid C \in \mathit{\Gamma}\}$ and, for an anti-sequent $s = \mathit{\Gamma} \dashv \mathit{\Delta}$, define $\mathit{nnf}(s) = \mathit{nnf}(\mathit{\Gamma} \cup \neg\mathit{\Delta})$. Furthermore, let $\tau[G]$ denote the root of a completion graph G. We define a mapping ξ which maps any proof of $\mathbf{SC}^c_{\mathcal{ALC}}$ to some complete completion graph. Let χ be a proof of $\mathbf{SC}^c_{\mathcal{ALC}}$ and s_χ be the end-sequent of χ. The mapping ξ is inductively defined as follows:

- If s_χ is a propositional axiom, then $\xi(\chi) = \langle V, E, \mathcal{L}\rangle$, where $V = \{v_0\}$, $E = \emptyset$, and $\mathcal{L}(v_0) = \mathit{nnf}(s_\chi)$.
- If s_χ results from an application of some binary rule ρ, and $\xi(\chi_1) = G_1 = \langle V_1, E_1, \mathcal{L}_1\rangle$ and $\xi(\chi_2) = G_2 = \langle V_2, E_2, \mathcal{L}_2\rangle$, where χ_1 is the proof of the left

premiss, χ_2 is the proof of the right premiss, and G_1 and G_2 are disjoint, then we distinguish several cases:

- If $\rho = (\textsc{Mix}, \forall)$, then s_χ is of form $\Gamma^{r_1}, \dots, \Gamma^{r_n} \dashv \Delta^{r_1}, \dots, \Delta^{r_n}, \forall r.C$. Then, $\xi(\chi) = G = \langle V, E, \mathcal{L} \rangle$, where $\tau[G] = v'$ ($v' \notin V_1 \cup V_2$), $V = V_1 \cup V_2 \cup \{v'\}$, $E = E_1 \cup E_2 \cup \{e\}$, for $e = (v', \tau[G_1])$, and the labelling function preserves the labels from G_1 and G_2 but additionally satisfies $\mathcal{L}(e) = r$ and $\mathcal{L}(v') = nnf(s_\chi)$.
- If $\rho = (\textsc{Mix}, \exists)$, then s_χ is of form $\Gamma^{r_1}, \dots, \Gamma^{r_n}, \exists r.C \dashv \Delta^{r_1}, \dots, \Delta^{r_n}$, $V = V_1 \cup V_2 \cup \{v'\}$, $E = E_1 \cup E_2 \cup \{e\}$, for $e = (v', \tau[G_1])$, and the labelling function preserves the labels from G_1 and G_2 but additionally satisfies $\mathcal{L}(e) = r$ and $\mathcal{L}(v') = nnf(s_\chi)$.
- If $\rho = (\textsc{Mix})$, then s_χ is of form $\Gamma_0, \Gamma^{r_1}, \dots, \Gamma^{r_n} \dashv \Delta_0, \Delta^{r_1}, \dots, \Delta^{r_n}$. Then, $\xi(\chi) = G = \langle V, E, \mathcal{L} \rangle$, where $\tau[G] = \tau[G_2]$, $V = V_1 \cup V_2$, $E = E_1 \cup E_2$, and $\mathcal{L}(\tau[G]) = nnf(s_\chi)$.

– If s_χ results from application of some unary rule, and $\xi(\chi_1) = G_1 = \langle V_1, E_1, \mathcal{L}_1 \rangle$, where χ_1 is proof of the upper sequent, then $\xi(\chi) = G = \langle V, E, \mathcal{L} \rangle$, where $V = V_1$, $E = E_1$, and the labelling function preserves the labels from G_1 but additionally satisfies $\mathcal{L}(\tau[G]) = \mathcal{L}_1(\tau[G]) \cup nnf(s_\chi)$.

Theorem 14. *Let χ be a proof in $\mathbf{SC}^c_{\mathcal{ALC}}$ and s_χ the end-sequent of χ. Then, there exists a complete completion graph $G = \langle V, E, \mathcal{L} \rangle$ such that $\xi(\chi) = G$ and $nnf(s_\chi) \subseteq \mathcal{L}(\tau[G])$.*

Example 15. In Figure 2, we compare a proof of $\mathbf{SC}^c_{\mathcal{ALC}}$ with its corresponding complete completion graph $G = \langle V, E, \mathcal{L} \rangle$. For better readability, we labelled each anti-sequent in the proof with subscripts of form $[s]_v$ which means that $nnf(s) \subseteq \mathcal{L}(v)$. Note that the completion graph represents a model of the concept $\iota(s_\chi)$, where s_χ is the end-sequent of the depicted proof. In fact, a model is given by $\mathcal{I} = \langle \Delta^\mathcal{I}, \cdot^\mathcal{I} \rangle$, where $\Delta^\mathcal{I} = \{v_i \mid 0 \leq i \leq 4\}$, $D^\mathcal{I} = \{v_3\}$, $C^\mathcal{I} = \{v_0, v_4\}$, $r^\mathcal{I} = \{(v_0, v_1), (v_0, v_2)\}$, and $p^\mathcal{I} = \{(v_1, v_3), (v_2, v_4)\}$. □

5 A Multi-Modal Rejection Calculus

As mentioned above, the development of our calculus for $\mathcal{ALC}$ is based on a rejection calculus for modal logic $\mathbf{K}$, as introduced by Goranko [6], by taking into account that $\mathcal{ALC}$ can be translated into a multi-modal version of $\mathbf{K}$. In this section, we lay down the relation of our calculus to Goranko's system, thereby generalising his calculus to the multi-modal case.

We start with describing the multi-modal logic $\mathbf{K}_m$ [20]. In general, the signature of multi-modal logics usually provide a countably infinite supply of different *modalities* which we identify by lower case Greek letters $\alpha, \beta, \dots$ as well as a countably infinite supply of *propositional variables* $p, q, \dots$. Formulae in the language of a multi-modal logic are then built up using the propositional connectives $\wedge$, $\vee$, $\neg$, $\top$, $\bot$ and the *modal operators* $[\alpha]$, where α is a modality. For every modality α, we define $\langle \alpha \rangle := \neg[\alpha]\neg$.

LOGICAL RULES

$$\frac{\Gamma, \varphi, \psi \dashv \Delta}{\Gamma, \varphi \wedge \psi \dashv \Delta}\,(\wedge, l) \qquad \frac{\Gamma \dashv \varphi, \Delta}{\Gamma \dashv \varphi \wedge \psi, \Delta}\,(\wedge, r)_1 \qquad \frac{\Gamma \dashv \psi, \Delta}{\Gamma \dashv \varphi \wedge \psi, \Delta}\,(\wedge, r)_2$$

$$\frac{\Gamma \dashv \varphi, \psi, \Delta}{\Gamma \dashv \varphi \vee \psi, \Delta}\,(\vee, r) \qquad \frac{\Gamma, \varphi \dashv \Delta}{\Gamma, \varphi \vee \psi \dashv \Delta}\,(\vee, l)_1 \qquad \frac{\Gamma, \psi \dashv \Delta}{\Gamma, \varphi \vee \psi \dashv \Delta}\,(\vee, l)_2$$

$$\frac{\Gamma \dashv \varphi, \Delta}{\Gamma, \neg\varphi \dashv \Delta}\,(\neg, l) \qquad \frac{\Gamma, \varphi \dashv \Delta}{\Gamma \dashv \neg\varphi, \Delta}\,(\neg, r) \qquad \frac{\Gamma \dashv \Delta}{\Gamma, \top \dashv \Delta}\,(\top) \qquad \frac{\Gamma \dashv \Delta}{\Gamma \dashv \bot, \Delta}\,(\bot)$$

$$\frac{\Gamma_0 \dashv \Delta_0 \qquad [\alpha_1]\Gamma_1, \ldots, [\alpha_n]\Gamma_n \dashv [\alpha_1]\Delta_1, \ldots, [\alpha_n]\Delta_n}{\Gamma_0, [\alpha_1]\Gamma_1, \ldots, [\alpha_n]\Gamma_n \dashv \Delta_0, [\alpha_1]\Delta_1, \ldots, [\alpha_n]\Delta_n}\,(\textsc{Mix})$$

where Γ_0, Δ_0 are disjoint sets of propositional variables.

$$\frac{\Gamma_k \dashv \varphi_k \ \cdots \ \Gamma_l \dashv \varphi_l \qquad [\alpha_1]\Gamma_1, \ldots, [\alpha_n]\Gamma_n \dashv [\alpha_1]\Delta_1, \ldots, [\alpha_n]\Delta_n}{[\alpha_1]\Gamma_1, \ldots, [\alpha_n]\Gamma_n \dashv [\alpha_1]\Delta_1, \ldots, [\alpha_n]\Delta_n, [\alpha_k]\varphi_k, \ldots, [\alpha_l]\varphi_l}\,(\textsc{Mix}^2)$$

where $1 \leq k \leq l \leq n$.

STRUCTURAL RULES

$$\frac{\Gamma, \varphi \dashv \Delta}{\Gamma \dashv \Delta}\,(w^{-1}, l) \quad \frac{\Gamma \dashv \Delta, \varphi}{\Gamma \dashv \Delta}\,(w^{-1}, r) \quad \frac{\Gamma, \varphi \dashv \Delta}{\Gamma, \varphi, \varphi \dashv \Delta}\,(c^{-1}, l) \quad \frac{\Gamma \dashv \varphi, \Delta}{\Gamma \dashv \varphi, \varphi, \Delta}\,(c^{-1}, r)$$

Fig. 3: Rules of a multi-modal variant of $\mathbf{SC}^c_{\mathcal{ALC}}$.

Following Goranko and Otto [22], let τ be the set of all modalities. A *Kripke interpretation* is a triple $\mathcal{M} = \langle W, \{R_\alpha\}_{\alpha\in\tau}, V\rangle$, where W is a non-empty set of *worlds*, $R_\alpha \subseteq W \times W$ defines an *accessibility relation* for each $\alpha \in \tau$, and V maps any propositional variable to a subset of W, i.e., V defines which propositional variables are true at which worlds. The pair $\langle W, \{R_\alpha\}_{\alpha\in\tau}\rangle$ defines the *Kripke frame* on which $\mathcal{M}$ is *based*. Given any Kripke interpretation $\mathcal{M} = \langle W, \{R_\alpha\}_{\alpha\in\tau}, V\rangle$, we define the *truth of a formula* φ *at a world* $w \in W$, denoted by $\mathcal{M}, w \models \varphi$, inductively in the usual manner. Furthermore, the notions of *validity* in a frame and validity in a class of frames is defined as usual (cf. Goranko and Otto [22] for a detailed account). $\mathbf{K}_m$ is the multi-modal logic consisting of all formulae which are valid in all Kripke frames.

A concept of $\mathcal{ALC}$ can be translated into a formula of $\mathbf{K}_m$ by viewing concepts of form $\forall r.C$ as modal formulae of form $[\alpha]C'$, where C' is the corresponding translation of the concept C. Each role name corresponds to one and only one modality. Furthermore, the propositional connectives of $\mathcal{ALC}$ can easily seen to be translated into the usual connectives of classical propositional logic. From a semantic point of view, interpretations of $\mathcal{ALC}$ correspond to Kripke interpretations if we identify the domain of the interpretation with the corresponding set of worlds of the Kripke interpretation. Furthermore, the interpretation of each role name corresponds to some accessibility relation. The extension of an $\mathcal{ALC}$ concept contains then exactly those worlds of the corresponding Kripke interpretation where the translation of the concept is satisfied.

Let C be a concept, $\mathcal{I}$ an interpretation, and let the translations be given by the formula φ and the Kripke interpretation $\mathcal{M}$, respectively. It holds that $\mathcal{M}, w \models \varphi$ iff $w \in C^{\mathcal{I}}$. For a full treatment of the translation, we refer to Schild [20] and Baader *et al.* [15].

A rejection system for $\mathbf{K}_m$ can now be defined as follows. Let a *multi-modal anti-sequent* $\Gamma \dashv \Delta$ be defined as in the case of $\mathcal{ALC}$, but with Γ and Δ being finite multi-sets of multi-modal formulae. $\Gamma \dashv \Delta$ is *refutable* if there exists a Kripke interpretation $\mathcal{M} = \langle W, \{R_\alpha\}_{\alpha \in \tau}, V \rangle$ and some $w \in W$ such that $\mathcal{M}, w \models \varphi$, for every $\varphi \in \Gamma$, but $\mathcal{M}, w \not\models \psi$, for every $\psi \in \Delta$. Axioms of this system are given by anti-sequents of form $\Gamma_0 \dashv \Delta_0$ with Γ_0, Δ_0 being disjoint sets of propositional variables and anti-sequents of form $[\alpha_1]\Gamma_1, \dots, [\alpha_n]\Gamma_n \dashv$. The rules of the resulting calculus are depicted in Figure 3 (for any multi-set Γ and modality α, we have $[\alpha]\Gamma := \{[\alpha]\varphi \mid \varphi \in \Gamma\}$; $\alpha_1, \dots, \alpha_n$ are pairwise different modalities).

Note that, e.g., $(\textsc{Mix}^2)$ corresponds to our rule $(\textsc{Mix}, \forall)$,

$$\frac{\Gamma \dashv \varphi \qquad \Box\Gamma \dashv \Box\Delta}{\Box\Gamma \dashv \Box\Delta, \Box\varphi}\; \mathit{MIX}^2_{\mathbf{K}},$$

where $\Box\Sigma := \{\Box\varphi \mid \varphi \in \Sigma\}$ for any multi-set Σ. We did not explicitly include here a corresponding rule for $\langle\alpha\rangle$ since this can be derived using $(\textsc{Mix}^2)$.

The following result can be shown:

Theorem 16. *A multi-modal anti-sequent is refutable iff it is provable.*

6 Conclusion

We presented a sequent-type calculus for deciding concept non-subsumption in $\mathcal{ALC}$. Sequent calculi are important means for proof-theoretic investigations. We pointed out that our calculus is in some sense equivalent to a well-known tableau procedure which is interesting from a conceptual point of view: as pointed out by Goranko [6], the reason why complementary calculi have been rarely studied may be found in the fact that often theories are recursively axiomatisable while being undecidable. Hence, for such logics, reasonable complementary calculi cannot be devised as the set of invalid propositions is not recursively enumerable there.

However, most of the description logics studied so far are decidable, hence they permit an axiomatisation of the satisfiable concepts. Indeed, the well-known tableau procedures for different description logics focus on checking satisfiability, while unsatisfiability is established by exhaustive search. For such logics, the sequent-style counterparts to tableau procedures are complementary calculi, since checking satisfiability can be reduced to checking invalidity. We note in passing that the tableau procedure for $\mathcal{ALC}$ has also been simulated by *hyperresolution* by Fermüller *et al.* [23].

As for future work, a natural extension of our calculus is to take $\mathcal{ALC}$ TBox reasoning into account. However, this seems not to be straightforward in view of existent tableau algorithms for it. We conjecture that a feasible approach would need to employ a more complicated notion of anti-sequent, which is related to the problem of non-termination of the respective tableau algorithm without blocking rules.

References

1. Łukasiewicz, J.: Aristotle's Syllogistic from the Standpoint of Modern Formal Logic. Clarendon Press (1957)
2. Bonatti, P.A.: A Gentzen System for Non-Theorems. Technical Report CD-TR 93/52, Technische Universität Wien, Institut für Informationssysteme (1993)
3. Tiomkin, M.L.: Proving Unprovability. In: Proc. LICS '88, IEEE Computer Society (1988) 22–26
4. Dyckhoff, R.: Contraction-Free Sequent Calculi for Intuitionistic Logic. Journal of Symbolic Logic **57**(3) (1992) 795–807
5. Kreisel, G., Putnam, H.: Eine Unableitbarkeitsbeweismethode für den Intuitionistischen Aussagenkalkül. Archiv für Mathematische Logik und Grundlagenforschung **3**(1-2) (1957) 74–78
6. Goranko, V.: Refutation Systems in Modal Logic. Studia Logica **53** (1994) 299–324
7. Skura, T.: Refutations and Proofs in S4. In Wansing, H., ed.: Proof Theory of Modal Logic, Kluwer (1996) 45–51
8. Oetsch, J., Tompits, H.: Gentzen-Type Refutation Systems for Three-Valued Logics with an Application to Disproving Strong Equivalence. In: Proc. LPNMR 2011, Springer (2011) 254–259
9. Wybraniec-Skardowska, U.: On the Notion and Function of the Rejection of Propositions. Acta Universitatis Wratislaviensis Logika 23 (2754) (2005) 179–202
10. Caferra, R., Peltier, N.: Accepting/rejecting Propositions from Accepted/rejected Propositions: A Unifying Overview. International Journal of Intelligent Systems **23**(10) (2008) 999–1020
11. Bonatti, P.A., Olivetti, N.: Sequent Calculi for Propositional Nonmonotonic Logics. ACM Transactions on Computational Logic **3**(2) (April 2002) 226–278
12. Reiter, R.: A Logic for Default Reasoning. Artificial Intelligence **13**(1-2) (1980) 81–132
13. Moore, R.C.: Semantical Considerations on Nonmonotonic Logic. In: Proc. IJCAI '83, William Kaufmann (1983) 272–279
14. McCarthy, J.: Circumscription – A Form of Non-Monotonic Reasoning. Artificial Intelligence **13**(1-2) (1980) 27–39
15. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
16. Rademaker, A.: A Proof Theory for Description Logics. Springer (2012)
17. Borgida, A., Franconi, E., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F.: Explaining ALC Subsumption. In: Proc. DL '99. Volume 22 of CEUR Workshop Proceedings. (1999)
18. Baader, F., Sattler, U.: An Overview of Tableau Algorithms for Description Logics. Studia Logica **69** (2001) 5–40
19. Takeuti, G.: Proof Theory. Studies in Logic and the Foundations of Mathematics Series. North-Holland (1975)
20. Schild, K.: A Correspondence Theory for Terminological Logics: Preliminary Report. In: Proc. IJCAI '91, Morgan Kaufmann Publishers Inc. (1991) 466–471
21. Horrocks, I.: The FaCT System. In: Proc. TABLEAUX '98, Springer (1998) 307–312
22. Goranko, V., Otto, M.: Model Theory of Modal Logic. In Blackburn, P., Wolter, F., van Benthem, J., eds.: Handbook of Modal Logic. Elsevier (2006) 255–325
23. Fermüller, C., Leitsch, A., Hustadt, U., Tammet, T.: Resolution Decision Procedures. In Robinson, A., Vorkonov, A., eds.: Handbook of Automated Reasoning. Elsevier (2001) 1791–1849

Introducing Real Variables and Integer Objective Functions to Answer Set Programming[⋆]

Guohua Liu, Tomi Janhunen, and Ilkka Niemelä

Helsinki Institute for Information Technology HIIT
Department of Information and Computer Science
Aalto University, FI-00076 AALTO, FINLAND
{Guohua.Liu,Tomi.Janhunen,Ilkka.Niemela}@aalto.fi

Abstract. Answer set programming languages have been extended to support linear constraints and objective functions. However, the variables allowed in the constraints and functions are restricted to integer and Boolean domains, respectively. In this paper, we generalize the domain of linear constraints to real numbers and that of objective functions to integers. Since these extensions are based on a translation from logic programs to mixed integer programs, we compare the translation-based answer set programming approach with the native mixed integer programming approach using a number of benchmark problems.

1 Introduction

Answer set programming (ASP) [14], also known as logic programming under *stable model* semantics [8], is a declarative programming paradigm where a given problem is solved by devising a logic program whose *answer sets* capture the solutions of the problem and then by computing the answer sets using *answer set solvers*. The paradigm has been exploited in a rich variety of applications [2].

Linear constraints have been introduced to ASP [1, 7, 11, 12] in order to combine the high-level modeling capabilities of ASP languages with the efficient constraint solving techniques developed in the area of constraint programming. In particular, a language ASP(LC) is devised in [11] which allows linear constraints to be used within the original ASP language structures. The answer set computation for ASP(LC) programs is based on mixed integer programming (MIP) where an ASP(LC) program is first translated into a MIP program and then the solutions of the MIP program are computed using a MIP solver. Finally, answer sets can be recovered from the solutions found (if any).

In this paper, we extend and evaluate the ASP(LC) language in the following aspects. First, we generalize the domain of variables allowed in linear constraints from integers to reals. Real variables are ubiquitous in applications, e.g., timing variables in scheduling problems. However, the MIP-based answer set computation confines the variables in linear constraints to the integer domain. We overcome this limitation by developing a translation of ASP(LC) programs to MIP programs so that constraints over

[⋆] The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.

real variables are enabled in the language. Second, we introduce MIP objective functions, i.e., linear functions of integer variables, to the ASP(LC) language. The original ASP(LC) language allows objective functions of Boolean variables only, but integer variables are more convenient than Booleans in many applications [11]. To model optimization problems in these areas, we enable MIP objective functions in ASP by giving semantics for ASP programs with these functions. Third, we compare ASP(LC) to MIP. This is interesting as ASP(LC) provides a richer language than MIP where ASP language structures are extended with linear constraints but the implementation technique is based on translating an ASP(LC) program to a MIP program to solve. We choose some representative problems, study the ASP(LC) and MIP encodings of the problems, and evaluate their computational performance by experiments.

The rest of the paper is organized as follows. Preliminaries are given in Section 2. Then we extend ASP(LC) language with real variables in Section 3, introduce MIP objective functions in Section 4, and compare ASP(LC) and native MIP formulations in Section 5. The experiments are reported in Section 6 followed by a discussion on the related work in Section 7. The paper is concluded by Section 8.

2 Preliminaries

In this section, we review the basic concepts of linear constraints, mixed integer programming, and the ASP(LC) language. A *linear constraint* is an expression of the form

$$\sum_{i=1}^{n} u_i x_i \sim k \tag{1}$$

where the u_i's and k are real numbers and the x_i's are variables ranging over real numbers (including integers). We distinguish the variables to be *real* and *integer* variables when necessary. The operator $\sim$ is in $\{<, \leq, \geq, >\}$. Constraints involving "$<$" and "$>$" are called *strict* constraints. A valuation ν from variables to numbers is a *solution* of (or *satisfies*) a constraint C of the form (1), denoted $\nu \models C$, iff $\sum_{i=1}^{n} u_i \nu(x_i) \sim k$ holds. A valuation ν is a solution of a set of constraints $\Pi = \{C_1, ..., C_m\}$, denoted $\nu \models \Pi$, iff $\nu \models C_i$ for each $C_i \in \Pi$. A set of linear constraints is *satisfiable* iff it has a solution.

A *mixed integer program* (or a *MIP program*), takes the form

$$\texttt{optimize} \quad \sum_{i=1}^{n} u_i x_i \tag{2}$$

$$\texttt{subject to} \; C_1, ..., C_m. \tag{3}$$

where the keyword `optimize` is `minimize` or `maximize`, u_i's are numbers, x_i's are variables, and C_i's are linear constraints. The operators in the constraints are in $\{\leq, =, \geq\}$. The function $\sum_{i=1}^{n} u_i x_i$ is called an *objective function*. The constraints $C_1, ..., C_m$ may be written as a set $\{C_1, ..., C_m\}$. A valuation ν is a solution of a MIP program iff $\nu \models \{C_1, ..., C_m\}$. A solution is *optimal*, iff it minimizes (or maximizes) the value of the objective function. The objective function could be empty (missing from a MIP program), in which case the function is trivially optimized by any solution. The keywords `optimize` and `subject to` may be omitted if the objective function is empty. The goal of MIP is to find the optimal solutions of a MIP program.

An *ASP(LC) program* is a set of rules of the form

$$a \leftarrow b_1, \ldots, b_n, \text{not}\, c_1, \ldots, \text{not}\, c_m, t_1, \ldots, t_l \tag{4}$$

where each a, b_i, and c_i is a propositional atom and each t_i, called a *theory atom,* is a linear constraint of the form (1). Propositional atoms and theory atoms may be uniformly called *atoms*. Atoms and atoms preceded by ”not” are also referred to as *positive* and *negative literals*, respectively. Given a program P, the set of propositional and theory atoms appearing in P are denoted by $\mathcal{A}(P)$ and $\mathcal{T}(P)$, respectively. For a rule r of the form (4), the *head* and the *body* of r are defined by $\mathrm{H}(r) = \{a\}$ and $\mathrm{B}(r) = \{b_1, \ldots, b_n, \text{not}\ c_1, \ldots, \text{not}\ c_m, t_1, \ldots, t_l\}$. Furthermore, the *positive, negative,* and *theory* parts of the body are defined as $\mathrm{B}^+(r) = \{b_1, \ldots, b_n\}$, $\mathrm{B}^-(r) = \{c_1, \ldots, c_m\}$, and $\mathrm{B}^t(r) = \{t_1, \ldots, t_l\}$, respectively. The body and the head of a rule could be empty: a rule without body is a *fact* whose head is true unconditionally and a rule without head is an *integrity constraint* enforcing the body to be false.

A set of atoms M satisfies an atom a, denoted $M \models a$, iff $a \in M$, and it satisfies a negative literal ’not a’, denoted $M \models \text{not}\, a$, iff $a \notin M$. The set M satisfies a set of literals $L = \{l_1, \ldots, l_n\}$, denoted $M \models L$, iff $M \models l_i$ for each $l_i \in L$. An *interpretation* of an ASP(LC) program P is a pair $\langle M, T\rangle$ where $M \subseteq \mathcal{A}(P)$ and $T \subseteq \mathcal{T}(P)$, such that $T \cup \bar{T}$ is satisfiable in linear arithmetics where $\bar{T} = \{\neg t \mid t \in \mathcal{T}(P) \text{ and } t \notin T\}$ and $\neg t$ denotes the constraint obtained by changing the operator of t to the complementary one. Two interpretations $I_1 = \langle M_1, T_1\rangle$ and $I_2 = \langle M_2, T_2\rangle$ are *equal*, denoted $I_1 = I_2$, iff $M_1 = M_2$ and $T_1 = T_2$. An interpretation $I = \langle M, T\rangle$ satisfies a literal l iff $M \cup T \models l$. An interpretation I satisfies a rule r, denoted $I \models r$, iff $I \models \mathrm{H}(r)$ or $I \not\models \mathrm{B}(r)$. An integrity constraint is satisfied by I iff $I \not\models \mathrm{B}(r)$. An interpretation I is a *model* of a program P, denoted $I \models P$, iff $I \models r$ for each $r \in P$.

Answer sets are defined using the concept of program *reduct* as follows.

Definition 1 (Liu et al. [11]). *Let P be an ASP(LC) program and $\langle M, T\rangle$ an interpretation of P. The* reduct *of P with respect to $\langle M, T\rangle$, denoted $P^{\langle M,T\rangle}$, is defined as $P^{\langle M,T\rangle} = \{\mathrm{H}(r) \leftarrow \mathrm{B}^+(r) \mid r \in P, \mathrm{H}(r) \neq \emptyset,\ \mathrm{B}^-(r) \cap M = \emptyset, \text{ and } \mathrm{B}^t(r) \subseteq T\}$.*

Definition 2 (Liu et al. [11]). *Let P be an ASP(LC) program. An interpretation $\langle M, T\rangle$ is an* answer set *of P iff $\langle M, T\rangle \models P$ and M is the subset minimal model of $P^{\langle M,T\rangle}$. The set of answer sets of P is denoted by $AS(P)$.*

Example 1. Let P be an ASP(LC) program consisting of the rules

$$a \leftarrow x - y \leq 2. \quad b \leftarrow x - y \geq 5. \quad \leftarrow x - y \geq 0.$$

The interpretation $I_1 = \langle\{a\}, \{x - y \leq 2\}\rangle$ is an answer set of P since $\{(x - y \leq 2), \neg(x - y \geq 5), \neg(x - y \geq 0)\}$ is satisfiable in linear arithmetics, $I_1 \models P$, and $\{a\}$ is the minimal model of $P^{I_1} = \{a \leftarrow .\}$. The interpretation $I_2 = \langle\{b\}, \{x - y \geq 5\}\rangle$ is not an answer set since $\{(x - y \geq 5), \neg(x - y \leq 2), \neg(x - y \geq 0)\}$ is unsatisfiable. Finally, $I_3 = \langle\emptyset, \{x - y \geq 0\}\rangle$ is not an answer set, since $I_3 \not\models P$. $\square$

Syntactically, theory atoms are allowed as heads of rules in the current implementation [11] for more intuitive reading and thus such rules are used in this paper. As regards their semantics, a rule with a theory atom as the head is equivalent to an integrity constraint, i.e., a rule $t \leftarrow a_1, \ldots, a_m, \text{not } b_1, \ldots, \text{not } b_n, t_1, \ldots, t_l$ where t is a theory atom is treated as the rule $\leftarrow a_1, \ldots, a_m, \text{not } b_1, \ldots, \text{not } b_n, t_1, \ldots, t_l, \neg t$ in answer set computation. Moreover, the semantics of ASP(LC) programs coincides with that of *normal* logic programs [8, 14] if no theory atoms are present.

Answer set computation for ASP(LC) programs is based on a translation to MIP programs [11]. We will refer to the translation as *MIP-translation* and denote the translation of a program P by $\tau(P)$. Due to space limitations, we skip a thorough review of $\tau(P)$ and focus on a fragment most relevant for this paper, i.e., the rules of the form

$$a \leftarrow t. \tag{5}$$

where a is an propositional atom or not present at all and t is a theory atom. Recall that a rule without head is an integrity constraint.

In the translation, special linear constraints called *indicator constraints* are used. An indicator constraint is of the form $d = v \rightarrow C$ where d is a *binary variable* (integer variable with the domain $\{0, 1\}$), v is either 0 or 1, and C is a linear constraint. An indicator constraint is *strict* if C is strict and *non-strict* otherwise. An indicator constraint can be written as a constraint (1) using the so-called *big-M* formulation.

For a program P consisting of simple rules (5) only, $\tau(P)$ is formed as follows:

1. For each theory atom t, we include a pair of indicator constraints

$$d = 1 \rightarrow t \qquad d = 0 \rightarrow \neg t \tag{6}$$

   where d is a new binary variable introduced for t. The idea is to use the variable d to represent the constraint t in the sense that, for any solution ν of the constraints in (6), $\nu(d) = 1$ iff $\nu \models t$. Thus d can be viewed as a kind of a *name* for t.
2. Assuming that $a \leftarrow t_1$, ..., $a \leftarrow t_k$ are all rules (5) that have a as head, we include

$$a - d_1 \geq 0, \quad \ldots, \quad a - d_k \geq 0, \tag{7}$$
$$d_1 + \ldots + d_k - a \geq 0 \tag{8}$$

   where $d_1, \ldots, d_k$ are the binary variables corresponding to $t_1, \ldots, t_k$ in (6). The constraints in (7) and (8) enforce that the joint head a holds iff some of the bodies $t_1, \ldots, t_k$ holds which is compatible with Clark's completion [3]. If $k = 1$, i.e., the atom a has a unique defining rule, the constraints of (7) and (8) reduce to $a - d_1 = 0$ which makes d_1 synonymous with a. Moreover, if the rule (5) is an integrity constraint, then $d_1 = 0$ is sufficient, as intuitively implied by $k = 1$ and $a = 0$.

In the implementation of $\tau(P)$, more variables and constraints are used to cover the rules of the general form (4). We refer the reader to [11] for details.

The solutions of the MIP-translation of a program capture its answer sets as follows. Let P be an ASP(LC) program and ν a mapping from variables to numbers. We define the *ν-induced interpretation* of P, denoted I_P^ν, by setting $I_P^\nu = \langle M, T \rangle$ where

$$M = \{a \mid a \in \mathcal{A}(P),\ \nu(a) = 1\} \text{ and} \tag{9}$$
$$T = \{t \mid t \in \mathcal{T}(P),\ \nu \models t\}. \tag{10}$$

Theorem 1 (Liu et al. [11]). *Let P be an ASP(LC) program.*

1. *If ν is a solution of $\tau(P)$, then $I^{\nu}_{P} \in AS(P)$.*
2. *If $I \in AS(P)$, then there is a solution ν of $\tau(P)$ such that $I = I^{\nu}_{P}$.*

Example 2. For the program P from Example 1, the translation $\tau(P)$ consists of:

$$\begin{array}{lll} d_1 = 1 \rightarrow x - y \leq 2, & d_1 = 0 \rightarrow x - y > 2, & a - d_1 = 0, \\ d_2 = 1 \rightarrow x - y \geq 5, & d_2 = 0 \rightarrow x - y < 5, & b - d_2 = 0, \\ d_3 = 1 \rightarrow x - y \geq 0, & d_3 = 0 \rightarrow x - y < 0, & d_3 = 0. \end{array}$$

For any solution ν of $\tau(P)$, we have $\nu(a) = 1$, $\nu(b) = 0$, and $\nu(x) - \nu(y) \leq 2$ which characterize the unique answer set $\langle \{a\}, \{x - y \leq 2\} \rangle$ of P. □

3 Extension with Real Variables

In this section, we first illustrate how strict constraints involved in the MIP-translation prevent the introduction of real variables in ASP(LC) programs. Motivated by these observations, we develop a translation of strict constraints to non-strict ones. Finally, we apply the translation to remove strict constraints from the MIP-translation so that real variables can be allowed in ASP(LC) programs.

3.1 Problems Caused by Real Variables

Real variables are widely used in knowledge representation and reasoning. However, the computation of answer sets based on the MIP-translation becomes problematic in their presence. The reason is that typical MIP systems do not fully support strict constraints involving real variables, e.g., by treating strict constraints as non-strict ones. Consequently, the correspondence between solutions and answer sets may be lost.

Example 3. Consider the condition that Tom gets a bonus if he works at least 8.25 hours and the fact that he works for that long. By formalizing these constraints we obtain an ASP(LC) program P consisting of the following rules:

$$\mathsf{bonus}(tom) \leftarrow h(tom) \geq 8.25. \tag{11}$$

$$\leftarrow h(tom) < 8.25. \tag{12}$$

In the above, the ground term $h(tom)$ is treated as a real variable recording the working hours of Tom and $\mathsf{bonus}(tom)$[1] is a ground (propositional) atom meaning that Tom will be paid a bonus. The MIP-translation $\tau(P)$ of P has the following constraints:

$$d_1 = 1 \rightarrow h(tom) \geq 8.25 \tag{13}$$

$$d_1 = 0 \rightarrow h(tom) < 8.25 \tag{14}$$

$$\mathsf{bonus}(tom) - d_1 = 0 \tag{15}$$

$$d_2 = 1 \rightarrow h(tom) < 8.25 \tag{16}$$

$$d_2 = 0 \rightarrow h(tom) \geq 8.25 \tag{17}$$

$$d_2 = 0 \tag{18}$$

[1] We use different fonts for function and predicate symbols, such as "h" and "bonus" in this example, for clarity.

Given $\tau(P)$ as input, CPLEX provides a solution ν where $\nu(\mathsf{bonus}(tom)) = \nu(d_1) = \nu(d_2) = 0$ and $\nu(h(tom) = 8.25)$. However $I_P^\nu = \langle \emptyset, \{h(tom) = 8.25\}\rangle$ is not an answer set of P since it does not satisfy the rule (11). This discrepancy is due to the fact that ν actually does not satisfy the strict constraint (14), but CPLEX treats this as the non-strict one $d_1 = 0 \rightarrow h(tom) \leq 8.25$ and unexpectedly gives ν as a solution. □

The current implementation of the MIP-translation [11] addresses only integer-valued constraints where the coefficients and variables range over integers. Given this restriction, strict constraints of the form $\sum_{i=1}^{n} u_i x_i < k$ (resp. $> k$) can be implemented as non-strict ones $\sum_{i=1}^{n} u_i x_i \leq k - 1$ (resp. $\geq k + 1$).

It might be tempting to convert the domain of a problem from reals to integers, e.g., by multiplying the constraints by 100 and by replacing the variables $h(tom)$ by another holding a hundredfold value. For the program P in Example 3 this would give rise to:

$$\mathsf{bonus}(tom) \leftarrow h'(tom) \geq 825. \tag{19}$$

$$\leftarrow h'(tom) < 825. \tag{20}$$

Thereafter constraints (13) and (14) could be rewritten as non-strict constraints:

$$d_1 = 1 \rightarrow h'(tom) \geq 825. \tag{21}$$

$$d_1 = 0 \rightarrow h'(tom) \leq 824. \tag{22}$$

This approach, however, does not work in general. First, the translated program cannot cover the domain of the original problem due to the continuity of real numbers. For example, the rules (21) and (22) do not give any information about the working hours 8.245 which is covered by (13) and (14). Second, determining the required coefficients is infeasible in general since the real numbers occurring in constraints can be specified up to arbitrary precision which could vary from problem instance to another.

Because CPLEX treats strict constraints as non-strict ones, the MIP-translation becomes inapplicable for answer set computation in the presence of real-valued variables. To enable such computations, a revised translation which consists of non-strict constraints only is needed. Such a translation is devised in sections to come.

3.2 Non-Strict Translation of Strict Constraints

We focus on strict constraints of a restricted form $y > 0$. This goes without loss of generality because any constraint $\sum_{i=1}^{n} u_i x_i > k$ can be rewritten as a conjunction of a non-strict constraint $\sum_{i=1}^{n} u_i x_i - y = k$ and a strict one $y > 0$ where y is fresh. Also, a constraint of the form $\sum_{i=1}^{n} u_i x_i < k$ is equivalent to $-\sum_{i=1}^{n} u_i x_i > -k$.

Lemma 1. *Let Γ be a set of non-strict constraints, $S = \{x_1 > 0, \ldots, x_n > 0\}$, and δ a new variable. Then, the set $\Gamma \cup S$ is satisfiable iff for any bound $b > 0$, the set $\Gamma \cup S_\delta \cup \{0 < \delta \leq b\}$ where $S_\delta = \{x_1 \geq \delta, \ldots, x_n \geq \delta\}$ is satisfiable.*

Proof. We prove the direction "⇒" since the other direction is obvious. Since $\Gamma \cup S$ is satisfiable, there is a valuation ν such that $\nu \models \Gamma$ and $\nu(x_i) > 0$ for each $1 \leq i \leq n$. Let $b > 0$ be any number and $m = \min\{\nu(x_1), \ldots, \nu(x_n)\}$. Then $\nu(x_i) \geq m$ holds for any $1 \leq i \leq n$ and $m > 0$. Two cases arise and need to be analyzed:

Case 1: If $m \leq b$, then $\Gamma \cup S_\delta \cup \{0 < \delta \leq b\}$ has a solution ν' which extends ν by the assignment $\nu'(\delta) = m$.

Case 2: If $m > b$, define ν' as an extension of ν such that $\nu'(\delta) = b$. Thus $b > 0$ implies $\nu' \models 0 < \delta \leq b$. Moreover, for any $1 \leq i \leq n$, $\nu'(x_i) = \nu(x_i) \geq \nu'(\delta) = b$ since $m > b$. Therefore $\nu' \models x_i \geq \delta$ for any $1 \leq i \leq n$.

It follows that $\nu' \models \Gamma \cup S_\delta \cup \{0 < \delta \leq b\}$. □

The result of Lemma 1 can be lifted to the case of indicator constraints since indicator constraints are essentially linear constraints.

Lemma 2. *Let Γ be a set of non-strict constraints,*

$$S = \{d_i = v_i \rightarrow x_i > 0 \mid 1 \leq i \leq n\} \tag{23}$$

a set of strict indicator constraints, and δ a new variable. Then, $\Gamma \cup S$ is satisfiable iff for any bound $b > 0$, $\Gamma \cup S_\delta \cup \{0 < \delta \leq b\}$ is satisfiable where

$$S_\delta = \{d_i = v_i \rightarrow x_i \geq \delta \mid 1 \leq i \leq n\}. \tag{24}$$

Lemma 2 shows that a set of strict indicator constraints can be transformed to a set of non-strict ones by introducing a new bounded variable $0 < \delta \leq b$. Below, we relax the last remaining strict constraint $\delta > 0$ to $\delta \geq 0$ using a MIP objective function.

Definition 3. *Let $\Pi = \Gamma \cup S$ be a set of constraints where Γ is a set of non-strict ones and S is the set of strict indicator constraints (23), δ a new variable, and $b > 0$ a bound. The* non-strict translation *of Π with respect to δ and b, denoted Π^b_δ, is:*

$$\begin{array}{ll} \texttt{maximize} & \delta \\ \texttt{subject to} & \Gamma \cup S_\delta \cup \{0 \leq \delta \leq b\} \end{array} \tag{25}$$

where S_δ is defined by (24).

Given Lemma 2 and Definition 3, the satisfiability of a set of constraints can be captured by its non-strict translation as formalized by the following theorem.

Theorem 2. *Let Π, S, and Π^b_δ be defined as in Definition 3. Then, Π is satisfiable iff Π^b_δ has a solution ν such that $\nu(\delta) > 0$.*

Theorem 2 enables the use of current MIP systems for checking the satisfiability of a set of strict constraints, i.e., by computing an optimal solution for the non-strict translation of the set and by checking if the objective function has a positive value.

3.3 Non-Strict Translation of Programs

Next, we develop the non-strict translation of ASP(LC) programs using Definition 3.

Definition 4. *Let P be an ASP(LC) program, δ a new variable, and $b > 0$ a bound. The* non-strict translation *of P with respect to δ and b, is $\tau(P)^b_\delta$ where $\tau(P)$ is the MIP-translation of P.*

We show that the solutions of $\tau(P)^b_\delta$ and $\tau(P)$ are in a tight correspondence.

Lemma 3. *Let P be an ASP(LC) program that may involve real variables, δ a new variable, and $b > 0$ a bound.*

1. *For any solution $\nu \models \tau(P)$, there is a solution $\nu' \models \tau(P)^b_\delta$ such that $\nu(a) = \nu'(a)$ for each $a \in \mathcal{A}(P)$, $\nu \models t$ iff $\nu' \models t$ for each $t \in \mathcal{T}(P)$, and $\nu'(\delta) > 0$.*
2. *For any solution $\nu \models \tau(P)^b_\delta$ where $\nu(\delta) > 0$, there is a solution ν' of $\tau(P)$ such that $\nu(a) = \nu'(a)$ for each $a \in \mathcal{A}(P)$ and $\nu \models t$ iff $\nu' \models t$ for each $t \in \mathcal{T}(P)$.*

Proof. We prove (i) and omit the proof of (ii) which is similar. Let ν be a solution of $\tau(P)$. Given ν, we extend $\tau(P)$ to $\tau'(P)$ by adding for each atom $a \in \mathcal{A}(P)$, a constraint $a = \nu(a)$, and for each theory atom $t \in \mathcal{T}(P)$ and the variable d introduced for t in (6), $d = \nu(d)$. It is clear that $\nu' = \nu$ is a solution of $\tau'(P)$. Let $\tau''(P)$ be the analogous extension of $\tau(P)^b_\delta$. Applying Theorem 2 to $\tau'(P)$, there is a solution ν'' of $\tau''(P)$ such that $\nu''(\delta) > 0$ and for each a, $\nu''(a) = \nu'(a) = \nu(a)$, and for each d, $\nu''(d) = \nu'(d) = \nu(d)$. The valuation ν'' is also a solution of $\tau(P)^b_\delta$, as $\tau(P)^b_\delta \subset \tau''(P)$. Note that for any $t \in \mathcal{T}(P)$ and the respective atom d, $\nu(d) = 1$ iff $\nu \models t$, and $\nu''(d) = 1$ iff $\nu'' \models t$. Then $\nu \models t$ iff $\nu'' \models t$ due to $\nu(d) = \nu''(d)$. □

Now, we relate the solutions of $\tau(P)^b_\delta$ and the answer sets of P. As a consequence of Lemma 3 and the generalization of Theorem 1 for real variables, we obtain:

Theorem 3. *Let P be an ASP(LC) program that may involve real variables, δ a new variable, and $b > 0$ a bound.*

1. *If ν is a solution of $\tau(P)^b_\delta$ such that $\nu(\delta) > 0$, then $I^\nu_P \in AS(P)$.*
2. *If $I \in AS(P)$, then there is a solution ν of $\tau(P)^b_\delta$ such that $I = I^\nu_P$ and $\nu(\delta) > 0$.*

Example 4. Let us revisit Example 3. By setting $b = 1$ as the bound, we obtain the non-strict translation $\tau(P)^1_\delta$ as follows:

$$\begin{array}{lll}
\texttt{maximize} & \delta & \\
\texttt{subject to} & 0 \le \delta \le 1 & \\
 & d_1 = 1 \to h(tom) \ge 8.25, & d_1 = 0 \to h(tom) + \delta \le 8.25, \\
 & d_2 = 1 \to h(tom) + \delta \le 8.25, & d_2 = 0 \to h(tom) \ge 8.25, \\
 & \mathsf{bonus}(tom) - d_1 = 0, & d_2 = 0.
\end{array}$$

For any optimal solution ν of $\tau(P)^1_\delta$, we have $\nu(\mathsf{bonus}(tom)) = \nu(d) = \nu(\delta) = 1$ and $\nu(h(tom)) \ge 8.25$ that corresponds to the intended answer set $\langle\{\mathsf{bonus}(tom)\}, \{h(tom) \ge 8.25\}\rangle$. We note that CPLEX provides exactly this solution for $\tau(P)^1_\delta$. □

It can be verified that the non-strict translation $\tau(P)^b_\delta$ reduces to the MIP-translation if the variables in P and the new variable δ are integers and the bound b is set to 1.

4 Extension with Objective Functions

In this section, we define optimal answer sets for ASP(LC) programs enhanced by *objective functions* of the form (2) and illustrate the resulting concept by examples.

Definition 5. *Let P be an ASP(LC) program with an objective function f and $\langle M, T\rangle \in AS(P)$. The answer set $\langle M, T\rangle$ is* optimal *iff there is a solution of $T \cup \bar{T}$ that gives the optimal value to f among the set of valuations*

$$\{\nu \mid \nu \models T \cup \bar{T} \textit{ for some } \langle M, T\rangle \in AS(P)\}.$$

Example 5. Let P be an ASP(LC) program

$$\texttt{minimize}\ x. \quad a \leftarrow x \geq 5. \quad b \leftarrow x \geq 7. \quad \leftarrow x < 5.$$

The answer sets of P are $I_1 = \langle\{a\}, \{x \geq 5\}\rangle$ and $I_2 = \langle\{a, b\}, \{x \geq 5, x \geq 7\}\rangle$. Let $T_1 = \{x \geq 5\}$ and $T_2 = \{x \geq 5, x \geq 7\}$. The solutions of $T_1 \cup \bar{T_1} = \{x \geq 5, x < 7\}$ admit a smaller value of $f(x) = x$ (i.e., 5) than any solution of $T_2 \cup \bar{T_2} = \{x \geq 5, x \geq 7\}$. Therefore the answer set I_1 is optimal. □

According to Definition 5, each optimal answer set identifies an optimal objective value. In other words, if the objective function is unbounded with respect to an answer set, then the answer set is not optimal. This is illustrated by our next example.

Example 6. Let P be an ASP(LC) program

$$\texttt{minimize}\ x. \quad a \leftarrow x \leq 4. \quad b \leftarrow x > 4.$$

The program P has two answer sets $I_1 = \langle\{a\}, \{x \leq 4\}\rangle$ and $I_2 = \langle\{b\}, \{x > 4\}\rangle$. Let $T_1 = \{x \leq 4\}$ and $T_2 = \{x > 4\}$. We have $T_1 \cup \bar{T_1} = T_1$ and $T_2 \cup \bar{T_2} = T_2$. Although $T_1 \cup \bar{T_1}$ admits smaller values of x than $T_2 \cup \bar{T_2}$, I_1 is not optimal, since x may become infinitely small subject to $T_1 \cup \bar{T_1}$. Therefore, P has no optimal answer set. □

For a program that does not involve real variables, we can establish an approach to computing the optimal answer sets as stated in the theorem below. This result essentially follows from Definition 5 and Theorem 1.

Theorem 4. *Let P be an ASP(LC) program involving integer variables only and f the objective function of P. Then $\langle M, T\rangle$ is an optimal answer set of P iff there is a solution $\nu \models \tau(P)$ such that $I_P^\nu = \langle M, T\rangle$ and ν gives the optimal value to f.*

However, when real variables are involved, the non-strict translation from ASP to MIP programs cannot be employed to compute the optimal answer sets, since the variable δ introduced in the translation may affect the optimal objective function value.

The ASP languages implemented in [6, 16] support objective functions of the form

$$\texttt{\#optimize}\,[a_1 = w_{a_1}, ..., a_m = w_{a_m}, \text{not } b_1 = w_{b_1}, ..., \text{not } b_n = w_{b_n}] \qquad (26)$$

where the keyword `optimize` is `minimize` or `maximize`, a_i and ”not b_i” are literals, and w_{a_i} and w_{b_i} are integer weights associated with the respective literals. The difference between the functions (2) and (26) is that x_i's in the former are integer variables whereas a_i's and b_i's in the latter are Boolean variables, i.e., propositional atoms from the ASP viewpoint. Proper objective functions facilitate modeling optimization problems as one needs not to encode integer variables in terms of Booleans.

Example 7. Let x be an integer variable taking a value from 1 to n and we want to minimize x. Using an objective function, this can be concisely encoded by:

$$\texttt{minimize}\ x. \quad x \geq 1. \quad x \leq n.$$

Following [14, 17], to encode the same using the function (26), we need:

$$\texttt{\#minimize}\ [x(1) = 1, \ldots, x(n) = n]. \tag{27}$$

$$x(i) \leftarrow \text{not}\, x(1), \ldots, \text{not}\, x(i-1), \text{not}\, x(i+1), \ldots \text{not}\, x(n). \quad 1 \leq i \leq n \tag{28}$$

where the Boolean variable $x(i)$ represents that x takes value i and the rules in (28) encode that x takes exactly one value from 1 to n. The size of the ASP(LC) encoding is constant, while that of the original ASP encoding is quadratic[2] in the size of the domain of x, i.e., the objective function of length n plus n rules of length n. □

Restriction to Boolean variables affects the performance of ASP systems when dealing with problems involving large domains. A detailed account can be found in [11].

5 A Comparison of ASP(LC) with MIP

In the ASP(LC) paradigm, a problem is solved indirectly, i.e., by first modeling it as an ASP program, then by translating the program to a MIP program, and by solving it. A question is how the approach compares with native MIP. In this section, we study the modeling capabilities of ASP(LC) and MIP languages. We focus on two widely used primitives in modeling: *reachability* and *disjunctivity*. For the former, we study a Hamiltonian Routing Problem (HRP) and for the latter, a Job Shop Problem (JSP). The main observation is that ASP(LC) can provide more intuitive and compact encodings in debt to its capability to model non-trivial logical relations. But, the compactness does not always offer computational efficiency as perceived in the sequel.

In the HRP, we have a network and a set of *critical* vertices. The goal is to route a package along a Hamiltonian cycle in the network so that the package reaches each critical vertex within a given vertex-specific time limit. The network is represented by a set of vertices $V = \{1, \ldots, n\}$ and a set of weighted edges E consisting of elements (i, j, d) where $1 \leq i, j \leq n$ and d is a real number representing that the delay of the edge from i to j is d. The set of critical vertices CV consists of pairs (i, t) where $1 < i \leq n$ and t is a real number representing that the time limit of vertex i is t.

The ASP(LC) encoding of HRP in Figure 1 is obtained by extending the encoding of Hamiltonian cycle problem [14] with timing constraints[3]. The rules (29)–(35) specify a Hamiltonian cycle. To model the timing constraints, we use a real variable $t(X)$ to denote the time when a vertex X is reached. Rules (36) and (37) determine the respective times of reaching vertices. The rule (38) ensures each critical vertex to be reached in time. The MIP program of HRP in Figure 2 is from [15] with extensions of timing constraints[4]. The binary variable x_{ij} represents whether an edge (i, j, d) is on the Hamiltonian cycle ($x_{ij} = 1$) or not ($x_{ij} = 0$) and the integer variable p_i denotes the

[2] Linear and logarithmic encodings can be achieved using *cardinality constraints* [6, 16] and *bit vectors* [13], respectively. But both are more complex than the given ASP(LC) encoding.

[3] A rule with the operator "=" in the head, such as (36) and (37), is a shorthand for a pair of rules with the operators "$\leq$" and "$\geq$" in their heads respectively.

[4] Disequalities and implications can be represented using the operators "$\leq$" and "$\geq$" [15].

$$\mathsf{hc}(X,Y) \leftarrow \mathsf{e}(X,Y,D),\ \text{not}\ \mathsf{nhc}(X,Y). \tag{29}$$
$$\mathsf{nhc}(X,Y) \leftarrow \mathsf{e}(X,Y,D_1),\ \mathsf{e}(X,Z,D_2),\ \mathsf{hc}(X,Z),\ Y \neq Z. \tag{30}$$
$$\mathsf{nhc}(X,Y) \leftarrow \mathsf{e}(X,Y,D_1),\ \mathsf{e}(Z,Y,D_2),\ \mathsf{hc}(Z,Y),\ X \neq Z. \tag{31}$$
$$\mathsf{initial}(1). \tag{32}$$
$$\mathsf{reach}(X) \leftarrow \mathsf{reach}(Y),\ \mathsf{hc}(Y,X),\ \text{not}\ \mathsf{initial}(Y),\ \mathsf{e}(Y,X,D). \tag{33}$$
$$\mathsf{reach}(X) \leftarrow \mathsf{hc}(Y,X),\ \mathsf{initial}(Y), \mathsf{e}(Y,X,D). \tag{34}$$
$$\leftarrow \mathsf{v}(X),\ \text{not}\ \mathsf{reach}(X). \tag{35}$$
$$t(1) = 0. \tag{36}$$
$$t(X) - t(Y) = D \leftarrow \mathsf{hc}(Y,X), \mathsf{e}(Y,X,D), X \neq 1. \tag{37}$$
$$t(X) \leq T \leftarrow \mathsf{critical}(X,T). \tag{38}$$

Fig. 1. An ASP(LC) encoding of HRP

$$\textstyle\sum_{(i,j,d)\in E} x_{ij} = 1 \qquad i \in V \tag{39}$$
$$\textstyle\sum_{(j,i,d)\in E} x_{ji} = 1 \qquad i \in V \tag{40}$$
$$1 \leq p_i \leq n \qquad i \in V \tag{41}$$
$$p_i \neq p_j \qquad i \in V,\ j \in V,\ i \neq j \tag{42}$$
$$p_j \neq p_i + 1 \qquad (i,j,d) \notin E,\ i \neq j \tag{43}$$
$$(p_i = n) \rightarrow (p_j \geq 2) \qquad (i,j,d) \notin E,\ i \neq j \tag{44}$$
$$r_1 = 0 \tag{45}$$
$$x_{ij} = 1 \rightarrow r_j - r_i = d_{ij} \qquad (i,j,d) \notin E \tag{46}$$
$$r_i \leq t_i \qquad (i,t) \in CV \tag{47}$$

Fig. 2. A MIP encoding of HRP

position of the vertex i on the cycle. The real variable r_i denotes the time of reaching vertex i. The constraints (39)–(44) encode a Hamiltonian cycle and (45)–(47) are the counterparts of (36)–(38) respectively.

In comparison, reachability is modeled by the recursive rules (33) and (34) in the ASP(LC) program. Since MIP language cannot express such recursion directly, the reachability condition is captured otherwise—by constraining the positions of the nodes in (41)–(44). Note that the node positions are actually irrelevant for the existence of a cycle. In fact, modeling Hamiltonian cycles in non-complete graphs is challenging in MIP and the above encoding is the most compact one to the best of our knowledge.

In the JSP, we have a set of tasks $T = \{1, ..., n\}$ to be executed by a machine. Each task is associated with an earliest starting time and a processing duration. The goal is to schedule the tasks so that each task starts at its earliest starting time or later, the processing of the tasks do not overlap, and all tasks are finished by a given deadline. Using ASP(LC) we model the problem in Figure 3 where the predicate $\mathsf{task}(I, E, D)$ denotes that a task I has an earliest starting time E and a duration D and the real variables $s(I)$ and $e(I)$ denote the starting and ending times of task I, respectively. The

$$s(I) \geq E \leftarrow \mathsf{task}(I, E, D). \tag{48}$$
$$e(I) - s(I) \geq D \leftarrow \mathsf{task}(I, E, D). \tag{49}$$
$$\leftarrow \mathsf{task}(I, E_1, D_1),\ \mathsf{task}(J, E_2, D_2),\ I \neq J,\ s(I) - s(J) \leq 0,\ s(J) - e(I) \leq 0. \tag{50}$$
$$e(I) \leq \mathit{deadline} \leftarrow \mathsf{task}(I, E, D). \tag{51}$$

Fig. 3. An ASP(LC) encoding of JSP

$$s_i \geq est_i \qquad i \in T \tag{52}$$
$$e_i - s_i \geq d_i \qquad i \in T \tag{53}$$
$$x_{ij} = 1 \rightarrow e_i - s_j < 0 \qquad i \in T,\ j \in T,\ i \neq j \tag{54}$$
$$x_{ij} = 0 \rightarrow e_i - s_j \geq 0 \qquad i \in T,\ j \in T,\ i \neq j \tag{55}$$
$$x_{ij} + x_{ji} = 1 \qquad i \in T,\ j \in T,\ i \neq j \tag{56}$$
$$e_i \leq \mathit{deadline} \qquad i \in T \tag{57}$$

Fig. 4. A MIP encoding of JSP

rule (48) says that a task starts at its earliest starting time or later. The rule (49) ensures each task being processed long enough. The rule (50) encodes the mutual exclusion of the tasks, i.e., for any two tasks, one must be finished before the starting of the other. The rule (51) enforces each task being finished by the deadline. Figure 4 adopts a recent MIP encoding of JSP [9] in CPLEX language, where the variables s_i and e_i represent the starting and ending times of task i, respectively; est_i and d_i are the earliest starting time and processing duration of task i; the binary variable x_{ij} denotes that task i ends before task j starts. The constraints (52), (53), and (57) are the counterparts of (48), (49), and (51) respectively. The constraints (54)–(56) exclude overlapping tasks.

In the ASP(LC) program of JSP, the mutual exclusion of $s(I) - s(J) \leq 0$ and $s(J) - e(I) \leq 0$ is expressed by one rule (50). In contrast, MIP language lacks direct encoding of relations between constraints and therefore, to encode the relation of $e_i < s_j$ and $e_j < s_i$, one has to first represent them by new variables x_{ij} and x_{ji} and then encode the relations of the variables (56). Note that, for computation, the ASP(LC) and the native MIP encodings are essentially the same, since the translation of the rule (50) includes two indicator constraints to represent the constraints $s(I) - s(J) \leq 0$ and $s(J) - e(I) \leq 0$, respectively, and additional constraints to encode the rule.

6 Experiments

We implemented the non-strict translation of ASP(LC) programs and the MIP objective functions by modifying the MINGO system [11]. The new system is called MINGO^r. We tested MINGO^r with a number of benchmarks[5]: the HRP and JSP detailed in Section 5,

[5] A prototype implementation of the MINGO^{r}system and benchmarks can be found under `http://research.ics.aalto.fi/software/asp/mingoR/`

the Newspaper, Routing Max, and Routing Min problems from [11], and the Disjunctive Scheduling problem from [4]. These problems were selected as they involve either reachability or disjunctivity. The original instances are revised to include real numbers. The objective functions of the optimization versions of HRP and JSP are to minimize the time of reaching some critical node and the ending time of some task, respectively. The optimization problems involve integers only. The experiments were run on a Linux cluster having 112 AMD Opteron and 120 Intel Xeon nodes with Linux 6.1. In each run, the memory is limited to 4GB and the cutoff time is set to 600 seconds.

In Table 1, we evaluate MINGO^r with different values of the parameter b, the bound used in the non-strict translation. The goal is to find a default setting for b. Table 1 suggests that $b = 10^{-6}$ is the best, considering the number of solved instances and the running time. We tested 100 random instances for each problem except Disjunctive Scheduling where we used the 10 instances given in [4]. Each instance was run 5 times and the average number of solved instances and running time are reported. We also include the average sizes of resulting *ground* programs in kilobytes to give an idea on the space complexity of the instances. It is also worth pointing out that none of the problems reported in Table 1 is solvable by existing ASP systems since they involve real variables and thus comparisons of MINGO^r with other ASP systems are infeasible.

Tables 2 and 3 provide comparisons of the translation-based ASP approach with native MIP approach, where both MINGO^r and CPLEX are run with default settings. For the HRP problem in Table 2, we tested 50 randomly generated graphs of 30 nodes for each *density* (the ratio of the number of edges to the number of edges in the complete graph). The results show that the instances with medium densities are unsolvable to CPLEX before the cutoff but MINGO^r can solve them in reasonable time. We also note that CPLEX performs better for the graphs of high densities. This is because the MIP program encodes the positions of nonadjacent nodes. For the JSP problem in Table 3, we tested 50 random instances for each number of tasks and MINGO^r is slower than CPLEX by roughly a order of magnitude but, in spite of this, scaling is similar.

In summary, some observations are in order. On one hand, the ASP(LC) language enables compact encodings in debt to its capability of expressing non-trivial logical relations. Thus some redundant information, such as the order of nodes in a cycle in HRP, can be left out in favor of computational performance. On the other hand, the translation of ASP(LC) programs into MIP is fully general and thus some unnecessary extra variables could be introduced. E.g., in the case of JSP, the structure of the translation is more complex than the native MIP encoding. As a consequence, the translation-based approach is likely to be slower due to extra time needed for propagation.

7 Related Work

Dutertre and de Moura [5] translate strict linear constraints into non-strict ones using a new variable δ in analogy to Lemma 1. However, the variable remains unbounded from above in their proposal. In contrast to this, an explicit upper bound is introduced by our translation. The bound facilitates computation: if the translated set of constraints has no solution ν with $\nu(\delta) > 0$ under a particular bound b, this result is conclusive and no

Benchmark	$b = 10^{-9}$		$b = 10^{-6}$		$b = 10^{-3}$		$b = 1$		$b = 10^{3}$		Size
	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time	
Disj. Scheduling	10	0.60	10	0.78	10	0.99	10	0.89	10	12.60	206
Ham. Routing	100	30.79	100	24.52	100	23.16	100	41.63	0	NA	155
Job Shop	100	9.76	100	9.56	100	25.98	100	14.61	10	66.08	387
Newspaper	100	22.64	100	21.61	93	77.90	100	40.76	0	NA	846
Routing Max.	100	0.11	100	0.14	100	0.25	100	0.55	100	0.69	7
Routing Min.	76	109.58	77	102.98	80	127.12	46	95.07	20	79.07	368

Table 1. The effect of the bound $b > 0$

Density	Decision		Optimization	
	MINGO^r	CPLEX	MINGO^r	CPLEX
10	0.03	0.01	0.07	0.01
20	0.05	0.01	0.12	0.01
30	0.92	NA	50.81	NA
40	41.62	NA	NA	NA
50	13.94	NA	NA	NA
60	64.91	NA	NA	NA
70	35.78	NA	NA	NA
80	8.02	95.40	NA	NA
90	181.33	24.74	NA	NA
100	146.18	13.88	NA	NA

Table 2. Hamiltonian Routing Problem

Tasks	Decision		Optimization	
	MINGO^r	CPLEX	MINGO^r	CPLEX
10	0.42	0.14	0.35	0.08
20	4.04	0.18	1.56	0.14
30	6.78	0.40	4.69	0.49
40	13.74	0.72	12.18	1.62
50	27.37	1.36	16.15	1.16
60	45.44	1.72	30.82	2.01
70	51.56	1.57	47.85	1.80
80	88.72	2.34	68.99	2.83
90	114.32	2.97	79.28	6.43
100	192.09	4.19	112.09	8.05

Table 3. Job Shop Problem

further computations are needed. Unbounded variables are problematic for typical MIP systems and thus having an upper bound for δ is important.

Given the extension of ASP(LC) programs with objective functions, MIP programs can be seen as a special case of ASP(LC) programs, i.e., for any MIP program P with an objective function (2) and constraints in (3), there is an ASP(LC) program P' whose objective function is (2) and whose rules simply list the constraints in (3) as theory atoms (facts in ASP terminology). Note that in this setting the non-strict translation of P' is identical to P, since P' involves non-strict constraints only.

In theory, all similar paradigms proposed in [1, 7, 12] cover real-valued constraints. Moreover, a recent ASP system CLINGCON [7] is implemented where constraints over integers are allowed in logic programs. But, to the best of our knowledge, there has not been any system that supports real-valued constraints nor integer-based objective functions. With the non-strict translation of ASP programs into mixed integer programs, we are able to implement these primitives in the context of ASP.

8 Conclusion and Future Work

In this paper, we generalize a translation from ASP(LC) programs to MIP programs so that linear constraints over real variables are enabled in answer set programming.

Moreover, we introduce integer objective functions to ASP(LC) language. These results extend the applicability of answer set programming. Finally, we compare the ASP approach with the native MIP approach and the results show that ASP extensions in question facilitate modeling and offer computational advantage at least for some problems. Our results suggest that MIP and ASP paradigms can benefit mutually. Efficient MIP formulations may be obtained by translating a compact ASP(LC) program. On the other hand, ASP language can be extended using MIP constraints and an objective function to deal with problems that are not directly solvable within standard ASP.

The future work will be focused on system development and experiments. E.g., we will study techniques to reduce the number of extra variables needed in translations and to stop the solving phase early (as soon as δ is positive). These goals aim at more efficient implementation of MINGO^{r}which also lacks a user-friendly front-end parser for the moment. Experiments that compare MINGO^{r}with other constraint logic programming systems [10] will also be conducted to provide insights into improving MINGO^r.

References

1. Balduccini, M.: Industrial-size scheduling with ASP+CP. In: LPNMR. (2011) 284–296
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Commun. ACM **54**(12) (2011) 92–103
3. Clark, K.L.: Negation as failure. Logics and Databases (1978) 293–322
4. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczynski, M.: The second answer set programming competition. In: LPNMR. (2009) 637–654
5. Dutertre, B., de Moura, L.M.: A fast linear-arithmetic solver for DPLL(T). In: CAV. (2006) 81–94
6. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in *gringo* series 3. In: LPNMR. (2011) 345–351
7. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: ICLP. (2009) 235–249
8. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP. (1988) 1070–1080
9. Grimes, D., Hebrard, E., Malapert, A.: Closing the open shop: Contradicting conventional wisdom. In: CP. (2009) 400–408
10. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. Journal of Logic Programming **19/20** (1994) 503–581
11. Liu, G., Janhunen, T., Niemelä, I.: Answer set programming via mixed integer programming. In: KR. (2012) 32–42
12. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. AMAI **53**(1-4) (2008) 251–287
13. Nguyen, M., Janhunen, T., Niemelä, I.: Translating answer-set programs into bit-vector logic. CoRR **abs/1108.5837** (2011)
14. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. AMAI **25**(3-4) (1999) 241–273
15. Niemelä, I.: Linear and integer programming modelling and tools. In Lecture Notes for the course on Search Problems and Algorithms, Helsinki University of Technology (2008)
16. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
17. You, J.H., Hou, G.: Arc-consistency + unit propagation = lookahead. In: ICLP. (2004) 314–328

Propositional Encoding of Constraints over Tree-Shaped Data

Alexander Bau [⋆] and Johannes Waldmann
HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany

Abstract. We present a functional programming language for specifying constraints over tree-shaped data. The language allows for Haskell-like algebraic data types and pattern matching. Our constraint compiler CO4 translates these programs into satisfiability problems in propositional logic. We present an application from the area of automated analysis of termination of rewrite systems, and also relate CO4 to Curry.

1 Motivation

The paper presents a high-level declarative language CO4 for describing constraint systems. The language includes user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order and polymorphic types. This language comes with a compiler that transforms a high-level constraint system into a satisfiability problem in propositional logic. This is motivated by the following.

Constraint solvers for propositional logic (SAT solvers) like Minisat [ES03] are based on the Davis-Putnam-Logemann-Loveland (DPLL) [DLL62] algorithm and extended with conflict-driven clause learning (CDCL) [SS96] and preprocessing. They are able to find satisfying assignments for conjunctive normal forms with 10^6 and more clauses in a lot of cases quickly. SAT solvers are used in industrial-grade verification of hardware and software.

With the availability of powerful SAT solvers, *propositional encoding* is a promising method to solve constraint systems that originate in different domains. In particular, this approach had been used for automatically analyzing (non-)termination of rewriting [KK04,ZSHM10,CGSKT12] successfully, as can be seen from the results of International Termination Competitions (most of the participants use propositional encodings).

So far, these encodings are written manually: the programmer has to construct explicitly a formula in propositional logic that encodes the desired properties. Such a construction is similar to programming in assembly language: the advantage is that it allows for clever optimizations, but the drawbacks are that the process is inflexible and error-prone.

[⋆] This author is supported by an ESF grant

This is especially so if the data domain for the constraint system is remote from the "sequence of bits" domain that naturally fits propositional logic. In typical applications, data is not a flat but hierarchical (e.g., using lists and trees), and one wants to write constraints on such data in a direct way.

Therefore, we introduce a constraint language CO4 that comes with a compiler to propositional logic. Syntactically, CO4 is a subset of Haskell [Jon03], including data declarations, case expressions, higher order functions, polymorphism (but no type classes). The advantages of re-using a high level declarative language for expressing constraint systems are: the programmer can rely on established syntax and semantics, does not have to learn a new language, can re-use his experience and intuition, and can re-use actual code. For instance, the (Haskell) function that describes the application of a rewrite rule at some position in some string or term can be directly used in a constraint system that describes a rewrite sequence with a certain property.

A constraint programming language needs some way of parameterizing the constraint system to data that is not available when writing the program. For instance, a constraint program for finding looping derivations for a rewrite system R, will not contain a fixed system R, but will get R as run-time input.

A formal specification of compilation is given in Section 2, and a concrete realization of compilation of first-order programs using algebraic data types and pattern matching is given in Section 3. In these sections, we assume that data types are finite (e.g., composed from `Bool`, `Maybe`, `Either`), and programs are total. We then extend this in section 4 to handle infinite (that is, recursive) data types (e.g., lists, trees), and partial functions. Note that a propositional encoding can only represent a finite subset of values of any type, e.g., lists of booleans with at most 5 elements, so partial functions come into play naturally.

We then treat in Section 5 briefly some ideas that serve to improve writing and executing CO4 programs. These are higher-order functions and polymorphism, as well as hash-consing, memoization, and built-in binary numbers.

Next, we give an application of CO4 in the termination analysis of rewrite systems: In Section 6 we describe a constraint system for looping derivations in string rewriting. We compare this to a hand-written propositional encoding [ZSHM10], and evaluate performance. The subject of Section 7 is the comparison of CO4 to Curry [Han11], using the standard N-Queens-Problem as a test case.

Our constraint language and compiler had been announced in short workshop contributions at *HaL 8* (Leipzig, 21 June 13) and *Haskell and Rewriting Techniques* (Eindhoven, 26 June 13). Here, we explain CO4 in more detail, and with fresh examples. Because of space restrictions, we still leave out some technicalities in Sections 2 and 3, and instead refer to the extended version [BW13].

2 Semantics of Propositional Encodings

In this section, we introduce CO4 syntax and semantics, and give the specification for compilation of CO4 expressions, in the form of an invariant (it should hold for all sub-expressions). When applied to the full input program, the specification implies that the compiler works as expected: a solution for the constraint system can be found via the external SAT solver. We defer discussion of our implementation of this specification to Section 3, and give here a more formal, but still high-level view of the CO4 language and compiler.

Evaluations on concrete data. We denote by $\mathbb{P}$ the set of expressions in the input language. It is a first-order functional language with algebraic data types, pattern matching, and global and local function definitions (using `let`) that may be recursive. The concrete syntax is a subset of Haskell. We give examples— which may appear unrealistically simple but at this point we cannot use higher-order or polymorphic features. These will be discussed in see Section 5.

```
data Bool = False | True
and2 :: Bool -> Bool -> Bool
and2 x y = case x of { False -> False ; True -> y }

data Maybe_Bool = Nothing | Just Bool
f :: Maybe_Bool -> Maybe_Bool -> Maybe_Bool
f p q = case p of
    Nothing -> Nothing
    Just x -> case q of
        { Nothing -> Nothing ; Just y -> Just (and2 x y) }
```

For instance, `f (Just x) Nothing` is an expression of $\mathbb{P}$, containing a variable `x`. We allow only *simple* patterns (a constructor followed by variables), and we require that pattern matches are *complete* (there is exactly one pattern for each constructor of the respective type). It is obvious that nested patterns can be translated to this form.

Evaluation of expressions is defined in the standard way: The domain of *concrete values* $\mathbb{C}$ is the set of data terms. For instance, `Just False` $\in \mathbb{C}$. A *concrete environment* is a mapping from program variables to $\mathbb{C}$. A *concrete evaluation function* $\mathsf{concrete\text{-}value} : E_{\mathbb{C}} \times \mathbb{P} \to \mathbb{C}$ computes the value of a concrete expression $p \in \mathbb{P}$ in a concrete environment $e_{\mathbb{C}}$. Evaluation of function and constructor arguments is strict.

Evaluations on abstract data. The CO4 compiler transforms an input program that operates on concrete values, to an *abstract program* that operates on *abstract values.* An abstract value contains propositional logic formulas that may contain free propositional variables. An abstract value represents a set of concrete values. Each assignment of the propositional values produces a concrete value.

We formalize this in the following way: the domain of abstract values is called $\mathbb{A}$. The set of assignments (mappings from propositional variables to truth values $\mathbb{B} = \{0, 1\}$) is called Σ, and there is a function $\mathsf{decode} : \mathbb{A} \times \Sigma \to \mathbb{C}$.

We now specify abstract evaluation. (The implementation is given in Section 3.) We use *abstract environments* $E_{\mathbb{A}}$ that map program variables to abstract values, and an *abstract evaluation function* $\mathsf{abstract\text{-}value} : E_{\mathbb{A}} \times \mathbb{P} \to \mathbb{A}$.

Allocators. As explained in the introduction, the constraint program receives known and unknown arguments. The compiled program operates on abstract values.

The abstract value that represents a (finite) set of concrete values of an unknown argument is obtained from an *allocator*. For a property $q : \mathbb{C} \to \mathbb{B}$ of concrete values, a q-allocator constructs an object $a \in \mathbb{A}$ that represents all concrete objects that satisfy q:

$$\forall c \in \mathbb{C} : q(c) \iff \exists \sigma \in \Sigma : c = \mathsf{decode}(a, \sigma).$$

We use allocators for properties q that specify c uses constructors that belong to a specific type. Later (with recursive types, see Section 4) we also specify a size bound for c. An example is an allocator for lists of booleans of length ≤ 4.

As a special case, an allocator for a singleton set is used for encoding a known concrete value. This *constant allocator* is given by a function $\mathsf{encode} : \mathbb{C} \to \mathbb{A}$ with the property that $\forall c \in \mathbb{C}, \sigma \in \Sigma : \mathsf{decode}(\mathsf{encode}(c), \sigma) = c$.

Correctness of constraint compilation. The semantical relation between an expression p (a concrete program) and its compiled version $\mathsf{compile}(p)$ (an abstract program) is given by the following relation between concrete and abstract evaluation:

Definition 1. *We say that $p \in \mathbb{P}$ is compiled* correctly *if*

$$\begin{aligned} \forall e \in E_{\mathbb{A}}\ \forall \sigma \in \Sigma : &\mathsf{decode}(\mathsf{abstract\text{-}value}(e, \mathsf{compile}(p)), \sigma) \\ &= \mathsf{concrete\text{-}value}(\mathsf{decode}(e, \sigma), p) \end{aligned} \tag{1}$$

Here we used $\mathsf{decode}(e, \sigma)$ as notation for lifting the decoding function to environments, defined element-wise by

$$\forall e \in E_{\mathbb{A}}\ \forall v \in \mathrm{dom}(e)\ \forall \sigma \in \Sigma : \mathsf{decode}(e, \sigma)(v) = \mathsf{decode}(e(v), \sigma).$$

Application of the Correctness Property. We are now in a position to show how the stages of CO4 compilation and execution fit together.

The top-level parametric constraint is given by a function declaration `main k u = b` where `b` (the *body*, a concrete program) is of type `Bool`. It will be processed in the following stages:

1. *compilation* produces an abstract program $\mathsf{compile}(b)$,
2. *abstract computation* takes a concrete parameter value $p \in \mathbb{C}$ and a q-allocator $a \in \mathbb{A}$, and computes the abstract value

$$V = \mathsf{abstract\text{-}value}(\{k \mapsto \mathsf{encode}(p), u \mapsto a\}, \mathsf{compile}(b))$$

3. *solving* calls the backend SAT solver to determine $\sigma \in \Sigma$ with $\mathsf{decode}(V, \sigma) =$ True. If this was successful,
4. *decoding* produces a concrete value $s = \mathsf{decode}(a, \sigma)$,
5. and optionally, *testing* checks $\mathsf{concrete\text{-}value}(\{k \mapsto p, u \mapsto s\}, b) =$ True.

The last step is just for reassurance against implementation errors, since the invariant implies that the test returns True. This highlights another advantage of re-using Haskell for constraint programming: one can easily check the correctness of a solution candidate.

3 Implementation of a Propositional Encoding

In this section, we give a realization for abstract values, and show how compilation creates programs that operate correctly on those values, as specified in Definition 1.

Encoding and Decoding of Abstract Values. The central idea is to represent an abstract value as a tree, where each node contains an encoding for a symbol (a constructor) at the corresponding position, and the list of concrete children of the node is a prefix of the list of abstract children (the length of the prefix is the arity of the constructor).

The encoding of constructors is by a sequence of formulas that represent the number of the constructor in binary notation.

We denote by F the set of propositional logic formulas. At this point, we do not prescribe a concrete representation. For efficiency reasons, we will allow some form of sharing. Our implementation (`satchmo-core`) assigns names to subformulas by doing the Tseitin transform [Tse83] on-the-fly, creating a fresh propositional literal for each subformula.

Definition 2. *The set of abstract values* $\mathbb{A}$ *is the smallest set with* $\mathbb{A} = \mathrm{F}^* \times \mathbb{A}^*$.

An element $a \in \mathbb{A}$ *thus has shape* $(\overrightarrow{f}, \overrightarrow{a})$ *where* $\overrightarrow{f}$ *is a sequence of formulas, called the* flags *of* a, *and* $\overrightarrow{a}$ *is a sequence of abstract values, called the* arguments *of* a. *Equivalently, in Haskell notation,*

```
data A = A { flags :: [F] , arguments :: [A] }
```

We introduce notation

- $\mathsf{flags} : \mathbb{A} \to \mathrm{F}^*$ *gives the flags of an abstract value*
- $\mathsf{flags}_i : \mathbb{A} \to \mathrm{F}$ *gives the i-th flag of an abstract value*
- $\mathsf{arguments} : \mathbb{A} \to \mathbb{A}^*$ *gives the arguments of an abstract value,*
- $\mathsf{argument}_i : \mathbb{A} \to \mathbb{A}$ *gives the i-th argument of an abstract value*

The sequence of flags of an abstract value encodes the number of its constructor. We use the following variant of a binary encoding: For each data type T with c constructors, we use as flags a set of sequences $S_c \subseteq \{0,1\}^*$ with $|S_c| = c$ and such that each long enough $w \in \{0,1\}^*$ does have exactly one prefix in S_c:

$$S_1 = \{\epsilon\}; \qquad \text{for } n > 1\text{:} \quad S_n = 0 \cdot S_{\lceil n/2 \rceil} \cup 1 \cdot S_{\lfloor n/2 \rfloor}$$

For example, $S_2 = \{0,1\}, S_3 = \{00, 01, 1\}, S_5 = \{000, 001, 01, 10, 11\}$. The lexicographic order of S_c induces a bijection $\mathsf{numeric}_c : S_c \to \{1, \ldots, c\}$.

The encoding function (from concrete to abstract values) is defined by

$$\mathsf{encode}_T(C(v_1, \ldots)) = (\mathsf{numeric}_c^-(i), [\mathsf{encode}_{T_1}(v_1), \ldots])$$

where C is the i-th constructor of type T, and T_j is the type of the j-th argument of C. Note that here, $\mathsf{numeric}_c^-(i)$ denotes a sequence of constant flags (formulas) that represents the corresponding binary string.

For decoding, we need to take care of extra flags and arguments that may have been created by the function merge (Definition 4) that is used in the compilation of `case` expressions.

We extend the mapping $\mathsf{numeric}_c$ to longer strings by $\mathsf{numeric}_c(u \cdot v) := \mathsf{numeric}_c(u)$ for each $u \in S_c, v \in \{0,1\}^*$. This is possible by the unique-prefix condition.

Given the type declaration `data Bool = False | True` the concrete value `True` can be represented by the abstract value $a_1 = ([x], [])$ and assignment $\{x = 1\}$, since `True` is the second (of two) constructors, and $\mathsf{numeric}_2([1]) = 2$. The same concrete value `True` can also be represented by the abstract value $a_2 = ([x, y], [a_1])$ and assignment $\{x = 1, y = 0\}$, since $\mathsf{numeric}_2([1, 0]) = 2$. This shows that extra flags and extra arguments are ignored in decoding.

We give a formal definition: for a type T with c constructors, $\mathsf{decode}_T((f, a), \sigma)$ is the concrete value $v = C_i(v_1, \ldots)$ where $i = \mathsf{numeric}_c(f\sigma)$, and C_i is the i-th constructor of T, and $v_j = \mathsf{decode}_{T_j}(a_j, \sigma)$ where T_j is the type of the j-th argument of C_i.

As stated, this is a partial function, since any of f, a may be too short. For this Section, we assume that abstract values always have enough flags and arguments for decoding, and we defer a discussion of partial decodings to Section 4.

Allocators for Abstract Values. Since we consider (in this section) finite types only, we restrict to *complete* allocators: for a type T, a complete allocator is an abstract value $a \in \mathbb{A}$ that can represent each element of T: for each $e \in T$, there is some σ such that $\mathsf{decode}_T(a, \sigma) = e$.

For some exemplary types, complete allocators are

type	complete allocator
`data Bool = False | True`	$a_1 = ([x_1], [])$
`data Ordering = LT | EQ | GT`	$a_2 = ([x_1, x_2], [])$
`data EBO = Left Bool | Right Ordering`	$a_3 = ([x_1], [([x_2, x_3], [])])$

where $x_1, \dots$ are (boolean) variables. We compute $\mathsf{decode}(a_3, \sigma)$ for $\sigma = \{x_1 = 0, x_2 = 1, x_3 = 0\}$): Since $\mathsf{numeric}_2([0]) = 1$, the top constructor is `Left`. It has one argument, obtained as $\mathsf{decode}_{\texttt{Bool}}(([x_2, x_3], []), \sigma)$. For this we compute $\mathsf{numeric}_2([1, 0]) = 2$, denoting the second constructor (`True`) of `Bool`. Thus, $\mathsf{decode}(a_3, \sigma) =$ `Left True`.

Compilation of Programs. In the following we illustrate the actual transformation of the input program (that operates on concrete values) to an abstract program (operating on abstract values).

Generally, compilation keeps structure and names of the program intact. For instance, if the original program defines functions f and g, and the implementation of g calls f, then the transformed program also defines functions f and g, and the implementation of g calls f.

Compilation of variables, bindings, and function calls is straightforward, and we omit details.

We deal now with pattern matches. They appear naturally in the input program, since we operate on algebraic data types. The basic plan is that *compilation removes pattern matches.* This is motivated as follows. Concrete evaluation of a pattern match (in the input program) consists of choosing a branch according to a concrete value (of the discriminant expression). Abstract evaluation cannot access this concrete value (since it will only be available after the SAT solver determines an assignment). This means that we cannot abstractly evaluate pattern matches. Therefore, they must be removed by compilation.

We restrict to pattern matches where patterns are *simple* (a constructor followed by variables) and *complete* (one branch for each constructor of the type).

Definition 3 (Compilation, pattern match).

Consider a pattern match expression e of shape ***`case`*** *d* ***`of`*** *$\{\dots\}$, for a discriminant expression d of type T with c constructors.*

We have $\mathsf{compile}(e) =$ ***`let`*** $x = \mathsf{compile}(d)$ ***`in`*** $\mathsf{merge}_c(\mathsf{flags}(x), b_1, \dots)$ *where x is a fresh variable, and b_i represents the compilation of the i-th branch.*

Each such branch is of shape $C\ v_1 \dots v_n \to e_i$, where C is the i-th constructor of the type T.

Then b_i is obtained as `let` $\{v_1 = \mathsf{argument}_1(x); \dots\}$ `in` $\mathsf{compile}(e_i)$.

We need the following auxiliary function that combines the abstract values from branches of pattern matches, according to the flags of the discriminant.

Definition 4 (Combining function). $\mathsf{merge} : F^* \times \mathbb{A}^c \to \mathbb{A}$ *combines abstract values so that* $\mathsf{merge}(\overrightarrow{f}, a_1, \dots, a_c)$ *is an abstract value* $(\overrightarrow{g}, z_1, \dots, z_n)$*, where*

- *number of arguments:* $n = \max(|\mathsf{arguments}(a_1)|, \dots, |\mathsf{arguments}(a_c)|)$
- *number of flags:* $|\overrightarrow{g}| = \max(|\mathsf{flags}(a_1)|, \dots, |\mathsf{flags}(a_c)|)$
- *combining the flags:*

$$\text{for } 1 \le i \le |\overrightarrow{g}|, \qquad g_i \leftrightarrow \bigwedge_{1 \le j \le c} (\mathsf{numeric}_c(\overrightarrow{f}) = j \to \mathsf{flags}_i(a_j)) \tag{2}$$

- *combining the arguments recursively:*

$$\text{for each } 1 \le i \le n, \qquad z_i = \mathsf{merge}(\overrightarrow{f}, \mathsf{argument}_i(a_1), \dots, \mathsf{argument}_i(a_c)).$$

Example 1. Consider the expression `case e of False -> u; True -> v`, where `e`,`u`,`v` are of type `Bool`, represented by abstract values $([f_e], []), ([f_u], []), ([f_v], [])$ with one flag an no arguments. The case expression is compiled into an abstract value $([f_r], [])$ where

$$\begin{aligned} f_r &= \mathsf{merge}_2([f_e], ([f_u], []), ([f_v], [])) \\ &= (\mathsf{numeric}_2(f_e) = 1 \to f_u) \land (\mathsf{numeric}_2(f_e) = 2 \to f_v) \\ &= (\overline{f_e} \to f_u) \land (f_e \to f_v) \end{aligned}$$

We refer to extend version [BW13] for the full specification of compilation, and proofs of correctness.

We mention already here one way of optimization: if all flags of the discriminant are constant (i.e., known during abstract evaluation, before running the SAT solver) then abstract evaluation will evaluate only the branch specified by the flags, instead of evaluating all, and merging the results. Typically, flags will be constant while evaluating expressions that only depend on the input parameter, and not on the unknown.

4 Partial encoding of Infinite Types

We discuss the compilation and abstract evaluation for constraints over infinite types, like lists and trees. Consider declarations

```
data N = Z | S N
double :: N -> N
double x = case x of { Z -> Z ; S x' -> S (S (double x')) }
```

Assume we have an abstract value a to represent `x`. It consists of a flag (to distinguish between `Z` and `S`), and of one child (the argument for `S`), which is another abstract value. At some depth, recursion must stop, since the abstract value is finite (it can only contain a finite number of flags). Therefore, there is a child with no arguments, and it must have its flag set to [False] (it must represent `Z`).

There is another option: if we leave the flag open (it can take on values False or True), then we have an abstract value with (possibly) a constructor argument missing. When evaluating the concrete program, the result of accessing a non-existing component gives a bottom value. This corresponds to the Haskell semantics where each data type contains bottom, and values like `S (S` $\perp$`)` are valid. To represent these values, we extend our previous definition to:

Definition 5. *The set of abstract values* $\mathbb{A}_\perp$ *is the smallest set with* $\mathbb{A}_\perp = \mathrm{F}^* \times \mathbb{A}_\perp^* \times \mathrm{F}$*, i.e. an abstract value is a triple of flags and arguments (cf. definition 2) extended by an additional* definedness constraint.

We write $\mathsf{def} : \mathbb{A}_\perp \to \mathrm{F}$ *to give the definedness constraint of an abstract value, and keep* flags *and* argument *notation of Definition 2.*

The decoding function is modified accordingly: $\mathsf{decode}_T(a, \sigma)$ for a type T with c constructors is $\perp$ if $\mathsf{def}(a)\sigma = \text{False}$, or $\mathsf{numeric}_c(\mathsf{flags}(a))$ is undefined (because of "missing" flags), or $|\,\mathsf{arguments}(a)|$ is less than the number of arguments of the decoded constructor.

The correctness invariant for compilation (Eq. 1) is still the same, but we now interpret it in the domain $\mathbb{C}_\perp$, so the equality says that if one side is $\perp$, then both must be. Consequently, for the application of the invariant, we now require that the abstract value of the top-level constraint under the assignment *is defined* and True. Abstract evaluation is extended to $\mathbb{A}_\perp$ by the following:

- *explicit bottoms*: a source expression `undefined` results in an abstract value $([], [], 0)$ (flags and arguments are empty, definedness is False)
- *constructors are lazy*: the abstract value created by a constructor application has its definedness flag set to True
- *pattern matches are strict*: the definedness flag of the abstract value constructed for a pattern match is the conjunction of the definedness of the discriminant with the definedness of the results of the branches, combined by merge.

5 Extensions for Expressiveness and Efficiency

We briefly present some enhancements of the basic CO4 language. To increase expressiveness, we introduce higher order functions and polymorphism. To improve efficiency, we use hash-consing and memoization, as well as built-in binary numbers.

More Haskell features in CO4. For formulating the constraints, expressiveness in the language is welcome. Since we base our design on Haskell, it is natural to include some of its features that go beyond first-order programs: higher order functions and polymorphic types.

Our program semantics is first-order: we cannot (easily) include functions as result values or in environments, since we have no corresponding abstract values for functions. Therefore, we instantiate all higher-order functions in a standard preprocessing step, starting from the main program.

Polymorphic types do not change the compilation process. The important information is the same as with monomorphic typing: the total number of constructors of a type, and the number (the encoding) of one constructor.

In all, we can use in CO4 a large part of the Haskell Prelude functions. CO4 just compiles their "natural" definition, e.g.,

```
and xs = foldl (&&) True xs ; a ++ b = foldr (:) b a
```

Memoization. We describe another optimization: in the abstract program, we use memoization for all subprograms. That is, during execution of the abstract program, we keep a map from (function name, argument tuple) to result. Note that arguments and result are abstract values. This allows to write "natural" specifications and still get a reasonable implementation.

For instance, The textbook definition of the lexicographic path order $>_{lpo}$ (cf. [BN98]) defines an order over terms according to some precedence. Its textbook definition is recursive, and leads to an exponential time algorithm, if implemented literally. By calling $s >_{lpo} t$ the algorithm still does only compare subterms of s and t, and in total, there are $|s| \cdot |t|$ pairs of subterms, and this is also the cost of the textbook algorithm with a memoizing implementation.

For memoization we frequently need table lookups. For fast lookups we need fast equality tests (for abstract values). We get these by *hash-consing*: abstract constructor calls are memoized as well, so that abstract nodes are globally unique, and structural equality is equivalent to pointer equality.

Memoization is awkward in Haskell, since it transforms pure functions into state-changing operations. This is not a problem for CO4 since this change of types only applies to the abstract program, and thus is invisible on the source level.

Built-in data types and operations. Consider the following natural definition:

```
not a   = case a of { False -> True ; True -> False }
```

Abstract values for `a`, `b`, and output, contain one flag each (and no arguments). CO4 will compile `not` in such a way that a fresh propositional variable is allocated for the output, and then emit two CNF clauses that assert the bi-implication between input and output (by Definition 4). This fresh variable is actually not necessary since we can invert the polarity of the input literal directly. To achieve this, Booleans and (some of) their operations are handled specially by CO4.

Similarly, we can model binary numbers as lists of bits:

```
data [] a = [] | a : [a] ; data Nat = Nat [ Bool ]
```

An abstract value for a k-bit number then is a tree of depth k. Instead of this, we provide built-in data types $\texttt{Nat}_k$ that represent a k-bit number as one abstract node with k flags, and no arguments. These types come with standard arithmetical and relational operations.

We remark that a binary propositional encoding for numbers is related to the “sets-of-intervals” representation that a finite domain (FD) constraint solver would typically use. A partially assigned binary number, e.g., $[*, 0, 1, *, *]$, also represents a union of intervals, here, $[4..7] \cup [20..23]$. Assigning variables can be thought of as splitting intervals. See Section 7 an application of CO4 to a typical FD problem.

6 Case study: Loops in String Rewriting

We use CO4 for compiling constraint systems that describe looping derivations in rewriting. W make essential use of CO4’s ability to encode (programs over) unknown objects of algebraic data types, in particular, of lists of unknown lengths, and with unknown elements.

The application is motivated by automated analysis of programs. A loop is an infinite computation, which may be unwanted behaviour, indicating an error in the program’s design. In general, it is undecidable whether a rewriting system admits a loop. Loops can be found by enumerating finite derivations.

Our approach is to write the predicate “the derivation d conforms to a rewrite system R and d is looping” as a Haskell function, and solve the resulting constraint system, after putting bounds on the sizes of the terms that are involved.

Previous work uses several heuristics for enumerations resp. hand-written propositional encodings for finding loops in string rewriting systems [ZSHM10].

We compare this to a propositional encoding via CO4. We give here the type declarations and some code examples. Full source code is available from `https://github.com/apunktbau/co4/blob/master/CO4/Test/Loop.standalone.hs`.

In the following, we show the data declarations we use, and give code examples.

- We represent symbols as binary numbers of flexible width, since we do not know (at compile-time) the size of the alphabet: `type Symbol = [ Bool ]`.
- We have words: `type Word = [Symbol]` , rules: `type Rule = (Word, Word)`, and rewrite systems `type SRS  = [Rule]`.
- A rewrite step $(p{+\!\!+}l{+\!\!+}s) \to_R (p{+\!\!+}r{+\!\!+}s)$, where rule (l, r) is applied with left context p and right context s, is represented by `Step p (l,r) s` where

  ```
  data Step = Step Word Rule Word
  ```

- a derivation is a list of steps: `type Derivation = [Step]`, where each step uses a rule from the rewrite system, and consecutive steps fit each other.
- a derivation is looping if the output of the last step is a subword of the input of the first step

  ```
    constraint :: SRS -> Looping_Derivation -> Bool
    constraint srs (Looping_Derivation pre d suf) =
      conformant srs d && eqWord (pre ++ start d ++ suf) (result d)
  ```

  This is the top-level constraint. The rewrite system `srs` is given at run-time. The derivation is unknown. An allocator represents a set of derivations with given maximal length (number of steps) and width (length of words).

Overall, the complete CO4 code consists of roughly 100 lines of code. The code snippets above indicate that the constraint system literally follows the textbook definitions. E.g., note the list-append $({+\!\!+})$ operators in `constraint`.

In contrast, Tyrolean Termination Tool 2 (TTT2, version 1.13) contains a hand-written propositional encoding for (roughly) the same constraint (`ttt2/src/processors/src/nontermination/loopSat.ml`) consisting of roughly 300 lines of (non-boilerplate) code. The TTT2 implementation explicitly allocates propositional variables (this is implicit in CO4), and explicitly manipulates indices (again, this is implicit in our ${+\!\!+}$).

Table 1 compares the performance of our implementation to that of TTT2 on some string rewriting systems of the Termination Problems Data Base collection. We restrict the search space in both tools to derivations of length 16 and words of length 16. All test were run on a Intel Core 2 Duo CPU with 2.20GHz and 4GB RAM.

We note that CO4 generates larger formulas, for which, in general, MiniSat needs more time to solve. There are rare cases where CO4's formula is solved faster.

Table 1: Finding looping derivations in rewrite systems.

	`Gebhardt/ 03`		`Gebhardt/ 08`		`Zantema_04/ z042`		`Zantema_06/ loop1`	
	CO4	TTT2	CO4	TTT2	CO4	TTT2	CO4	TTT2
#vars	132232	23759	132168	23696	248990	32180	132024	21880
#clauses	448543	39541	448351	39445	854949	50150	447935	35842
solving	192s	10s	11s	30s	7.5s	3s	8.3s	4s

7 A Comparison to Curry

We compare the CO4 language and implementation to that of the functional logic programming language Curry [Han11], and its PAKCS implementation (using the SICSTUS prolog system).

A common theme is that both languages are based on Haskell (syntax and typing), and extend this by some form of non-determinism, so the implementation has to realize some form of search.

In Curry, nondeterminism is created lazily (while searching for a solution). In CO4, nondeterminism is represented by additional boolean decision variables that are created beforehand (in compilation).

The connection from CO4 to Curry is easy: a CO4 constraint program with top-level constraint `main :: Known -> Unknown -> Bool` is equivalent to a Curry program (query) `main k u =:= True where u free`.

In the other direction, it is not possible to translate a Curry program to a CO4 program since it may contain locally free variables, a concept that is currently not supported in CO4. For doing the comparison, we restrict to CO4 programs.

Example 2. We give an example where the CO4 strategy seems superior: the n queens problem.

We compare our approach to a Curry formulation (taken from the PAKCS online examples collection) that uses the CLPFD library for finite-domain constraint programming. Our CO4 formulation uses built-in 8-bit binary numbers (Section 5) but otherwise is a direct translation. Note that with 8 bit numbers we can handle board sizes up to 2^7: we add co-ordinates when checking for diagonal attacks.

Table 2 shows the runtimes on several instances of the n queens problem. CO4's runtime is the runtime of the compiled program in addition to the runtime of the SAT-solver. The runtimes for PAKCS were measured using the `:set +time` flag after compiling the Curry program in the PAKCS evaluator.

Fig. 1: Two approaches to solve the n queens problem

CO4 source code

```
constraint n xs =
     all (\ x -> le (nat8 1) x
              && le x n ) xs
  && all_safe xs

all_safe xs = case xs of
  []    -> True
  x:xs' -> safe x xs' (nat8 1)
        && all_safe xs'

safe x ys p = case ys of
  []      -> True
  y : ys' ->
       no_attack x y p
    && safe x ys' (increment p)

no_attack x y p =
  neq x y && neq (add x p) y
          && neq x (add y p)

le          = leNat8
neq a b     = not (eqNat8 a b)
add         = plusNat8
increment x = add x (nat8 1)
```

Curry source code

```
import CLPFD

queens options n l =
       gen_vars n =:= l &
       domain l 1 (length l) &
       all_safe l &
       labeling options l

all_safe [] = success
all_safe (q:qs) = safe q qs 1
                & all_safe qs

safe _ [] _ = success
safe q (q1:qs) p = no_attack q q1 p
                 & safe q qs (p+#1)

no_attack q1 q2 p = q1 /=# q2
                  & q1 /=# q2+#p
                  & q1 /=# q2-p

gen_vars n = if n==0
             then []
             else var : gen_vars (n-1)
                  where var free
```

Table 2: Time for finding one solution of the n queens problem

n	8	12	16	20	24	32	64	128
CO4	0.08s	0.16s	0.31s	0.57s	0.73	1.59s	10.8s	53.1s
Curry/PAKCS	0.02s	0.13s	0.43s	8.54s	>10m	>10m	>10m	>10m

The PAKCS software also includes an implementation of the n queens problem that does not use the CLPFD library. As this implementation already needs 6 seconds to solve a $n = 8$ instance, we omit it in the previous comparison.

8 Discussion

In this paper we described the CO4 constraint language and compiler that allows to write constraints on tree-shaped data in a natural way, and to solve them via propositional encoding.

We presented the basic ideas for encoding data and translating programs, and gave an outline of a correctness proof for our implementation.

We gave an example where CO4 is used to solve an application problem from the area of termination analysis. This example shows that SAT compilation has advantages w.r.t. manual encodings.

We also gave an experimental comparison between CO4 and Curry, showing that propositional encoding is an interesting option for solving finite domain (FD) constraint problems.

Work on CO4 is ongoing. Our immediate goals are, on the one hand, to reduce the size of the formulas that are built during abstract evaluation, and on the other hand, to extend the source language with more Haskell features (e.g., type classes).

References

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, New York, NY, USA, 1998.

[BW13] Alexander Bau and Johannes Waldmann. Propositional encoding of constraints over tree-shaped data. *CoRR*, abs/1305.4957, 2013.

[CGSKT12] Michael Codish, Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Sat solving for termination proofs with recursive path orders and dependency pairs. *J. Autom. Reasoning*, 49(1):53–93, 2012.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[Han11] Michael Hanus. Functional logic programming: From theory to curry. Technical Report, Christian-Albrechts-Universität Kiel, 2011.

[Jon03] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.

[KK04] Masahito Kurihara and Hisashi Kondo. Efficient BDD encodings for partial order constraints with application to expert systems in software verification. In Robert Orchard, Chunsheng Yang, and Moonis Ali, editors, *IEA/AIE*, volume 3029 of *Lecture Notes in Computer Science*, pages 827–837. Springer, 2004.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[Tse83] G.S. Tseitin. On the complexity of derivation in propositional calculus. In Jörg Siekmann and Graham Wrightson, editors, *Automation of Reasoning*, Symbolic Computation, pages 466–483. Springer Berlin Heidelberg, 1983.

[ZSHM10] Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *SOFSEM*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.

Coverage Driven Test Generation and Consistency Algorithm

Jomu George Mani Paret and Otmane Ait Mohamed

Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve West, Montreal, Quebec, H3G 1M8, Canada
{jo_pare,ait}@ece.concordia.ca

Abstract. Coverage driven test generation (CDTG) is an essential part of functional verification where the objective is to generate input stimuli that maximize the functional coverage of a design. CDTG techniques analyze coverage results and adapt the stimulus generation process to improve the coverage. One of the important component of CDTG based tools is the constraint solver. The efficiency of the verification process depends on the performance of the solver. The speed of the solver can be increased if inconsistent values can be removed from the domain of input variables. In this paper, we propose a new efficient consistency algorithm called GACCC-op (generalized arc consistency on conjunction of constraints-optimized) which can be used along with the constraint solver of CDTG tools. With the proposed algorithm, the time to generate solutions was reduced by 19% in the case of 3-Sat instances.

1 Introduction

In electronic design automation (EDA), functional verification is the task of verifying whether the hardware design conforms the required specification. This is a complex task which consumes the majority of the time and effort in most of the electronic system design projects. Many studies show that up to 70% of design development time and resources are spent on functional verification[11]. In coverage based test generation techniques, coverage tools are used side by side with a stimulus generator (constraint solver) in order to assess the progress of the verification plan during the verification cycle. Coverage analysis allows for the modification of the directives for the stimulus generators and the targeting of areas of the design that are not covered well. This process of adapting the directives of stimulus generator, according to the feedback based on coverage reports, is called Coverage Driven Test Generation(fig 1). CDTG is time consuming and an exhaustive process, but essential for the completion of the verification cycle.

The most important component in CDTG is the constraint solver. The efficiency of a CDTG is heavily dependent on the constraint solver. The stimulus generation methods of CDTGs are similar to a constraint satisfaction problem (CSP). But the CSPs arising from stimulus generation are different from typical CSPs [9]. One striking difference is the existence of variables with huge domains. Another difference is the requirement to produce multiple different solutions, distributed uniformly, for the same CSP. Hence available general purpose constraint solvers cannot be used along with CDTG tools.

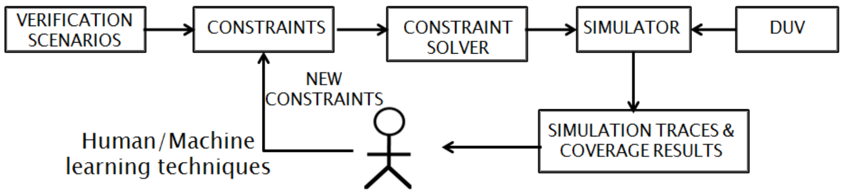


Fig. 1. Coverage Driven Test Generation Technique

Let us look at one example involving verification of the floating-point unit present in microprocessors. Stimulus generation for floating-point unit verification involves targeting corner cases, which can often be solved only through complex constraint solving. Hence the main task of the constraint solver is to generate a set of input stimulus that comprises a representative sample of the entire space, taking into account the many corner cases. Consider a floating point unit with two input operands. This potentially yields 400 (20^2) cases that must be covered, assuming 20 major FP instruction types (e.g. +/-zero, +/-min denorm). With four floating point instructions (addition, subtraction, division and multiplication) there is about 1600 cases to be covered. The probability that a CDTG tool will generate a sequence that covers a particular combination is very low (1:2^{17})[12]. Hence a CDTG tool will take many hours to generate the input stimuli required to attain the needed coverage.

Certain values in the domain of input variables of the constraints used for stimulus generation cannot be part of the solution. The inconsistent values can be found out by consistency search and can be removed from the domain of input variables. If the reduced domain is given to the constraint solver of CDTG, then the solutions for CSP can be generated in less time and with reduced memory consumption. In this paper, we propose a consistency search algorithm which can be used along with CDTG tools, to reduce the domain of input variables. The remainder of this paper is organized as follows. We will explain some of the related work in Section 2. Section 3 describes the proposed consistency algorithm. Finally, we present our experimental results in Section 4, and give some concluding remarks and future work in Section 5.

2 Related work

Existing research in CDTG have been focused on improving the input stimuli generated by CDTG tools. All high-end hardware manufacturers use CDTG to produce input stimulus. Some manufacturers of less complex designs rely on electronic design automation (EDA) tool vendors (e.g.Cadence, Mentor Graphics and Synopsys) for their stimulus generation needs. Those EDA tools, in turn, are based on internally developed constraint solvers [13]. Others such as Intel [15], adapt external off-the-shelf solvers to the stimulus generation problem. Some manufacturers such as IBM rely on proprietary constraint solvers developed in-house to solve this problem [12].

One approach for solving CSP is based on removing inconsistent values from the domain of variables till the solution is obtained. These methods are called consistency techniques. The most widely used consistency technique is called arc consistency (AC). The arc consistency algorithms are divided into two categories: coarse-grained algorithms and fine-grained algorithms. Coarse grained algorithms are algorithms in which the removal of a value from the domain of a variable will affect all other variables in the problem. The first consistency algorithms AC-1[14] and AC-3[14] belong to this category. These two consistency algorithms are succeeded by AC2000 [8], AC2001-OP [4], AC3-OP [3] and AC3d [10].

Fine grained consistency algorithms are algorithms in which removal of a value from the domain of a variable 'X' will affect only other variables which are related to the variable 'X'. Since only the variables that are affected by change in domain value are revisited, this algorithm is faster than coarse grained algorithms. Algorithms AC-4 [16], AC4-OP [2], AC-5 [17] and AC-6 [5] belong to this category. AC-7[6] is an algorithm developed based on AC-6. It uses the knowledge about the constraint properties to reduce the cost of consistency check.

All the above algorithms are developed for binary constraints. GAC-scheme [7] is a consistency algorithm developed for n-arity (n variables are there in the constraint) constraints. It is the extension of AC-7 for n-arity constraints. Conjunctive Consistency [7] enforces GAC-scheme on conjunctions of constraints. We chose GAC scheme on conjunction of constraints for our purpose because:

1. We need to eliminate as much invalid domain values as possible. This can be done by conjunction of constraints.
2. GAC scheme do not require any specific data structure.
3. The constraints used in CDTG can have more than two variables and GAC-scheme can handle constraint of n-arity.
4. The constraints used in CDTG are not of a fixed type and GAC-scheme can be used with any type of constraints.

3 Consistency Algorithm

3.1 Preliminaries

Tuple: A tuple τ on an ordered set of variables is an ordered list which contains values for all the variables. X(τ) represents the set of variables in the tuple τ.

Constraint: X(C_i) represents the set of variables in the constraint C_i. A constraint C_i on an ordered set of variables gives the list of allowed tuples for the set of variables.

Constraint Network: A constraint network is defined as a tuple N = ⟨X,D,C⟩ where

X is a set of n variables X = $\{x_1,\ldots, x_n\}$
D is a finite set of domains for the n variables = {D(x_1),. . ., D(x_n)}
C is a set of constraints between variables=$\{C_1,\ldots,C_k\}$
where n and k are non zero positive integers.

Valid Tuple: The value of variable x in a tuple τ is denoted by τ[x]. A tuple τ on C_i is valid iff $\forall$x $\in$X(C_i), τ[x] $\in$D(x) and τ satisfies the constraint C_i.

Support: If a $\in$D(x_i) and τ be a valid tuple on C_j, then τ is called a support for (x_i, a) on C_j.

Arc Consistency: A value a $\in$D(x_i) is consistent with C_i iff x_i $\in$X(C_i) and $\exists\tau$ such that τ is a support for (x_i, a) on C_i. C_i is arc consistent iff $\forall x_i$ $\in$X(C_i),D(x_i)$\neq$0 and $\forall$a $\in$D(x_i), a is consistent with C_i.

Generalized Arc Consistency of a network: A CSP is generalized arc consistent iff $\forall C_i \in C$ is arc consistent.

Conjunctive Consistency: If X(S_j)=X(C_1)$\cup\ldots\cup$X(C_k) where X(C_i)= set of variables in C_i, then S_j is conjunctively consistent iff $\forall$a $\in$D(x_k), x_k $\in$X(S_j) and there exists a tuple τ such that a=$\tau[x_k]$ and τ is a support $\forall x_k$.

Conjunctive Consistency of a network: Let P=<X, D, S> be a constraint network. P is conjunctive consistent network iff $\forall S_j \in S$ is conjunctive consistent.

3.2 GACCC

In GACCC[7], first a variable in a conjunction of constraint is selected and the variable will be assigned a value from its domain. The algorithm will generate tuples in lexicographical order (the selected variable value will not change) and check whether the tuple satisfies the constraint. The tuples are generated until all the tuples are generated or a tuple which satisfies the constraint is generated. If there is no tuple which satisfies the constraint for the selected variable value, then that variable value is inconsistent and removed from the variable domain. The process is repeated for all the domain values of the selected variable, then for all the variables in the constraint and for all the constraints in the constraint network.

To illustrate the idea discussed above, let us consider the following CSP: set of variables $X = \{m, n, o, p, q\}$, domain of the variables D(m)={1, 2}, D(n)={2, 3}, D(o)={1, 2}, D(p)={1, 3}, D(q)={2, 3} and the constraints $C1 : m{+}n{+}o{+}p = 7$ and $C2 : m{+}o{+}q = 9$. The consistency search (for conjunction of constraints) for $m = 1$ has to go through 16 tuples to find out that value is not consistent because each of the remaining variables (n, o, p, q) has two variables in the domain.

3.3 Intuitive Idea of GACCC-op

In consistency check, if any one constraint is not satisfied, the tuple generated is inconsistent with the conjunction set. We can reduce the consistency search by using this property. Initially for a given variable, we consider the constraint with lowest number of variables and contains the specified variable. We generate tuples for the above constraint and search for consistency. If the tuple generated for the smallest constraint is not consistent then all the tuples generated for the conjunction of constraints are also not consistent. If the tuple generated for the smallest constraint is consistent, then only we need to generate the tuples for the conjunction of constraints (The tuple generated for conjunction of constraints should contain the tuple which is consistent with the smallest constraint). Since the number of variables in the smallest constraint is less when compared to tuple for conjunction of constraints, consistency can be checked in less number of iterations.

In the above CSP, $C2$ is the smallest constraint in the set, which has 3 variable and the variable m. Consistency check is first performed on this constraint. In 4(2x2) iterations we can find that $m = 1$ is inconsistent with the constraint $C2$. Hence m=1 is inconsistent for the conjunction of constraints. The tuples for a variable in conjunction of constraints is generated only if the smallest constraint containing the variable is satisfied by the tuple. Consider another set of constraints $C3 : m+n+o+p = 8$ and $C4 : m+o+q = 6$. By GACCC we have to generate 8 tuples to find a consistent tuple. By using the new algorithm we need only 5(4 iterations for $C3$ and 1 for conjunction of $C3$ and $C4$) iterations to find the tuple which satisfies the constraints. So by using the proposed algorithm consistency check can be finished in less number of iteration when compared to GACCC.

So the difference between GACCC and GACCC-op are as follows:

1. In GACCC the support list is made by using some existing variable order scheme. In GACCC-op we propose a new variable ordering scheme in which the consistency search start with the variable, which is present in the constraint with the lowest arity and has the largest number of domain values.
2. In GACCC during consistency search of a domain value of a variable, the tuples generated will contain all the variable in the conjunction set. In GACCC-op the consistency search for a variable x will begin with tuples which contain only variables from the smallest constraint(C_s)(C_s should contain the variable x). If there is a tuple which satisfies the constraint C_s, only then GACCC-op generates tuples with all the variable in the conjunction set.

3.4 GACCC-op

Let us start the discussion with the main program (Algorithm 1). First the data structures (lastSc, supportlist, deletionlist and Sclast) must be created and initialized. Sclast, supportlist, deletionlist and lastSc are initialized in such a way that

1. Sclast contains the last tuple returned by the function **SeekValidSupportSet** as a support for variable value
2. supportlist contains all tuples that are support for variable value
3. deletionlist contains all variable values that are inconsistent
4. lastSc is the last tuple returned by the function **SeekValidSupport** as a support for variable value

Then for each set of constraints, for each variable present in the constraints, all the domain values of the variable are put in supportlist. The domain values of

Algorithm 1 PROPOSED CONSISTENCY Algorithm

```
 1: for each constraint set do
 2:    for each variable in set do
 3:       for each domain value of variable do
 4:          Add to support stream(S,y,b)
 5:       end for
 6:    end for
 7: end for
 8: while support stream $\neq$ nil do
 9:    $\sigma$ = SeekInferableSupport(S,y,b)
10:    if $\sigma$ = nil then
11:       c = smallest constraint containing variable y
12:       while found soln || checked all tuples do
13:          $\sigma*$ = lastSc(C,y,b)
14:          if $\sigma*$ = nil then
15:             LOOP2: $\sigma*$ = SeekValidSupport (C,y,b,$\sigma*$)
16:             if $\sigma*$ = nil then
17:                DeletionStream (y,b)
18:             else
19:                if variables in all the constraints are same then
20:                   Add to Sclast(S,y,b)
21:                else
22:                   Add to lastSc(C,y,b)
23:                   go to LOOP1
24:                end if
25:             end if
26:          else
27:             if Sclast(S,y,b)$\neq$ nil then
28:                $\sigma**$ = Sclast(S,y,b)
29:                go to LOOP1
30:             else
31:                $\sigma**$ = nil
32:             end if
33:          end if
34:          LOOP1: $\lambda$* = SeekValidSupportSet(S,y,b,$\sigma**$)
35:          if $\lambda$*$\neq$ nil then
36:             Add to Sclast(S,y,b)
37:          else
38:             go to LOOP2
39:          end if
40:       end while
41:    end if
42: end while
```

the variables in a conjunction set are added to supportlist using the following heuristics:

1. Find the lowest arity constraint(C_l) in the conjunction set.
2. Find a variable (x_l) where the variable and C_l is not added to the list, the variable is in C_l, has the highest number of domain values.
3. Add the domain values of the selected variable (x_l), variable and the constraint to the list.
4. Repeat step 2 until all the variables in the constraint C_l are considered.
5. If there is any variable to be added to the list from the conjunction set, then find the next highest arity constraint and repeat step 2.

This supportlist is used to find the support(support is a tuple which satisfies the constraint) for each variable value in the constraint set. For each value in supportlist the algorithm will try to find a valid support by using the function **SeekInferableSupport**. Function **SeekInferableSupport** checks whether an already checked tuple is a support for (y,b). If there is no valid support to be inferred then we will search for a valid support.

For every value 'b', for a variable 'y' in X(C), lastSc(C,y,b) is the last tuple returned by **SeekValidSupport** as a support for (y,b) if **SeekValidSupport**(C,y,b) has already been called or empty otherwise. The above two functions help to avoid checking several times whether the same tuple is a support for the constraint or not. If the search is new we look for support from the first valid tuple.

If no valid tuple is found then the variable value is not consistent with the constraint. Hence it is not consistent with constraint set. This variable value will be deleted from the domain of the variable by the function **DeletionStream**(y,b).

Algorithm 2 SeekInferableSupport

```
1: SeekInferableSupport (in S:constraint; in y:variable; in b:value):tuple
2: while support stream $\neq$ nil do
3:   if Sclast(var(S,y),$\tau$[y]) = b then
4:     zigma = Sclast(S,y,b)
5:   else
6:     zigma = nil
7:   end if
8:   return zigma
9: end while
```

If a tuple is returned by lastSc(C,y,b), we will check for Sclast(S,y,b). Sclast(S,y,b) is the last tuple returned by **SeekValidSupportSet** as a support for (S,y,b) if **SeekValidSupportSet** has already been called or empty otherwise. If a tuple is returned we start the search for support for conjunction constraint set from that tuple, else we will start search from the first valid tuple for the conjunction set, with variables in constraint C has the values of the tuple from lastSc(C,y,b). If the **SeekValidSupportSet** returns empty then we will call function **SeekValidSupport** and repeat the process until a valid tuple for the for conjunction constraint set is found or the lastSc(C,y,b) returns empty. If the lastSc(C,y,b) returns empty then the variable value is deleted the function **DeletionStream**(y,b). The above processes will be repeated until both the deletionlist and supportlist are empty.

The function **SeekInferableSupport**(Algorithm 2) ensures that the algorithm will never look for a support for a value when a tuple supporting this value has already been checked. The idea is to exploit the property: "If (y,b) belongs to a tuple supporting another value, then this tuple also supports (y,b)".

Algorithm 3 SeekValidSupport

```
1: SeekValidSupport (in C:constraint; in y:variable; in b:value; in $\tau$:tuple):tuple
2: if $\tau \neq$ nil then
3:    zigma = NextTuple(C,y,b,$\tau$)
4: else
5:    zigma = FirstTuple(C,y,b)
6: end if
7: zigma1 = SeekCandidateTuple(C,y,b,$\tau$)
8: solution found = false
9: while (zigma1 $\neq$ nil) and (not solution found) do
10:    if zigma1 satisfies constraint C then
11:       solution found = true
12:    else
13:       zigma1= NextTuple(C,y,b,zigma1)
14:       zigma1 = SeekCandidateTuple(C,y,b,zigma1)
15:    end if
16:    return zigma1
17: end while
```

After the function **SeekInferableSupport** fails to find any previously checked tuple as a support for (y,b) on the constraint C, the function **SeekValidSupport** (Algorithm 3) is called to find a new support for (y,b). But the function has to avoid checking tuples which are already checked. This is taken care by using the function **SeekCandidateTuple**. The function **NextTuple** will generate new tuples in a lexicographical order which can be a valid support for the constraint variable value.

Function **SeekCandidateTuple**(C,y,b,τ)(Algorithm 4) returns the smallest candidate greater than or equal to τ. For each index from 1 to $|X(C)|$ **SeekCandidateTuple** verifies whether τ is greater than lastSc (λ). If τ is smaller than λ, the search moves forward to the smallest valid tuple following τ, else to the valid tuple following λ. When the search moves to the next valid tuple greater than τ or λ, some values before the index may have changed. In this cases we again repeats the previous process to make sure that we are not repeating a previously checked tuple.

Algorithm 4 SeekCandidateTuple

```
1: SeekCandidateTuple (**in** C:constraint; **in** y:variable; **in** b:value; **in** $\tau$:tuple):tuple
2: k = 1
3: **while** ($\tau \neq$ nil) **and** (k$\leq X(C)$) **do**
4:    **if** lastc(var(C,k),$\tau$[k])$\neq$ nil **then**
5:       $\lambda$ = lastSc(var(C,k),$\tau$[k])
6:       split = 1
7:       **while** $\tau$[split] = $\lambda$[split] **do**
8:          split = split+1
9:       **end while**
10:      **if** $\tau$[split] < $\lambda$[split] **then**
11:         **if** split < k **then**
12:            ($\tau$,k')= **NextTuple**( C,y,b,$\lambda$)
13:            k = k'+1
14:         **else**
15:            ($\tau$,k')= **NextTuple**( C,y,b,$\lambda$)
16:            k = min(k'-1, k)
17:         **end if**
18:      **end if**
19:   **end if**
20:   k = k+1
21: **end while**
22: **return** $\tau$
```

Algorithm 5 SeekValidSupportSet

```
1: SeekCandidateTuple (**in** S:constraint set; **in** y:variable; **in** b:value; **in** $\tau$:tuple):tuple
2: **if** $\tau\neq$ nil **then**
3:    zigma = **NextTuple**(S,y,b,$\tau$,$\theta$)
4: **else**
5:    zigma = **FirstTuple**(S,y,b)
6: **end if**
7: zigma1 = **SeekCandidateSet**(S,y,b,$\tau$,$\theta$)
8: solution found = **false**
9: **while** (zigma1 $\neq$ nil) **and** (not solution found) **do**
10:   **if** zigma1 satisfies constraint set S **then**
11:      solution found = **true**
12:   **else**
13:      zigma1= **NextTuple**(S,y,b,zigma1,$\theta$)
14:      zigma1 = **SeekCandidateSet**(D,y,b,zigma1,$\theta$)
15:   **end if**
16:   **return** zigma1
17: **end while**
```

The function **SeekValidSupportSet**(Algorithm 5) is called to find a new support for (y,b) on the conjunction of constraints. But the function has to avoid checking tuples which are already checked. This is taken care by using the function **SeekCandidateSet**. This function is similar to the function **SeekCandidateTuple**. The function **SeekCandidateSet** returns the smallest tuple which is a support of the conjunction of constraints.

Algorithm 6 DeletionStream

```
1: SeekCandidateTuple (in y:variable; in b:value)
2: if Sclast(var(C,y),$\tau$[y])= b then
3:     Add to supportlist (S,(var(C,x)),a) where x$\neq$ y and $\tau$[x]=a
4:     delete $\lambda$ from Sclast
5: end if
```

If there is no support for a variable value, then that variable value is deleted from the variable domain by the function **DeletionStream**(Algorithm 6). The function also checks whether any tuple in Sclast contains the variable value. If there is such a tuple, then all the variable value in the tuple is added to supportlist to find new support.

3.5 Heuristic for generating Conjunction set

The CSPs associated with the verification scenarios have large number of constraints, large domain for each input variables and many of the constraints have the same variables. The pruning capability by consistency search can be increased, by combining/conjuncting a large number of constraints together. If a large number of constraints are conjuncted, the variables in the tuple increases and the number of tuples that has to be generated also increases. So there should be a limit to the number of constraints conjuncted together. Similarly the number of variables in the tuple has to be regulated to prevent the tuple from becoming very large. For conjunction of constraints to be effective in reducing the domain values, the constraints in the conjunction set should have a certain number of variables in common. The number of constraints (k), number of variable in the conjunction set (j) and the number of variable common to all the constraints in the conjunction set (i) depends on the CSP and the machine capacity. So there should be a heuristic based on the parameters i, j and k to determine which constraints can be combined together to make the conjunction set.

The heuristic for grouping constraints into conjunctive sets is as follows:

1. Initially there will be 'n' conjunctive sets($\boldsymbol{S}$), each containing a single constraint (where n is the total number of constraints in the CSP).
2. If there exists two conjunctive sets S1, S2 such that variables in S1 is equal to variables in S2, then remove S1 and S2 and add a new set which is conjunction of all the constraints in S1 and S2.

3. If there exist two conjunctive sets S1, S2 such that (a) S1, S2 share at least i variables (b) the number of variables in S1 $\cup$ S2 is less than j (c) the total number of constraints in S1 and S2 is less than k then remove S1 and S2 and add a new set which is conjunction of all the constraints in S1 and S2.
4. Repeat 2 and 3 until no more such pairs exist.

3.6 Correctness of the algorithm

To show the correctness of the algorithm it is necessary to prove that every inconsistent value is removed (completeness) and that no consistent value is removed by the algorithm (soundness) when the algorithm terminates. Moreover, we need to prove that the algorithm terminates.

Lemma 1. *Algorithm will terminate.*

Proof. The algorithm consists of a for loop and two while loops. The generation of elements for the list called *support stream(S,y,b)* uses a for loop. The number of domain values, variable and constraints are finite. Hence the elements generated for the list is finite and the for loop will terminate. The pruning process for the domain values uses a while loop. During each cycle, one element is removed from the list. The elements are added to this list only when a value is removed from some domain. Thus, it is possible to add only a finite number of elements to the list (some elements can be added repeatedly). Hence the while loop will terminate. The algorithm uses a while loop to find support for a variable value in a constraint. The algorithm generates tuples in lexicographic order starting for the smallest one. Since the number of possible tuples for a constraint is finite, the while loop will terminate when it finds a valid support tuple or when all the tuples are generated.

Lemma 2. *SeekCandidateTuple will not miss any valid tuple during the generation of next tuple.*

Proof. Consider that there is a candidate tuple σ' between σ and the tuple returned by the function NextTuple. This implies that $\sigma'[1...k] = \sigma[1...k]$ else σ' will the tuple returned by NextTuple. Hence σ' should be smaller than λ (lines 10-11). If σ' is smaller than λ then that tuple is already generated and checked for consistency. So σ' cannot be a tuple between σ and the tuple returned by the function NextTuple.

Another possibility is that there can be a candidate tuple σ' between σ and λ. Then $\sigma'[1...k]$ should be equal to $\lambda[1...k]$ (lines 7-11). This is not possible candidate since λ is not a valid support tuple.

Lemma 3. *The algorithm does not remove any consistent value from the domain of variables.*

Proof. A value is removed from the domain of a variable only if the value is not arc consistent i.e. there is no valid support tuple for the variable value. Thus, the algorithm does not remove any consistent value from the variables' domains so the algorithm is sound.

Lemma 4. *When the algorithm terminates, then the domain of variables contain only arc consistent values (or some domain is empty).*

Proof. Every value in the domain has to pass the consistency test and inconsistent values will be deleted. When an inconsistent value is deleted and if the deleted value is part of a valid support tuple, then all variable values in that tuple are checked for consistency again. Hence when the algorithm terminates only consistent values remain in the domain.

3.7 Complexity of the algorithm

Lemma 5. *The worst case time complexity of the algorithm is* $\mathbf{O}(en^2d^n)$.

Proof. The worst-case time complexity of GACCC-op depends on the arity of the constraints involved in the constraint network. The greater the number of variables involved in a constraint, the higher the cost to propagate it. Let us first limit our analysis to the cost of enforcing GAC on a single conjunction constraint, S_i , of arity n (n = $|X(S_i)|$) and d = size of the domain of the variable. For each variable $x_i \in X(S_i)$, for each value a$\in D(x_i)$, we look for supports in the space where $x_i = a$,which can contain up to d^{n-1} tuples. If the cost to check whether a tuple satisfies the constraint is in **O**(n), then the cost for checking consistency of a value is in $\mathbf{O}(nd^{n-1})$. Since we have to find support for nd values, the cost of enforcing GAC on S_i is in $O(n^2d^n)$. If we enforce GAC on the whole constraint network, values can be pruned by other constraints, and each time a value is pruned from the domain of a variable involved in S_i, we have to call **SeekValidSupportSet** on S_i. So, S_i can be revised up to nd times. Fortunately, additional calls to **SeekValidSupportSet** do not increase its complexity since, $last(S_i, y, b)$ ensures that the search for support for (x_i, a) on S_i will never check twice the same tuple. Therefore, in a network involving e number of constraints with arity bounded by n, the total time complexity of GACCC-op is in $\mathbf{O}(en^2d^n)$.

Lemma 6. *The worst case space complexity of the algorithm is* $\mathbf{O}(en^2d)$.

Proof. Consistency search generates at most one valid support tuple for each variable value. Then there are at most nd tuples in memory for a constraint. One tuple will contain n elements. Then the set of all tuples which are a valid support for a constraint can be represented in $\mathbf{O}(n^2d)$. Therefore, in a network involving e constraints with arity bounded by n, the total space complexity of GACCC-op is in $\mathbf{O}(en^2d)$.

4 Experimental Results

We implemented the proposed algorithm in C++. The tool will take SystemVerilog constraints and the domain of the input variables as input and generates the reduced domain as output. For our purpose we considered a subset of SystemVerilog constraints which can be given as input to the tool. Our tool can handle

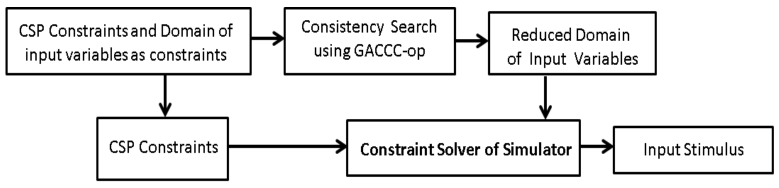

CSP Constraints and Domain of input variables as constraints
Consistency Search using GACCC-op
Reduced Domain of Input Variables
CSP Constraints
Constraint Solver of Simulator
Input Stimulus

Fig. 2. CDTG with consistency

unary constraint, binary constraints and some high order constraints. The high order constraints considered includes arithmetic, logical, mutex and implication constraints. The proposed consistency search algorithm is used along with existing CDTG as shown in fig 2. As shown earlier in fig 1 the different verification scenarios are converted to constraints. We used SystemVerilog to model the scenarios as constraints. The domain of the input variables are also specified as constraints. These constraints and the domain of the variables are given to the consistency check tool(based on GACCC-op). The reduced domain obtained from the tool and the SystemVerilog constraints are then given to the constraint solver of CDTG tools. The output of the solver is the input stimulus required for verification of the DUV.

No: of Variables	No: of Constraints	No of tuples with GACCC	No of tuples with GACCC-op	%improvement in time
10	14	98	76	12.34
12	14	96	70	10.66
14	14	103	82	11.46
18	30	168	120	19.86
20	30	170	131	17.96
20	40	256	216	17.43

Table 1. Time for consistency search for 3-SAT problem instances

We report on experiments we performed with different CSP models. The first is a model for the 3-SAT problems[1] with different number of variables. The SAT problems with a set of clauses are converted into CSPs containing the same set of variables. In our case, we set i=2, k=2 and j=5(i, j and k are the values from the heuristic for generating conjunction set) and generated the conjunction set. Hence the model contained some conjunction set which has 2 variables shared between member constraints. The results are shown in Table 1. For each problem the experiment is repeated for 20 instances. We implemented the GAC-scheme on conjunction of constraints and the proposed algorithm using the C++ language. The result shows that the proposed algorithm attains consistency faster than the existing algorithm.

In order to show the effect of consistency check on constraint solvers associated with CDTG, we took three different CSP benchmark problems, Langford

Series, Magic Sequence and Golomb Ruler. The three CSPs are modeled using SystemVerilog (modeling language used by CDTGs). The SystemVerilog constraints are then used for consistency search. The reduced input variable domain are generated by the consistency search. This reduced domain is then used by the CDTG tool VCS to generate the CSP solutions. From the Table 2 we can see that the time to solve the three CSPs are reduced after giving the reduced domain. In the cases of Magic Sequence the time is significantly reduced, because, after the domain reduction the number of domain values in most of the variables is reduced to one. Since the domain of input variables are reduced, the search space which has to be covered by the solver is reduced. This helps the solver to generate the solutions for CSP in less time and with reduced memory consumption.

5 Conclusions

Existing CDTG tools take large amount of time and memory to generate the required test cases. In this paper, we presented a consistency check algorithm which helps CDTG tools to reduce the memory consumption and time required to generate the test cases. The results showed that the proposed algorithm helps in getting solution faster and with reduced memory consumption. For illustration purposes, we provided the analysis of the Magic Sequence, Langford Series, Golomb Ruler and 3-SAT problem. The requirements of constraint solvers associated with CDTG open the doors to many interesting and novel directions of research. Some worth mentioning are, generating all possible solutions and uniformity in randomization. In future we would like to propose a methodology based on consistency search which will be able to attain 100% coverage at a faster rate with fewer iterations.

			Improvement After Domain Reduction	
Benchmark Problem	No: of Variables	No: of Domain Values	Time (%)	Memory (%)
Langford Series	6	3	10.0	23.5
	8	4	21.4	27.7
	14	7	25.0	40.8
Golomb Ruler	3	4	8.3	23.2
	4	7	7.1	28.2
	5	12	9.5	39.1
	6	18	13.8	73.1
Magic Sequence	4	4	30.0	50.0
	5	5	40.0	71.6
	7	7	55.0	73.3
	8	8	62.5	81.5

Table 2. Results for benchmark CSP problems using VCS

References

1. http://www.satlib.org.
2. Marlene Arangú and Miguel Salido. A fine-grained arc-consistency algorithm for non-normalized constraint satisfaction problems. *Int. J. Appl. Math. Comput. Sci.*, 21(4):733–744, December 2011.
3. Marlene Arangu, Miguel A. Salido, and Federico Barber. Ac3-op: An arc-consistency algorithm for arithmetic constraints. In *Proceedings of the 2009 conference on Artificial Intelligence Research and Developmen*, pages 293–300, Amsterdam, 2009. IOS Press.
4. Marlene Arangú, Miguel A. Salido, and Federico Barber. Ac2001-op: An arc-consistency algorithm for constraint satisfaction problems. In *IEA/AIE (3)*, pages 219–228, 2010.
5. Christian Bessière. Arc-consistency and arc-consistency again. *Artif. Intell.*, 65:179–190, January 1994.
6. Christian Bessière, Eugene C. Freuder, and Jean-Charles Regin. Using inference to reduce arc consistency computation. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1*, pages 592–598, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.
7. Christian Bessière and Jean-Charles Régin. Local consistency on conjunctions of constraints. In *Proceedings of the ECAI'98 Workshop on Non-binary constraints*, pages 53–59, Brighton, UK, 1998.
8. Christian Bessire. Refining the basic constraint propagation algorithm. In *In Proceedings IJCAI01*, pages 309–315, 2001.
9. Eyal Bin, Roy Emek, Gil Shurek, and Avi Ziv. Using a constraint satisfaction formulation and solution techniques for random test program generation. *IBM Systems Journal*, 41(3):386–402, 2002.
10. Marc R. C. van Dongen. Ac-3d an efficient arc-consistency algorithm with a low space-complexity. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 755–760, London, UK, UK, 2002. Springer-Verlag.
11. Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using bayesian networks. In *Design Automation Conference, 2003. Proceedings*, pages 286 – 291, june 2003.
12. Laurent Fournier, Yaron Arbetman, and Moshe Levinger. Functional verification methodology for microprocessors using the genesys test-program generator. application to the x86 microprocessors family. In *Design, Automation and Test in Europe Conference and Exhibition 1999. Proceedings*, pages 434–441, 1999.
13. Mahesh Iyer. Race a word-level atpg-based constraints solver system for smart random simulation. In *Test Conference, 2003. Proceedings. ITC 2003. International*, volume 1, pages 299 – 308, 30-oct. 2, 2003.
14. Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
15. Anna Moss. Constraint patterns and search procedures for cp-based random test generation. In *Proceedings of the 3rd international Haifa verification conference on Hardware and software: verification and testing*, HVC'07, pages 86–103, Berlin, Heidelberg, 2008. Springer-Verlag.
16. Mohr Roger and Henderson Thomas. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
17. Pascal Van Hentenryck, Yves Deville, and Choh Teng. A generic arc consistency algorithm and its specializations. Technical report, Providence, RI, USA, 1991.

Towards a Verification Framework for Haskell by Combining Graph Transformation Units and SAT Solving

Marcus Ermler

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany

maermler@informatik.uni-bremen.de

Abstract. In this paper, we propose a new approach for the verification of Haskell programs by combining graph transformation units and SAT solving. Therefore, function equations, known properties, and the property to be proven are translated into graph transformation units to perform structural induction proofs. In general, the automation of this process is highly nondeterministic because in each rewriting step several rules could be applied, also those that may lead in the wrong direction. To tackle this deficiency we translate the derivation process of graph transformation into propositional formulas and explore, thereby, the whole state space up to a certain bound.

Keywords: Haskell, verification, graph rewriting, SAT encoding

1 Introduction

This paper is an extended version of an unpublished informal workshop contribution submitted to HART2013 [9] where we propose a new approach for proving the correctness of programs written in the functional programming language Haskell. This extended version includes an improved translation to propositional formulas based on typed nodes, more theoretical results, and detailed remarks, examples, and figures.

We introduce a prototypical framework for verifying properties of Haskell programs via structural induction proofs based on graph transformation units [13] and their combination with SAT solving [3]. This seems to us worthwhile for four reasons: **(1)** Graph rewriting techniques have proven successful in term rewriting (cf. [16]) and functional programming (cf. [4, 12]); **(2)** For the functional programming language CLEAN [4] the theorem prover SPARKLE [15] was developed and both have a semantic based on graph rewriting; **(3)** Automatically applying function equations (or their graph transformation rule counterparts) is highly nondeterministic, but a translation of rule applications to propositional formulas can tackle this nondeterminism such as shown in [14, 10]; **(4)** Verification with model checkers and SAT solvers is a hot topic in general, but not much studied in graph transformation. Besides our SATaGraT tool, the

GROOVE tool [17] is the only tool in the graph transformation area that directly combines graph transformation with model checking and verification. In [1], an encoding of term graph rewriting into dynamic logic is proposed. Furthermore, in [2] model checking for graph transformation is introduced via a translation of specifications of the graph transformation machine AGG into Bogor models.

A graph transformation unit consists of an initial and a terminal graph class expression describing the permitted in- and outputs, a set of graph transformation rules and a control condition for guiding the rule application. Haskell function equations that are converted into graph transformation rules and a property in equation form are translated into graph transformation units for the base case and the inductive step. In both cases, the left-hand side of the property is the input of the graph transformation unit and the right-hand side is its output. If both graph transformation units find a successful derivation, i.e. a derivation from the input to the output graph, then the property has been proven.

Why is rule application highly nondeterministic and how can a translation to SAT tackle this deficiency? In each derivation step several rules can be applied, but only some of them yield a proper result whereas the rest leads in the wrong direction. It is possible that a rule, e.g. the identity in addition, is applied infinitely often because of automating the verification. So one would need a backtracking mechanism to continue at another point of the derivation. By using a translation to SAT the whole state space up to a certain bound is generated and a SAT solver like MiniSat [7] performs a backtracking based on the DPLL procedure [6].

We devise a new translation of the derivation process of graph transformation units into propositional formulas directly yielding formulas in conjunctive normal form (the conversion to CNF via the Tseitin transformation [18] was a bottleneck of the previous version), the standard input format for today's SAT solvers. Furthermore, we reduce the formula sizes by introducing node types. The translation of graph transformation to SAT, the application of MiniSat, and the evaluation of the output is done fully automatically in our tool SATaGraT (***SAT*** *solving* ***a****ssists* ***Gra****ph* ***T****ransformation* Engine) [10, 8]. We extend SATaGraT by our new translation to SAT and our proposed verification framework. Finally, we test the framework.

The paper is organized as follows. Section 2 explains the connection between Haskell and graph transformation and Section 3 details how graph transformation is translated into propositional formulas. The main ideas behind our framework are described in Section 4. In Section 5, we present conducted experiments. Section 6 contains the conclusion.

2 From Haskell to Graph Transformation

In this section, we detail the basic notions of graphs, graph transformation, and graph transformation units and how these concepts can be applied to Haskell and term rewriting.

Graphs, Graph Morphisms, and Matches. We use *edge labeled directed graphs without multiple edges* and with a finite set of typed nodes. For a finite set Σ of edge labels and a set $\mathcal{T}$ of types such a graph is a triple $G = (V, E, t)$ where $V \subseteq \{1, \ldots, n\} = [n]$ for some $n \in \mathbb{N}$ is a finite set of *nodes*, $E \subseteq V \times \Sigma \times V$ is a set of *labeled edges*, and $t \colon V \to \mathcal{T}$ is a mapping that maps each node v in V to a type *type* in $\mathcal{T}$. n is called the *size* of G. The components V, E, t are also denoted by V_G, E_G, and t_G. In some cases it is useful to have a special node type *arbitrary* that indicates an arbitrary node type. We call an edge (v, x, v) a *loop*. A special graph is the *empty graph* $\emptyset = (\emptyset, \emptyset, \emptyset)$. We call G a *subgraph* of H, denoted by $G \subseteq H$, if $V_G \subseteq V_H$, $E_G \subseteq E_H$, and $t_G(v) = t_H(v)$ for all $v \in V_G$.

Furthermore, we use injective graph morphisms for the matching. Let G, H be two graphs as defined above. An *injective graph morphism* $g \colon G \to H$ is an injective mapping $g_V \colon V_G \to V_H$, that is structure-, label- and type-preserving, i.e. for each edge $(v, x, v') \in E_G$, $(g_V(v), x, g_V(v')) \in E_H$ such that $t(v) = t(g_V(v))$ and $t(v') = t(g_V(v'))$. An injective graph morphism $g \colon G \to H$ yields the image $g(G) = (g_V(V_G), g_E(E_G)) \subseteq H$ with $g_E(E_G)) = \{(g_V(v), x, g_V(v')) \mid (v, x, v') \in E_G\}$ called the *match* of G in H. In the following, we will write $g(v)$ and $g(e)$ for nodes $v \in V_G$ and edges $e \in E_G$ because the type of the argument indicates the indices V and E.

Remark 1 (Haskell terms and graphs). Our approach considers a subset of Haskell consisting of predefined data types (like Int, Char, String or Lists) and functions defined by functions equations without guards or local definitions. Terms are represented by trees where nodes are typed with function symbols, constants, or variables. The outermost function name is the root, and constants and variables are leafs. The outgoing edges of nodes with function symbols are labeled with the argument positions of the corresponding function arguments. Otherwise the terms `xs ++ ys` and `ys ++ xs` would have the same tree representation. In drawings, we use rectangles for function symbols, circles for constants and variables, and draw the node types inside the nodes. Node numbers are drawn in bold, right or below right outside the nodes whereas the numbers in normal font represent function argument positions. The drawing of node numbers is optional.

Figure 1 shows an example for a term graph with node numbers. Here, we represent the Haskell term `length ([] ++ ys)` by the graph $G = (V, E, t)$ where $V = \{1, 2, 3, 4\}$, $E = \{(1, 1, 2), (2, 1, 3), (2, 2, 4)\})$, and $t = \{1 \mapsto \texttt{length}, 2 \mapsto \texttt{++}, 3 \mapsto \texttt{[]}, 4 \mapsto \texttt{ys})$. Figure 2 shows the same graph without node numbers.

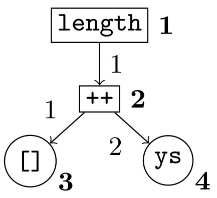


Fig. 1. A term graph with node numbers

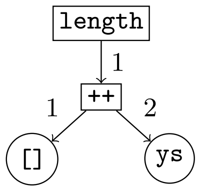


Fig. 2. A term graph

Rules and Their Application. A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs: the *left-hand side* L, the *gluing graph* K, and the *right-hand side* R. In our approach, we only consider rules with $E_K = \emptyset$ and an invariant node set, i.e. $V_L = V_K = V_R$. For that reason, we simplify the rule notation to $r = (L \to R)$, denote the set of nodes of a rule r by V_r, and its size by $size(r)$.

The application of a rule to a graph works as follows. Let $r = (L \to R)$ be a rule, G a graph, and $g\colon L \to G$ an injective graph morphism. Remove the edges in $g(L)$ from G yielding D and add R disjointly to D. Finally, glue R and D as follows. (1) Merge each $v \in V_R$ with $g(v)$. (2) If there is an edge $(v, x, v') \in E_R$ with $v, v' \in V_R$ and an edge $(g(v), x, g(v')) \in E_D$ then these edges are identified. The application of a rule r to a graph G with respect to an injective graph morphism g yielding a graph H is denoted by $G \underset{r,g}{\Longrightarrow} H$. This is called *rule application* or *direct derivation* and fits into the double-pushout approach (cf. [5]). The sequential composition $d = G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \cdots \underset{r_n,g_n}{\Longrightarrow} G_n$ of n direct derivations for some $n \in \mathbb{N}$ is called a *derivation*, shortly denoted by $G_0 \overset{*}{\underset{P}{\Longrightarrow}} G_n$ if $r_1, \ldots, r_n \in P$. In general, rule matching corresponds to the well-known subgraph isomorphism problem and is NP-complete but by using a fixed number of non-parametrized rules the matching and, hence, the number of matches of a rule r in a graph V is polynomially bounded.

Remark 2 (From Haskell function equations to rules). In the following, we denote by $tree(\texttt{t})$ the tree representation of a term `t`. A Haskell function equation `l=r` can be easily converted into a graph transformation rule by translating the left-hand side and the right-hand side into trees, i.e. `l=r` is translated into $tree(\texttt{l}) \to tree(\texttt{r})$ because Haskell function equations match the left-hand side and replace the match by the right-hand side.

As defined above, we use rules that add and delete edges for transforming graphs. To handle the addition and deletion of nodes we employ a simple trick: deleted and unused nodes are marked with a special label named *del* (or 1 in the propositional encoding). By removing this label, one can add the corresponding node. For reasons of readability, we allow in drawings the deletion of nodes instead of using a special label.

The function equation `[] ++ ys = ys` is translated into the graph transformation rule $(\texttt{++})_1$ drawn in Figure 3. The same rule without drawing deleted nodes can be found in Figure 4. Please note, that the term `ys` is a placeholder for an arbitrary term of the same type. For example, the term `xs ++ ys` could be matched. Moreover, we need for the SAT encoding an additional node $\overline{v}$ of an arbitrary type that is connected with the node v_{ofs} typed with the outermost function symbol of the left-hand side. If this node is deleted, the edge between $\overline{v}$ and v_{ofs} is redirected to the new outermost function symbol, variable, or constant.

An example for a derivation can be found in Figure 5. Here, the rule $(\texttt{++})_1$ is applied to the term graph of the term `length ([] ++ x:xs)` w.r.t. the mapping $g = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 1\}$. The result is the term graph of `length x:xs`. Finally, the graph transformational counterpart of `length x:xs = 1 + length`

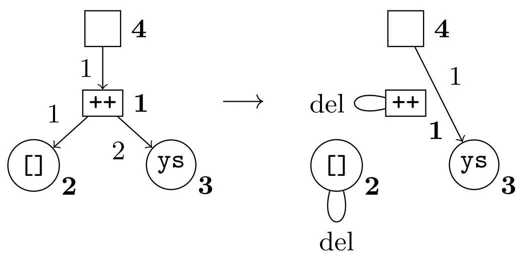


Fig. 3. The rule $(++)_1$ where deleted nodes are drawn

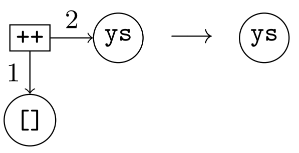


Fig. 4. The rule $(++)_1$ where deleted nodes are not drawn

xs, called $length_1$ in the following, is applied yielding the last term graph of the derivation. To apply $length_1$ it is necessary that the second graph contains two additional and new nodes of type 1 and +.

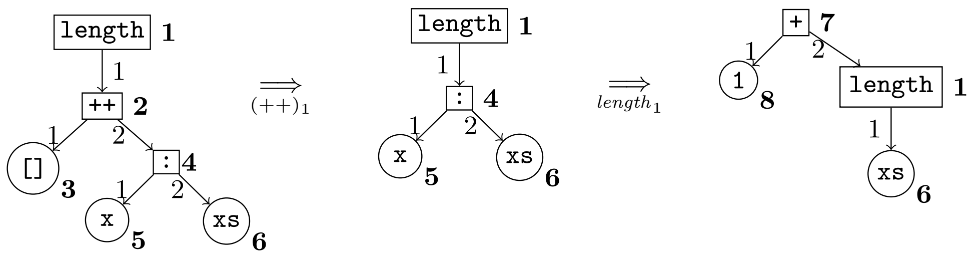


Fig. 5. Result of applying $(++)_1$ and $length_1$ to *tree*(`length ([] ++ x:xs)`)

Graph Transformation Units. A *graph transformation unit* [13] (gtu for short) is a system $gtu = (I, P, C, T)$ where I and T are graph class expressions to specify the *initial* and *terminal* graphs, P is a set of rules, and C is a control condition. The graph transformation unit *gtu* specifies all derivations from initial to terminal graphs that are allowed by the control condition. Such derivations are called *successful.*

Control conditions guide the rule application and restrict their nondeterminism. Each control condition C specifies a language $L(C) \subseteq P^*$. The elements of $L(C)$ are called *rule sequences.* The finite sublanguage $L(C)(m) = \{w \in L(C) \mid |w| = m\}$ describes the set of all rule sequences in $L(C)$ with length m. In this paper, we use regular expressions as control conditions. The set of all rule sequences of a regular expression *regexp* can be easily generated by using the corresponding regular language $L(regexp)$. More precisely, every rule r is a control condition that requires one application of r. For control conditions c, c_1, c_2, the expressions $c_1; c_2$, $c_1|c_2$, and c^* are control conditions where $c_1; c_2$ means to apply c_1 before c_2, $c_1|c_2$ means to apply c_1 or c_2, and c^* means to iterate the application of c arbitrarily but finitely often. A rule set $P = \{r_1, \ldots, r_n\}$ can also be used as control condition, it abbreviates the expression $r_1| \ldots |r_n$.

Graph class expressions are used to specify the initial and terminal graphs of derivations. Typical examples are the sets of all unlabeled graphs or subsets Δ of Σ specifying terminally labeled graphs. Concrete graphs can also be considered as graph class expressions, i.e. each graph G specifies itself. We use single graphs as initial and terminal graphs, they indicate the left-hand side and the right-hand side of a property to be proven.

Remark 3 (Term rewriting via graph transformation units). The derivation from Figure 5 can be expressed via the following simple graph transformation unit: $gtu = (I, P, C, T)$ where $I = tree($`length ([] ++ x:xs)`$)$, $P = \{(++)_1, length_1\}$, $C = (++)_1 \; ; \; length_1$, $T = tree($`1 + length xs`$)$, and $length_1 = tree($`length x:xs`$) \to tree($`1 + length xs`$)$. This can be noted down in the following way (cf. Figure 6):

initial:	$tree($`length ([] ++ x:xs)`$)$
rules:	$(++)_1$
	$length_1$
cond.:	$(++)_1 \; ; \; length_1$
terminal:	$tree($`1 + length xs`$)$

Fig. 6. A sample unit *gtu*

3 Translating the Semantic of Graph Transformation Units into Propositional Formulas

In this section, we show how the semantic of graph transformation units can be described by propositional formulas. For this, we have to translate the graph class expressions of initial and terminal graphs and the derivation process into propositional formulas. A satisfying variable assignment to the formula represents one of the successful derivations from initial to terminal graphs.

Representing Graphs and Graph Sequences as Propositional Formulas. To describe the edges of a derivation $G_0 \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \cdots \underset{r_m,g_m}{\Longrightarrow} G_m$ we have to add a $k \in \mathbb{N}$ to each edge that determines the derivation step in which the edge occurs. For this, every propositional formula with variable set $\{edge(e,k) \mid e \in [n] \times \Sigma \times [n], k \in [m]\}$ represents a sequence $G_1, \ldots, G_m$ of graphs for each variable assignment f satisfying the formula, i.e. the graph G_k contains the edge e if and only if $f(edge(e,k)) = TRUE$. A single initial graph G in the kth derivation step can be described by the formula

$$\mathsf{graph}(\mathsf{G},\mathsf{k}) = \bigwedge_{(v,a,v')\in E_G} edge(v,a,v',k) \wedge \bigwedge_{(v,a,v')\in([n]\times\Sigma\times[n])-E_G} \neg edge(v,a,v',k).$$

Example 1. Then, the graph from Figure 2, called G_0 in the following, is expressed via the following formula (where $E_0 = \{(1,1,2), 2,1,3), (2,2,4)\}$)

$$edge(1,1,2,0) \wedge edge(2,1,3,0) \wedge edge(2,2,4,0) \wedge \bigwedge_{e \in ([n]\times\Sigma\times[n]) - E_0} \neg(e,0).$$

Remark 4 (Variables do not contain node types). Please note, that it is not necessary for the variables to contain node types. Why is that? The node types are fixed by the type mapping and are not changed during a derivation, i.e. each node–if deleted or not–has the same type at the beginning and at the end of a derivation.

Remark 5 (Bound for additional nodes). If rules allow the addition of nodes, we have to guarantee that the initial graph contains a sufficient number of extra nodes. These nodes are labeled with *del* at the beginning of a derivation. The bound can be determined by the rule that adds a maximum number of nodes to a graph and the number of derivation steps. For example, if we have two rules adding two and three nodes, respectively, and we have ten derivation steps. Then the inital graph has to contain $3 * 10$ extra nodes because in the worst case the rule that adds three nodes is applied in each derivation step.

Representing Rule Applications as Propositional Formulas. Please remember, that the set of nodes is invariant during the derivation process, i.e. all graphs in the derivation have the same set of nodes V and node numbers $[n]$. For applying a rule, we have to compute the set of all injective matchings from the left-hand side of the rule into the considered graph. Because of the definition of injective matchings, nodes of the left-hand side can only match nodes of the same type in the considered graph. Nodes of arbitrary type like the top node in Figure 3 can match every node in a graph.

For a rule $r = (L \to R)$ the set of injective graph morphisms from r to the set of nodes $[n]$ is denoted by $\mathcal{M}(r,n)$. Let $k \in \mathbb{N}$ be a derivation step, G_{k-1} be a graph of size n, $r = (L \to R)$ be a rule, and $g \in \mathcal{M}(r,n)$. The application of r to G_{k-1} with respect to g is then expressed by the following formulas[1]

- $\mathsf{morph(r,g,k)} = morph(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_L} edge(g(v),a,g(v'),k-1)$,
- $\mathsf{rem(r,g,k)} = rem(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_L - E_R} \neg edge(g(v),a,g(v'),k)$,
- $\mathsf{add(r,g,k)} = add(r,g,k) \leftrightarrow \bigwedge_{(v,a,v')\in E_R} edge(g(v),a,g(v'),k)$,
- $\mathsf{keep(r,g,k)} = keep(r,g,k) \leftrightarrow \Big(\bigwedge_{(v,a,v')\notin g(E_L\cup E_R)} \big(edge(v,a,v',k-1) \leftrightarrow edge(v,a,v',k)\big)\Big)$ where $g(E_L \cup E_R) = \{(g(v),a,g(v')) \mid (v,a,v'), \in E_L \cup E_R\}$,
- $\mathsf{apply(r,g,k)} = apply(r,g,k) \leftrightarrow \big(morph(r,g,k) \wedge rem(r,g,k) \wedge add(r,g,k) \wedge keep(r,g,k)\big)$.

[1] Please note, that in the following the expressions in bold typewriter font like $\mathsf{morph(r,g,k)}$ or $\mathsf{rem(r,g,k)}$ abbreviate propositional formulas whereas the italic font is used for variables (e.g. $morph(r,g,k)$, $rem(r,g,k)$, or $edge(v,a,v',k)$).

The formula morph describes that g is a graph morphism from L to G_{k-1}, i.e. it describes a matching of L in G_{k-1}. The removal of the images of every edge of the left-hand side L from G_{k-1} is expressed by rem. The addition of edges of the right-hand side R is described by add. That edges that have been neither deleted nor added must be kept, corresponds to the formula keep. Finally, apply describes the whole application of r to G_{k-1} with respect to g.

In our tool, all formulas are directly expressed as CNF. The conversion to CNF can be done in at most quadratic time.

Lemma 1. *Let r be a rule, g be an injective graph morphism, and $k \in \mathbb{N}$ be a derivation step. Then the formulas* $\mathsf{morph}(\mathsf{r}, \mathsf{g}, \mathsf{k})$, $\mathsf{rem}(\mathsf{r}, \mathsf{g}, \mathsf{k})$, $\mathsf{add}(\mathsf{r}, \mathsf{g}, \mathsf{k})$, $\mathsf{keep}(\mathsf{r}, \mathsf{g}, \mathsf{k})$, *and* $\mathsf{apply}(\mathsf{r}, \mathsf{g}, \mathsf{k})$ *can be converted into CNF in at most quadratic time.*

The following theorem states that a satisfying assignment to apply corresponds to a direct derivation, i.e a rule application.

Theorem 1. $G_{k-1} \underset{r,g}{\Longrightarrow} G_k$ *if and only if it there is a satisfying assignment to* $\mathsf{graph}(\mathsf{G_{k-1}}, \mathsf{k}-1) \wedge \mathsf{apply}(\mathsf{r}, \mathsf{g}, \mathsf{k}) \wedge \mathsf{graph}(\mathsf{G_k}, \mathsf{k})$.

Example 2. Please note, that we use in the SAT encodings the label 1 instead of *del*. But for reasons of readability we write *del*. For the rule $(++)_1$ in Figure 3, the first graph in Figure 5, and the graph morphism $g = \{1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 4, 4 \mapsto 1)\}$ we get the following formulas. Please note, that the node typing of the left-hand side nodes corresponds under the mapping g to the node typing of the considered graph.

- $morph((++)_1, g, 1) \leftrightarrow \Big(edge(2,1,3,0) \wedge edge(2,2,4,0) \wedge edge(1,1,2,0)\Big)$
- $rem((++)_1, g, 1) \leftrightarrow \Big(\neg edge(2,1,3,1) \wedge \neg edge(2,2,4,1) \wedge \neg edge(1,1,2,1\Big)$
- $add((++)_1, g, 1) \leftrightarrow \Big(edge(1,1,4,1) \wedge edge(2,del,2,1) \wedge edge(3,del,3,1)\Big)$
- $keep((++)_1, g, 1) \leftrightarrow \bigwedge_{(v,a,v') \in E_{keep}} \Big(edge(v,a,v',0) \leftrightarrow edge(v,a,v',1)\Big)$

where

$$\begin{aligned} E_{keep} &= ([n] \times \Sigma \times [n]) - g(E_{L_{(++)_1}} \cup E_{R_{(++)_1}}) \\ &= ([n] \times \Sigma \times [n]) - \{(2,1,3),(2,2,4),(1,1,2),(1,1,4),(2,del,2),(3,del,3)\}. \end{aligned}$$

because the left-hand side edges of $(++)_1$ are $E_{L_{(++)_1}} = \{(1,1,2),(1,2,3),(4,1,1)\}$ and the right-hand side edges are $E_{R_{(++)_1}} = \{(4,1,3),(1,del,1),(2,del,2)\}$. For example, the edge $(4,2,2)$ is kept because it is neither matched, nor added, nor removed.

Translation of Graph Transformation Units into Propositional Formulas. The kth derivation step is expressed via

$$\mathsf{step}(\mathsf{n}, \mathsf{r}, \mathsf{k}) = \bigvee_{g \in M(r,n)} apply(r, g, k).$$

Here, for each mapping $g \in M(r,n)$, the possible applications of r to G_{k-1} are described. Each of these *apply*-variables corresponds to a formula that expresses the matched, added, deleted, and kept edges. This fact results in the following formula

$$\mathsf{stepEdges}(\mathsf{n},\mathsf{r},\mathsf{k}) = \bigwedge_{g \in M(r,n)} \Big(\mathsf{apply}(\mathsf{r},\mathsf{g},\mathsf{k}) \wedge \mathsf{morph}(\mathsf{r},\mathsf{g},\mathsf{k}) \wedge \mathsf{rem}(\mathsf{r},\mathsf{g},\mathsf{k}) \wedge \mathsf{add}(\mathsf{r},\mathsf{g},\mathsf{k}) \wedge \mathsf{keep}(\mathsf{r},\mathsf{g},\mathsf{k})\Big)$$

where the *apply*-variables are included in $\mathsf{apply}(\mathsf{r},\mathsf{g},\mathsf{k})$.

Let $gtu = (I, C, P, T)$ be a graph transformation unit with single graphs G_I and G_T as initial and terminal graphs. Then for all natural numbers m, n and for all $r_1 \cdots r_m \in L(C)(m)$ a derivation of length m from an initial graph of size n that follows $r_1 \cdots r_m$ is expressed via

$$\mathsf{f}_{\mathsf{gtu}}(\mathsf{n},\mathsf{r}_1 \cdots \mathsf{r}_\mathsf{m}) = \mathsf{graph}(\mathsf{G}_\mathsf{I},0) \wedge \bigwedge_{k=1}^{m} \Big(\mathsf{step}(\mathsf{n},\mathsf{r}_\mathsf{k},\mathsf{k}) \wedge \mathsf{stepEdges}(\mathsf{n},\mathsf{r}_\mathsf{k},\mathsf{k})\Big) \wedge \mathsf{graph}(\mathsf{G}_\mathsf{T},\mathsf{m}).$$

The following theorem states that a satisfying assignment to $\mathsf{f}_{\mathsf{gtu}}$ corresponds to a successful derivation from an initial to a terminal graph.

Theorem 2. *Let $gtu = (G_I, P, C, G_T)$ be a graph transformation unit with single initial and terminal graphs G_I and G_T of size n. Then there is a successful derivation $G_I \underset{r_1,g_1}{\Longrightarrow} G_1 \underset{r_2,g_2}{\Longrightarrow} \cdots \underset{r_m,g_m}{\Longrightarrow} G_T$ from the inital graph G_I to the terminal graph G_T guided by the rule sequence $r_1 \cdots r_m \in L(C)$ if and only if there is a satisfying assignment to $\mathsf{f}_{\mathsf{gtu}}(\mathsf{n},\mathsf{r}_1 \cdots \mathsf{r}_\mathsf{m})$.*

If we assume that morph, rem, add, keep, and apply are already in CNF, such as stated in Lemma 1, then we can formulate the following statement.

Lemma 2. *Let $gtu = (G_I, P, C, G_T)$ be a graph transformation unit with single initial and terminal graphs G_I and G_T of size n and $r_1 \cdots r_m \in L(C)$ be a rule sequence. Then the following holds:*

1. *For all $k \in \{1, \ldots, m\}$, $step(n, r_k, k)$ and $stepEdges(n, r_k, k)$ are in CNF.*
2. *$f_{gtu}(n, r_1 \cdots r_m)$ is in CNF.*

The set of all derivations of length m from a graph with node set $[n]$ is described by

$$\mathsf{f}_{\mathsf{all}}(\mathsf{n},\mathsf{m}) = \bigvee_{r_1 \cdots r_m \in L(C)(m)} \mathsf{f}_{\mathsf{gtu}}(\mathsf{n},\mathsf{r}_1 \cdots \mathsf{r}_\mathsf{m})$$

where each of the subformulas $\mathsf{f}_{\mathsf{gtu}}$ can be generated and solved separately. This may lead to a speed-up in the processing time in some cases and leaves open the possibility of using parallelization, i.e. each $\mathsf{f}_{\mathsf{gtu}}$ could be solved on a separate core.

If we add the empty rule $empty = (\emptyset \to \emptyset)$ to the rule set, such as shown in [14], every derivation of a length less than $p(n)$ can be prolonged by empty steps to the length $p(n)$. Note, that a successful derivation of length $p(n)$ with empty steps is also successful if the empty steps are removed. For this, we can express the set of all polynomial derivations by $f_{all}(n, p(n))$.

Remark 6. We introduce new variables like $morph(r, g, k)$ or $add(r, g, k)$ to exactly reconstruct computed derivations. In previous versions of SATaGraT, only *edge*-variables are used where all informations about applied rules and morphisms get lost and have to be reconstructed via an algorithm that delivers for some rule sets ambiguous results (cf. [10]). For example, the previous morph was described by $\mathsf{morph}(\mathsf{r}, \mathsf{g}, \mathsf{k}) = \bigwedge_{(v,a,v')\in E_L} edge(g(v), a, g(v'), k - 1)$.

4 Structural Induction via Graph Transformation Units

We want to sketch the main ideas of how graph transformation can be employed for structural induction proofs by verifying a well-known list property in our framework. The corresponding implementation is currently in a prototypical stage and supports the mentioned subset of Haskell (cf. Section 2).

The function `(++)` given by the equations

```
(++) [] ys = ys
(++) (x:xs) ys = x:xs ++ ys
```

can be easily translated into graph transformational rules $(++)_1$ and $(++)_2$ (cf. Figures 4 and 7).

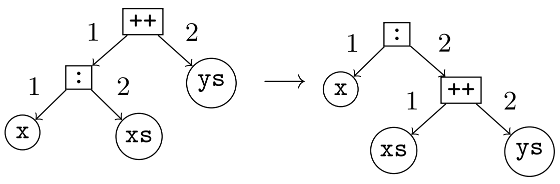


Fig. 7. The rule $(++)_2$

As described in Section 2, Haskell terms are represented by trees where the outermost function name is the root and constant names and variables are the leafs. The function `length` has also a simple rule representation which can be generated in the same way as shown above. To simplify the example, we are using the following definition of `length` instead of the Prelude definition in Data.List.genericLength:

```
length [] = 0
length (x:xs) = 1 + length xs
```

where the corresponding rules are abbreviated with $length_1$ and $length_2$.

A property $p = $ (`l=r`) consists of two Haskell terms of the same type and may contain variables of a variable set. We write p(`xs`) $=$ (`l(xs)=r(xs)`) to indicate that the left-hand side and the right-hand side contain a common variable `xs`. A simple structural induction for list properties can be defined in the following way.

Definition 1 (Simple Structural Induction). *Let $p(\texttt{xs}) = (\texttt{l(xs)=r(xs)})$ be a list property. Then the property holds for all finite lists* `xs` *if and only if $p(\texttt{[]}) = (\texttt{l([])=r([])})$ holds and $p(\texttt{xs})$ implies that $p(\texttt{x:xs}) = (\texttt{l(x:xs)=r(x:xs)})$ holds.*

One interesting property to be proven by a structural induction proof could be, whether the length of a list concatenation is the sum of the separate list lengths, i.e.

`length (xs ++ ys) = length xs + length ys` for all finite lists `xs` and `ys`.

This can be proven by using two graph transformation units: One unit for the base case and the other for the inductive step where we perform an induction over `xs`. The input of both units is the left-hand side of the property, i.e. `length ([] ++ ys)` and `length (x:xs ++ ys)`, respectively, and the output is the right-hand side, i.e. `length [] + length ys` and `length x:xs + length ys`, respectively. All known function equations can be used as rules, in the inductive step the hypothesis is added as additional rule. This idea can be stated in the following way.

Definition 2. *Let $p(\texttt{xs}) = (\texttt{l(xs)=r(xs)})$ be a list property with induction variable* `xs` *and let P be a set of graph transformational rules representing Haskell function equations. Then the base case unit and the inductive step unit are defined as follows.*

- *$base(p(\texttt{[]})) = (tree(\texttt{l([])}), P, P^*, tree(\texttt{r([])}))$*
- *$step(p(\texttt{x:xs})) = (tree(\texttt{l(x:xs)}), P \cup \{hyp_1, hyp_2\}, C_{step}, tree(\texttt{r(x:xs)}))$ where $hyp_1 = (tree(\texttt{l(xs)}) \to tree(\texttt{r(xs)}))$, $hyp_2 = (tree(\texttt{r(xs)}) \to tree(\texttt{l(xs)}))$, and $C_{step} = P^*\ ;\ (hyp_1 \mid hyp_2)\ ;\ P^*$.*

If it is found in both cases a derivation from the initial to the terminal graph, the property has been proven. This can be formulated in the following way.

Theorem 3. *Let $p(\texttt{xs})$ be a property, $base(p(\texttt{[]}))$ be a base case unit, and $step(p(\texttt{x:xs}))$ be an inductive step unit. If there is a successful derivation in $base(p(\texttt{[]}))$ as well as in $step(p(\texttt{x:xs}))$, then the property holds.*

Why is that? Obviously, the rule arrows can be reversed, i.e. one can apply the rules also from right to left, and, thus, the derivation process can be reversed, i.e. it also exists a derivation from the terminal to the initial graph. Hence, both sides are equal and the property has been verified. This is stated in the following lemma. Let r be a rule, then r' is obtained by reversing the arrow in r.

Lemma 3. *Let $gtu = (G_I, P, C, G_T)$ be a base case unit (or an inductive step unit) and $r_1 \cdots r_m \in L(C)$. Then $G_I \underset{r_1 \cdots r_m}{\Longrightarrow} G_T$ if and only if $G_T \underset{r'_m \cdots r'_1}{\Longrightarrow} G_I$.*

Let us continue with the example. The unit for the base case can be found in Figure 8 where $identity_{add}$ is the rule counterpart of `0 + x = x`, and $length'_1 =$

initial:	*tree*(`length ([] ++ ys)`)
rules:	$(++)_1$, $length'_1$, $identity_{add}$
cond.:	$\big((++)_1 \mid length'_1 \mid identity_{add}\big)^*$
terminal:	*tree*(`length [] + length ys`)

Fig. 8. *base case* unit

initial:	*tree*(`length (x:xs ++ ys)`)
rules:	$(++)_2$, len_2, len'_2, $assoc_{add}$, *hypothesis*
cond.:	$\big((++)_2 \mid len_2 \mid len'_2 \mid assoc_{add}\big)^*$
	; *hypothesis*
	; $\big((++)_2 \mid len_2 \mid len'_2 \mid assoc_{add}\big)^*$
terminal:	*tree*(`length x:xs + length ys`)

Fig. 9. *inductive step* unit

tree(`0`) $\rightarrow$ *tree*(`length []`), i.e. the rule *length* with a reversed arrow.[2] The inductive step unit in Figure 9 has two additional rules, the hypothesis (see Figure 11) and a rule for the associativity of addition. For reasons of space, we omit the hypothesis rule with a reversed arrow. The rule $length_2$ and the same rule but with an reversed arrow are abbreviated by len_2 and len'_2, respectively. Now, we can formulate the following proposition.

Proposition 1. *If there is a successful derivation in the base case unit from Figure 8 as well as in the inductive step unit from Figure 9, then the property*

`length (xs ++ ys) = length xs + length ys`

holds for all finite lists `xs` *and* `ys`*.*

Proof. One successful derivation for the base case proof can be found in Figure 10. The first graph is the initial graph, the second graph is derived by applying the rule $(++)_1$, the third graph is the result of the application of the rule for the identity of addition, and, finally, the fourth graph is the terminal graph derived by applying the rule $length'_1$.

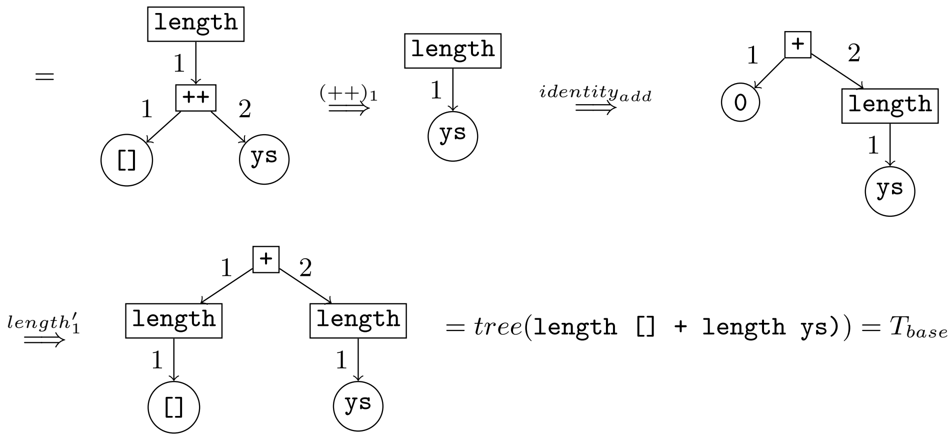
One successful derivation for the inductive step can be found in Figure 12. The first graph is the initial graph, the second graph is derived by applying the rule $(++)_2$, the third graph is the result of applying the rule len_2, the fourth graph is derived by applying the *hypothesis* rule, the fifth graph is derived by applying $assoc_{add}$ and, finally, the terminal graph is the result of applying len'_2.

5 Experiments

In [9], it was only possible to prove the base case of the `length`-property given in Section 2. The inductive step unit leads to a stack overflow because of the formula size. The set Σ for edge labels really blows up the formulas and, therefore, we introduce nodes with types. This significantly reduces the formula sizes because Σ now only consists of argument positions. Hence, our tool allows to prove base

[2] Please note, that we choose for reasons of comprehensibility an adequate and simple rule set in both cases. In general, the rule set can consist of much more function equations and properties.

$I_{base} = tree($`length ([] ++ ys)`$)$



$= tree($`length [] + length ys)`$) = T_{base}$

Fig. 10. Base case proof performed by the *base case* unit

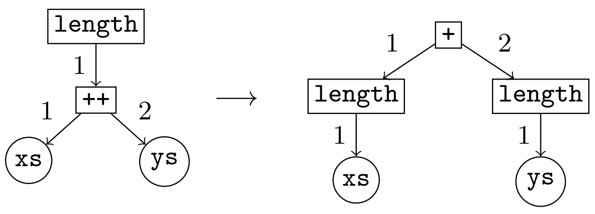


Fig. 11. The rule *hypothesis*

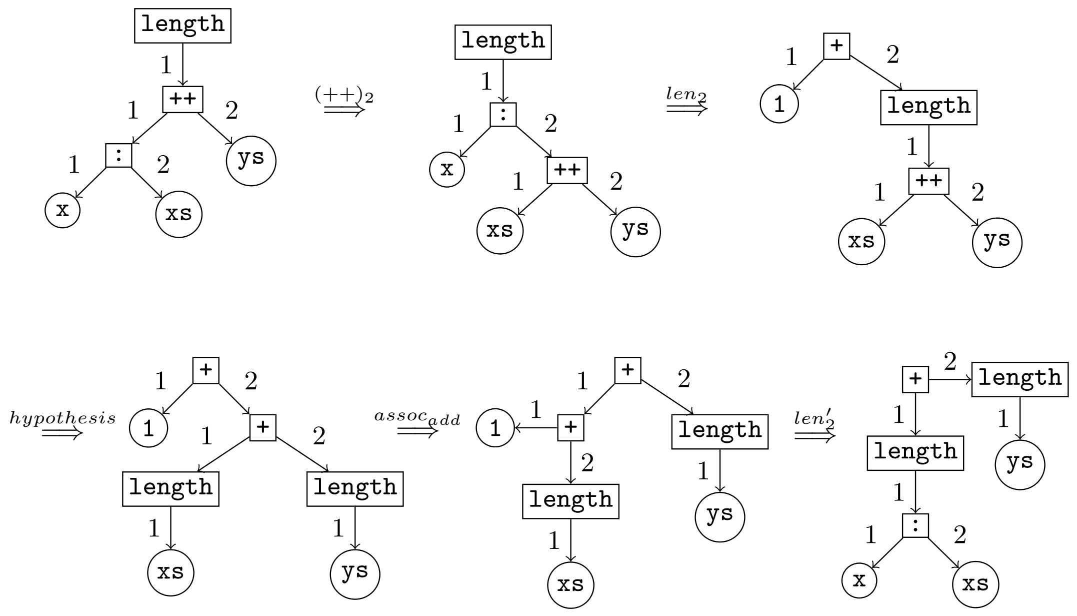


Fig. 12. Inductive step performed by the *inductive step* unit

cases and inductive steps of various properties. We supplemented the proposed approach to our SATaGraT tool where a detailed system description can be found in [8]. The table below (cf. Table 1) presents results for well-known list properties. The properties were tested under Ubuntu 10.04 LTS on an AMD 2.0 GHz with 4GB RAM where lemma 4 is proven by a direct proof via lemma 3.

Lemma	Strategy	Base case	Inductive step
`length (xs ++ ys) = length xs + length ys`	induction	8 sec	90 sec
`xs ++ (ys ++ zs) = (xs ++ ys) ++ zs`	induction	0.3 sec	17 sec
`xs ++ [] = xs`	induction	1 sec	1 sec
`[] ++ (xs ++ []) = xs`	direct proof	0 sec	

Table 1. Proven lemmata

6 Conclusion

In this paper, we have introduced an approach for automatically verifying Haskell programs by means of graph transformation units and SAT solving. Our first experiments with this technique nurture the hope that it can be employed for verification proofs in Haskell.

We are planning to undertake further investigations in the future. At the moment, we are working on an automatic translation of a subset of Haskell programs and properties to be proven into graph transformation units. On the one hand, we are thinking about extending this framework step-by-step by further aspects of Haskell like lambda abstractions, local definitions, or user-defined data structures. On the other hand, we could use the preprocessing described in [11] to transform Haskell programs into equivalent Haskell programs consisting only of a subset of Haskell constructs. Hence, it would be sufficient to apply our approach only to this subset. In addition, it could be of interest to compare the GROOVE tool [17] with our approach because GROOVE also allows to explore state spaces.

Acknowledgments. The author is grateful to Hans-Jörg Kreowski for his idea that node deletion can be realized by a special label and to the anonymous referees of WFLP 2013 for their helpful comments. Furthermore, the author would like to thank the anonymous referees and participants of HART2013 for their helpful comments regarding the informal workshop contribution.

References

1. Balbiani, P., Echahed, R., Herzig, A.: A dynamic logic for termgraph rewriting. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 59–74. Springer (2010)

2. Baresi, L., Rafe, V., Rahmani, A. T., Spoletini, P.: An efficient solution for model checking graph transformation systems. Electr. Notes Theor. Comput. Sci. 213(1), 3–21 (2008)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
4. Brus, T. H., van Eekelen, M. C. J. D., van Leer, M. O., Plasmeijer, M. J.: Clean: A language for functional graph writing. In: Kahn, G. (ed.) FPCA. LNCS, vol. 274, pp. 364–384. Springer (1987)
5. Corradini, A., Ehrig, H., Heckel, R., Löwe, M., Montanari, U., Rossi, F.: Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations, pp. 163–245. World Scientific (1997)
6. Davis, M., Logemann, G., Loveland, D. W.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
7. Eén, N., Sörensson, N.: An extensible SAT solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer (2004)
8. Ermler, M.: A SAT-based graph rewriting and verification tool implemented in Haskell. In: WFLP 2013 (2013), accepted for presentation
9. Ermler, M.: Towards a verification framework for Haskell by combining graph transformation units and SAT solving. In: Rose, K., Waldmann, J. (eds.) HART 2013. pp. 26–33 (2013), http://www.imn.htwk-leipzig.de/HART2013/
10. Ermler, M., Kreowski, H.-J., Kuske, S., von Totth, C.: From graph transformation units via MiniSat to GrGen.NET. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 153–168. Springer (2012)
11. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for haskell by term rewriting. ACM Trans. Program. Lang. Syst. 33(2), 7 (2011)
12. Jones, S. L. P.: The Implementation of Functional Programming Languages. Prentice-Hall (1987)
13. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – An overview. In: Degano, P., Nicola, R. D., Meseguer, J. (eds.) Concurrency, Graphs and Models, LNCS, vol. 5065, pp. 57–75. Springer (2008)
14. Kreowski, H.-J., Kuske, S., Wille, R.: Graph transformation units guided by a SAT solver. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 27–42. Springer (2010)
15. Mol, M. D., Eekelen, M. V., Plasmeijer, R.: Theorem proving for functional programmers - SPARKLE: A functional theorem prover. In: IFL 2001, Selected Papers. LNCS, vol. 2312, pp. 55–72. Springer (2001)
16. Plump, D.: Term graph rewriting. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools, pp. 1–61. World Scientific (1999)
17. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J. L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer (2004)
18. Tseitin, G.: On the complexity of derivation in propositional calculus. In: Studies in Constructive Mathematics and Mathematical Logic, Part 2. pp. 115–125 (1968)

FOBS-X: An Extensible Hybrid Functional-Object-Oriented Scripting Language

James Gil de Lamadrid

Bowie State University, Bowie, MD, USA, 20715
jgildelamadrid@bowiestate.edu

Abstract. A language FOBS-X (Extensible FOBS) is described. This language is an interpreted language, intended as a universal scripting language. An interesting feature of the language is its ability to be extended, allowing it to be adapted to new scripting environments.
The interpretation process is structured as a core-language parser back-end, and a macro processor front-end. The macro processor allows the language syntax to be modified. A configurable library is used to help modify the semantics of the language, adding the required capabilities for interacting in a new scripting environment.
This paper focuses on the macro capability of the language. A macro extension to the language has been developed, called the standard extension, that gives FOBS-X a friendlier syntax. It also serves as a convenient tool for demonstrating the macro expansion process.

Keywords: hybrid, scripting, functional, object-oriented.

1 Introduction

The object-oriented programming paradigm and the functional paradigm both offer valuable tools to the programmer. Many problems lend themselves to elegant functional solutions. Others are better expressed in terms of communicating objects. FOBS-X is a single language with the expressive power of both paradigms allowing the user to tackle both types of problems, with fluency in only one language. FOBS-X is a modification to the FOBS language described in Gil de Lamadrid and Zimmerman [4]. The modification involves simplifications to the pointers used in the scoping rules.

FOBS-X has a distinctly functional flavor. In particular, it is characterized by the following features:

- A single, simple, elegant data type called a FOB, that functions both as a function and an object.
- Stateless programming. In the runtime environment, mutable objects are not allowed. Mutation is accomplished, as in functional languages, by the creation of new objects with the required changes.

- A simple form of inheritance. A *sub-FOB* is built from another *super-FOB*, inheriting all attributes from the super-FOB in the process.
- A form of scoping that supports attribute overriding in inheritance. This allows a sub-FOB to replace data or behaviors inherited from a super-FOB.
- A macro expansion capability, enabling the user to introduce new syntax.

As with many scripting languages FOBS-X is weakly typed, a condition necessitated by the fact that it only has one data type. However, with interpreted languages the line between parsing and execution is more blurred than with compiled languages, and the necessity to perform extensive type checking before execution becomes less important.

Several researchers have built hybrid language systems, in an attempt to combine the functional and object-oriented paradigms, but have sacrificed referential transparency in the process. Yau et al. [7] present a language called PROOF. PROOF tries to fit objects into the functional paradigm with little modification to take into account the functional programming style. The language D by Alexandrescu [1] is a rework of the language C transforming it into a more natural scripting language similar to Ruby and Javascript.

Two languages that seek to preserve functional features are FLC by Beaven et al. [2], and FOOPS by Goguen and Mesegner [5]. FOOPS is built around the addition of ADTs to functional features. We feel that the approach of FLC is conceptually simpler. In FLC, classes are represented as functions. This is the basis for FOBS also. In FOBS we have, however, removed the concept of the class. In a stateless environment, the job of the class as a "factory" of individual objects, each with their own state, is not applicable. In stateless systems a class of similar objects is better represented as a single prototype object that can be copied with slight modifications to produce variants.

Scripting languages have tended to shy away from the functional paradigm. Several object-oriented scripting languages such as Python [3] are available. Although mostly object-oriented, its support for functional programming is decent, and includes LISP characteristics such as anonymous functions and dynamic typing. However, Python lacks referential transparency. We consider this as one of the important advantages of FOBS-X. In the design of FOBS-X, we also felt that a simpler data structure could be used to implement objects and the inheritance concept, than was used in this popular language.

2 Language Description

FOBS-X is built around a core language, core-FOBS-X. Core-FOBS-X has only one type of data: the FOB. A *simple FOB* is a quadruplet,

$$[m\ i \text{ -> } e \wedge \rho]$$

The FOB has two tasks. Its first task is to bind an identifier, *i*, to an expression, *e*. The *e-expression* is unevaluated until the identifier is accessed. Its second task is to supply a return value when invoked as a function. ρ (the ρ-*expression*) is an unevaluated expression that is evaluated and returned upon invocation.

The FOB also includes a modifier, *m*. This modifier indicates the visibility of the identifier. The possible values are: "`+", indicating public access, "`~", indicating protected access, and "`$", indicating argument access. Identifiers that are protected are visible only in the FOB, or any FOB inheriting from it. An argument identifier is one that will be used as a formal argument, when the FOB is invoked as a function. All argument identifiers are also accessible as public.

For example, the FOB

```
[`+x -> 3 ^ 6]
```

is a FOB that binds the variable *x* to the value 3. The variable *x* is considered to be public, and if the FOB is used as a function, it will return the value 6.

Primitive data is defined in the FOB-X library. The types *Boolean*, *Char*, *Real,* and *String* have constants with forms close to their equivalent *C* types. The *Vector* type is a container type, with constants of a form close to that of the ML list. For example, the vector

```
["abc", 3, true]
```

represents an ordered list of a string, an integer, and a Boolean value. Semantically, a vector is more like the *Java* type of the same name. It can be accessed as a standard list, using the usual *car*, *cdr,* and *cons* operations, or as an array using indexes, and is implemented as a Perl List structure. Unlike the *Java* type, the FOBS-X type is immutable. The best approximation to the mutate operation is the creation of a brand new modified vector.

There are three operations that can be performed on any FOB. These are called *access*, *invoke*, and *combine*.

An access operation accesses a variable inside a FOB, provided that the variable has been given a public or argument modifier. As an example, in the expression

```
[`+x -> 3 ^ 6].x
```

the operator "." indicates an access, and is followed by the identifier being accessed. The expression would evaluate to the value of *x*, which is 3.

An invoke operation invokes a FOB as a function, and is indicated by writing two adjacent FOBs. In the following example

```
[`$y -> _ ^ y.+[1]] [3]
```

a FOB is defined that binds the variable *y* to the empty FOB and returns the result of the expression *y* + 1, when used as a function. When used as a function, since *y* is an argument variable, the binding of the variable *y* to the empty FOB is considered only a default binding. This binding is replaced by a binding to the actual argument, 3. To do the addition, *y* is accessed for the FOB bound to the identifier "+", and this FOB is invoked with 1 as its actual argument. The result of the invocation is 4.

In an invocation, it is assumed that the second operand is a vector. This explains why the second operand in the above example is enclosed in square braces. Invocation involves binding the actual argument to the argument variable in the FOB, and then evaluating the ρ–expression, giving the return value.

A combine operation is indicated with the operator ";". It is used to implement inheritance. In the following example

```
[`+x -> 3 ^ _] ; [`$y -> _ ^ x.+[y]]          (1)
```

two FOBs are combined. The *super-FOB* defines a public variable *x*. The *sub-FOB* defines an argument variable *y*, and a ρ-expression. Notice that the sub-FOB has unrestricted access to the super-FOB, and is allowed access to the variable *x*, whether modified as public, argument or protected.

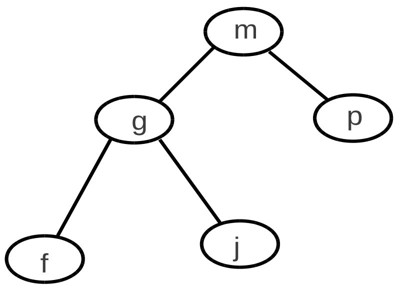


Fig. 1. (a) Example binary search tree.

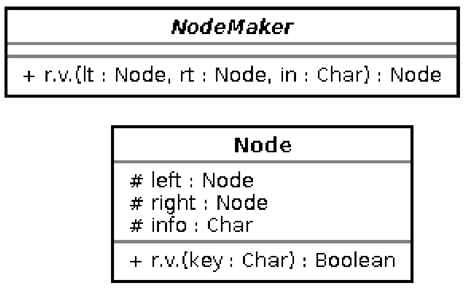

NodeMaker
+ r.v.(lt : Node, rt : Node, in : Char) : Node
Node
# left : Node
# right : Node
# info : Char
+ r.v.(key : Char) : Boolean

Fig. 1. (b) Example FOB structure.

Multiple combine operations result in *FOB stacks*, which are compound FOBs. For example, the following code creates a FOB with an attribute *x* and a two argument function that multiplies its arguments together. The code then uses the FOB to multiply 9 by 2.

```
([`+x -> 5 ^ _] ; [`$a -> _ ^ _] ;
  [`$b -> _ ^ a.*[b]]) [9, 2]
```

In the invocation, the arguments are substituted in the order from top to bottom of the FOB stack, so that the formal argument *a* would be bound to the actual argument 2, and the formal argument *b* would be bound to 9.

In addition to the three FOBS-X operations, many operations on primitive data are defined in the FOBS-X library. These operations include the usual arithmetic, logic, and string manipulation operations. In addition, conversion functions provide conversion from one primitive type to another, when appropriate.

We present a larger example to demonstrate how FOBS code might be used to solve more complex programming problems. In this example we build the binary search tree, shown in Fig. 1(a)., and then search it for the character 'f'. In the FOBS-X solution, we construct a FOB with the structure shown in Fig. 1(b). The *Node* FOB is the prototype that is copied to create *Node* objects. The method called *r.v.* indicates the return value of the FOB.

The FOBS-X code for the example follows.

```
## definition of the NodeMaker FOB
([NodeMaker ->
  [`$lt -> _ ^ _] ;
  [`$rt -> _ ^ _] ;
  [`$in -> _ ^ _] ;
  [`~Node ->
    [`~left -> lt ^ _] ;
    [`~right -> rt ^ _] ;
    [`~info -> in ^ _] ;
    [`~_ -> _ ^
      [`~a1 -> info.=[key] ^ _] ;
      [`~a2 -> FOBS.isEmpty[left].|[a1].if[false,
        left[key]] ^ _];
      [`~a3 -> FOBS.isEmpty[right].|[a1].if[false,
        right[key]]^_];
      [`+a4 -> a1.|[a2].|[a3] ^ _]).a4] ^ _]
  ^ Node] ;
## build the tree
[`+tree ->
  NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
    NodeMaker['j', _, _]], NodeMaker['p', _, _]]
^_]
## search for 'f'
.tree['f']
#.
```

(2)

This code has two types of elements: a FOB expression, and macro directives. Macro directives begin with the "#" character, and are expanded by the macro preprocessor. The two seen here are the comment directive, "##", and the *end of expression* directive, "#.".

The top-level FOB defines a FOB *NodeMaker*, and the search tree, *tree*. The top-level FOB is accessed for the tree, and it is searched for the value "f", using the invoke operator.

NodeMaker creates a FOB with the required attributes, and a return value that does a search. The return value uses the local variables *a1*, *a2*, *a3*, and *a4* to save the results of the comparison with the node, the left child, the right child, and the final result, respectively.

Often it is necessary to compare a FOB with the empty FOB, as in Example (2), where it must be determined if the two subtrees are empty. This is done using code like `FOBS.isEmpty[left]` that uses a function from the library FOB, FOBS.

FOBS-X code is run by feeding FOBS-X expressions to the interpreter. Each FOBS-X expression produces as its value a single FOB, which is returned back to the user and printed. This example would cause the value `true` to be printed.

3 Core-FOBS-X Design Issues

Expression evaluation in FOBS-X is fairly straight forward. Three issues, however, need some clarification. These issues are: the semantics of the redefinition of a variable, the semantics of a FOB invocation, and the interaction between dynamic and static scoping.

3.1 Variable overriding

A FOB stack may contain several definitions of the same identifier, resulting in overriding. For example, in the following FOB

```
[`$m -> 'a' ^ m.toInt[]] ; [`+m -> 3 ^ m]
```

the variable *m* has two definitions; in the super-FOB it is defined as an argument variable, and in the sub-FOB another definition is stacked on top with *m* defined as a public variable. The consequence of stacking on a new variable definition is that it completely overrides any definition of the same variable already in the FOB stack, including the modifier. In addition, the new return value becomes the return value of the full FOB stack.

3.2 Argument substitution

As mentioned earlier, the invoke operator creates bindings between formal and actual arguments, and then evaluates the ρ-expression of the FOB being invoked. At this point we give a more detailed description of the process.

Consider the following FOB that adds together two arguments, and is being invoked with values 10 and 6.

```
([`$r -> 5 ^ _] ; [`$s -> 3 ^ r.+[s]]) [10, 6]
```

The result of this invocation is the creation of the following FOB stack

```
[`$r -> 5 ^ _] ;
[`$s -> 3 ^ r.+[s]] ;
[`+r -> 6 ^ r.+[s]] ;
[`+s -> 10 ^ r.+[s]]
```

In this new FOB the formal arguments are now public variables bound to the actual arguments, and the return value of the invoked FOB has been copied up to the top of the FOB stack. The return value of the original FOB can now be computed easily with this new FOB by doing a standard evaluation of its ρ-expression, yielding a value of 16.

3.3 Variable scope, and expression evaluation

Pure lexical scoping does not cope well with variable overriding, as understood in the object-oriented sense, which typically involves dynamic message binding. To address this issue, FOBS-X uses a hybrid scoping system which combines lexical and dynamic scoping.

Consider the following FOB expression.

```
[`~y -> 1^_] ;
[`~x ->
  [`+n -> y + m ^ n] ;
  [`~m -> 2 ^_]
^_] ;
[`~z -> 3 ^x.n]
```

(3)

This expression defines a FOB stack that is three deep, containing declarations for a protected variable *y*, with value 1, a protected variable *x* with a FOB stack as its value, and a protected variable *z* with the value 3 as its value. The stack that is the value of *x* consists of two FOBs, one defining a public variable *n*, and one defining a protected variable *m*.

We are currently mostly interested in the FOB stack structure of Expression (3), and can represent it graphically with the *stack graph*, given in Fig. 2. In the stack graph each node represents a simple FOB, and is labeled with the variable defined in the FOB. Three types of edges are used to connect nodes: the *s-pointer*, the *t-pointer*, and the *γ-pointer*.

The s-pointer describes the lexical nested block structure of one FOB defined inside of another. The s-pointer for each node points to the FOB in which it is defined. For example *m* is defined inside of the FOB *x*.

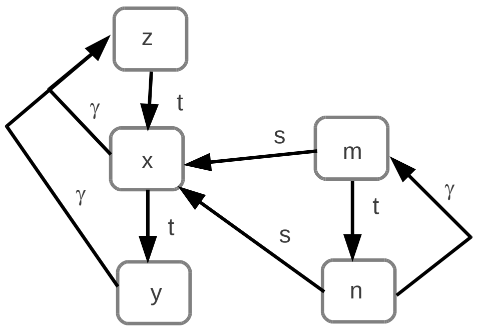


Fig. 2. Stack graph of Example (3).

The t-pointer for each node points to the super-FOB of a FOB. It describes the FOB stack structure of the graph. In Fig. 2 there are basically two stacks: the top level stack consists of nodes *z, x,* and *y,* and the nested stack consisting of nodes *m,* and *n*.

The γ-pointer is a back pointer, that points up the FOB stack to the top. This provides an easy efficient mechanism for finding the top of a stack from any of the nodes in the stack.

If the FOB *z* were invoked, it would access the FOB *n* for the value of *n*. This would cause the expression *y* + *m* to be evaluated, a process that demonstrates the use of all three pointers when searching for the value of *y*.

The process of resolving a reference in FOBS-X first examines the current FOB stack. The top of the current stack is reached by following the γ-pointer. Then the t-pointers are used to search the stack from top to bottom. If the reference is still unresolved, the s-pointer is used to find the FOB stack enclosing the current stack. This enclosing stack now becomes the current stack, and is now searched in the same fashion, from top to bottom, using the γ-pointer to find the top of the stack, and the t-pointers to descend to the bottom.

4 Library Structure

The FOBS-X library contains definitions for both *primitive FOBs,* and *utility FOBs,* which are shown in Fig. 3. These FOBs are all used to define the primitive data types of the FOBS-X language.

Utility FOBs are an organizational tool with similarities to *mix-in classes* described by Page-Jones [6]. Operations that are common among several FOBs are collected into utility FOBs. The utility FOBs are then stacked into other primitive FOBs in the library.

The primitive FOBs of the library mix in the utility FOBs to provide themselves with necessary operations. Fig. 3 shows the mix-in connections in the library, using UML The utility FOBs are shown at the top of the diagram, and the primitive FOBs that inherit from them are shown on the bottom.

There are four utility FOBs in the library:

Numeric: contains the usual arithmetic operations.

Comparable: contains relational operators.

Eq: contains operations for comparing FOBs for equality.

Printable: contains an operation to generate a print-string for a FOB.

The FOB *FOBS* will eventually contain operations for interacting with the outside environment, enabling the language to be used for scripting.

As a more detailed example, consider the primitive FOB *Boolean,* and the utility FOB *Eq*. The contents of these two FOBs are given in Table 1. The table gives the definitions contained in the FOBs, and a brief description of their function. Table 1 shows that the *Boolean* FOB contains operators for the *if* function, the *and* function, the *or* function, and the *not* function. Fig. 3 shows that the FOB *Boolean* inherits from the utility FOBs *Printable,* giving it the ability to be printed, and *Eq*, giving it the ability to be compared with the equals operator. Table 1 shows that the *Eq* FOB provides the operations *equal*, and *not-equal*, to that end.

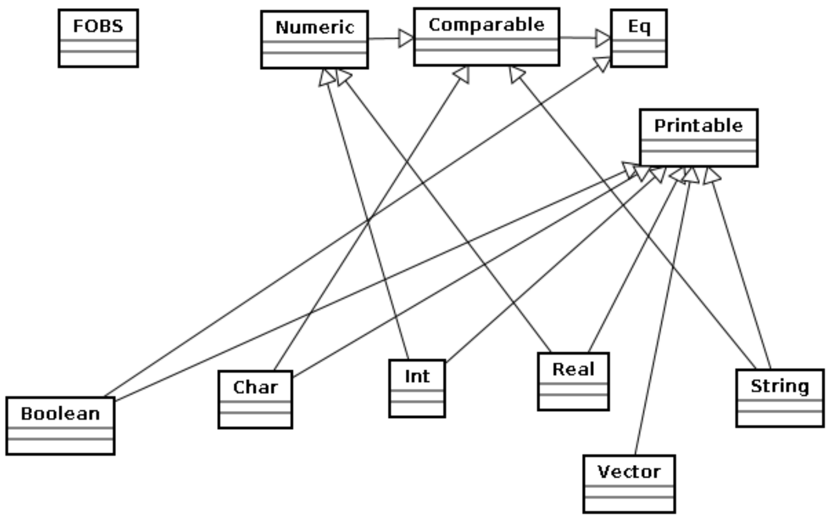


Fig. 3. Mix-in structure of the FOBS library.

Table 1. Operations for the FOBs *Eq*, and *Boolean*.

Library FOB	**Operation**	**Description**
`Boolean`	`b.if[x, y]`	If boolean value *b* is true, return *x*, otherwise return *y*
	`b.&[x]`	Return the boolean value of the expression $b \wedge x$
	`b.|[x]`	Return the boolean value of the expression $b \vee x$
	`b.![]`	Return the boolean value of the expression $\neg b$
`Eq`	`e.=[x]`	Return the boolean value of the expression $e = x$
	`e.!=[x]`	Return the boolean value of the expression $e \neq x$

5 Macro Expansion

FOBS-X allows extensions to its syntax using a macro processor. Macros in FOBS-X are quadruplets $<S_1 \rightarrow S_2$: P, $d>$. The semantics of the quadruplet notation is as follows.

- S_1: the search string, which includes wild-card tokens.
- S_2: the replacement string, which includes wild-card tokens.
- *P*: the priority of the macro, with priority 19 being highest priority, and priority 0 being the lowest.
- *d*: the direction of the scan, with *r* indicating right-to-left, and *l* indicating left-to-right.

The quadruple gives a rewrite rule in which an occurrence of a search string is replaced by a replacement string. Incorporated into the rule is an operator precedence and associativity. As an example, a FOBS-X macro to turn multiplication into an infix operator might appear as follows.

```
< #?multiplicand * #?multiplier →
  ( #?multiplicand .* [ #?multiplier ] ) : 18 , l >
```
(4)

As is often the case in macros, the above macro contains wild-card tokens. These tokens are distinguished from normal literal tokens by the fact that they begin with the "#" character. Wild-card tokens are named, in order to allow references to their bindings. In the example the wild-card tokens `are #?multiplicand`, and `#?multiplier`.

Interpreting the example macro, we start with the search string, S_1 = `#?multiplicand * #?multiplier`. To successfully match this string, first the wild-card token `#?multiplicand` is used to match what is called an *atom*. Then a token "*" is matched, and finally another atom must match the wild-card token `#?multiplier`.

Following the successful match, the replacement string, S_2 = `( #?multiplicand . * [ #?multiplier ] )` is used to replace the matched text, with a string composed of the text matched by the wild-cards, and a few other delimiting characters.

As an example, suppose that the search sting matched the text `x * y`. The wild card `#?multiplicand` would, as a result, be bound to the text `x`, and the wild-card `#?multiplier` would be bound to the text `y`. Substituting these bindings into the replacement string, would yield the replacement text `(x.*[y]).`

From Example (4), we see that a priority of 18 is specified for the multiplication macro. The priority of the macro is used to establish the precedence of operators defined as macros. This macro defines an operator with an extremely high precedence.

To implement priority in the macros, the FOBS-X macro processor does a multi-layered expansion. In this process macro definitions are queued by their priority. Each rule is then matched in the order in which it occurs in the queue. If written correctly, high priority macros consume operands, and then their expansions become operands for lower priority macros, implementing a precedence hierarchy.

Direction in the macro is used to implement associativity for macro defined operators. In the multiplication example, it is indicated that the direction is left-to-right. This macro rule would then be applied by searching for the search string from the left of a FOBS-X expression to its right. Right-to-left direction indicates that the search string is matched moving from the right of the FOBS-X expression to its left. If written correctly, direction can be used to implement associativity. For example, with left-to-right direction, the left most occurrence of an operator would become an operand for the next occurrence of the operator to the right.

It Is possible to have nested macros. This happens in one of two ways: The input text may contain a macro invocation inside of another invocation, or a macro definition may contain an invocation of another macro in its replacement string. In either case, once the macro has been expanded, the replacement text still contains macro invocations. To handle this situation, each time a rule triggers, the macro processor pushes the replacement text on to the front of the input, so that any further processing will reexamine the replacement text, as well as any remaining input.

As mentioned, in FOBS-X macros match text in units of atoms. The term *atom* is often used to refer to the simple tokens of a programming language, such as identifier, constants, or delimiters. In FOBS-X the term is used in a slightly more extended manner. Certainly simple tokens are atoms. However, in addition we allow the macro processor to process compound atoms, consisting of a sequence of atoms enclosed in balanced bracketing symbols. The recognized bracketing symbols are "(", ")", "{", "}", "[", and "]".

6 Macro Files

The syntax for macro definition is demonstrated by the following example that defines the multiplication macro from Example (4).

```
## numeric multiply operator
#defleft
  #?op1 * #?op2
#as
  ( #?op1 .:*: [ #?op2 ] )
#level
  9
#end
#defleft
  :*:
#as
  *
#level
  0
#end
```

(5)

There are two macro definitions in this example. Each definition begins with the directive `#defleft`, and ends with the directive `#end`. The markers `#as`, and `#level` separate the search string from the replacement string, and the replacement string from the priority, respectively. The direction of the macro is indicated be the beginning marker: `#defleft` for left-to-right, and `#defright` for right-to-left.

The example illustrates a common technique used when defining macros that simply move an operator from a prefix position to an infix position. Simply moving the operator leaves it vulnerable to being rematched by the same macro rule again, resulting in infinite macro expansion. The solution is to split the process of replacement between two macro rules. The top rule does indeed move the operator “*”, but also changes it to the string “:*:”. After the top macro rule has finished its work and has been removed from the priority queue, the bottom macro, which is still in the priority queue because of its lower priority, changes the name of the operator form “:*:” back to just “*”.

7 The Standard Extension

The syntax in core-FOBS-X is a little cumbersome. It has been designed with minimalistic notation, allowing a concise formal description. It is not necessarily attractive to the programmer. Standard extension (SE) FOBS-X attempts to rectify this situation. In particular, SE-FOBS-X includes constructs to implement the following

- Allow infix notation for most operators.
- Eliminate the cumbersome syntax associated with declaring a FOB.
- Introduce English keywords to replace some of the more cryptic notation.
- Allow some parts of the syntax to be optionally omitted.

SE-FOBS-X is a language defined entirely using the macro processor. It demonstrates the flexibility of the FOBS-X macro capability to almost entirely rework the syntax of the language, without touching the back-end of the interpreter.

In the previous section we presented Example (5), a macro that converted multiplication from a prefix operation into an infix operation. In SE-FOBS-X this has been done for most operators in the library, using the same technique described in that section.

The definition of a FOB in core-FOBS-X is cryptic, relying on a small set of delimiter characters to structure the definition. In SE-FOBS-X a more verbose mechanism has been provided, using keywords as delimiter. At the same time, SE-FOBS-X eliminates the need to fill in all information when describing a FOB, when there are parts that are not being used. Below, as an example, is one of the macros that helps streamline FOB definitions.

```
#defleft
    fob { #?id ret { #*ret } \ #*x }
#as
    ( [ `~ #?id -> _ ^ #*ret ] ; fob { #*x } )
#level
    3
#end
```

(6)

This macro defines two keywords, *fob*, and *ret*, that delimit the beginning of a FOB definition, and delimit the return value, respectively. This new syntax also allows a *fob* structure that is missing e-expression information, so that the text

```
fob{x ret{3 * 5}\}
```

can be converted into the core-FOBS-X expression

```
([`~x -> _ ^ (3.*[5])] ; _)
```

In this transformation the macro rule fills in the modifier for the variable *x*, and a default value for *x*, which is the empty FOB. The multiplication macro rules convert the multiplication into its infix form.

Close inspection of Example (6) reveals that the *fob* syntax of SE-FOBS-X is set up to to define full FOB stacks, as opposed to just single FOBs. The "\" syntax is designed to introduce new *fob* structures, recursively, in order to string together several FOBs into a stack as shown in the following example.

```
fob{
      public x val{3} \
      y val{5} ret{x + y} \
}
```

which expands to the FOB stack

```
([`+x -> 3 ^ _] ; ([`~y -> 5 ^ (x.+[y])] ; _))
```

Each element of the FOB stack is enclosed in the *fob* structure, and terminated by the "\" symbol. Each element can contain an optional modifier keyword, such as `public`, an optional *val* structure, giving its e-expression, and an optional *ret* structure, providing the ρ-expression.

8 An SE-FOBS-X Example

We present a larger example to demonstrate how SE-FOBS-X code is used to solve more complex programming problems. This example is the SE-FOBS-X version of Example (2).

```
#use #SE
## definition of the NodeMaker FOB
 (fob{
  NodeMaker
  val{
    fob{
      argument lt \
      argument rt \
      argument in \
      Node
      val{
        fob{
          left val {lt} \
          right val {rt} \
          info val {in} \
          argument key
          ret{
            (fob{
              a1 val {info = key} \
              a2
              val{
                if {nofob left | a1}
                then {false}
                else {left[key]}
              } \
              a3
              val{
                if {nofob right | a1 | a2}
                then {false}
                else {right[key]}
              } \
              public a4 val{a1 | a2 | a3} \
                  } ).a4
          } \
        }
      }
      ret {Node} \
    }
  } \
  ## build the sample tree
  public tree
  val{
    NodeMaker['m', NodeMaker['g', NodeMaker['f', _, _],
      NodeMaker['j', _, _]], NodeMaker['p', _, _]]
  } \
} )
## use the main FOB tree variable to search for 'f'
.tree['f']
#.
#!
```

(7)

This code has two types of elements: a FOB expression, and macro directives. The four macro directives seen here are the comment directive, “##”, the *end of expression* directive, “#.”, the *end of script* directive, “#!”, and the “#use” directive. The “#use” directive installs an extension with the given name, by locating the macro file and processing it, and also locating the corresponding library modules and loading them into the FOBS primitive FOB.

Several Features of SE-FOBS-X are demonstrated by the example. These include more readable syntax for primitive FOBS-X functions, such as the *if* construct that translates into an invocation of the Boolean primitive FOB function of the same name, and the `nofob` operator that translates into a call to the FOBS FOB function *isEmpty*. Also shown is the use of the SE-FOBS-X modifier names `public`, and `argument`, instead of the more cryptic core-FOBS-X equivalents. Finally, the example demonstrates the use of the SE-FOBS-X *fob-val-ret* structure, with optional *val* and *ret* parts.

9 Conclusion

We have presented a core FOBS-X language. This language is designed as the basis of a universal scripting language. It has a simple syntax and semantics.

FOBS-X is a hybrid language, which combines the tools and features of object oriented languages with the tools and features of functional languages. In fact, the defining data structure of FOBS-X is a combination of an object and a function. The language provides the advantages of referential transparency, as well as the ability to easily build structures that encapsulate data and behavior. This provides the user the choice of paradigms.

Core-FOBS-X is the core of an extended language, SE-FOBS-X, in which programs are translated into the core by a macro processor. This allows for a language with syntactic "sugar", that still has the simple semantics of our core-FOBS-X language.

Because of the ability to be extended, which is utilized by SE-FOBS-X, the FOBS-X language gains the flexibility that enables it to be a universal scripting language. The language can be adapted syntactically, using the macro capability, to new scripting applications. In the future, the library will also be adaptable, allowing the user to add the operations necessary to adapt it to interact with a new environment.

References

1. Alexandrescu, A.: **The D Programming Language**. Adison Wesley (2010)
2. Beaven, M., Stansifer, R., Wetlow, D.: **A Functional Language with Classes. In: Lecture Notices in Computer Science**, 507, (1991)
3. Beazley, D., Van Rossum, G**.: Python; Essential Reference**. New Riders Publishing, Thousand Oaks, CA, (1999)
4. Gil de Lamadrid, J., Zimmerman, J.: Core FOBS: A Hybrid Functional and Object-Oriented Language. In: **Computer Languages, Systems & Structures**, 38, (2012)
5. Goguen, J. A., Mesegner, J.: Unifying Functional, Object-Oriented, and Relational Programming with Logical Semantics. In: **Research Directions in Object-Oriented Programming**, MIT Press, pp. 417-478 (1987)
6. Page-Jones, M.: **Fundamentals of Object-Oriented Design in UML**. Addison Wesley, pp. 327-336 (2000)
7. Yau, S.S., Jia, X., Bae, D. H.: Proof: A Parallel Object-Oriented Functional Computation Model. In: **Journal of Parallel Distributed Computing**, 12 (1991)

Towards Parallel Constraint-Based Local Search with the X10 Language

Danny Munera[1], Daniel Diaz[1], and Salvador Abreu[2]

[1] University of Paris 1-Sorbonne, France
Danny.Munera@malix.univ-paris1.fr, Daniel.Diaz@univ-paris1.fr
[2] Universidade de Évora and CENTRIA, Portugal
spa@di.uevora.pt

Abstract. In this study, we started to investigate how the Partitioned Global Address Space (PGAS) programming language X10 would suit the implementation of a Constraint-Based Local Search solver. We wanted to code in this language because we expect to gain from its ease of use and independence from specific parallel architectures. We present the implementation strategy, and search for different sources of parallelism. We discuss the algorithms, their implementations and present a performance evaluation on a representative set of benchmarks.

1 Introduction

Constraint Programming has been successfully used to model and solve many real-life problems in diverse areas such as planning, resource allocation, scheduling and product line modeling [16, 17]. Classically constraint satisfaction problems (CSPs) may be solved exhaustively by complete methods which are able to find all solutions, and therefore determine whether any solutions exist. However efficient these solvers may be, a significant class of problems remains out of reach because of exponential growth of search space, which must be exhaustively explored. Another approach to solving CSPs entails giving up completeness and resorting to (meta-) heuristics which will guide the process of searching for solutions to the problem. Solvers in this class make choices which limit the search space which actually gets visited, enough so to make problems tractable. For instance a complete solver for the *magic squares* benchmark will fail for problems larger than 15×15 whereas a local search method will easily solve a 100×100 problem instance within the lower resource bounds. On the other hand, a local search procedure may not be able to find a solution, even when one exists.

However, it is unquestionable that the more computational resources are available, the more complex the problems that may be solved. We would therefore like to be able to tap into the forms of augmented computational power which are actually available, as conveniently as feasible. This requires taming various forms of explicitly parallel architectures.

Present-day parallel computational resources include increasingly multi-core processors, General Purpose Graphic Processing Units (GPGPUs), computer clusters and grid computing platforms. Each of these forms requires a different programming model and the use of specific software tools, the combination of which makes software development even more difficult.

The foremost software platforms used for parallel programming include POSIX Threads [1] and OpenMP [15] for shared-memory multiprocessors and multicore CPUs, MPI [20] for distributed-memory clusters or CUDA [14] and OpenCL [10] for massively parallel architectures such as GPGPUs. This diversity is a challenge from the programming language design standpoint, and a few proposals have emerged that try to simultaneously address the multiplicity of parallel computational architectures.

Several modern language designs are built around the Partitioned Global Address Space (PGAS) memory model, as is the case with X10 [19], Unified Parallel C [7] or Chapel [5]. Many of these languages propose abstractions which capture the several forms in which multiprocessors can be organized. Other, less radical, approaches consist in supplying a library of inter-process communication which relies on and uses a PGAS model.

In our quest to find a scalable and architecture-independent implementation platform for our exploration of high-performance parallel constraint-based local search methods, we decided to experiment with one of the most promising new-generation languages, X10 [19].

The remainder of this article is organized as follows: Section 2 discusses the PGAS Model and briefly introduces the X10 programming language. Section 3 introduces native X10 implementations exploiting different sources of parallelism of the Adaptive Search algorithm. Section 4 presents an evaluation of these implementations. A short conclusion ends the paper.

2 X10 and the Partitioned Global Address Space (PGAS) model

The current arrangement of tools to exploit parallelism in machines are strongly linked to the platform used. As it was said above, two broad programming models stand out in this matter: *distributed* and *shared memory* models. For large distributed memory systems, like clusters and grid computing, Message Passing Interface (MPI) [20] is a de-facto programming standard. The key idea in MPI is to decompose the computation over a collection of processes with private memory space. These processes can communicate with each other through message passing, generally over a communication network.

With the recent growth of many-core architectures, the shared memory approach has increased its popularity. This model decomposes the computation in multiple threads of execution sharing a common address space, communicating with each other by reading and writing shared variables. Actually, this is the model used by traditional programming tools like Fortran or C through libraries like *pthreads* [1] or OpenMP [15].

The PGAS model tries to combine the advantages of the two approaches mentioned so far. This model extends shared memory to a distributed memory setting. The execution model allows having multiple processes (like MPI), multiple threads in a process (like OpenMP), or a combination (see Figure 1). Ideally, the user would be allowed to decide how tasks get mapped to physical resources. X10 [19], Unified Parallel C [22] and Chapel [5] are examples of PGAS-enabled languages, but there exist also PGAS-based IPC libraries such as GPI [12], for use in traditional programming languages. For the experiments described herein, we used the X10 language.

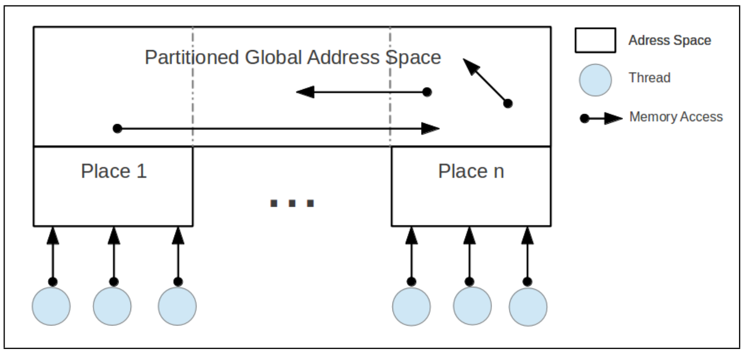


Fig. 1. PGAS Model

X10 [19] is a general-purpose language developed by IBM, which provides a PGAS variation: Asynchronous PGAS (APGAS). APGAS extends the PGAS model making it flexible, even in non-HPC platforms [18]. Through this model X10 can support different levels of concurrency with simple language constructs.

There are two main abstractions in the X10 model: *places* and *activities*. A *place* is the abstraction of a virtual shared-memory process, it has a coherent portion of the address space together with threads (activities) that operate on that memory. The X10 construct for creating a place in X10 is *at*, and is commonly used to create a place for each processing unit in the platform. An *activity* is the mechanism to abstract the single threads that perform computation within a place. Multiple activities may be active simultaneously in a place.

X10 implements the major components of the PGAS model, by the use of places and activities. However, the language includes other interesting tools with the goal of improving the abstraction level of the language. Synchronization is supported thanks to various operations such as *finish*, *atomic* and *clock*. The operation *finish* is used to wait for the termination of a set of activities, it behaves like a traditional barrier. The constructs *atomic* ensures an exclusive access to a critical portion of code. Finally, the construct *clock* is the standard way to ensure the synchronization between activities or places. X10 supports the distributed array construct, which makes it possible to divide an array into sub-arrays which are mapped to available places. Doing this ensures a local access from each place to the related assigned sub-array. A detailed examination of X10, including tutorial, language specification and examples can be consulted at http://x10-lang.org/.

3 Native X10 Implementations of Adaptive Search

In order to take advantage of the parallelism it is necessary to identify the sources of parallelism of the Adaptive Search algorithm. In [4], the authors survey the state-of-the-art of the main parallel meta-heuristic strategies and discuss general design and implementation principles. They classify the decomposition of activities for parallel work in two main groups: *functional parallelism* and *data parallelism* (also known as OR-parallelism and AND-parallelism in the Logic Programming community).

On the one hand, in *functional parallelism* different tasks run on multiple compute instances across the same or different datasets. On the other hand, *data parallelism* refers to the methods in which the problem domain or the associated search space is decomposed. A particular solution methodology is used to address the problem on each of the resulting components of the search space. This article reports on our experiments concerning both kinds of parallelism applied to the Adaptive Search method.

3.1 Sequential Implementation

Our first experiment with AS in X10 was to develop a sequential implementation corresponding to a specialized version of the Adaptive Search for permutation problems [13][3].

Figure 2 shows the class diagram of the basic X10 project. The class *ASPermutSolver* contains the Adaptive Search permutation specialized method implementation. This class inherits the basic functionality from a general implementation of the Adaptive Search solver (in class *AdaptiveSearchSolver*), which in turn inherits a very simple Local Search method implementation from the class *LocalSearchSolver*. This class is then specialized for different parallel approaches, which we experimented with. As we will see below, we experimented with two versions of Functional Parallelism (FP1 and FP2) and a Data Parallelism version (called Random Walk, i.e. RW).

Moreover, a simple CSP model is described in the class *CSPModel*, and specialized implementations of each CSP benchmark problem are contained in the classes *PartitModel, MagicSquareModel, AllIntervallModel* and *CostasModel*, which have all data structures and methods to implement the error function of each problem.

Listing 1.1 shows a simplified skeleton code of our X10 sequential implementation, based on Algorithm 1. The core of the Adaptive Search algorithm is implemented in the method *solve*. The *solve* method receives a *CSPModel* instance as parameter. On line 8, the CSP variables of the model are initialized with a random permutation. On the next line the total cost of the current configuration is computed. The *while* instruction on line 10 corresponds to the main loop of the algorithm. The *selectVarHighCost* function (Line 12) selects the

[3] In a permutation problem, all N variables have the same initial domain of size N and are subject to an implicit *all-different* constraint. The associated algorithm is reported in the appendix.

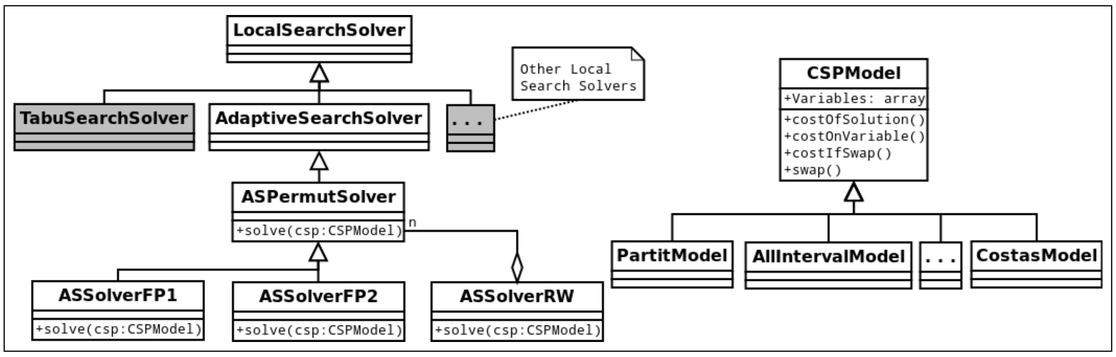


Fig. 2. X10 Class Diagram basic project

variable with the maximal error and saves the result in the *maxI* variable. The *selectVarMinConflict* function (Line 13) selects the best neighbor move from the highest cost variable *maxI*, and saves the result in the *minJ* variable. Finally, if no local minimum is detected, the algorithm swaps the variables *maxI* and *minJ* (permutation problem) and computes the total cost of the resulting new configuration (Line 16). The solver function ends if the *totalCost* variable equals 0 or when the maximum number of iterations is reached.

Listing 1.1. Simplified AS X10 Sequential Implementation

```
 1  class ASPermutSolver {
 2    var totalCost: Int;
 3    var maxI: Int;
 4    var minJ: Int;
 5
 6    public def solve (csp: CSPModel): Int {
 7      . . . local variables . . .
 8      csp.initialize();
 9      totalCost = csp.costOfSolution();
10      while (totalCost != 0) {
11        . . . restart code . . .
12        maxI = selectVarHighCost (csp);
13        minJ = selectVarMinConflict (csp);
14        . . . local min tabu list, reset code . . .
15        csp.swapVariables (maxI, minJ);
16        totalCost = csp.costOfSolution ();
17      }
18      return totalCost;
19    }
20  }
```

3.2 Functional Parallel Implementation

Functional parallelism is our first attempt to parallelize the Adaptive Search algorithm. The key aim for this implementation is to decompose the problem into different tasks, each task working in parallel on the same data. To achieve this objective it is necessary to change the inner loop of the sequential Adaptive Search algorithm.

In this experiment, we decided to change the structure of the *selectVarHighCost* function, because therein lies the most costly activities performed in the inner loop. The most important task performed by this function is to go through the variable array of the CSP model to compute the cost of each variable (in order to select the variable with the highest cost). A X10 skeleton implementation of *selectVarHighCost* function is presented in Listing 1.2.

Listing 1.2. Function selVarHighCost in X10

```
public def selectVarHighCost( csp : CSPModel ) : Int {
  . . . local variables . . .
  // main loop: go through each variable in the CSP
  for (i = 0; i < size; i++) {
    . . . count marked variables . . .
    cost = csp.costOnVariable (i);
    . . . select the highest cost . . .
  }
  return maxI; // (index of the highest cost)
}
```

Since this function must process the entire variable vector at each iteration, it is then natural to try to parallelize this task. For problems with many variables (e.g. the magic square problem involves N^2 variables) the gain could be very interesting. We developed a *first approach* (called FP1), in which `n` single activities are created at each iteration. Each activity processes a portion of the variables array and performs the required computations. The X10 construct *async* was chosen to create individual *activities* sharing the global array. Listing 1.3 shows the X10 skeleton code for the *first approach* of the *functional parallelism* in the function *selectVarHighCost*.

Listing 1.3. First approach to *functional parallelism*

```
 1  public def selectVarHighCost (csp : CSPModel) : Int {
 2    // Initialization of Global variables
 3    var partition : Int = csp.size/THNUM;
 4    finish for(th in 1..THNUM){
 5      async{
 6        for (i = ((th−1)*partition); i < th*partition; i++){
 7          . . . calculate individual cost of each variable . . .
 8          . . . save variable with higher cost . . .
 9        }
10      }
11    }
12    . . . terminate function: merge solutions . . .
13    return maxI; //(Index of the higher cost)
14  }
```

In this implementation the constant `THNUM` on line 4 represents the number of concurrent activities that are deployed by the program. On the same line, the keyword *finish* ensures the termination of all spawned activities. Finally, the construct *async* on line 5 spawns independent individual tasks to cross over a portion of the variable array (sentence *for* on line 6). With this strategy we face up with a well known problem of functional parallelism: the overhead due to the management of fine-grained activities. As expected results are not good enough (see Section 4 for detailed results).

In order to limit the overhead due to activity creation, we implemented a *second approach* (called FP2). Here the n working activities are created at the very beginning of the solving process, just before the main loop of the algorithm. These activities are thus available for all subsequent iterations. However, it is necessary to develop a synchronization mechanism to assign tasks to the working activities and to wait for their termination. For this purpose we created two new classes: *ComputePlace* and *ActivityBarrier*. *ComputePlace* is a compute instance, which contains the functionality of the working activities. *ActivityBarrier* is a very simple barrier developed with X10 monitors (X10 concurrent package).

Listing 1.4 shows the X10 implementation of the *second approach*.

Listing 1.4. Second approach to *functional parallelism*

```
public class ASSolverFP1 extends ASPermutSolver{
  val computeInst : Array[ComputePlace];
  var startBarrier : ActivityBarrier;
  var doneBarrier : ActivityBarrier;

  public def solve(csp : CSPModel):Int{
    for(var th : Int = 1; th <= THNUM ; th++)
      computeInst(th)[4] = new ComputePlace(th , csp);

    for(id in computeInst)
      async computeInst(id).run();

    while(total_cost!=0){
      . . . restart code . . .
      for(id in computeInst)
        computeInst(id).activityToDo = SELECVARHIGHCOST;

      startBarrier.wait(); // send start signal
      // activities working...
      doneBarrier.wait(); // work ready
      maxI=terminateSelVarHighCost();
      . . . local min tabu list, reset code . . .
    }
    // Finish activities
    for(id in computeInst)
      computeInst(id).activityToDo = FINISH;

    startBarrier.wait();
    doneBarrier.wait();
```

[4] Remark: in X10 the array notation is *table(index)* instead of *table[index]* as in C.

```
30        return totalCost;
31     }
32  }
```

This code begins with the definition of three global variables on lines 2-4: *computeInst*, *startBarrier* and *doneBarrier*; *computeInst* is an array of *ComputePlace* objects, one for each working activity desired. *startBarrier* and *doneBarrier* are *ActivityBarrier* instances created to signalize the starting and ending of the task in the compute place. On lines 7-11, before the main loop `THNUM` working activities are created and started over an independent X10 activity. When the algorithm needs to execute the *selectVarHighCost* functionality, the main activity assigns this task putting a specific value into the variable *activityToDo* in the corresponding instance of the *ComputePlace* class (lines 15 and 16), then the function *wait()* is executed over the barrier *startBarrier* to notify all working activities to start (line 18). Finally, the function *wait()* is executed over the barrier *doneBarrier* to wait the termination of the working activities (line 20). Then on line 21 the main activity can process the data with the function *terminateSelVarHighCost*. When the main loop ends, all the working activities are notified to end and the *solve* function returns (lines 25-30). Unfortunately, as we will see below, the improvement of this *second approach* is not important enough (and, in addition, it has its own overhead due to synchronization mechanisms).

3.3 Data Parallel Implementation

A straightforward implementation of data parallelism in the Adaptive Search algorithm is the multiple independent Random Walks (IRW) approach. The idea is to use isolated sequential Adaptive Search solver instances dividing the search space of the problem through different random starting points. This strategy is also known as Multi Search (MPSS, Multiple initial Points, Same search Strategies) [4] and has proven to be very efficient [6, 11].

The key of this implementation is to have several independent and isolated instances of the Adaptive Search Solver applied to the same problem model. The problem is distributed to the available processing resources in the computer platform. Each solver runs independently (starting with a random assignment of values). When one instance finds a solution it is necessary to stop all other running instances. This is achieved using a termination detection communication strategy. This simple parallel version has no inter-process communication, making it *Embarrassingly* or *Pleasantly Parallel*. The skeleton code of the algorithm is shown in the Listing 1.5.

Listing 1.5. Adaptive Search *data parallel* X10 implementation

```
public class ASSolverRW{
   val solDist : DistArray[ASPermutSolver];
   val cspDist : DistArray[CSPModel];
   def this( ){
      solDist=DistArray.make[ASPermutSolver](Dist.makeUnique());
      cspDist=DistArray.make[CSPModel](Dist.makeUnique());
```

```
 7     }
 8     public def solve(){
 9        val random = new Random();
10        finish for(p in Place.places()){
11           val seed = random.nextLong();
12           at(p) async {
13              cspDist(here.id) = new CSPModel(seed);
14              solDist(here.id) = new ASPermutSolver(seed);
15              cost = solDist(here.id).solve(cspDist(here.id));
16              if (cost==0){
17                 for (k in Place.places())
18                    if (here.id != k.id)
19                       at(k) async{
20                          solDist(here.id).kill = true;
21                       }
22              }
23           }
24        }
25        return cost;
26     }
27  }
```

For this implementation the *ASSolverRW* class was created. The algorithm has two global distributed arrays: *solDist* and *cspDist* (lines 2 and 3). As explained in Section 2, the *DistArray* class creates an array which is spread across multiple X10 places. In this case, an instance of *ASPermutSolver* and *CSPModel* are stored at each available place in the program. On lines 5 and 6 function *make* creates and initializes the ditributed vector in the region created by the function *Dist.makeUnique()* (*makeUnique* function creates a distribution over a region that maps every point in the region to a distinct place, and which maps some point in the region to every place). On line 10 a *finish* operation is executed over a *for* loop that goes through all the places in the program (*Place.places()*). Then, an activity is created in each place with the sentence *at(p) async* on line 12. Into the *async* block, a new instance of the solver (*new ASPermutSolver(seed)*) and the problem (*new CSPModel(seed)*) are created (lines 13 and 14) and a random seed is passed. On line 15, the solving process is executed and the returned cost is assigned to the *cost* variable. If this cost is equal to 0, the solver in a place has reached a valid solution, it is then necessary to send a termination signal to the remaining places (lines 16- 22). For this, every place (i.e. every solver), checks the value of a *kill* variable at each iteration. When it becomes equal to *true* the main loop of the solver is broken and the activity is finished. To set a *kill* remote variable from any X10 place it was necessary to create a new activity into each remaining place (sentence *at(k) async* on line 19) and into the *async* block to change the value of the *kill* variable. On line 18, the sentence *if (here.id != k.id)* filters all places which are not the winning one (*here*). Finally, the function returns the solution of the fastest place on line 25.

4 Performance Analysis

In this section, we present and discuss our experimental results of our X10 implementations of the Adaptive Search algorithm. The testing environment used was a non-uniform memory access (NUMA) computer, with 2 Intel Xeon W5580 CPUs each one with 4 hyper-threaded cores running at 3.2GHz as well as a system based on 4 16-core AMD Opteron 6272 CPUs running at 2.1GHz.

We used a set of benchmarks composed of four classical problems in constraint programming: the magic square problem (MSP), the number partitioning problem (NPP) and the all-interval problem (AIP), all three taken from the CSPLib [8]; also we include the Costas Arrays Problem (CAP) introduced in [9], which is a very challenging real problem. The problems were all tested on significantly large instances. The interested reader may find more information on these benchmarks in [13].

It is worth noting, at the software level, that the X10 runtime system can be deployed in two different backends: Java backend and C++ backend; they differ in the native language used to implement the X10 program (Java or C++), also they present different trade-offs on different machines. Currently, the C++ backend seems relatively more mature and faster for scientific computation. Therefore, we have chosen it for this experimentation.

Regarding the stochastic nature of the Adaptive Search behavior, several executions of the same problem were done and the times averaged. We ran 100 samples for each experimental case in the benchmark.

In this presentation, all tables report raw times in seconds (average of 100 runs) and relative speed-ups. These tables respect the same format: the first column identifies the problem instance, the second column is the execution time of the problem in the sequential implementation, the next group of columns contains the corresponding speed-up obtained with a varying number of cores (places), and the last column presents the execution time of the problem with the highest number of places.

4.1 Sequential Performance

Even if our first goal in using X10 is parallelism, it is interesting to compare the sequential X10 implementation with a reference implementation: our low-level and highly optimized C version initially used in [2, 3] and continuously improved since then. The X10 implementation appears to be 3 to 5 times slower than the C version: this is not a prohibitive price to pay, if one takes into account the possibilities promised by X10 for future experimentation.

A possible explanation of the difference between the performances of both implementations is probably the richness of the X10 language (OOP, architecture abstractions, communication abstractions, etc.). Also, maybe it is necessary to improve our X10 language skills good enough to get the best performance of this tool.

4.2 Functional Parallel Performance

Table 1 shows the results of the *first version* of the *functional parallelism* X10 implementation. Only two benchmarks (2 instances of MSP and CAP) are presented. Indeed, we did not investigate this approach any further since the results are clearly not good. Each problem instance was executed with a variable number of activities (`THNUM = 2, 4` and `8`). It is worth noting, that the environmental X10 variable *X10_NTHREADS* was passed to the program with an appropriate value to each execution. This variable controls the number of initial working threads per place in the X10 runtime system.

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
`MSP-100`	11.98	0.86	0.95	0.77	15.49
`MSP-120`	24.17	1.04	0.97	0.98	24.65
`CAP-17`	1.56	0.43	0.28	0.24	6.53
`CAP-18`	12.84	0.51	0.45	0.22	57.16

Table 1. Functional Parallelism – first approach (timings and speed-ups)

As seen in Table 1, for all the treated cases the obtained speed-up is less than 1 (i.e. a slowdown factor), showing a deterioration of the execution time due to this parallel implementation. So, it is possible to conclude that no gain time is obtainable in this approach. To analyze this behavior it is important to return to the description of the Listing 1.3. As already noted, the parallel function *selVarHighCost* in this implementation are located into the main loop of the algorithm, so `THNUM` activities are created, scheduled and synchronized at each iteration in the program execution, being a very important source of overhead. The results we obtained suggest that this overhead is larger than the improvement obtained by the implementation of this parallel strategy.

Turning to the *second approach*, Table 2 shows the results obtained with this strategy. Equally, the number of activities spawn, in this case at the beginning, was varied from 2 to 8.

Problem instance	time (s) seq.	speed-up with k places 2	4	8	time (s) 8 places
`MSP-100`	11.98	1.15	0.80	0.86	13.87
`MSP-120`	24.17	1.23	0.94	0.63	38.34
`CAP-17`	1.56	0.56	0.30	0.25	6.35
`CAP-18`	12.84	0.74	0.39	0.27	46.84

Table 2. Functional Parallelism – second approach (timings and speed-ups)

Even if the results are slightly better, there is no noticeable speed-up. This is due to a new form of overhead due to the synchronization mechanism which is used in the inner loop of the algorithm to assign tasks and to wait for their termination (see Listing 1.4).

4.3 Data Parallel Performance

Table 3 and Figure 3 document the speedups we obtained when resorting to data parallelism. Observe that, for this particular set of runs, we used a different hardware platform, with more cores than for the other runs.

Problem instance	time (s) seq.	speed-up with k places 8	16	24	32	time (s) 32 places
`AIP-300`	56.7	4.7	7.1	9.9	10.0	5.6
`NPP-2300`	6.6	6.1	9.8	10.5	12.0	0.5
`MSP-200`	365	8.3	12.2	13.6	14.6	24.9
`CAP-20`	731	5.6	12.0	16.1	20.5	35.7

Table 3. Data Parallelism (timings and speed-ups)

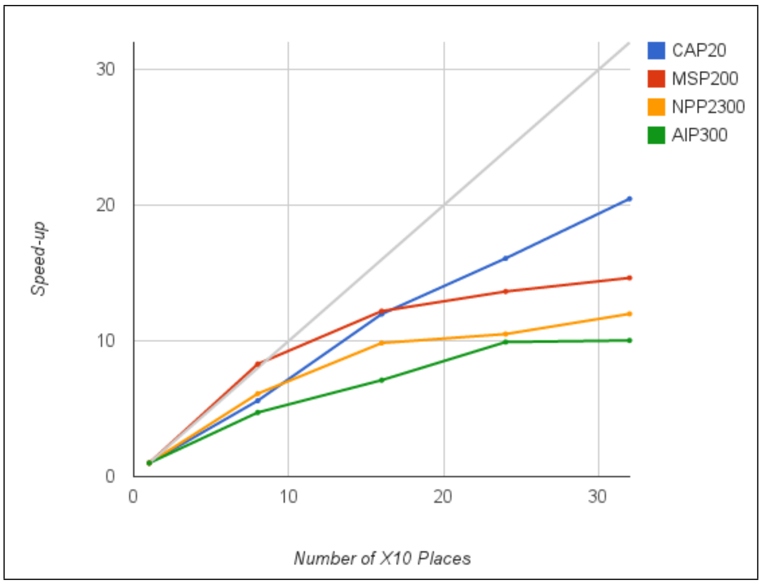


Fig. 3. Speed-ups for the most difficult instance of each problem

The performance of data parallel version is clearly above the performance of the functional parallel version. The resulting average runtime and the speed-ups obtained in the entire experimental test performed seems to lie within the predictable bounds proposed by [21]. The Costas Arrays Problem displays remarkable performance with this strategy, e.g. the CAP reaches a speed-up of 20.5 with 32 places. It can be seen that the speed-up increases almost linearly with the number of used places. However, for other problems (e.g. MSP), the curve clearly tends to flat when the number of places increases.

5 Conclusion and Future Work

We presented different parallel X10 implementations of an effective Local Search algorithm, Adaptive Search in order to exploit various sources of parallelism. We first experimented two functional parallelism versions, i.e. trying to divide the inner loop of the algorithm into various concurrent tasks. This turns out to yield no speed-up at all, most likely because of the bookkeeping overhead (creation, scheduling and synchronization) that is incompatible with such a fine-grained level of parallelism.

We then proceeded with a data parallel implementation, in which the search space is decomposed into possible different random initial configurations of the problem and getting isolated solver instances to work on each point concurrently. We got a good level of performance for the X10 data-parallel implementation with monotonously increasing speed-ups in all problems we studied, although they taper off after some point.

The main result we draw from this experiment, is that X10 has proved a suitable platform to exploit parallelism in different ways for constraint-based local search solvers. These entail experimenting with different forms of parallelism, ranging from single shared memory inter-process communication to a distributed memory programming model. Additionally, the use of the X10 implicit communication mechanisms allowed us to abstract away from the complexity of the parallel architecture with a very simple and consistent device: the *distributed arrays* and the *termination detection system* in our data parallel implementation.

Considering that straightforward forms of parallelism seem to get lower gains as we increase the number of cores, we want to look for ways of improving on this situation. Future work will focus on the implementation of a cooperative Local Search parallel solver based on data parallelism. The key idea is to take advantage of the many communications tools available in this APGAS model, to exchange information between different solver instances in order to obtain a more efficient and, most importantly, scalable solver implementation. We also plan to test the behavior of a cooperative implementation under different HPC architectures, such as the many-core Xeon Phi, GPGPU accelerators and grid computing platforms.

References

1. David Butenhof. *Programming With Posix Threads*. Addison-Wesley Professional, 1997.

2. Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In Kathleen Steinhöfel, editor, *Stochastic Algorithms: Foundations and Applications*, pages 342–344. Springer Berlin Heidelberg, London, 2001.
3. Philippe Codognet and Daniel Diaz. An Efficient Library for Solving CSP with Local Search. In *5th international Conference on Metaheuristics*, pages 1–6, Kyoto, Japan, 2003.
4. Teodor Gabriel Crainic and Michel Toulouse. Parallel Meta-Heuristics. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, number May, pages 497–541. Springer US, 2010.
5. Cray Inc. *Chapel Language Specification Version 0.91.* 2012.
6. Daniel Diaz, Salvador Abreu, and Philippe Codognet. Targeting the Cell Broadband Engine for constraint-based local search. *Concurrency and Computation: Practice and Experience (CCP&E)*, 24(6):647–660, 2011.
7. Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. *Wiley: UPC: Distributed Shared Memory Programming - Tarek El-*. Wiley, 2005.
8. I.P. Gent and T. Walsh. "CSPLib: a benchmark library for constraints. Technical report, 1999.
9. Serdar Kadioglu and Meinolf Sellmann. Dialectic Search. In *Principles and Practice of Constraint Programming (CP)*, volume 5732, pages 486–500, 2009.
10. Khronos OpenCL Working Group. *OpenCL Specification.* 2008.
11. Rui Machado, Salvador Abreu, and Daniel Diaz. Parallel Local Search : Experiments with a PGAS-based programming model. In *12th International Colloquium on Implementation of Constraint and Logic Programming Systems*, pages 1–17, Budapest, Hungary, 2012.
12. Rui Machado and Carsten Lojewski. The Fraunhofer virtual machine: a communication library and runtime system based on the RDMA model. *Computer Science - R&D*, 23(3-4):125–132, 2009.
13. Danny Múnera, Daniel Diaz, and Salvador Abreu. Experimenting with X10 for Parallel Constraint-Based Local Search. In Ricardo Rocha and Christian Theil Have, editors, *Proceedings of the 13th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2013)*, August 2013.
14. NVIDIA. CUDA C Programming Guide, 2013.
15. OpenMP. The OpenMP API specification for parallel programming.
16. Francesca Rossi, Peter Van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence.* Elsevier Science, 2006.
17. Camille Salinesi, Raul Mazo, Olfa Djebbi, Daniel Diaz, and Alberto Lora-michiels. Constraints : the Core of Product Line Engineering. In *Conference on Research Challenges in Information Science (RCIS)*, number ii, pages 1–10, Guadeloupe, French West Indies, France, 2011.
18. Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier Tardieu. The Asynchronous Partitioned Global Address Space Model. In *The First Workshop on Advances in Message Passing*, pages 1–8, Toronto, Canada, 2010.
19. Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification - Version 2.3. Technical report, 2012.
20. Mark Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI : The Complete Reference.* The MIT Press, 1996.
21. Charlotte Truchet, Florian Richoux, and Philippe Codognet. Prediction of parallel speed-ups for las vegas algorithms. 2013.
22. UPC Consortium, editor. *UPC Language Specifications.* 2005.

Algorithm 1 Adaptive Search Base Algorithm

Input: problem given in CSP format:

- set of variables $V = \{X_1, X_2 \cdots\}$ with their domains
- set of constraints C_j with error functions
- function to project constraint errors on vars (positive) cost function to minimize
- T: Tabu tenure (number of iterations a variable is frozen on local minima)
- RL: number of frozen variables triggering a reset
- MI: maximal number of iterations before restart
- MR: maximal number of restarts

Output: a solution if the CSP is satisfied or a quasi-solution of minimal cost otherwise.

```
 1: $Restart \leftarrow 0$
 2: repeat
 3:     $Restart \leftarrow Restart + 1$
 4:     $Iteration \leftarrow 0$
 5:     Compute a random assignment $A$ of variables in $V$
 6:     $Opt\_Sol \leftarrow A$
 7:     $Opt\_Cost \leftarrow cost(A)$
 8:     repeat
 9:         $Iteration \leftarrow Iteration + 1$
10:         Compute errors constraints in $C$ and project on relevant variables
11:         Select variable $X$ with highest error: $MaxV$
12:                                                         ▷ not marked Tabu
13:         Select the move with best cost from $X$: $MinConflictV$
14:         if no improvement move exists then
15:             mark $X$ as Tabu for $T$ iterations
16:             if number of variables marked Tabu $\geq RL$ then
17:                 randomly reset some variables in $V$
18:                                                 ▷ and unmark those Tabu
19:             end if
20:         else
21:             swap($MaxV$,$MinConflictV$),
22:                                             ▷ modifying the configuration $A$
23:             if $cost(A) < Opt\_Cost$ then
24:                 $Opt\_Sol \leftarrow A$
25:                 $Opt\_Cost \leftarrow costs(A)$
26:             end if
27:         end if
28:     until $Opt\_Cost = 0$ (solution found) or $Iteration \geq MI$
29: until $Opt\_Cost = 0$ (solution found) or $Restart \geq MR$
30: $output(Opt\_Sol, Opt\_Cost)$
```

Constraint-based Approach for an Early Inspection of the Feasibility of Cyber Physical Systems

Benny Höckner[1], Petra Hofstedt[1], Thilo Vörtler[2], Peter Sauer[1], and Thomas Hinze[1]

[1] Brandenburg University of Technology, Cottbus
{benny.hoeckner,petra.hofstedt,peter.sauer,thomas.hinze}@tu-cottbus.de
[2] Fraunhofer Institute for Integrated Circuits IIS
Design Automation Division EAS
Zeunerstraße 38, 01069 Dresden, Germany
thilo.voertler@eas.iis.fraunhofer.de

Abstract. Cyber Physical Systems (CPS) are technical devices typically consisting of numerous interacting electronic and mechanical components. A CPS incorporates many forms of control and monitoring based on a specific selection of physical quantities like temperature and acceleration. When designing a CPS from scratch, the question arises how to identify an optimal setting of components and interaction schemes. Due to the large number of combinations, resulting properties of system candidates like energy consumption or overall feasibility are hard to predict.
We present a constraint-based approach to model and inspect configurations of a CPS in an early stage of its development cycle. In addition, utilisation of constraints facilitates consideration of incomplete systems towards identification of valid configurations. A case study concerning a CPS for fall detection using temperature and acceleration sensors demonstrates the practicability of our approach.

1 Motivation

Nowadays, computer systems can be found in most aspects of human life. They emerged as an essential part of daily life in industrialised countries. For example, the percentage of costs for electronics and software in cars is currently already more than a third and assumed to increase in the future [1]. However, where previously relatively closed embedded systems have done their work, we now have to deal with a set of embedded systems. These systems are much more powerful and connected to each other, which leads to a higher complexity of the resulting entire system. In order to cope with this complexity, new methods to analyse and to create such systems are needed. This led to the concept of Cyber Physical Systems (CPS) [2].

CPS means that calculations and physical processes influence each other. In addition, a CPS differs from a traditional embedded system by using a lot of heterogeneous and independently developed components, which are interconnected to each other. A variety of different application areas benefits from these CPS. Examples include the control of manufacturing systems and production processes, traffic control systems, medical devices and systems for monitoring patient conditions or assistance systems in the field of assisted living.

In order to be able to fulfil the increasing requirements for these applications, embedded systems must be rich in functionality based on an efficient design. Therefore, CPS are often built by connecting multiple components or even by composition of complete embedded systems. In addition, also non-functional behaviour as timing, costs, and power consumption have to be taken into account. Developers of such CPS are therefore faced with entirely new design challenges.

The properties of a CPS as a whole depend on both the properties of the system components as well as the systems architecture. For the developer of a CPS, it is difficult to know in advance how a system behaves with certain components. In the beginning, the developer has only a raw idea about the system currently under design, e.g. he may know which components are mandatory, but does not know how exactly to configure them, or if there are some unused reserve available to add further optional components. So we are looking for a method to be able to iteratively deduce the structure and the behaviour of a system starting from an incomplete configuration. As a possible solution, we present a constraint-based approach, implemented in a prototypical tool. At the moment, this tool just covers specific CPS, which we will describe later, but it already demonstrates its benefits. The developer can specify an individual CPS and gets a Boolean answer whether the current configuration is feasible or not. Beyond that, constraints allow the developer to specify incomplete systems by replacing unknown values, for instance for a sensor configuration, by variables. The underlying solver then is able to evaluate all given constraints to calculate valid domains for those variables, which the developer in turn can use to iteratively complete the system.

The main advantage here lies in the possibility to save time and costs for prototyping, because only those configurations are generated that own the desired properties. This prevents the developer from wasting time for testing systems with inappropriate configurations.

This paper is structured as follows. Section 2 describes related work in the field of optimizing non-functional requirements, such as power consumption and performance. Section 3 is dedicated to the general modelling approach for the system components. The implementation of our approach is described in Section 4. A case study based on the real word application of a smart vest is shown in Section 5. Conclusions and an outlook are given in Section 6.

2 Related Work

The evaluation of non-functional requirements such as power consumption or timing can be divided into simulation-based and static approaches. Simulation-based approaches (see e.g. [3]) are based on running an executable system description e.g. in a system description language. Languages such as SystemC [4] allow to simulate complete systems including software and hardware. Simulations may also include the timing behaviour. The timing can be modelled at different levels of abstraction depending on the needed simulation performance and accuracy [5]. Although these approaches necessitate a complete system model, the evaluation is done using real execution traces of the system.

Static approaches, in contrast, work on simplified models of the components. Therefore, they are applicable before a real system model exists and enable an early optimization regarding desired properties.

Hang, Manolios, and Papavasileiou [6] discussed a constraint-based synthesis of cyber-physical architectural models using an IMT (Integer linear programming Modulo Theories) solver. Here, models are specified using the modelling language CoBaSA in a declarative way. Exclusively scheduling problems are described by real-time constraints.

Similarly, in our approach we use interval constraint-based methods for the system description but aim at taking further and more general non-functional system requirements into consideration.

Energy- and/or temperature-constrained scheduling is done using e.g. model checking approaches [7], local search [8], or constraint programming [9]. Chuo, Liu, Li, and Bagherzadeh [10] present the IMPACCT design tool along with a methodology for power-aware embedded systems which combine static components and simulation. The designer gives as system input a model of the application and timing- and power-constraints. Furthermore, he provides a model of the target architecture with components from a library instantiated and configured by the user. The tool performs power-aware scheduling [11] using a heuristic local search approach on a graph of timing constraints between tasks. It supports a simulation back-end for the integrated evaluation of the system under design. In contrast to these approaches, we abstract from the scheduling level and aim at the feasibility of the system model. Future extensions of our tool might include the application of constraint-based methods for modelling the different system levels.

In [12], Katoen, Noll, Wu, Santen, and Seifert present a method for determining the software/hardware deployment that minimizes the expected energy consumption. Their method combines constraint-solving techniques – the SMT (Satisfiability Modulo Theories) solver Z3 – for generating deployment candidates and probabilistic analyses of Markov chains for computing the expected energy consumption of the respective deployment. In contrast, we consistently apply the constraint-based approach also to the modelling of non-functional requirements such as timing behavior and power consumption.

3 Modelling a Cyber Physical System

Obtaining an early perception of the overall power consumption caused by the system after assembly is one of the main applications of our approach. Further, it is required that the system in its current configuration is valid, which means that all existing functional and non-functional requirements are fulfilled. Primarily it would be nice if we could specify systems which are partially incomplete and could receive suggestions for the missing parts. This, for example, would be useful to find out, whether it is possible to add one or two additional components which make a system more reliable and/or useful, or not. Another possible intention could be to estimate the CPU frequency which ensures the functionality of the system. This would help to save free CPU capacity, which again reduces the overall systems power consumption and increases the battery lifetime. To be able to gather such information we need specific data about a CPS which is discussed in this section, as well as the behavioural profile.

3.1 General Structure of a System

In the following, we assume that a system consists of these types of components:

- One CPU or microcontroller running a software application (or an operating system).
- At least one bus connecting external components (sensors, actuators) to the CPU, or another module like a bluetooth device. Main characteristics of such components is that they occupy one of the microcontrollers peripheral interfaces.
- There should be at least one sensor or actuator connected to a bus to collect information from the environment or to perform interactions with it.

At the moment, we do not model inter-CPU communication in our model, hence we assume that a system just consists of one single CPU. However, this is no general restriction of our approach. We assume further that every bus is connected to exactly one CPU and every sensor to exactly one bus. If the system would consist of more than one CPU, it could be divided into appropriate subsystems.

To estimate the overall power consumption of such systems we need certain static and dynamic parameters of our components, e.g. the supply voltage, the states of the system and their statistical frequency and the currents of the components. To allow a more precise approximation for the power consumption we distinguish at least between two power modes for every component. So, a component can either be in an active or inactive/standby mode.

To be more specific, a CPU or microcontroller is determined by static units like the supply voltage Vcc, the processor clock frequency f and current tuples for each supported mode, which would be in this case I_{active} and $I_{inactive}$. Each tuple further consists of two values, a lower bound $\mathtt{I_{low}}$ for the current, and an upper bound $\mathtt{I_{high}}$. These values will later used to form an interval for the current of the component. Some data sheets provide additionally a standard value for the current. The developer here can decide himself, whether he uses the standard value as a bound or not. In addition, every microcontroller consists of several peripheral interfaces PI_i, each supporting various protocols $Prot_i$ like *UART*, I^2C or *SPI*. The buses, sensors, and modules consist of a set of possible configurations C_i and a configuration index `CfgIndex`, which indicates with which configuration the component is running. Analogously to the CPU, we distinguish between two different modes, each described by a power consumption value ($\texttt{P}_{\texttt{active}}$ and $\texttt{P}_{\texttt{inactive}}$)[3] for the buses, and sensors or modules only support specific protocols within individual configurations. Moreover, a system is described by several states, whereas each state marks some components to be active while others may not. This could be described by adding a set of active states S to the sensor or module components. Thus the structure of a system can be formalized as follows:

- $CPU = (\texttt{Vcc}, \texttt{f}, I_{active}, I_{inactive}, PI_1, \dots, PI_m)$
- $I = (\texttt{I}_{\texttt{low}}, \texttt{I}_{\texttt{high}})$
- $PI_i = \{Prot_1, \dots, Prot_n\}$

- $BUS = (\texttt{CfgIndex}, \{C_1^{BUS}, \dots, C_p^{BUS}\})$
- $C_i^{BUS} = (\texttt{f}, \texttt{P}_{\texttt{active}}, \texttt{P}_{\texttt{inactive}}, PI)$

- $Sensor = (\texttt{CfgIndex}, S, \{C_1^{Sensor}, \dots, C_q^{Sensor}\})$
- $C_i^{Sensor} = (\texttt{Vcc}, \texttt{f}, I_{active}, I_{inactive}, \{Prot_1, \dots, Prot_k 1\})$
- $S = \{\texttt{StateId}_1, \dots, \texttt{StateId}_{\texttt{r}}\}$

- $Module_i = (\texttt{CfgIndex}, S, \{C_1^{Module}, \dots, C_s^{Module}\})$
- $C_i^{Module} = (\texttt{Vcc}, I_{active}, I_{inactive}, \{Prot_1, \dots, Prot_k 2\})$

In addition, we need mappings to determine the connections between components:

- $connectedBUS(CPU, p) \rightarrow \{b \mid$ *bus b with a configuration connected to the given CPU using a protocol p*$\}$
- $connectedSensors(bus) \rightarrow \{s \mid$ *sensor s with configuration connected to the given bus*$\}$
- $connectedModule(CPU, p) \rightarrow \{m \mid$ *module m with a configuration connected to the given CPU using a protocol p*$\}$

Beyond these mappings, we can define additional rules which are fulfilled by a *correctly designed system.* For instance:

- There should be at most as many buses or modules connected to the microcontroller as peripheral interfaces are available.
- Modules should only be connected to interfaces, if they have at least one type of protocol in common.
- All sensors on the same bus have to use the same protocol, which further should also be supported by the corresponding peripheral interface.

[3] The usage of the combination of the voltage and the current for some components, while for others directly employing the power consumption is caused by the availability of these value within the later example. Actually, there should be no difference in switching between both representations.

3.2 Cost Parameters and Behavioural Profile

Within the parameters of the aforementioned components, we also take the following quantitative parameters into consideration, in order to determine a valid system including its expected power consumption, and potential lifetime.

- An approximated number of lines of code (LOC) the CPU needs to process a single sensor data packet for each type of sensor.
- The approximated number of LOC to retrieve a data packet for each type of sensor.
- The bus overhead to transmit a data value packet.
- Each system state is executed for a statistically derived duration.

All these values are allowed to vary between different sensor or bus types.

With regard to the analysis of the estimated power consumption of the system, it is important to safely exclude invalid systems. The validity for a system is only restricted to the following rules:

- The CPU must be able to process the data of all connected sensors, and
- each bus must be able to transmit the data of its sensors.

4 Modelling Cyber Physical Systems using Constraints

Based on the model introduced in Section 3, we will now present some interesting key points of the implementation of our prototypical tool. We describe how a system can be modelled using constraints and what kind of benefit we obtain in comparison to an imperative implementation.

We are using the ECL^{i}PS^e Prolog solver [13] with the IC library. This library allows to model integer and real interval arithmetic constraints in one solver, which perfectly fits for our needs. On the one hand we have to deal with floating point numbers, which can be represented directly by real variables, to calculate units like the power consumption. In this way, we avoid an artificial transformation of those variables into finite domain variables, which would cause a loss of accuracy. On the other hand, we are mainly interested in lower and upper bounds for these variables. The internal work-flow of the Prolog program for finding a valid solution mainly consists of two mandatory steps and one optional step:

4.1 Step 1 - Validation of the Architectural Design

Here, we ensure that the given system is valid with respect to the design (cf. Subsection 3.1 *correctly designed system*).

The encoding of a CPS in Prolog is based on nested lists. Let us sketch the procedure in brief. As already mentioned, a system consists of various buses and modules, so we represent each of them by a list, grouped together in another list, whereby the location in the list indicates to which peripheral interface of the microcontroller this component is connected. Listing 4.1 shows the general pattern, which contains a list for a bus and each module. Here the bus would be connected to the first microcontroller interface, the module to the second one and so on.

Listing 4.1 Idea of encoding a system in PROLOG

```
[ [[CfgIndex] | SensorList], % bus
  [[CfgIndex, ActiveStates]], % module
  ...]                        % further buses and modules
```

To be able to post constraints, which guarantee the correct design, is now quite easy if we are using some predicates, which help us to extract the needed variables and constants. For instance, take a look at Listing 4.2, where we can see how to ensure that the protocol of the bus or module is supported by the current peripheral interface. To be able to distinguish between different cases, i.e. between buses and modules, we use the structure of the lists (see Line 6 and Line 14). So we can be sure, that in Line 8 we are dealing with a module like a bluetooth device, or, in the other case, in Line 15 with a bus. At the same time, we extract the variable which stores the configuration index of the component. This variable is either bound to a specific value indicating which configuration is used, or it is currently unbound. In this case the domain contains all values, which are still feasible. To force that the component protocol is an element of the supported protocols of the current interface we use the member predicate (cf. Line 9 and Line 16). In Line 17, another predicate ensures that the connected sensors also apply the same protocol, respectively. Since the supported protocols of the components are just facts, which have to be given, e.g. via data sheets, these values are outsourced in *constant predicates*. These predicates ensure that the second argument gets bound to the protocol specified by the index (first argument). Advantage of using such predicates instead of constants is that they do not have to be passed as parameters all the time. An example for such an predicate is shown in Listing 4.4 and will be explained later.

4.2 Step 2 - Ensuring Functional and Non-functional Requirements

After finishing the first step, we can assume that we have a well designed system, which now needs to be checked for its functional and non-functional requirements. Since these requirements are mostly of physical quality, like the systems power consumption or the minimum needed bus frequency to transport all the sensor data, they can also be excellently represented using arithmetic constraints. Note that the power consumption of a component cannot be precisely predicted. This is why we are using intervals with a lower and an upper bound as defined in Section 3.

Listing 4.2 Ensuring the usage of a correct bus protocol

```
% Ensure that the given components are using protocols,
% which are supported by the peripheral interfaces.
valPeriCompProtocols(_, []).
valPeriCompProtocols([Interface | ITail], [Com | BTail]) :-
    % Ensure, that the current component is a module.
    Com = [[CfgIndex, _] | []],
    !,
    getComponentProtocols(CfgIndex, ModuleProtocols),
    member(ModuleProtocols, Interface),
    ...
.
valPeriCompProtocols([Interface | ITail], [Com | BTail]) :-
    % Ensure, that the current component is a bus.
    Com = [[CfgIndex] | Sensors],
    getBusProtocol(CfgIndex, BusProtocol),
    member(BusProtocol, Interface),
    valSensorProtocols(Sensors, BusProtocol),
    valPeriCompProtocols(ITail, BTail)
.
```

To allow a more accurate estimation of the power consumption, we consider different states of the systems. Within these states the behaviour may change widely, because sensors are allowed to switch into standby mode, which means that they are inactive to save power, while other components may wake up. The cpu and the buses then have to process or transmit a different amount of data, which also affects the power consumption and of course the validity of the system design. This fact is also supported by the current prototype. Since we also want to predict the overall systems lifetime, which mostly depends on the systems overall power consumption, we need stochastic information about the state's average execution time. This information can then be used to weight the individual power consumptions. Listing 4.3 shows the idea of how the distinction between the active and inactive mode especially works for sensors, but of course this idea is applicable also for modules. The predicate *calcPConsumpSensors* gets a list of sensor's, whereas each sensor knows its states where it is active (*StatesWhenActive*, Line 3), the current system state *StateNo*, which power consumption should be calculated, and a variable *Psensors* for the final power consumption. In Line 4 we ensure that the current system state is within the list of the sensors active states. If this clause fails, the PROLOG engine will backtrack and consider the next clause in Line 12. The calculation of the power consumption for a sensor is performed in the Lines 7 - 9.

In Line 7 and Line 8 we see two further examples for constant predicates. A schematic implementation for those predicates is shown in Listing 4.4. As noticeable in Line 2 these predicates are implemented using a specific constraint named *element*, so that the given *Index* variable currently does not have to be

Listing 4.3 Implementation to regard different component states

```
calcPConsumpSensors([], _, 0.0).
calcPConsumpSensors([Asensor | Tail], StateNo, Psensors) :-
    Asensor = [CfgIndex, StatesWhenActive],
    member(StateNo, StatesWhenActive),
    !,
    % Use power consumption for active mode
    getSensorVoltage(CfgIndex, UsValue),
    getSensorCurrent(CfgIndex, IddValue),
    Psensors $= IddValue * UsValue + Ptail,
    calcPConsumpSensors(Tail, StateNo, Ptail)
.
calcPConsumpSensors([Sensor | Tail], StateNo, Psensors) :-
    % Use power consumption for standby mode
    ...
.
```

bound to a specific value. Therefore the domain of the *Value* variable can also contain all possible values, which will all adopted in the subsequent computation of the program. The constants are values for the configuration related to a specific unit like the current or voltage of a component, and can mostly be found in the corresponding data sheets.

Listing 4.4 Using constant predicates to encode constants.

```
getConstantPredicate(Index, Value) :-
    element(Index,[Const_1, Const_2, ..., Const_n],Value),
.
```

After all necessary information has been accumulated into one single equation, which estimates the systems overall power consumption, we are able to evaluate the possible system life time. It should be mentioned here, that we currently do not consider further energy consuming properties like the battery self-discharge or the influence of the environmental temperature.

4.3 Step 3 - Selecting Specific Values for the Unknown Components

The program is able to determine solutions, by values and sets of values for known and unknown components. Since these sets might also be very large, this optional step should select the most relevant values. For instance, a system with one unknown sensor may have enough resources to clock a sensor with at most 200 Hz then of course it would also be possible to clock it with a slower sample rate like 100 Hz or 50 Hz and so on, but these values mostly do not provide any new information for the user. Therefore, it seems to be more useful to select specific values. Please note, that the meaningful values are not always the largest values within the valid range, e.g. a bus should be clocked with the smallest available frequency since this almost implies a smaller power consumption.

5 Application to an Example System

Now we will demonstrate the features of our tool in its current state via a case study. The study is derived from a real world low power embedded system platform, which for instance can be used for fall detection of a human person. The example system will be introduced and motivated in Subsection 5.1, and then we demonstrate in Subsection 5.2 how the used components can be modelled. Finally, we will show some evaluation results for different input assignments in Subsection 5.3.

5.1 A Practical Example – The Smart Vest

A recent major challenge in the western world is the increasing proportion of elderly people in the population. This is accompanied by an increased burden of the health care systems. *Smart clothing* defines a branch of research within this area.

Our example system is a smart vest CPS designed for elderly people including acceleration and temperature sensors to detect falls. Several similar systems have been developed and are partly commercially available (cf. [14]).

As elderly people certainly want to be able to live independently and self-determined as long as possible, a smart vest can be used to monitor people in residential homes in a more discreet way, since for them a fall often has more serious consequences than for younger people. In a residential home, the staff should be made aware of falls to be able to check if everything is still ok. A sketch of the example system is shown in Figure 1, and consists of the following components:

- One TIMSP430 8-bit microcontroller [15].
- Two ADXL345 acceleration sensors [16] to detect falls.
- One SHT21 humidity-temperature sensor [17] to detect if the vest is worn.
- An I^2C bus to connect the sensors to the microcontroller [18].
- A (RN41) bluetooth radio module [19] to transmit information to the staff of the residential homes. This module is directly connected to the microcontroller.
- A standard 3.3 V, 2000 mAh battery to provide the power for the complete system.

In such a system low power design is especially important, since it has direct influence on the weight (battery size) and the runtime of the overall system. But

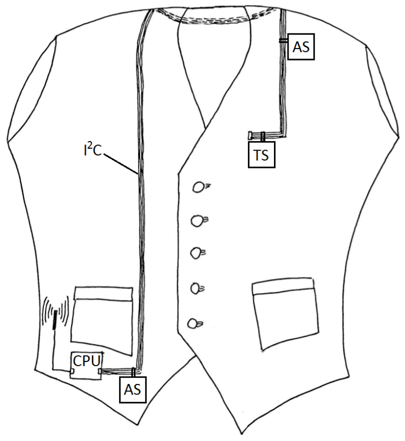


Fig. 1. A sketch of an intelligent vest prototype.

on the other hand the sensors have to operate with an appropriate sample rate, so that a reliable fall detection can be guaranteed. This again stands in conflict to a higher sample rate, which implies an higher overall power consumption. Furthermore, we should ensure that the CPU is able to process all values or the bus is able to transmit all sensor values. Such information can be perfectly encoded using constraints.

5.2 Modelling the Components

In the following, we describe the parameters used to model the system components.

TIMSP430 Microcontroller and Application Software: The microcontroller including the running application mainly influences the power consumption of the overall system. Figure 2 shows an overview of the program execution. After the system initialization (*initialize system*) the system is in State 1 (Standby state), where only the temperature sensor gets requested (*request temperature sensor*) and processed (*process temp. value*). If a person dresses the vest, then the temperature may rise about 34° C and the system should switch to State 2 (Fall detection state). In this state all three connected sensors will be requested (*request temperature and acceleration sensors*) and their values processed (*process values*). If a fall is detected, the system will switch to State 3, where the bluetooth module will send a specific help message. If no fall is detected, then it will be checked whether the vest is still carried by a person. Note that the microcontroller puts itself into a *sleep mode* (*low power mode 3 (LPM3)*) to save some power, until new sensor values need to be requested. Note, that the controller will wake up automatically.

The overall power consumed by the microcontroller depends on the current application and the sample rates of the sensors. The more extensive the application, the more energy is consumed. The same applies to the number of sensor

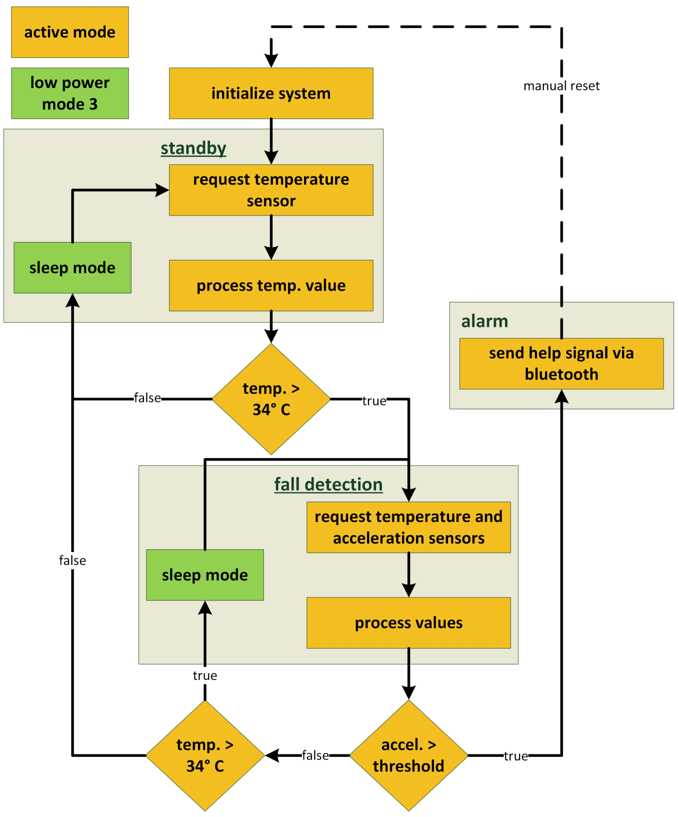


Fig. 2. Program loop of a possible fall detection system.

values. If more values are processed then the microcontroller operates longer in the power state *active mode (AM)*. The power consumption, furthermore, depends on the clock frequency of the processor and the supply voltage, which has been taken from the data sheet, whose typical and maximum values have been used. As we apply an interval solver, both values can be easily used in the system. The power consumption for the LPM3 has also been taken from the data sheet, and, however, does not depend on the system clock frequency.

Acceleration Sensors: The acceleration sensors are modelled following the operating principle of an ADXL345 acceleration sensor [16]. This sensor supports different sample rates, where with each sample value data packets are created. The sample rate ranges from 6.25 Hz up to 3200 Hz. Depending on the selected sample rate the power consumption of the sensor varies. Each packet consists of 6 Bytes of data (48 bits), which need to be transferred via the bus to the microcontroller. Due to the overhead of the I^2C protocol, the transmission of one sensor value takes 81 I^2C cycles. Additionally, these sensors also could utilise a standby mode to consume as few power as possible, which is done in State 1.

Temperature Sensor: The temperature sensor is modelled in accordance with the SHT21 (humidity and temperature sensor) [17]. For this scenario it is sufficient to just allow one possible sample rate about 1 Hz, i.e. every second one measurement is done. This low sample rate furthermore permits to neglect the self-heating effect of this sensor.

I^2C Bus: The bus system is connecting the sensors with the microcontroller and can be clocked either with 100 kHz or 400 kHz. Due to the aforementioned protocol overhead, the actual data rate, for sensors is lower than 100,000 or 400,000 bits/s. An average power consumption for transmitting data has been calculated for the bus system using a SPICE model. This value is independent of the actual data. If the bus does not transmit anything, then it consumes no power. Note that the bus speed also influences the time to transmit the sensor values to the microcontroller, i.e. the request states.

Bluetooth Data Module: The data values for this component are based on the RN41 module, which also could put into a power saving standby mode. The power consumption model of this component has been simplified so that we assume, that the module constantly transmits values with the same power consumption independent from the current environment.

Our tool only generates functionally correct solutions, i.e. the rules specified in Subsection 3.2. Furthermore, it directly tries to reduce the number of possible solutions by selecting the maximum possible sensor sample rates and the minimum possible bus frequencies.

5.3 Evaluation of the Results

The example system was evaluated for different scenarios to show the possibilities and power of the tool. It was assumed, that the application running on the microcontroller takes about 2,000 processor cycles to request one sensor value and around 5,000 to process it. Note that this is no general restriction of our approach, however we here assume that worst case execution times of program functions are available.

In the first scenario the clock frequency of the microcontroller is fixed to 2.4576 MHz, as well as the structure of the system, i.e. the developer does not want to add any new components. Therefore, the tool has to determine the overall power consumption and to estimate the systems lifetime regarding to the mentioned battery. With lifetime we mean the duration, where the system is able to perform a full emergency call. In consequence, if the power of the battery is so low that this would not be possible then the battery is declared as empty. The acceleration sensors were clocked to 50 Hz and the I^2C bus to 100 kHz. The tool then estimates a overall power consumption between 9.455 mW and 9.516 mW and in the best case the battery would last for around 761 hours ($\approx$ 31 days).

In the second scenario the developer wants to know whether it would be possible to add a third acceleration sensor, so that the reliability of the fall detection would increase. This implies that the sensor at least should be clocked with a sampling rate greater or equals than 25 Hz. Furthermore, the developer needs system lifetime of at least 4 weeks. The tool here is able to answer that "question" and determines a sample rate of at most 200 Hz for the new sensor.

The last scenario could take place in a quite early stage of the system design. Here the developer has the fewest information about the later system available. He only knows, that he needs two acceleration sensors, a temperature sensor and a bluetooth module. We assume that he does not know on which sample rates the sensors should operate, how the CPU has to be clocked and did not even thought about a correct design, i.e. he does not know which bus protocols he has to use. In this case, the optional third step of the tool should not be used, due to the fact that it would try to maximize the sample rates of the sensors, which would therefore cause a maximization of the CPU frequency, but the tool is still able to produce some outputs. The developer receives the information that the CPU must be clocked at least with a speed of 0.95500 MHz and one of the acceleration sensors can be clocked with 400 Hz at most. Furthermore, the system would last between 24 and 34 days, and the bus has to use the I^2C protocol and the bluetooth module the UART protocol.

6 Conclusion

In this paper we presented an approach to check the feasibility of CPS by constraints. We outlined that they are not only useful to check specific properties, but rather help to search the design space for valid configurations with respect to non-functional requirements, like for instance the power consumption. We further have shown that it is possible to respect dynamic units to retrieve more specific statements.

The applicability of our approach was shown by a real world case study. The usage of constraints is very promising, due to the fact that once the system is modelled, nearly every combination of known and unknown components a user could think of can be handled. Moreover, the approach allows to investigate the system design space statically and partly dynamically. It is also possible to optimize some criteria to reduce the number of possible solutions, which mainly should help the user to extract meaningful information faster.

We will investigate further ways to make this tool more general, e.g. to use a model driven development approach, where the user might be able to specify its own system with its own requirements and constraints. This could also open up the possibility to consider some economic properties like the cost of a specific component, so that the system will be constructed with the cheapest component available which fulfils the requirements. Furthermore, we want to extend our approach for larger and more detailed system models with broader design alternatives.

References

1. Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 33–42, New York, NY, USA, 2006. ACM.

2. Edward A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363–369, 2008.
3. Alexander Viehl, Björn Sander, Oliver Bringmann, and Wolfgang Rosenstiel. Integrated requirement evaluation of non-functional system-on-chip properties. In *Forum on Specification and Design Languages (FDL)*, pages 105–110. IEEE, 2008.
4. IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, 2012.
5. David C. Black and Jack Donovan. *SystemC: From the Ground Up*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
6. Christine Hang, Panagiotis Manolios, and Vasilis Papavasileiou. Synthesizing cyber-physical architectural models with real-time constraints. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV)*, volume 6806 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2011.
7. Weixun Wang, Xiaoke Qin, and Prabhat Mishra. Temperature- and energy-constrained scheduling in multitasking systems: a model checking approach. In Vojin G. Oklobdzija, Barry Pangle, Naehyuck Chang, Naresh R. Shanbhag, and Chris H. Kim, editors, *International Symposium on Low Power Electronics and Design (ISLPED), 2010, Austin, Texas, USA*, pages 85–90. ACM, 2010.
8. Ganesh Lakshminarayana, Anand Raghunathan, Niraj K. Jha, and Sujit Dey. Power management in high-level synthesis. *IEEE Transactions VLSI Systems*, 7(1):7–15, 1999.
9. Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek. An application of constraint programming to superblock instruction scheduling. In Peter J. Stuckey, editor, *Principles and Practice of Constraint Programming, 14th International Conference (CP) 2008, Sydney, Australia*, volume 5202 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2008.
10. Pai H. Chou, Jinfeng Liu, Dexin Li, and Nader Bagherzadeh. Impacct: Methodology and tools for power-aware embedded systems. *Design Automation for Embedded Systems*, 7(3):205–232, 2002.
11. Jinfeng Liu, Pai H. Chou, Nader Bagherzadeh, and Fadi J. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In Jan Madsen, Jörg Henkel, and Xiaobo Sharon Hu, editors, *Ninth International Symposium on Hardware/Software Codesign (CODES) 2001, Copenhagen, Denmark*, pages 153–158. ACM, 2001.
12. Joost-Pieter Katoen, Thomas Noll, Hao Wu, Thomas Santen, and Dirk Seifert. Model-based energy optimization of automotive control systems. In Enrico Macii, editor, *Design, Automation and Test in Europe (DATE)*, pages 761–766. EDA Consortium San Jose, CA, USA / ACM DL, 2013.
13. The ECL^{i}PS^e Constraint Programming System. `http://eclipseclp.org/`, 2013. last visited 2013-06-28.
14. Overview on Smart Shirts. `http://en.wikipedia.org/wiki/Smart_shirt`, 2013.
15. MSP430 Low Power Microcontroller. `http://www.ti.com/430brochure`, May 2013.
16. ADXL345 Digital Accelerometer. `http://www.analog.com/static/imported-files/data_sheets/ADXL345.pdf`, May 2013.
17. SHT21 Humidity and Temperature Sensor. `http://www.sensirion.com/fileadmin/user_upload/customers/sensirion/Dokumente/Humidity/Sensirion_Humidity_SHT21_Datasheet_V3.pdf`, July 2013.
18. IC Bus. `http://en.wikipedia.org/wiki/IC2B2C`, May 2013.
19. Bluetooth module RN41. `http://ww1.microchip.com/downloads/en/DeviceDoc/rn-41-ds-v3.42r.pdf`, July 2013.

Simplifying the Development of Rules Using Domain Specific Languages in DROOLS

Ludwig Ostermayer, Geng Sun, Dietmar Seipel

University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{ludwig.ostermayer, dietmar.seipel}@uni-wuerzburg.de
geng.sun@stud-mail.uni-wuerzburg.de

Abstract. The open–source business logic integration platform DROOLS supports a declarative, rule–based approach for problem solving. However, rules are implemented in a JAVA–based way in DROOLS, which is difficult to understand for non–programmers. The rules for a given scenario are usually provided in natural language by non–programmers, for instance business analysts; for controlling the correctness of the implemented rules, it is crucial to integrate these business analysts into the coding phase.
To bridge the gap between programmers and non–programmers, Domain Specific Languages (DSLs) allow for implementing rules in a language that is closer to natural language. In a DSL, rules can be written, read, and modified much easier, even by non–programmers. DROOLS offers a DSL editor, but both developing a DSL and implementing rules within the DSL is still difficult.
Thus, in this paper, we present a tool, DSLR Generator, which simplifies the creation of DSLs and the implementation of rules within the DSL. A graphical user interface supports the user step by step in the development process. Reusable and generic DSL templates can be used to write rules in a more readable format. The maintenance of the meta–data for the rules in DROOLS is supported, too.
With DSLs, it becomes easier to write rules in DROOLS that can also be parsed in PROLOG. Although the evaluation mechanism of DROOLS is much different from the resolution mechanism of PROLOG, we can support the rule development process by PROLOG technology: the rules could be analyzed, verified, or transformed by PROLOG software.

Keywords. Domain Specific Lagnuages, DROOLS, PROLOG, Logic Programming, Business Rules.

1 Introduction

The business rules approach [7] provides methodologies for system development that create applications as *white boxes*, whose inner business logic is visible, because it is separated into business rules written with a simple rule language, or even in natural language. Meanwhile, Business Rule Management Systems (BRMS) have been developed, which can define, deploy, execute, monitor, and maintain the variety and complexity of decision logic that is used by operational systems within an organization or enterprise.

Unlike a general–purpose language, such as Java, a Domain Specific Language (DSL) is a programming language of limited expressiveness for a particular problem domain [6]. It can be distinguished between internal and external DSLs. Examples of external DSLs are programming languages dedicated to a particular purpose, such as SQL for relational database languages and HTML for displaying web pages.

The well–known BRMS DROOLS [8] supports the development of a DSL and the editing of rules written in a DSL using only basic editors. But, the DSL editor provides only a few features; e.g., a content assist is not available as in the open development platform Eclipse, that is very popular within the JAVA community. To develop a DSL with the DSL editor of DROOLS, most of the DSL must be typed word by word without support or guidance. Moreover, there is no component which supports the editing of rules for a developed DSL, which makes it very difficult for domain experts who are not programmers to write DSL rules. Therefore, we have developed a tool called DSLR Generator, which is intended to improve the development process within DROOLS, since it is able to generate a DSL and DSL rules semi–automatically.

The rest of the paper is organized as follows: In Section 2, we give an overview of DROOLS, the used rule language, and the development of a DSL in DROOLS. In Section 3, we present the DSLR Generator with its components and illustrate our approach by an example from the financial sector. In Section 4, we describe a PROLOG–based analysis, that can be used during the development phase to find anomalies or errors. Finally, we summarize our work in Section 5.

2 The Business Logic Integration Platform DROOLS

The BRMS DROOLS consists of several modules. The core module EXPERT of DROOLS is basically a *production rule system*. The module GUVNOR provides a web based BRMS and the module FUSION deals with complex event processing. The module PLANNER works as a planning engine for optimizing scheduling problems.

2.1 The Core Module DROOLS EXPERT

The module EXPERT provides the inference engine of DROOLS for production rules based on an improved implementation of the Rete algorithm, called ReteOO. Facts are stored in a *working memory*, the rules are loaded by DROOLS in a *production memory*. During the inference process, facts can be modified and retracted, and new facts can be asserted. Conflicting rules are managed by an *agenda* that uses a *conflict resolution strategy*, see Figure 1 from [9]. Rules written in the DROOLS Rule Languange and are saved in simple text files with the extension .drl. A `package` is a set of rules. The package name itself is only a namespace and is not related to the folders. `import` statements work just like in JAVA.

DROOLS automatically imports classes from the JAVA package of the same name. In DROOLS, we have full access to the functionalities of almost all JAVA libraries. Global variables can be defined with `globals`. Complex logic can be outsourced and used

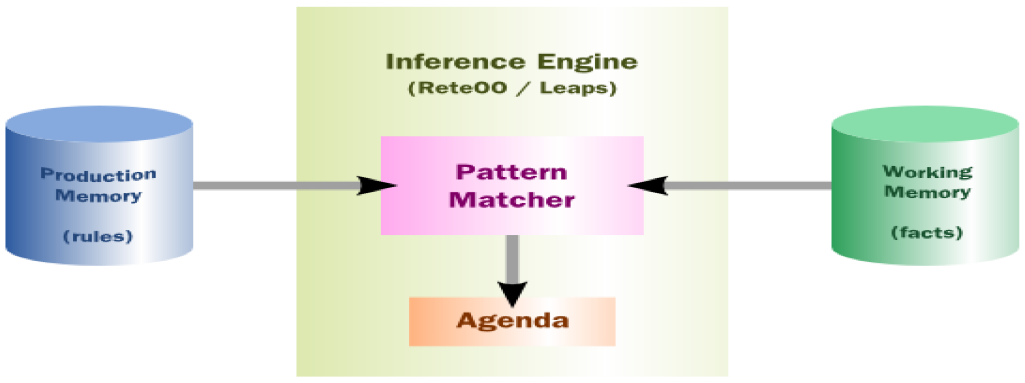


Fig. 1: High–Level View of the Production Rule System DROOLS EXPERT [9].

within a rule by a reference via the statement `functions`. A rule must have a unique name within a package. If a rule is added to package already containing a rule with the same name, then the new rule replaces the old one. Listing 1.1 shows a simple rule in DROOLS Rule Language format. The left hand side of a rule, which starts with the keyword `when`, is the *condition* part. Conditions consist of patterns that try to match facts in the working memory; patterns can have constraints to narrow down the set of potentially matching facts. The right hand side of a rule, which starts with the keyword `then`, is basically an *action* block, that can contain almost arbitrary JAVA statements. All methods or attributes of classes within the working memory can be accessed; external classes can be accessed after an import statement, too.

Listing 1.1: A Rule in the DROOLS Rule Language Format

```
package LoanApproval

rule "microfinance"
when
   application: Application(loan_amount < 10000,
      duration_year <= 5 )
   customer: Customer(credit_report == "good")
then
   application.approval();
   application.setInterest_rate(0.028);
end
```

2.2 Development of Rules in Domain Specific Languages in DROOLS

Rules in a Domain Specific Language are developed in DROOLS in two steps. First, the expressions of the DSL are designed, and then the mapping of the expressions to the rule language of DROOLS is written into a file with extension .dsl. Listing 1.2 shows a fragment of a .dsl file, where `[when]` indicates the scope of the expression as condition, and `[then]` is used for actions, respectively. The part before the single equality sign "=" is an expression of the DSL, and the part behind is the translation to the DROOLS Rule Language format.

Listing 1.2: A Fragment of a .dsl File

```
[when] The customer with monthly_income is greater than {value1}
       and credit_report is {value2} =
       customer: Customer(
          monthly_income > {value1}, credit_report == {value2})
```

In a second step, we write rules consisting of expressions of the DSL that are defined in the .dsl file just mentioned. The rules written in the DSL are saved to a file with the extension .dslr. Listing 1.3 shows a single rule, where we have used the expression from Listing 1.2 in the condition part of the rule.

For processing the rules defined in the .dslr file, DROOLS uses the mapping information contained in the .dsl file to internally transform the rules into the DROOLS Rule Language format.

Listing 1.3: A DSL Rule

```
rule "microfinance"
when
   The loan with loan_amount is smaller than or equal to 5000
   and duration_year is no more than 3
   The customer with monthly_income is greater than 2000
   and credit_report is "good"
then
   The loan will be approved
   Set the interest_rate of the loan to 0.025
end
```

DROOLS provides only a simple DSL editor. However, the editor lacks user friendliness and functionality. Most of the content must be typed word by word and there is no possibility to reuse created components conveniently. For instance, a content assist is not available. A package explorer for JAVA classes, attributes or methods would help greatly in the DSL design process. Additionally, after the completion of the DSL development, there is no component in DROOLS that can be used to create rules within the DSL. Therefore, creating DSLs and working with DSLs is still difficult in DROOLS, and non–programmers can not be incorporated easily and benefit as much as possible from the DSL approach. This is exactly where our tool DSLR Generator comes into play.

3 The Tool DSLR Generator

DSLR Generator supports in a few guided steps to edit rules, that have a readable format and a correct syntax. First, we describe how the user can develop a DSL with the aid of generic templates and what we mean by templates in this scenario. Then, we describe shortly the various graphical editors that help in the construction of syntactical correct rules in the DSL. Finally, we give a brief example illustrating the usage.

3.1 DSL Templates

We provide several generic templates that contain the mapping information between a natural language expression and the corresponding code in DROOLS. For instance, we can use the template shown in Listing 1.4 to generate a simple condition in DSL.

Listing 1.4: A DSL Template

```
The #instance with #field is smaller than or equal to {value} =
   #instance: #class(#field  <= {value})
```

For creating rules, keywords and parameters in a template can be replaced. A keyword starts with #; e.g., #field is a keyword. We already provide several keywords in the DSLR Generator, but more can be added. Parameters must be enclosed with curly brackets "{" and "}". If brackets appear literally in the language expression or the rule code, then they have to be escaped with a preceding backslash "\"; the same holds for magic characters of Java's pattern syntax.

Listing 1.5 gives an example how to use the template from above. In the DSL expression of Listing 1.4, the key word #instance has been replaced by loan, a name of an instance of the JAVA class Loan, which replaces #class. The name of an instance in a DSL statement can be different from the name of an actual instance in the JAVA code. The condition in our example is satisfied if the attribute loan_amount of an instance of Loan has a value greater than value. The key word #field has been replaced by loan_amount, the name of an attribute of the class Loan. We will see in Subsection 3.3 how we support the rule creation using the templates. E.g., once a class is selected to replace #instance, only attributes of the selected class can be assigned to #field. Further templates can be added easily.

Listing 1.5: A Simple Condition in DSL

```
The loan with loan_amount is smaller than or equal to 5000 =
   loan: Loan(loan_amount <= 5000).
```

We provide JAVA annotations to accomplish multilingual DSLs. The desired language expression can be chosen by setting a language flag in a config.property file. Listing 1.6 shows two annotations to reference the attribute loan_amount in plain English or German. Instead of loan_amount, like in Listing 1.5, #field would be replaced by the value of the annotation of the chosen language, e.g., amount of loan for English. This further improves the readability of rules. Additionally we created the annotation @Invisible() to controll the visibility of classes, attributes and methodes within the Basic DSL Editor.

Listing 1.6: An Annotation of a Language Expression

```
@EnExpression(value = "amount of loan")
@GerExpression(value = "Kredithoehe")
private double loan_amount;
```

3.2 Template Files

A template for a DSL is stored in a JAVA object of the type `Template`. All relevant key words are attributes of `Template`. There are two additional lists of classes `TemplateNode`: one is a list of conditions, the other is a list of actions. A `TemplateNode` is a container for several DSL mappings, like in Listing 1.5.

To improve the readability, we map template objects into a common format. Our approach currently supports two formats: XML and JavaScript Object Notation (JSON). For the mapping between JAVA and XML, we use the JAVA Architecture for XML Binding (JAXB) [12], and for the mapping between JAVA and JSON we use the open–source Java library FASTJSON [1]. According to the file extension, the Basic DSL Editor can automatically load the template file. The user only needs to specify a template file path in the file `config.properties`.

Listing 1.7 shows a DSL Template in XML. In addition to keywords, there is explicit a simple condition and an action. Every key word in a DSL statement is represented by an element, e.g., the key word `#instance` by a element with the tag `<instance>`. There are two DSL statement categories: condition and action; they are represented by elements, too. Each of it can contain several `<dsl>` elements. A DSL statement is tagged in XML by `<expression>`, the corresponding code in DROOLS simply by `<code>`. DSL statements are grouped by a domain; this is represented in XML by the child `<domain>` of `<condition>` and `<action>`, respectively. We do this, because during the development phase we want to filter templates by domains.

Listing 1.7: A Template in XML

```
<template>
   <class>#class</class>
   <com_parameter>{value}</com_parameter>
   <instance>#instance</instance>
   <field>#field</field>
   <obj>#i</obj>
   <condition>
      <domain>Common</domain>
      <dsl>
         <expression>
            The #instance with #field is smaller than
            or equal to {value}
         </expression>
         <code>
            #instance:#class(#field <= {value})
         </code>
      </dsl>
   </condition> ...
   <action>
      <domain>Common</domain>
      <dsl>
         <expression>
            Set the #field of the #instance to {value}
         </expression>
```

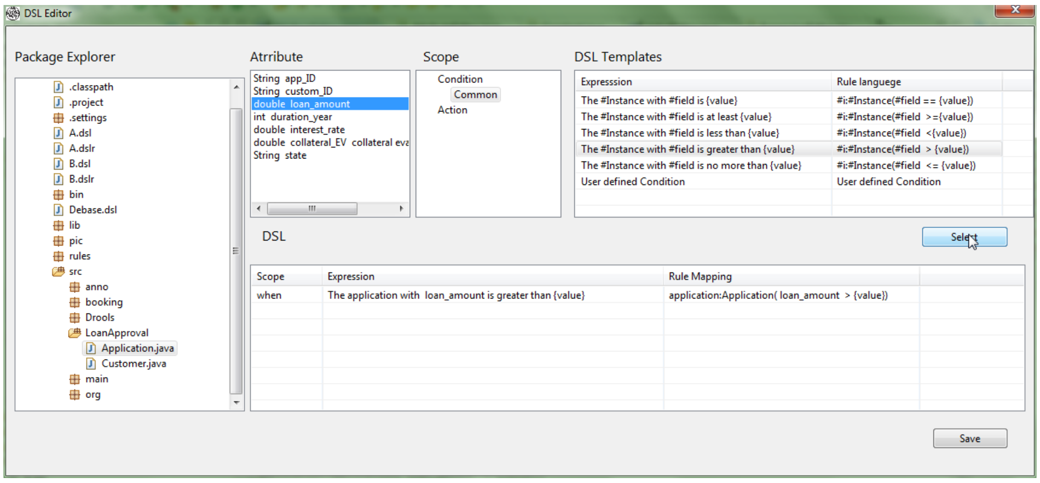
```
          <code>
             #instance.setField({value});
          </code>
       </dsl>
    </action> ...
</template>
```

The JSON representation is less verbose, but identical to the XML representation. With the `@XmlType` annotation of JAXB, the user can customize the XML already in JAVA. FASTJSON provides customizing options with the `@JSONType` annotations for JSON.

3.3 Creating Rules with DSL Templates

There are four components that can be used during the rule creation process; every component has a graphical user interface (GUI), see Figures 2 – 6.

Basic DSL *Editor.* In this editor the templates defined in `templates.xml` are available, and simple DSL statements can be created, see Figure 2. The user may customize his or her own templates. Additionally, there is a package explorer, with which the user can access JAVA classes as well as attributes and methods conveniently in a selected package. This editor can generate simple conditions, and actions in DSL. Domain experts can cooperate with programmers; they can take care of the language expression, and the programmer takes care of the corresponding code. If attributes and methods in JAVA already have been annotated, as described in Subsection 3.1, then it is possible for non–programmers to work with this editor largely independently.



Package Explorer:
- .classpath
- .project
- .settings
- A.dsl
- A.dslr
- B.dsl
- B.dslr
- bin
- Debase.dsl
- lib
- pic
- rules
- src
  - anno
  - booking
  - Drools
  - LoanApproval
    - Application.java
    - Customer.java
  - main
  - org

Atrribute:
- String app_ID
- String custom_ID
- double loan_amount
- int duration_year
- double interest_rate
- double collateral_EV collateral eva
- String state

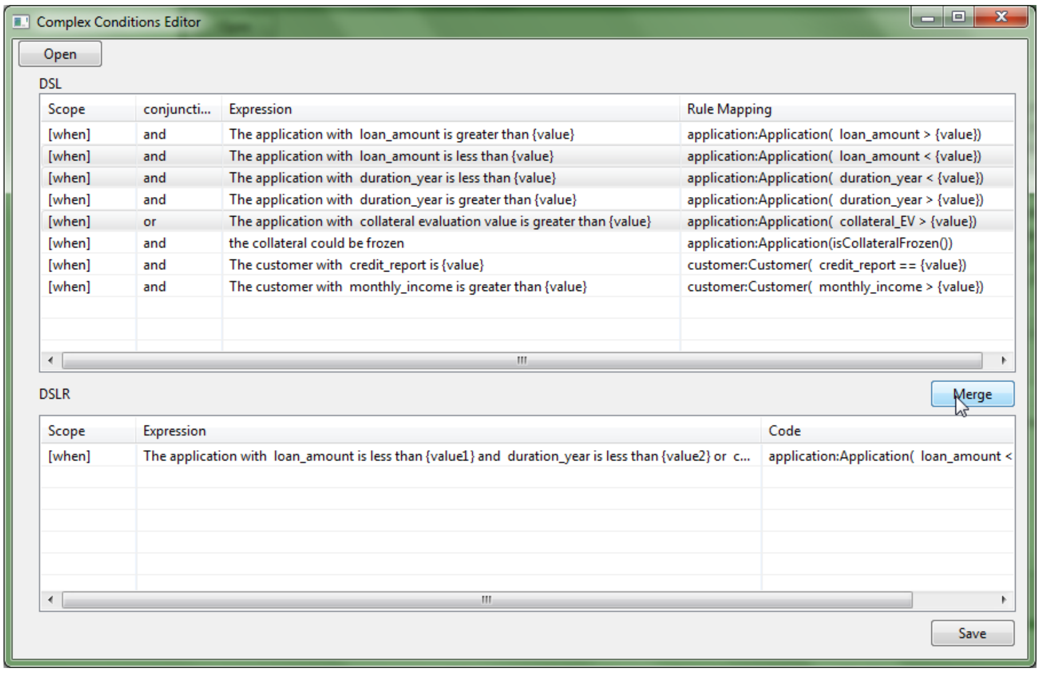
Scope:
- Condition
  - Common
- Action

Expresssion	Rule languege
The #Instance with #field is {value}	#i:#Instance(#field == {value})
The #Instance with #field is at least {value}	#i:#Instance(#field >={value})
The #Instance with #field is less than {value}	#i:#Instance(#field <{value})
The #Instance with #field is greater than {value}	#i:#Instance(#field > {value})
The #Instance with #field is no more than {value}	#i:#Instance(#field <= {value})
User defined Condition	User defined Condition

Scope	Expression	Rule Mapping
when	The application with loan_amount is greater than {value}	application:Application(loan_amount > {value})

Fig. 2: The Basic DSL Editor.

Complex Condition Editor. In terms of reusability and complexity, it is important to create simple conditions first. Complex conditions and actions are broken into atomic parts. Our editor supports this procedure, see Figure 3. The user can compose single conditions to complex conditions using conjunction and disjunction. Several conditions with connectives can be selected in a table and merged simply by hitting the *Merge* button. For building complex conditions with different connectives, it is sometimes necessary to use brackets to make the logic clear. The result is saved to a file with the extension .dsl.

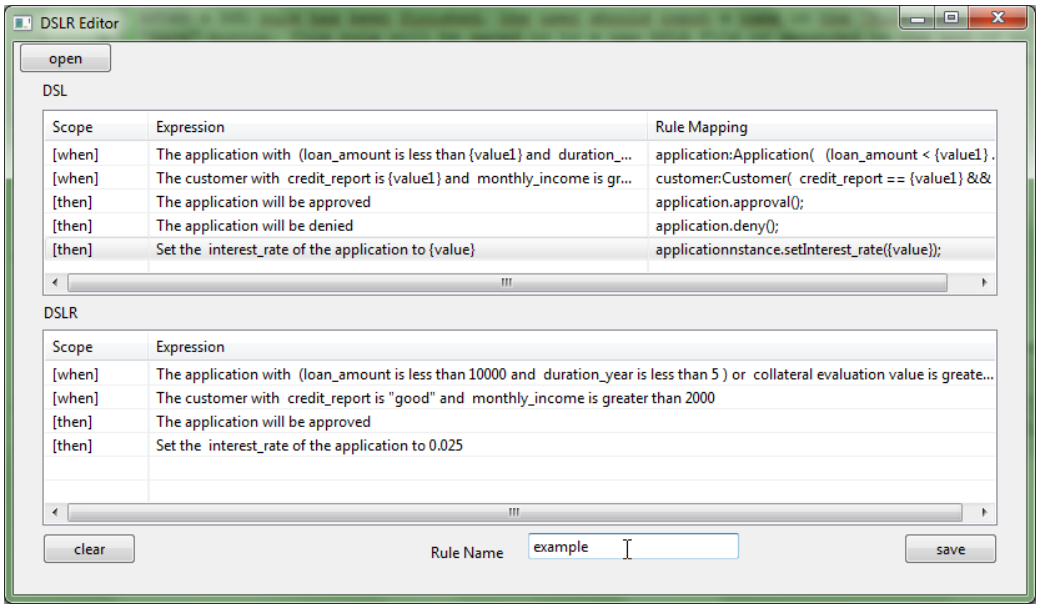


Scope	conjuncti...	Expression	Rule Mapping
[when]	and	The application with loan_amount is greater than {value}	application:Application(loan_amount > {value})
[when]	and	The application with loan_amount is less than {value}	application:Application(loan_amount < {value})
[when]	and	The application with duration_year is less than {value}	application:Application(duration_year < {value})
[when]	and	The application with duration_year is greater than {value}	application:Application(duration_year > {value})
[when]	or	The application with collateral evaluation value is greater than {value}	application:Application(collateral_EV > {value})
[when]	and	the collateral could be frozen	application:Application(isCollateralFrozen())
[when]	and	The customer with credit_report is {value}	customer:Customer(credit_report == {value})
[when]	and	The customer with monthly_income is greater than {value}	customer:Customer(monthly_income > {value})

Scope	Expression	Code
[when]	The application with loan_amount is less than {value1} and duration_year is less than {value2} or c...	application:Application(loan_amount <

Fig. 3: The Complex Condition Editor.

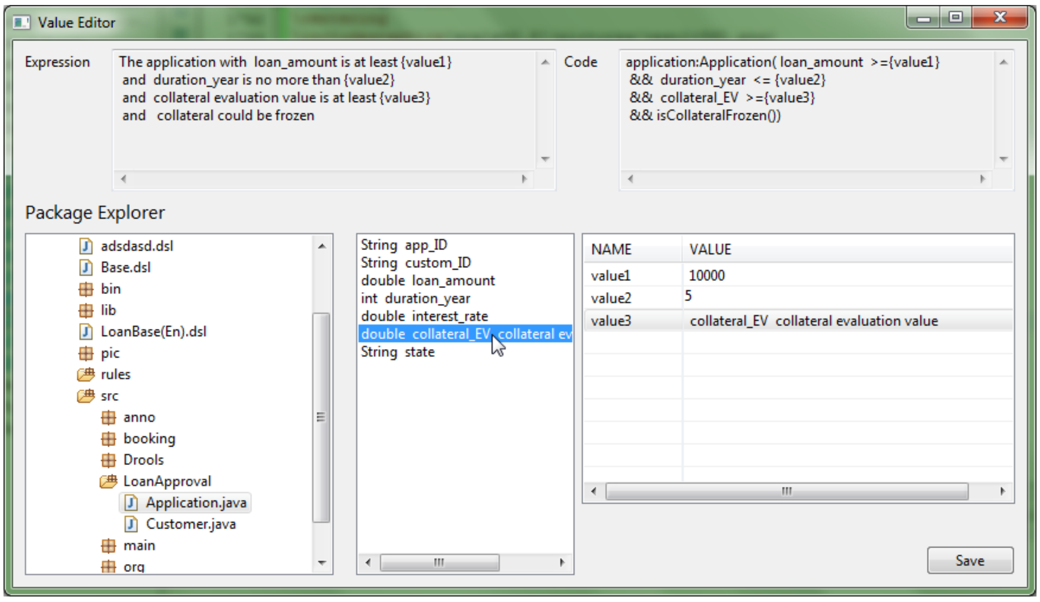
DSL *Rule Editor.* After loading a .dsl file, the user can select conditions and actions to create rules. For selecting a DSL table item, the user can double click it. If there are parameters in this DSL sentence, then a *Value Editor* will popup, see Figure 5. All parameters will be displayed in a table. The user can directly assign a value to a parameter, or he can find an attribute of a class and assign it to a parameter within a Package Explorer. Double clicking a class displays the list of all its attributes. Selecting a value cell and double clicking on an attribute puts the attribute into the cell. Finally, the DSL rule must be named before it can be saved into a new .dslr file or appended at the end of an existing .dslr file.



Scope	Expression	Rule Mapping
[when]	The application with (loan_amount is less than {value1} and duration_...	application:Application((loan_amount < {value1} .
[when]	The customer with credit_report is {value1} and monthly_income is gr...	customer:Customer(credit_report == {value1} &&
[then]	The application will be approved	application.approval();
[then]	The application will be denied	application.deny();
[then]	Set the interest_rate of the application to {value}	applicationnstance.setInterest_rate({value});

Scope	Expression
[when]	The application with (loan_amount is less than 10000 and duration_year is less than 5) or collateral evaluation value is greate...
[when]	The customer with credit_report is "good" and monthly_income is greater than 2000
[then]	The application will be approved
[then]	Set the interest_rate of the application to 0.025

Fig. 4: The DSL Rule Editor.



```
The application with  loan_amount is at least {value1}
 and  duration_year is no more than {value2}
 and  collateral evaluation value is at least {value3}
 and  collateral could be frozen
```

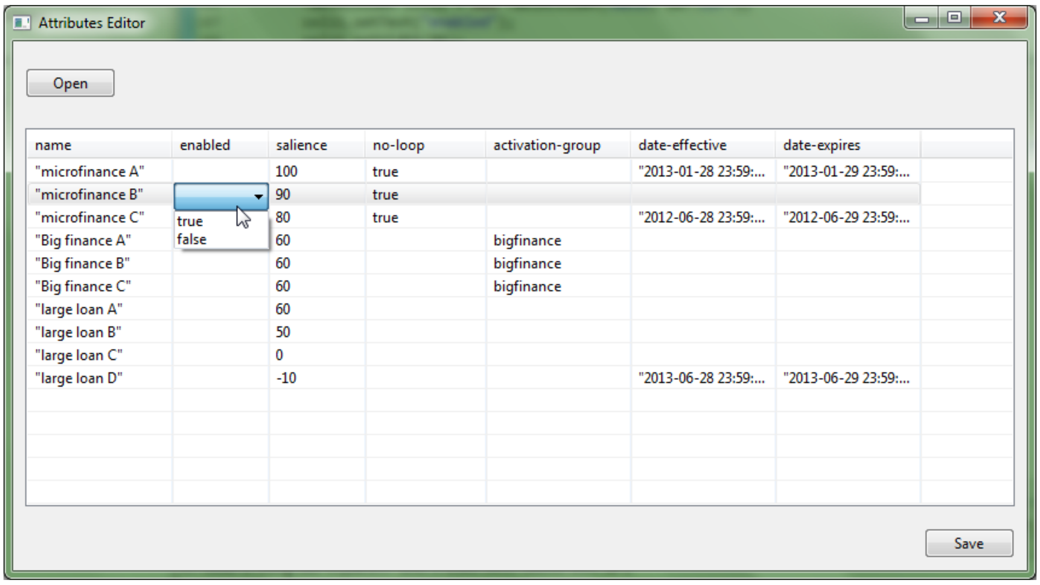
```
application:Application( loan_amount  >={value1}
 &&  duration_year  <= {value2}
 &&  collateral_EV  >={value3}
 && isCollateralFrozen())
```

```
String  app_ID
String  custom_ID
double  loan_amount
int  duration_year
double  interest_rate
double  collateral_EV  collateral ev
String  state
```

NAME	VALUE
value1	10000
value2	5
value3	collateral_EV collateral evaluation value

Fig. 5: The Value Editor.

Attribute Editor. This editor gives an overview over values of special attributes of rules in DROOLS, e.g., the attribute `salience` to control a rule's priority, see Figure 6. The user can modify these attributes easily and control the behavior of the complete set of rules in DROOLS.



name	enabled	salience	no-loop	activation-group	date-effective	date-expires
"microfinance A"		100	true		"2013-01-28 23:59:...	"2013-01-29 23:59:...
"microfinance B"		90	true			
"microfinance C"		80	true		"2012-06-28 23:59:...	"2012-06-29 23:59:...
"Big finance A"		60		bigfinance		
"Big finance B"		60		bigfinance		
"Big finance C"		60		bigfinance		
"large loan A"		60				
"large loan B"		50				
"large loan C"		0				
"large loan D"		-10			"2013-06-28 23:59:...	"2013-06-29 23:59:...

Fig. 6: The Attribute Editor.

3.4 Case Study: Loan Approval

In the following, we will illustrate the development process with DSLR Generator by an example from the financial sector [3]. A bank wants to approve a customer's request for a loan automatically. For making a decision, certain parameters must be taken into account: the amount to be borrowed, the duration of the loan, the value of the collateral, and the credit history of the customer. After a loan has been approved, an interest rate will be set. A business analyst may have arranged several rules in natural language. For lack of space we demonstrate only two:

"microfinance". A loan with an amount of less than or equal to 5000 EUR and a duration shorter than or equal to 3 years to a customer with good credit report and monthly income greater than 2000 EUR, will be approved, and the interest rate is set to 7.5%.

"deny". A loan with an amount of more than 5000 EUR and a duration longer than 5 years to a customer with weak credit report and monthly income less than 3000 EUR, will be denied.

First, the programmer designs the entities, in our example the classes `Loan` and `Customer`, that cover all parameters of the application. Then, templates for the DSL can be designed. In doing this, the business analyst breaks the rules into single conditions and actions, and the programmer then only needs to implement the business logic of simple conditions and actions. Using meaningful annotations right from the start and in cooperation with the business analyst improves the readability of the templates and simplifies the selection of classes, attributes and methods without a programmer's help.

Now, the programmer can use the Basic DSL Editor to generate the following simple conditions and actions:

Listing 1.8: Macro Expansion from DSL Format to DROOLS Format

```
[when] The loan with loan_amount is smaller than
       or equal to {value} =
       loan:Loan( loan_amount  <= {value} )
[when] The loan with duration_year is smaller than
       or equal to {value} =
       loan:Loan( duration_year  <{value} )
[when] The customer with monthly_income is greater
       than {value} =
       customer:Customer( monthly_income > {value} )
[when] The customer with credit_report is {value2} =
       customer:Customer( credit_report == {value2} )
...
[then] Set the interest_rate of the loan to {value} =
       loan.setInterest_rate( {value} );
[then] The loan will be approved = loan.approval();
...
```

With the Complex Condition Editor, simple conditions from Listing 1.8 can be combined using connectives. This can be done by the business analyst, because the conditions are already in the readable format of the DSL. The result is saved to the file `LoanApproval.dsl`. After loading `LoanApproval.dsl` with DSLR Generator, the business analyst can finally generate the following rules:

Listing 1.9: LoanApproval.dslr

```
package LoanApproval
expander LoanApproval.dsl

rule "microfinance"
when
   The loan with loan_amount is smaller than or equal to 5000
   and duration_year is no more than 3
   The customer with monthly_income is greater than 2000
   and credit_report is "good"
then
   The loan will be approved
   Set the interest_rate of the loan to 0.075
end
```

```
rule "deny"
when
   The loan with loan_amount is greater than 5000
   and duration_year is greater than 5
   The customer with monthly_income is less than 3000
   and credit_report is "weak"
then
   The loan will be denied
end
```

After the delivery of the project, certain maintenance tasks could be carried out easily by the business analyst, such as modifying a parameter of a rule, disabling a rule, or even creating a new rule with DSLR Generator, as the following example shows.

"1% Finance". The bank plans to make a *"1% Finance"* promotion in the Christmas Season of 2013: a loan with an amount less than or equal to 5000 EUR and a duration of not more than 2 years to a customer with a good credit report and monthly income greater than 1000 EUR, will be approved, and the interest rate will be set to 1%. With DSLR Generator, a business analyst can implement this new requirement easily:

Listing 1.10: 1% Finance

```
rule "1%finance"
when
    The loan with loan_amount is smaller than or equal to 5000
    and duration_year is no more than 2
    The customer with monthly_income is greater than 1000
    and credit_report is "good"
then
    The loan will be approved
    Set the interest_rate of the loan to 0.010
end
```

The fixed period, in which the rule is active, can be set in the Attribute Editor via values for `date-effective` and `date-expires`.

If the rule *"1%finance"* can fire, then the rule *"microfinance"* can fire, too. Here, the question is which interest rate should be set: 1% or 7.5% ? A solution is to set values for the attribute `salience` in the Attribut Editor to prioritize the rules.

Problematic ambiguity and anomalies between rules can not be avoided by DSLR Generator. But as we will see in the next section, we want to improve this situation by transferring experience with analyzing anomalies in ontologies from [4] to rules in DROOLS and extend the analysis w.r.t. DROOLS specific concepts.

4 PROLOG–Based Analysis

Although the evaluation mechanism of DROOLS is much different from the resolution mechanism of PROLOG, we can support the rule development process by PROLOG technology. We can check for *anomalies*, such as duplicates of rules, contradictions, and ambiguities in a set of rules or analyze the templates in the file `template.xml` that contains the mapping information from the DSL to DROOLS. Moreover, we can use ideas from [13] for the visualization of the derivation process via proof trees. In the following we detail some parts of the PROLOG based analysis.

Templates. The PROLOG call

```
template_to_prolog(+Template, -FN)
```

loads a template file in XML into PROLOG. The representation of an XML element in PROLOG is a term in field notation; we can analyze this term using the query, transformation and update language FNQUERY of the DDK [15, 17]. The PROLOG call

```
dsl_anomaly(+DSL1, +DSL2, -Anomaly)
```

checks `<dsl>` elements for anomalies, such as duplicates, see Listing 1.11, or an occurrence of a keyword in an expression, but not in the corresponding code, and vice versa. The path expressions `Dsl_i/expression` select sub–elements of `Dsl_i`. Additionally, we can ensure that there is a corresponding element in the XML representation for every keyword used in an expression.

Listing 1.11: Check for Duplicates

```
dsl_anomaly(Dsl_1, Dsl_2, Anomaly) :-
   ( X := Dsl_1/expression,
     Y := Dsl_2/expression
   ; X := Dsl_1/code,
     Y := Dsl_2/code )
   equivalent(X, Y),
   Anomaly = redundant(X, Y).
```

Rule Mapping. We can transform rules created with DSLR Generator to similar rules in PROLOG and analyze them for DROOLS specific anomalies. The transformation process is similar to the transformation process from PROLOG to JAVA as proposed in [14] – in a slightly reversed manner. A JAVA class is mapped to a PROLOG compound term whose functor is derived from the name of the class. Class attributes of primitive type become atomic arguments of the term (or numbers/ strings), otherwise compound terms on their own. Parameter changes via getter and setter methods in DROOLS are implemented simply by accessing the corresponding argument of the term in PROLOG. Thereby, the mapping information contained in the underlying .dsl file is used for the unidirectional mapping of the rules to PROLOG. However, we do not map every method with its complete functionality; it is enough to keep track of how objects in the working memory are asserted, retracted, or modified. For instance, methods of application data, i.e., classes not in direct reference to the working memory in DROOLS, are ignored. Since methods of application data usually do not influence facts in the working memory, there is no need to map their functionality. A constraint of a pattern is an expression that always returns a boolean value; this can, e.g., be a comparison of values or a boolean return value of a JAVA method. Such methods are mapped to terms whose functors are the method names and with a single argument that represents the boolean return value. Comparators are mapped to the corresponding comparators in PROLOG. Access to internals of DROOLS via methods like `assertObject()` or `retractObject()` can easily be simulated by their counterparts in PROLOG. The PROLOG call

```
dslr_to_prolog(+Dslr_File, +Dsl_File, -Prolog)
```

maps a .dslr file created with DSLR Generator to a set of PROLOG rules. Listing 1.12 shows the rule `microfinance` of Listing 1.9 after the mapping to PROLOG.

Listing 1.12: Rule `microfinance` in PROLOG

```
microfinance(Loan_1, Customer, Loan_2) :-
   Loan_1 = application(_, Cid, Loan, Duration, _, _, _),
   call(Loan_1),
   Loan =< 5000,
   Duration =< 3,
   Customer = customer(Cid, _, Credit_Report, Monthly_Income),
   call(Customer),
   Monthly_Income > 2000,
   Credit_Report = good,
   State = approved,
   Loan_2 = application(_, Cid, Loan, Duration, _, _, State).
```

Notice that the transformed rules usually cannot be executed in PROLOG, since the action parts have a procedural semantics. But, we can perform certain types of analysis on the transformed rules.

Loops. The following non–trivial example of a DROOLS specific anomaly has to do with loops. A loop occurs in the following simplified situation. Assume that there is a rule with the name `loop` containing a single condition `cond` referencing a fact with the identifier `ident`, and in the action block there is only a `modifyObject(ident)` instruction. The instruction in the action block notifies DROOLS that the `ident` fact has been modified, and DROOLS fires all appropriate rules again, for instance the rule `loop`. Thus, DROOLS will enter the action block of the rule `loop` again, i.e., we have a loop. It is possible to disable loops by setting the `no-loop` attribute of a rule to `true`; then the rule can not fire again. Anomalies – as in the described scenario – typically occur during the development process.

With DSLR Generator, we can ensure a correct rule syntax and a readable format, with PROLOG we concentrate on the semantics. Before saving a .dslr file in DSLR Generator, we can call PROLOG to detect anomalies with

```
drools_anomaly(+Prolog, -Anomaly).
```

The anomalies are reported on backtracking in `Anomaly`. If there is no anomaly, then the saving progress can be finished successfully. Otherwise, a warning with details about the anomaly is displayed, and the saving progress could be interrupted.

5 Conclusions

We have presented a tool, DSLR Generator, for handling DSLs within DROOLS. It simplifies the creation of DSLs and the implementation of rules within the DSL. Graphical user interfaces support the user in the rule development with reusable and generic DSL templates, that can be used to create rules in a more abstract and readable format. The maintenance of the meta–data for the rules in DROOLS is supported, too. Furthermore, with DSLR Generator domain experts without programming experience can complete parts of the development and maintain rules without requiring help from the IT.

In the future, we are planning to provide a library of templates and to extend our PROLOG–based anomaly analysis.

References

1. Alibabatech. *What is fastjson.* `http://code.alibabatech.com/wiki/display/FastJSON/Home`
2. L. Amador. *Drools Developer's Cookbook.* Packt Publishing, 2012.
3. M. Bali. *Drools JBoss Rules 5.0 Flow.* July 2009, `http://www.packtpub.com/article/drools-jboss-rules-5.0-flow-part1`
4. J. Baumeister, D. Seipel. *Anomalies in Ontologies with Rules.* Journal of Web Semantics: Science, Services and Agents on the World Wide Web 8, 2010, No. 1, pp. 55–68.
5. P. Browne. *JBoss Drools Business Rules.* Packt Publishing, 2009.
6. M. Fowler. *Domain–Specific Languages.* Addison–Wesley, 2011.
7. B. v. Halle. *Business Rules Applied.* Wiley, 2002.
8. JBoss Community. *Drools – The Business Logic Integration Platform.* `http://www.jboss.org/drools/`.
9. JBoss Drools Team. *Drools 5.5.0 Final: Drools Expert.* `http://www.jboss.org/drools/documentation`.
10. Json.org *Introducing JSON* `http://www.json.org/`
11. K. Kaczor, G. Nalepa, L. Lysik, K. Kluza. *Visual Design of Drools Rule Bases Using the XTT2 Method.* Studies in Computational Intelligence, Semantic Methods for Knowledge Management and Communication, Springer, 2011.
12. Oracle. *Java Architecture for XML Binding (JAXB).* `http://www.oracle.com/technetwork/articles/javase/ index-140168.html`
13. L. Ostermayer, D. Seipel. *Knowledge Engineering for Business Rules in* PROLOG. Proc. Workshop on Logic Programming (WLP), 2012.
14. L. Ostermayer, D. Seipel. *A Prolog Framework for Integrating Business Rules into Java Applications.* Proc. 9th Workshop on Knowledge Engineering and Software Engineering (KESE), 2013.
15. D. Seipel. *Processing* XML*–Documents in Prolog.* Proc. 17th Workshop on Logic Programming (WLP), 2002.
16. D. Seipel, M. Hopfner, B.Heumesser: *Analyzing and Visualizing* PROLOG *Programs Based on* XML *Representations.* Proc. International Workshop on Logic Programming Environments (WLPE), 2003.
17. D. Seipel. *The* DISLOG *Developers' Kit (*DDK*).* `http://www1.informatik.uni- wuerzburg.de/database/DisLog/`
18. L. Wunderlich. *Java Rules Engines.* enwickler.press, 2006.

A SAT-Based Graph Rewriting and Verification Tool Implemented in Haskell

Marcus Ermler

University of Bremen, Department of Computer Science
P.O.Box 33 04 40, 28334 Bremen, Germany
maermler@informatik.uni-bremen.de

Abstract. This paper presents a SAT-based graph rewriting and verification tool implemented in the functional programming language Haskell. Graphs, matchings, single rewriting steps, and complete derivations are translated into propositional formulas where a satisfying assignment represents one of the successful derivations. Furthermore, graph properties like “Is there a Hamiltonian path?” or “Is the graph Eulerian?” can be expressed in a graph transformation system and, then, proven by a SAT solver via a translation of graph rewriting to propositional formulas. We outline the tool concepts and its main components. Moreover, we give a short motivation why Haskell is the language of choice for our approach.

Keywords: graph rewriting, SAT solving, verification, Haskell

1 Introduction

Usually, graph related problems like the Hamiltonian path problem or the travelling salesperson problem are expressed and solved in common programming languages like Haskell, C++, or Java, in some cases by applying special approaches like parallelization, heuristics, or SAT solving. But an implementation of graph algorithms in such languages is often not straightforward. One has to describe graphs in internal data structures and has to implement methods or functions that operate on these data structures. Thereby, one introduces a new layer instead of directly working on graphs. Graph rewriting techniques have been proposed to directly describe and compute graph problems on the graph level what significantly eases the implementation of graph algorithms. Therefore, there is a need for appropriate graph rewriting tools that allow the implementation and execution of graph algorithms. Examples for such machines are the graph rewriting engine GrGen.Net [4] or the GROOVE tool [7].

The application of graph rewriting rules is in general nondeterministic because in each derivation step several matches and rules could be applied. Using control conditions for guiding the rule application is a first step to restrict the nondeterminism. But especially in case of NP-complete graph problems this is not sufficient because one gets an exponential number of derivations but only a small part of them yield proper results. Hence, a translation to propositional formulas seems promising to benefit from fast solving techniques implemented in modern SAT solvers.

In this paper, we present our SATaGraT tool[1] for implementing and solving graph problems in a SAT-based graph rewriting and verification framework. Graph transformation units [5] are employed as graph rewriting systems and their semantic is translated into propositional formulas. By using this approach it is also possible to verify certain graph properties like “Is there a Hamiltonian path?” or “Is the graph Eulerian?”. A first implementation of the tool based on the translation in [6] is introduced in the author’s diploma thesis [1] and has been extended to the SATaGraT tool where an early version is described in [3]. Another translation to propositional formulas is introduced in a contribution to WFLP 2013 [2].

Main contributions. The implementation is now restructured in three separate processing steps: Preprocessing for generating propositional formulas, processing for finding a successful derivation, and postprocessing for visualization. The new formulas and the proposed verification approach from [2] are supplemented. Furthermore, first steps for translations into CSP and SMT are done (cf. Section 3.2). Moreover, this paper contains the first detailed description of SATaGraTs functionality and main concepts.

The paper is organized as follows. Graph rewriting and its translation into propositional formulas is explained in Section 2. In Section 3, we give an overview of the tool concepts, motivate the application of Haskell for graph rewriting and propositional formulas, and present conducted experiments. Section 4 contains the conclusion.

2 From Graph Rewriting to SAT

SATaGraT uses *edge labeled directed graphs without multiple edges* and with a finite node set. For a finite set Σ of labels, such a graph is a pair $G = (V, E)$ where $V = \{1, \dots, n\}$ for some $n \in \mathbb{N}$ is a finite set of nodes and $E \subseteq V \times \Sigma \times V$ is a set of labeled edges. For the matching of subgraphs in graphs, we use *injective graph morphisms* that are structure- and label-preserving.

For transforming graphs, we use rules that add and delete edges. To handle the addition and deletion of nodes, deleted and unused nodes are marked with a special label named `deleted`. Such a *rule* $r = (L \to R)$ consists of two graphs: the *left-hand side* L and the *right-hand side* R where the node set remains invariant, i.e. $V_L = V_R$. The *rule application* of r to a graph G works as follows: Search for a subgraph H that is isomorphic to L, i.e. find a match $g(L)$ in G. If such a subgraph is found, delete the edges of $g(L)$ and add the edges of $g(R)$.

A *graph transformation unit* is a system $gtu = (I, P, C, T)$ where I and T are classes of initial and terminal graphs, P is a set of rules, and C is a control condition. A unit specifies all derivations from initial to terminal graphs that are

[1] SATaGraT is an abbreviation for ***SAT*** *solving* ***a****ssists* ***Gra****ph* ***T****ransformation* Engine. For downloading and testing our tool, please visit the SATaGraT home page: **www.informatik.uni-bremen.de/∼maermler/satagrat/index.html**.

allowed by the control condition. We use regular expressions as control conditions supplemented by the as-long-as-possible operator.

The semantic of graph transformation units is translated into propositional formulas. Single rule applications are expressed via four formulas: (1) one for describing matches of left-hand sides into graphs, (2) one for expressing the added edges, (3) one for describing deleted edges, and (4) one for describing those edges that are not added or deleted. Moreover, single derivations and all possible derivations are expressed via formulas. To get an impression how these formulas work and look like, we explain the formula for matchings from [6]:

$$\mathsf{morph}(\mathsf{r},\mathsf{g},\mathsf{k}) = \bigwedge_{(v,a,v')\in E_L} edge(g(v),a,g(v'),k-1).$$

All edges of the left-hand side of the rule r must have a match in the graph of the $(k-1)$th derivation step w.r.t. to a graph morphism g.

3 Implementation

In the following, we motivate the application of Haskell to graph rewriting and propositional formulas, describe the main components of SATaGraT, detail the three processing steps of the tool, and give some experimental results.

3.1 Why is Haskell the Language of Choice?

The implementation of the formulas from [6], [3], and [2] is in most cases straightforward and can easily be expressed via primitive recursion or using higher-order functions like `foldr`. For example, the formula morph for the matching can be implemented by using `foldr` in the following way.

```
morph :: Rule -> GraphMorphism -> DerivationStep -> Tree PropEdge
morph r g k
 = foldr (\ (v,a,v') edges
           -> TAnd (TLeaf $ makePropEdge (image g v) a (image g v') (k-1)) edges)
        TTrue edges
 where edges = Set.toList $ getEdges $ getLeftHandSide r
```

Graphs and graph rewriting rules can also be implemented in a straightforward way. All implementations are near to their mathematical description. Therefore, the author decided to use Haskell as language of choice.

3.2 Main Components

SATaGraT has been implemented in Haskell, uses MiniSat 2[2], Funsat 0.6.0[3], GrGen.NET 3.0, and Glasgow Haskell Compiler 6.12 and has been tested on an Intel 3.2 GHz with 8GB RAM. Optional are the CSP Solver Sugar 1.14.7[4], Limboole 0.2[5], and the SMT Solver Yices 1.029[6]. SATaGraT consists of the fol-

[2] http://minisat.se/
[3] http://hackage.haskell.org/package/funsat
[4] http://bach.istc.kobe-u.ac.jp/sugar/
[5] http://fmv.jku.at/limboole/
[6] http://yices.csl.sri.com/

lowing five main components.

- **Graph rewriting.** Modules for graphs, graph morphisms, rules, control conditions and graph transformation units are implemented.
- **Propositional formulas.** Three different translations can be utilized: **(1)** In [6], the first translations to SAT were introduced where used rules and morphisms can only be reconstructed via an algorithm that yields in some cases ambiguous results (cf. [3]). But the formulas are still smaller as those proposed in [2]; **(2)** in [2], the translations yield formulas in conjunctive normal form by introducing additional variables. These variables allow to easily reconstruct applied rules and morphisms for each derivation step. Moreover, this makes it possible to solve subformulas and, in this way, results in a speed-up of the processing time in some cases. But, the new variables lead to greater formulas compared with those in [6]; **(3)** the translations in [3] generate smaller formulas as those in [6, 2] and supplements the as-long-as-possible operator. But, the generated formulas are not in CNF.
- **Solvers.** SATaGraT provides the application of the SAT solvers MiniSat, Limboole, and Funsat. Furthermore, we adopt the CSP solver Sugar and the SMT solver Yices. At the moment, both solvers are only applied on propositional formulas. For coming SATaGraT versions it is planned to find translations of graph rewriting into CSP and SMT.
- **Verification.** One can prove existentially quantified properties like "Is there a Hamiltonian path?" or "Is the graph Eulerian?". In [2], we propose an approach of how all quantified properties over terms can be proven.
- **Examples.** One can find examples for various graph problems like the Hamiltonian path problem, job-shop scheduling, or the vertex cover problem.

3.3 Processing Steps

In this subsection, we describe the processing steps of the current SATaGraT version. The processing steps in the earlier versions differ in some aspects like generating whole formulas instead of subformulas or using no final visualization in GrGen.NET. In the system descriptions in Figure 1, Figure 2, and Figure 3 data is surrounded by rectangles with rounded corners and operations on data are surrounded by rectangles.

Preprocessing: From Graph Transformation Units to Rule Sequences. A graph transformation unit and a polynomial bound for the state space exploration are the input of SATaGraT (cf. Figure 1). The set $L(C)(p(n))$ in Figure 1 denotes the language that results from a control condition C with the restriction to a word length of exact $p(n)$. Each of these words is a rule sequence with length $p(n)$ and describes a sequence of rule applications from initial to terminal graphs.

Processing: Finding a Solution via a Translation into Propositional Formulas and SAT Solving. One of the rule sequences is chosen (see Figure 2) and together with the graph transformation unit and the polynomial translated into

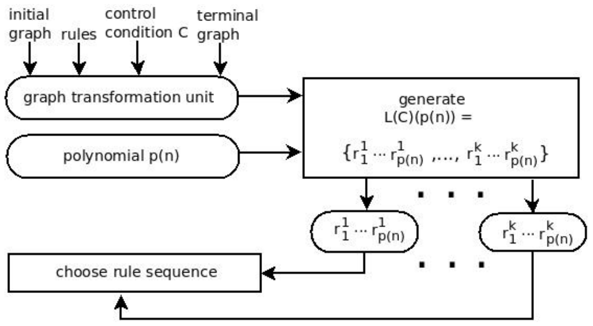

initial graph
rules
control condition C
terminal graph
graph transformation unit
polynomial p(n)
generate L(C)(p(n)) =
$\{r_1^1 \cdots r_{p(n)}^1, \ldots, r_1^k \cdots r_{p(n)}^k\}$
$r_1^1 \cdots r_{p(n)}^1$
$r_1^k \cdots r_{p(n)}^k$
choose rule sequence

Fig. 1. Preprocessing in SATaGraT

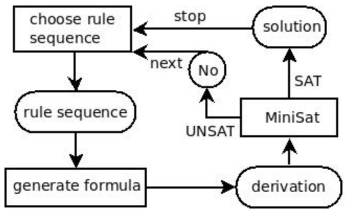


Fig. 2. Processing: Finding a solution

a formula in CNF that represents a derivation. Then, the generated formula is fed into MiniSat, a fast, competitive, and easy-to-use SAT solver. If the formula is satisfiable, MiniSat delivers a satisfying variable assignment and stops further computations. In the negative case, a simple *No* is sent and the next, not yet processed rule sequence is selected. This process runs as long as no solution is found or all possible rule sequences are already processed. A satisfying assignment states that the derivation is successful, i.e. we can reach a terminal graph for the given input graph by sequentially applying the rules of the rule sequence.

Postprocessing: Visualization by GrGen.NET. If a solution is found, the derivation is directly extracted from the variable assignment to present it the user in two different ways (see Figure 3): (1) the applied rules and matches in ev-

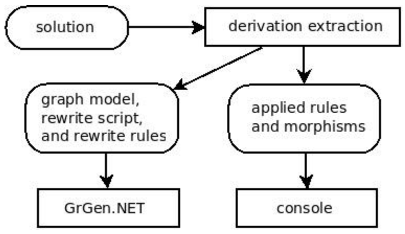

solution
derivation extraction
graph model, rewrite script, and rewrite rules
applied rules and morphisms
GrGen.NET
console

Fig. 3. Postprocessing

$\|V\|$	$\|E\|$	k	VC?	SATaGraT 2011	SATaGraT 2012
7	8	2	no	5	5
9	12	2	no	30	32
11	14	4	yes	96	34.5
13	20	3	yes	366	112
13	18	3	no	357	456
15	24	3	yes	> 3600	438

Fig. 4. Results for instances of VC

ery derivation step are displayed on the console and (2) the necessary files to run GrGen.NET are generated, in particular a graph model, a graph rewrite script consisting of the computed rules and matches, and rewrite rules; finally, GrGen.NET is executed.

3.4 Experiments

We conducted experiments for the NP-complete *vertex cover (VC) problem* (see Figure 4). The first three columns list the number of nodes, edges, and members of the vertex cover instance. The fourth column indicates whether the corresponding formula is satisfiable or not. The last two columns each contain the overall computation time in seconds. The overall time includes the generation time of the formula as well as the solving time of MiniSat. *SATaGraT 2012* contains the translation of [2] and delivers for the test cases the answer *yes* faster than the 2011 version. On the other hand, SATaGraT 2011 that is based on [6, 3] is a little bit faster in giving a *no* answer. Both is a result of the facts described in Section 3.2.

4 Conclusion

In this paper, we have presented a SAT-based rewriting and verification tool by giving an overview of its main concepts and a system description. Further conducted experiments can be found in [1, 3, 2]. We are planning to make it more usable by means of a graphical user interface for the input of graph transformation units and a visualization of rewriting steps. A first idea is to extend the GROOVE tool [7] by our SAT approach because it offers an appropriate GUI and visualization. At the moment, we can prove properties for single graphs like "Is the graph Eulerian". In future, we want to transfer the ideas from [2] to graphs for proving all quantified properties like "For all graphs G . . . ".

Acknowledgments. The author is grateful to the anonymous referees for their helpful comments and suggestions.

References

1. Ermler, M.: Untersuchung des dynamischen Verhaltens von Graphtransformationseinheiten mit Hilfe eines SAT-Solvers. Master's thesis, University of Bremen (2010)
2. Ermler, M.: Towards a verification framework for Haskell by combining graph transformation units and SAT solving. In: WFLP 2013 (2013), accepted for presentation
3. Ermler, M., Kreowski, H.-J., Kuske, S., von Totth, C.: From graph transformation units via MiniSat to GrGen.NET. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 153–168. Springer (2012)
4. Geiß, R., Kroll, M.: GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 568–569. Springer (2008)
5. Kreowski, H.-J., Kuske, S., Rozenberg, G.: Graph transformation units – An overview. In: Degano, P., Nicola, R. D., Meseguer, J. (eds.) Concurrency, Graphs and Models, LNCS, vol. 5065, pp. 57–75. Springer (2008)
6. Kreowski, H.-J., Kuske, S., Wille, R.: Graph transformation units guided by a SAT solver. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 27–42. Springer (2010)
7. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Pfaltz, J. L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer (2004)

Euler/X: A Toolkit for Logic-based Taxonomy Integration

Mingmin Chen[1], Shizhuo Yu[1], Nico Franz[2], Shawn Bowers[3], and Bertram Ludäscher[1]

[1] Dept. of Computer Science, UC Davis, {michen,szyu,ludaesch}@ucdavis.edu
[2] School of Life Sciences, Arizona State University, nico.franz@asu.edu
[3] Dept. of Computer Science, Gonzaga University, bowers@gonzaga.edu

Abstract. We introduce Euler/X, a toolkit for logic-based taxonomy integration. Given two taxonomies and a set of alignment constraints between them, Euler/X provides tools for detecting, explaining, and reconciling inconsistencies; finding all possible merges between (consistent) taxonomies; and visualizing merge results. Euler/X employs a number of different underlying reasoning systems, including first-order reasoners (Prover9 and Mace4), answer set programming (DLV and Potassco), and RCC reasoners (PyRCC8). We demonstrate the features of Euler/X and provide experimental results showing its feasibility on various synthetic and real-world examples.

1 Introduction

Biological taxonomies are hierarchical representations used to specify formal classifications of organismal groups (e.g., species, genera, families, etc.). While the names used for organismal groups (i.e., *taxa*) are regulated by various *Codes* of nomenclature, it is widely recognized that names alone are not sufficiently granular to integrate taxonomic entities occuring in related classifications [8,5,2]. Thus additional information is required to relate taxonomic entities across taxonomies. These relationships can then be used to compare different taxonomies and integrate multiple taxonomies into a single hierarchical representation.

The first attempts to provide formal reasoning over taxonomies were made in the MoReTax project [1], which introduced the use of RCC-5 relations [10] for defining relationships (articulations) among taxonomic concepts. RCC-5 provides five basic relations for defining *congruence*, *proper inclusion*, *inverse proper inclusion*, *overlap*, and *exclusion* among pairs of sets. These comparative relations are intuitive to taxonomic experts who assert them and who may also express ambiguity in their assessment among concept pairs by using disjunctions of articulations: when the exact relation is unknown to the expert, she can choose multiple RCC-5 relations of which one is assumed true. The MoReTax approach was formalized in first-order logic and implemented in CleanTax [12]. This system implemented RCC-5 reasoning using the first-order theorem provers

Mace4 and Prover9 [9], but also adding three taxonomic covering assumptions—*non-emptiness*, *sibling disjointness*, and *parent coverage*—to achieve a working environment for taxonomic reasoning.

Here we demonstrate the EULER/X toolkit which offers a suite of interactive reasoning and visualization programs that extend the capabilities of CLEANTAX while improving scalability. EULER/X also adds new reasoning approaches to CLEANTAX including ASP (Answer Set Programming [6]) and a specialized RCC-8 reasoner [11]. The toolkit implements a comprehensive taxonomy import, merge, and visualization workflow, with new features such as (1) PostgreSQL input of the original taxonomies and expert-asserted articulations [4], (2) detection of alignment inconsistencies, (3) diagnosis of inconsistency provenance (based on provenance semirings [7]) and interactive repair, (4) alignment ambiguity reduction, and (5) visualization of merged taxonomies based on a set of inferred, *maximally informative relationships* (MIR) that reflect (6) one or multiple possible worlds scenarios for taxonomy integration. We illustrate these features using an abstract example that embodies various of the aforementioned challenges (inconsistency, ambiguity, multiple possible worlds) while maintaining close resemblance with real-life use cases [5,3].

EULER/X encodes the input taxonomies, articulations, and constraints and feeds various inferences problems to different reasoners (the "X" in EULER/X), then translates the output from those reasoners to suit user needs. The main technical contribution are the ASP and other encodings, the use of provenance, and result visualization, applied to real-world taxonomy integration problems.

2 System Demonstration

Example. To demonstrate EULER/X, we introduce a simple example (Fig. 1) of two taxonomies T_1 (original) and T_2 (revised). Each taxonomy includes only two levels (genus and species) and ten constituent taxonomic concepts (1_A, 1_B, 2_A, ...). Moreover there are six initial, expert-asserted articulations that connect the respective entities. Three of these include disjunctions ('or'), reflecting the expert's uncertainty as to the precise relationship among concept pairs, and one leads to an inconsistency (though the expert is not yet aware of this error). Comparable, real-life examples are provided in [3].

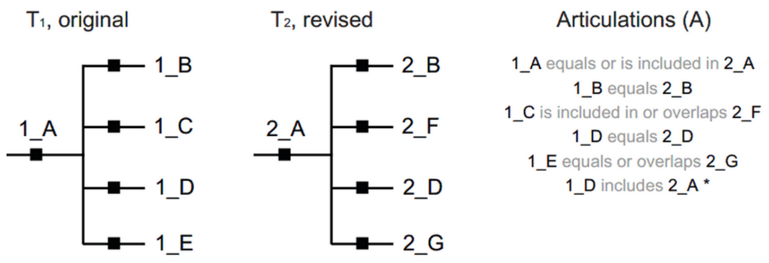


Fig. 1: Abstract example with two succeeding taxonomic classifications T_1, T_2 and a set of expert-asserted articulations (A) among taxonomic concepts. Three articulations are disjunctive; one ('*') leads to an inconsistency. T_2 (revised) builds on T_1 (original) but is a modification of T_1; it reuses T_1 entities but views and arranges them differently.

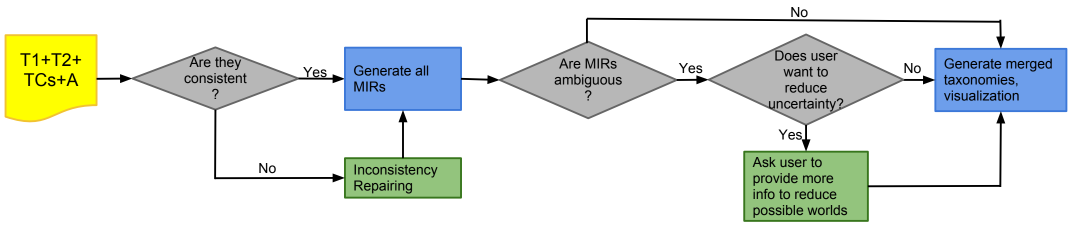

T1+T2+
TCs+A
Are they consistent ?
Yes
No
Generate all MIRs
Inconsistency Repairing
Are MIRs ambiguous ?
Yes
No
Does user want to reduce uncertainty?
No
Yes
Generate merged taxonomies, visualization
Ask user to provide more info to reduce possible worlds

Fig. 2: EULER/X workflow overview: Input taxonomies $\mathsf{T}_1, \mathsf{T}_2$ together with expert articulations A and other taxonomic constraints TCs yield MIRs, merged taxonomies, and visualization products.

Workflow Overview. EULER/X will ingest the example input (Fig. 1) into PostgreSQL in the form of three simple spreadsheets: (1) a table that uniquely identifies each of the ten taxonomic concepts; (2) a table that incorporates each set of five concepts into its respective taxonomy (T_1, T_2) via *is_a* parent/child relationships (e.g., 1_B *is_a* 1_A, etc.); and (3) a table with the six input articulations (A). The user also specifies a set of taxonomic constraints (TCs), e.g., *coverage*. The system then guides the user through an interactive workflow (Fig. 2) that includes the following major functions: consistency checking (including inconsistency explanation and repair), MIR generation, ambiguity representation (possible worlds[4]) and reduction, and lastly output of the merged taxonomies, including visualization and explanation of newly inferred MIR. Jointly, these functions enable the expert to obtain and comprehend a maximally consistent and unambiguous tabular and graphic representation of the merged taxonomy. Alternative reasoners—Prover9/Mace4 (FOL), DLV, Pottasco (ASP), and PyRCC8 (RCC)—are integrated into the workflow to suit specific reasoning challenges.

Consistency Checking and Inconsistency Repair. The example (Fig. 1) is computable in EULER/X using either FOL or ASP reasoners (Fig. 2). The first processing step focuses on testing the consistency of the input alignment (A). In our use case, the EULER/ASP and EULER/FO both infer that the input is inconsistent. In particular, EULER/FO provides a black-box explanation that "1_D includes 2_A" is inconsistent with the remaining articulations, and recommends removing this articulation to obtain a consistent alignment. In contrast, EULER/ASP offers a white-box explanation, stating that "1_D includes 2_A" (implying that 1_D is a high-level, inclusive taxonomic concept) is inconsistent with "1_A equals or is included in 2_A" and "1_D *is_a* 1_A" (jointly asserting that 1_D is a low-level, non-inclusive concept). Thus one can repair the inconsistency simply by deleting the articulation "1_D includes 2_A". Based on subsequent EULER/X reasoning (MIR), we will find that the correct 1_D/2_A articulation is "1_D is included in 2_A".

Generating MIR and Possible World Visualizations. Once the input example's inconsistency is repaired, EULER/X will proceed to generate all *maximally informative relations* (MIR; see Thau *et al.*, 2009 [12]) among taxonomic concept pairs. The interaction of the three articulations involving disjunction

[4] In each possible world, the relation of any two taxa is one of the RCC5.

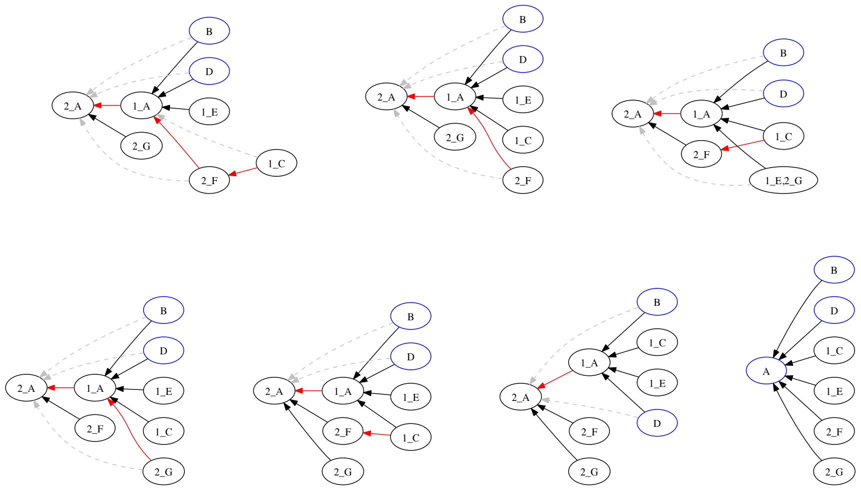


Fig. 3: Set of possible worlds pw0, pw1, . . . , pw6 (here: reduced containment graphs RCGs) resulting from the MIR inferred by EULER/X based on the repaired input example: blue nodes show congruent, merged concept in both taxonomies; black nodes show concept unique to each taxonomy; black edges show input *is_a* relations; dashed grey edges show redundant *is_a* relations; and red edges show newly inferred *is_a* relations. Note that RCGs do not represent concept overlap.

(Fig. 1) form an inherently ambiguous input alignment, which results a total of seven equally consistent "possible world" solutions. These possible worlds can be displayed using a simple "reduced containment graph" (Fig. 3).

Facilitating interactive ambiguity reduction. Although the seven possible worlds (Fig. 3) accurately reflect the resolving power of the input alignment (Fig. 1), the user may now have the ability and desire to reduce the inherent ambiguity by selectively eliminating certain (apparently improbable) possible worlds. This is facilitated by the EULER/X feature of ambiguity reduction. At run time, EULER will ask the user more questions (generated by a decision tree function) via the pop-out interactive windows which allow the user to select the preferred answer.

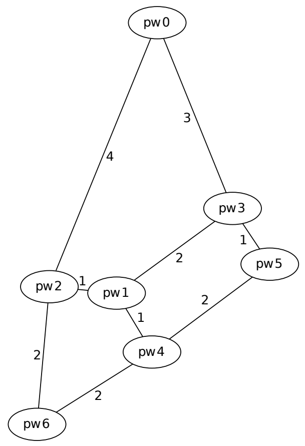


Fig. 4: Visualization of possible worlds for the input example, where the distance between two possible worlds is the shortest distance traceable in the graph (e.g., the distance between worlds 5 and 6 is 4).

Visual clustering of similar possible worlds. We can expect some use cases with larger sized input taxonomies and multiple inherent ambiguities to yield large numbers of possible worlds. EULER/X offers a visual representation of the cumulative possible worlds "universe" via a distance matrix (Fig. 4). As shown in Fig. 3, our input example has seven possible worlds. We can compute pairwise distances among these by integrating the numbers of MIRs in which they differ and thereby generating a network that summarizes the similarities and differences.

Additional features. EULER/X also provides information on the *provenance* of a newly generated MIR relation. Moreover the program can provide users with a consensus perspective of all possible worlds, i.e., specifying what is true in all of them, or how often a particular MIR occurs across all possible worlds.

3 Performance Results

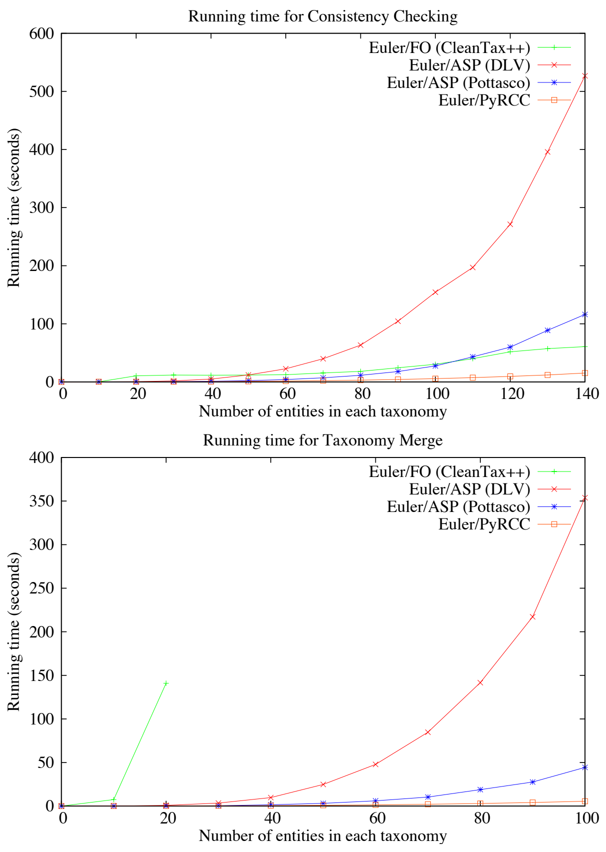


Fig. 5: Running times for consistency checking (*top*) and taxonomy merge (*down*) on synthetic taxonomies (balanced taxonomy trees of depth 8 with "is included in" articulations, resulting in a single possible world).

We tested the performance and scalability of different reasoning approaches, including EULER/FO (Prover9/Mace4), EULER/ASP (DLV and Pottasco), and EULER/PyRCC (PyRCC8). Tests used both real-life and simulated examples as well as performed both consistency checks and MIR and possible worlds computation. The running time was measured using increasingly larger input datasets. All examples were tested on an 8-core, 32GB-memory Linux server.

While EULER/FO checks consistency by calling Mace4 once and then generates each MIR by calling Prover9[5] (for $m * n$ MIR's assuming there are m, n entities in each taxonomy), the other EULER tools only invoke the reasoner once to check consistency and merge taxonomies (MIR and possible world generation). This is why EULER/FO is faster for consistency checking (specifically, EULER/FO is slower than EULER/ASP (Pottasco) when the number of entities in each taxonomy is less than 100, but faster when it is more than 100), but very slow in MIR generation as shown in Fig. 5. For taxonomy merge, PyRCC8 is faster than Potassco, Potassco is faster than DLV, and DLV is much faster than FO. However, note that EULER/PyRCC is not capable of applying the same

[5] To get a MIR, Prover9 is called to answer yes or no to the five base relation questions

merge as the other tools since the coverage constraints cannot be asserted using RCC-5. When considering all three taxonomic constraints, the Pottasco-based Euler is the fastest and reasonably good overall, since it can perform taxonomy merge for realistic taxonomies of 100 entities in half a minute.

4 Conclusions and Future Directions

Euler/X is open source and can be downloaded from BitBucket[6]. Planned developments include: (1) support for incremental changes to alignments; (2) an improved ASP-based tool, using the results from PyRCC8; (3) development of a user-friendly GUI; and (4) exploration of other reasoners, e.g., those developed for OWL.

Acknowledgements. We thank the anonymous reviewers for their helpful comments. Work supported in part by NSF awards IIS-1118088 and DBI-1147273.

References

1. W. G. Berendsohn. *MoReTax: Handling Factual Information Linked to Taxonomic Concepts in Biology.* Schriftenreihe für Vegetationskunde 39:1-113, 2003.
2. B. Boyle, N. Hopkins, Z. Lu, J. A. R. Garay, D. Mozzherin, T. Rees, N. Matasci, M. L. Narro, W. H. Piel, S. J. Mckay, et al. The taxonomic name resolution service: an online tool for automated standardization of plant names. *BMC Bioinformatics*, 14:16, 2013.
3. N. M. Franz and J. Cardona-Duque. Description of two new species and phylogenetic reassessment of *Perelleschus* Wibmer & O'Brien, 1986 (Coleoptera: Curculionidae), with a complete taxonomic concept history of *Perelleschus* sec, 2013.
4. N. M. Franz and R. K. Peet. Towards a language for mapping relationships among taxonomic concepts. *Systematics and Biodiversity*, 7(1):5–20, 2009.
5. N. M. Franz, R. K. Peet, and A. S. Weakley. On the use of taxonomic concepts in support of biodiversity research and taxonomy. *Systematics Association Special Volume*, 76:63, 2008.
6. M. Gelfond. Answer sets. In F. van Harmelen, V. Lifschitz, and B. Porter, editors, *Handbook of Knowledge Representation*, pages 285–316. Elsevier, 2008.
7. T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
8. J. Kennedy, R. Kukla, and T. Paterson. Scientific names are ambiguous as identifiers for biological taxa: Their context and definition are required for accurate data integration. In *Data Integration in the Life Sciences (DILS)*, LNCS 3615, 2005.
9. W. McCune. Prover9 and Mace4, 2005–2010. www.cs.unm.edu/~mccune/prover9.
10. D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In *Knowledge Representation and Reasoning (KR)*, 1992.
11. M. Sioutis. A RCC8 based qualitative spatial reasoner written in pure python. pypi.python.org/pypi/PyRCC8, 2012.
12. D. Thau, S. Bowers, and B. Ludäscher. Merging Sets of Taxonomically Organized Data Using Concept Mappings under Uncertainty. In *Ontologies, DataBases, and Applications of Semantics*, LNCS 5871, 2009.

[6] https://bitbucket.org/eulerx/euler-project

Debate Games in Logic Programming

Chiaki Sakama

Department of Computer and Communication Sciences
Wakayama University, Sakaedani, Wakayama 640-8510, Japan
sakama@sys.wakayama-u.ac.jp

Abstract. A *debate game* provides an abstract model of debates between two players based on the formal argumentation framework. This paper presents a method of realizing debate games in logic programming. Two players have their knowledge bases represented by extended logic programs and build claims using arguments associated with those programs. A player revises its knowledge base with arguments posed by the opponent player, and tries to refute claims by the opponent. During a debate game, a player may claim false or incorrect arguments as a tactic to win the game. The result of this paper provides a new formulation of debate games in a non-abstract argumentation framework associated with logic programming. Moreover, it provides a novel application of logic programming to modelling social debates which involve argumentative reasoning, belief revision and dishonest reasoning.

1 Introduction

Logic programming and argumentation are two different frameworks for knowledge representation and reasoning in artificial intelligence (AI). In his seminal paper, Dung [4] points out a close connection between the two frameworks and shows that a logic program can be considered as a schema for generating arguments. Since then, several attempts have been made for integrating the two frameworks ([1, 12, 8, 20]; see [9] for an overview).

A line of research of formal argumentation is concerned with the dialectical process of two or more players who are involved in a discussion [3]. Along this line, Sakama [18] introduces a *debate game* between two players based on the formal argumentation framework. In a debate game, a player makes the initial claim, then the opponent player tries to refute it by building a counter-claim. A debate continues until one cannot refute the other, and the player who makes the last claim wins the game. A debate game has unique features such that (i) each player has its own argumentation framework as its background knowledge, (ii) during a debate each player revises its argumentation framework by new arguments provided by the opponent player, and (iii) a player may claim inaccurate or even false arguments as a tactic to win a debate. The study [18] formulates debate games using the abstract argumentation theory of [4].

The abstract argumentation theory has an advantage that it is not bound to any particular representation for arguments on the one hand, but on the other hand it does not specify how arguments are generated from the underlying knowledge base and what conclusions are yielded by those arguments. In [2] the authors argue that "Argumentation, as it happens in the world around us, is almost never completely abstract. ...

Instead, the arguments one encounters in daily life consist of *reasons* that support particular *claims*. These reasons can formally be modelled in the form of *rules*, that are instances of underlying *argumentation schemes* [14]." In this respect, debate games based on the abstract argumentation theory need yet another formulation based on non-abstract argumentation frameworks.

With this motivation, this paper uses logic programming as an underlying representation language and formulates debate games in a non-abstract argumentation framework. In this framework, each player has a knowledge base represented by an extended logic program, and builds claims using arguments which can contain information brought by the opponent as well as information in the player's program. During a game, a player may use *dishonest* claims to refute the opponent, while a player must be self-consistent in its claims. The proposed framework provides an abstraction of real-life debates and realizes a formal dialogue system in logic programming. The rest of this paper is organized as follows. Section 2 reviews a framework of argument-based logic programming. Section 3 introduces debate games in logic programming and investigates formal properties. Section 4 discusses related issues and Section 5 concludes the paper.

2 Arguments in Logic Programming

In this paper we consider the class of extended logic programs [10]. An *objective literal* is a ground atom B or its explicit negation $\neg B$. We define $\neg\neg B = B$. A *default literal* is of the form $not\, L$ where L is an objective literal and not is *negation as failure* (NAF). An *extended logic program* (or simply a *program*) P is a finite set of *rules* of the form:

$$L_0 \leftarrow L_1, \ldots, L_m,\ not\, L_{m+1}, \ldots, not\, L_n$$

where each L_i $(0 \leq i \leq n)$ is an objective literal. The literal L_0 is the *head* of the rule and the conjunction $L_1, \ldots, L_m,\ not\, L_{m+1}, \ldots, not\, L_n$ is the *body* of the rule. A rule r is *believed-true* in P if $r \in P$. A rule containing default literals is called a *default rule*. A rule $L \leftarrow$ with the empty body is also called a *fact* and is identified with a literal L.

Let Lit be the set of all objective literals in the language of a program. A set S $(\subset Lit)$ is *consistent* if $L \in S$ implies $\neg L \notin S$ for any $L \in Lit$. The semantics of a program is given by its *answer sets* [10]. First, let P be a program containing no default literal and $S \subset Lit$. Then, S is an *answer set* of P if S is a consistent minimal set satisfying the condition that for each rule of the form $L_0 \leftarrow L_1, \ldots, L_m$ in P, $\{L_1, \ldots, L_m\} \subseteq S$ implies $L_0 \in S$. Second, given any program P (possibly containing default literals) and $S \subset Lit$, a *reduct* of P with respect to S (written P^S) is defined as follows: a rule $L_0 \leftarrow L_1, \ldots, L_m$ is in P^S iff there is a rule of the form $L_0 \leftarrow L_1, \ldots, L_m,\ not\, L_{m+1}, \ldots, not\, L_n$ in P such that $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$. Then, S is an *answer set* of P if S is an answer set of P^S. A program may have none, one or multiple answer sets in general. A program is *consistent* if it has an answer set; otherwise, it is *inconsistent*.

Definition 2.1. ([12, 20]) An *argument* associated with a program P is a finite sequence $A = [r_1; \cdots; r_n]$ of rules $r_i \in P$ such that (i) for every $1 \leq i \leq n$, for every objective literal L_j in the body of r_i there is a rule r_k $(k > i)$ such that the head of r_k is L_j. (ii) No two distinct rules in the sequence have the same head.

The head of a rule in an argument A is called a *conclusion* of A, and a default literal $not\, L$ in the body of a rule in A is called an *assumption* of A. We write $assum(A)$ for the set of assumptions and $concl(A)$ for the set of conclusions of an argument A. By the condition (i) of Definition 2.1, every objective literal in the body of a rule r_i is justified by the consequence of a rule that appears later in the sequence. The condition (ii) keeps an argument from containing circular sequences of rules. A *subargument* of A is a subsequence of A which is an argument. An argument A with a conclusion L is a *minimal argument for* L if there is no subargument of A with the conclusion L. An argument is *minimal* if it is minimal for some literal L. The minimality condition presents that an argument does not include rules which do not contribute to conclude some particular literal L.

Remark: In this paper, we slightly abuse the notation and use the same letter A to denote the *set* of rules included in an argument A. Thus, $P \cup A$ means the set of rules included either in a program P or in an argument A.

Example 2.1. Let P be the program:

$$\begin{aligned}&p \leftarrow q,\\&\neg p \leftarrow not\ q,\\&q \leftarrow,\\&r \leftarrow s.\end{aligned}$$

Then, the following facts hold.

- The minimal argument for p is $A_1 = [p \leftarrow q\,;\, q \leftarrow]$, $concl(A_1) = \{p, q\}$, and $assum(A_1) = \emptyset$.
- The minimal argument for $\neg p$ is $A_2 = [\neg p \leftarrow not\, q]$, $concl(A_2) = \{\neg p\}$ and $assum(A_2) = \{not\, q\}$.
- The minimal argument for q is $A_3 = [\, q \leftarrow]$, $concl(A_3) = \{q\}$ and $assum(A_3) = \emptyset$.
- r and s have no minimal arguments.

Proposition 2.1. *Let P be a consistent program containing no default literal. Then, for any argument A associated with P, $concl(A) \subseteq S$ holds for the answer set S of P.*

Proof. Let P' be the program which is obtained by replacing every negative literal $\neg L$ in P with a new atom L' that is uniquely associated with $\neg L$. As P is consistent, P' has the least model S' iff P has the answer set S where $\neg L$ in S is replaced by the atom L' in S'. Let A' be an argument associated with P'. Then, $A' \subseteq P'$ implies $concl(A') \subseteq S'$ by the monotonicity of deduction. By replacing L' with $\neg L$, $A \subseteq P$ implies $concl(A) \subseteq S$. □

Definition 2.2. ([12, 20]) Let A_1 and A_2 be two arguments.

- A_1 *undercuts* A_2 if there is an objective literal L such that L is a conclusion of A_1 and $\mathit{not}\ L$ is an assumption of A_2.
- A_1 *rebuts* A_2 if there is an objective literal L such that L is a conclusion of A_1 and $\neg L$ is a conclusion of A_2.
- A_1 *attacks* A_2 if A_1 undercuts or rebuts A_2.
- A_1 *defeats* A_2 if A_1 undercuts A_2, or A_1 rebuts A_2 and A_2 does not undercut A_1.

An argument is *coherent* if it does not attack itself. A set S of arguments is *conflict-free* if no argument in S attacks an argument in S. Given a program P, we denote the set of minimal and coherent arguments associated with P by $Args(P)$.

If an argument A_1 undercuts another argument A_2, then A_1 denies an assumption of A_2. This means that the assumption conflicts with the evidence to the contrary, and A_1 defeats A_2 in this case. If A_1 rebuts A_2, on the other hand, two arguments support contradictory conclusions. In this case, the attack relation is symmetric and A_1 defeats A_2 under the condition that A_2 does not undercut A_1. The coherency condition presents self-consistency of an argument. By definition, if $A \in Args(P)$ then the set A of rules is consistent.

Example 2.2. In the program P of Example 2.1, the following facts hold.

- $Args(P) = \{A_1,\ A_2,\ A_3\}$.
- A_1 and A_3 undercut (and also defeat) A_2.
- A_1 rebuts A_2 and A_2 rebuts A_1.
- $\{A_1, A_3\}$ is conflict-free, but $\{A_1, A_2\}$ and $\{A_2, A_3\}$ are not.
- The argument $A_4 = [p \leftarrow q\,;\ \neg p \leftarrow \mathit{not}\, q;\ q \leftarrow]$ is incoherent.

Proposition 2.2. *Let P be a consistent program. For any argument $A \in Args(P)$, if A is not defeated by any argument associated with P, then $concl(A) \subseteq S$ for any answer set S of P.*

Proof. Let A^+ be the set of rules obtained from A by removing every default literal in A. When A is not defeated by any argument associated with P, $A^+ \subseteq A^S$ for any answer set S of P, where A^S is the reduct of A wrt S. By Proposition 2.1, for any argument A^S associated with P^S, $concl(A^S) \subseteq S$ for any answer set S of P. Since $A^+ \subseteq A^S$ implies $concl(A^+) \subseteq concl(A^S)$, the result holds. □

In Example 2.1, A_1 and A_3 are defeated by no argument, then $concl(A_1)$ and $concl(A_3)$ are subsets of the answer set $\{p, q\}$ of P.

3 Debate Games in Logic Programming

3.1 Debate Games

A debate game involves two players. Each player has its knowledge base defined as follows.

Definition 3.1 (player). A *player* has a knowledge base $K = (P, O)$ where P is a consistent program representing the player's belief and O is a set of rules brought by another player. In particular, the initial knowledge base of a player is $K = (P, \emptyset)$.

In this paper, we identify a player with its knowledge base. We represent two players by K_1 and K_2. For a player K_1 (resp. K_2), the player K_2 (resp. K_1) is called the *opponent*.

Definition 3.2 (revision). Let $K = (P, O)$ be a player and A an argument. Then, *revision* of K with A is defined as

$$rev(K, A) = (P \setminus R,\ O \cup A)$$

where $R = \{\, r \mid$ there is a literal L in $concl(A)$ such that *not* L is in the body of a rule r and A is not defeated by any argument associated with $P \cup O \cup A\,\}$.

The function rev is iteratively applied to a player. We represent the result of the i-th revision of K by $K^i = (P^i, O^i)$ $(i \geq 0)$, that is, $K^i = (P^i, O^i) = rev(K^{i-1}, A_i)$ $(i \geq 1)$ for arguments $A_1, \ldots, A_i$ and $K^0 = (P^0, O^0) = (P, \emptyset)$.

Note that we handle A as a set here. By definition, revision adds rules A to O while it removes default rules R from P. When a player K cannot defeat the new argument A, the player is obliged to accept it and removes default rules R that have assumptions conflicting with conclusions of A. The reason of separating P and O is to distinguish belief originated in a player's program from information brought by the opponent player. A player having a knowledge base after the i-th revision is represented by K^i, but we often omit the superscript i when it is unimportant in the context.

Definition 3.3 (claim). Let $K_1 = (P_1, O_1)$ and $K_2 = (P_2, O_2)$ be two players.

1. The *initial claim* is a pair of the form: $(\texttt{in}(A), \_)$ where $A \in Args(P_1)$. It is read that "the player K_1 claims the argument A."
2. A *counter-claim* is a pair of the form: $(\texttt{out}(B), \texttt{in}(A))$ where $A \in Args(P_k \cup O_k)$ and $B \in Args(P_l \cup O_l)$ $(k,\ l = 1,\ 2;\ k \neq l)$. It is read that "the argument B by the player K_l does not hold because the player K_k claims the argument A".

The initial claim or counter-claims are simply called *claims*. A claim $(\texttt{in}(A), \_)$ or $(\texttt{out}(B), \texttt{in}(A))$ by a player is *refuted* by the claim $(\texttt{out}(A), \texttt{in}(C))$ with some argument C by the opponent player.

Definition 3.4 (debate game). Let $K_1^0 = (P_1^0, O_1^0)$ and $K_2^0 = (P_2^0, O_2^0)$ be two players. Then, an *admissible debate* Δ is a sequence of claims: $[(\texttt{in}(X_0), \_), (\texttt{out}(X_0), \texttt{in}(Y_1)), (\texttt{out}(Y_1), \texttt{in}(X_1)), \ldots, (\texttt{out}(X_i), \texttt{in}(Y_{i+1})), (\texttt{out}(Y_{i+1}), \texttt{in}(X_{i+1})), \ldots]$ such that

(a) $(\texttt{in}(X_0), \_)$ is the initial claim by K_1^0 where $X_0 \in Args(P_1^0)$.
(b) $(\texttt{out}(X_0), \texttt{in}(Y_1))$ is a claim by K_2^1 where $K_2^1 = rev(K_2^0, X_0) = (P_2^1, O_2^1)$ and $Y_1 \in Args(P_2^1 \cup O_2^1)$.
(c) $(\texttt{out}(Y_{i+1}), \texttt{in}(X_{i+1}))$ is a claim by K_1^{i+1} where $K_1^{i+1} = rev(K_1^i, Y_{i+1}) = (P_1^{i+1}, O_1^{i+1})$ and $X_{i+1} \in Args(P_1^{i+1} \cup O_1^{i+1})$ $(i \geq 0)$.

(d) $(\mathtt{out}(X_i), \mathtt{in}(Y_{i+1}))$ is a claim by K_2^{i+1} where $K_2^{i+1} = rev(K_2^i, X_i) = (P_2^{i+1}, O_2^{i+1})$ and $Y_{i+1} \in Args(P_2^{i+1} \cup O_2^{i+1})$ ($i \geq 0$).

(e) for each $(\mathtt{out}(U), \mathtt{in}(V))$, V defeats U.

(f) for each $\mathtt{out}(Z)$ in a claim by K_1^i (resp. K_2^i), there is $\mathtt{in}(Z)$ in a claim by K_2^j such that $j \leq i$ (resp. K_1^j such that $j < i$).

(g) both $\bigcup_{i \geq 0}\{\, X_i \mid X_i \subseteq P_1^0\}$ and $\bigcup_{j \geq 1}\{\, Y_j \mid Y_j \subseteq P_2^0\}$ are conflict-free.

Let $\mathit{\Gamma}_n$ ($n \geq 0$) be any claim. A *debate game* $\mathit{\Delta}$ (*for an argument* X_0) is an admissible debate between two players $[\mathit{\Gamma}_0, \mathit{\Gamma}_1, \ldots]$ where the initial claim is $\mathit{\Gamma}_0 = (\mathtt{in}(X_0), \_\_)$ and $\mathit{\Gamma}_m \neq \mathit{\Gamma}_{m+2k}$ ($m \geq 0; k > 0$). A debate game $\mathit{\Delta}$ for an argument X_0 *terminates* with $\mathit{\Gamma}_n$ if $\mathit{\Delta} = [\mathit{\Gamma}_0, \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n]$ is an admissible debate and there is no claim $\mathit{\Gamma}_{n+1}$ such that $[\mathit{\Gamma}_0, \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_n, \mathit{\Gamma}_{n+1}]$ is an admissible debate. In this case, the player who makes the last claim $\mathit{\Gamma}_n$ *wins* the game.

By definition, (a) the player K_1^0 starts a debate with the claim $\mathit{\Gamma}_0 = (\mathtt{in}(X_0), \_\_)$. (b) The player K_2^0 then revises its knowledge base with X_0, and responds to the player K_1^0 with a counter-claim $\mathit{\Gamma}_1 = (\mathtt{out}(X_0), \mathtt{in}(Y_1))$ based on the revised knowledge base K_2^1. In response to $\mathit{\Gamma}_1$, the player K_1^1 revises its knowledge base and builds a counter-claim $\mathit{\Gamma}_2 = (\mathtt{out}(Y_1), \mathtt{in}(X_1))$. A debate continues by iterating revisions and claims ((c),(d)), and (e) in each claim an argument V of $\mathtt{in}(V)$ defeats an argument U of $\mathtt{out}(U)$. (f) A player can refute not only the preceding claim of the opponent player, but any previous claim of the opponent. (g) During a debate game, arguments which come from a player's own program must be conflict-free, that is, each player must be self-consistent in its claims. Note that a player K_l^i ($l = 1, 2;\ i \geq 1$) can construct arguments using rules included in arguments O_l^i posed by the opponent player as well as rules in its own program P_l^i. This means that conclusions of arguments claimed by a player may change nonmonotonically during a game. If a player K_l^i claims $(\mathtt{out}(A), \mathtt{in}(B))$ which is refuted by a counter-claim $(\mathtt{out}(B), \mathtt{in}(C))$ by the opponent, then the player K_l^j ($i < j$) can use rules in the argument C for building a claim. Once the player K_l^j uses rules in C, it implies that K_l^j withdraws some conclusions of the argument B previously made by K_l^i (because B is defeated by C). Thus, two different claims by the same player may conflict during a game. The condition (g) states that such a conflict is not allowed among arguments which consist of rules from a player's original program P^0. In a debate game, a player cannot repeat the same claim ($\mathit{\Gamma}_m \neq \mathit{\Gamma}_{m+2k}$), otherwise arguments may go round in circles.

A debate game is represented as a directed tree in which the root node represents the initial claim, each node represents a claim, and there is a directed edge between two nodes $\mathit{\Gamma}_i$ and $\mathit{\Gamma}_j$ if the former refutes the latter. Figure 1 represents a debate game $\mathit{\Delta} = [\mathit{\Gamma}_0, \mathit{\Gamma}_1, \ldots, \mathit{\Gamma}_6]$ in which the player K_1^0 makes the initial claim $\mathit{\Gamma}_0$, the player K_2^1 makes a counter-claim $\mathit{\Gamma}_1$, the player K_1^1 refutes $\mathit{\Gamma}_1$ by $\mathit{\Gamma}_2$, and the player K_2^2 refutes $\mathit{\Gamma}_2$ by $\mathit{\Gamma}_3$. At this stage, K_1^2 cannot refute $\mathit{\Gamma}_3$ but refutes $\mathit{\Gamma}_1$ by $\mathit{\Gamma}_4$. The player K_2^3 cannot refute $\mathit{\Gamma}_4$ but refutes $\mathit{\Gamma}_0$ by $\mathit{\Gamma}_5$. Then, K_1^3 refutes $\mathit{\Gamma}_5$ by $\mathit{\Gamma}_6$. The player K_2^4 cannot refute $\mathit{\Gamma}_6$ and other claims by the opponent. As a result, the player K_1^3 wins the game. In what follows, we simply say "a debate game" instead of "a debate game for an argument X_0" when the argument X_0 in the initial claim is clear or unimportant in the context.

$$\begin{tikzcd}
 & & & \mathit{\Gamma}_0 \text{ (by } K_1^0\text{)} & & \\
 & & \mathit{\Gamma}_1 \text{ (by } K_2^1\text{)} \arrow[ur] & & \mathit{\Gamma}_5 \text{ (by } K_2^3\text{)} \arrow[ul] & \\
 & \mathit{\Gamma}_2 \text{ (by } K_1^1\text{)} \arrow[ur] & & \mathit{\Gamma}_4 \text{ (by } K_1^2\text{)} \arrow[ul] & & \mathit{\Gamma}_6 \text{ (by } K_1^3\text{)} \arrow[ul] \\
\mathit{\Gamma}_3 \text{ (by } K_2^2\text{)} \arrow[ur] & & & & &
\end{tikzcd}$$

Fig. 1. Debate game

Proposition 3.1. *Let $\mathit{\Gamma}$ be a claim of either $(\mathtt{in}(U), \_)$ or $(\mathtt{out}(V), \mathtt{in}(U))$ in a debate game. Then, U has a single answer set S such that $concl(U) = S$ and $concl(V) \not\subseteq S$.*

Proof. Since U is minimal and coherent, U has a single answer set S such that $concl(U) = S$. As U defeats V, there is a rule $r \in V$ such that the head of r is not included in S. □

Proposition 3.2. *Every debate game terminates.*

Proof. By definition, each player cannot repeat the same claim in a debate game. Since the number of minimal and coherent arguments associated with a propositional program is finite, the result holds. □

Example 3.1. Suppose a dispute between a prosecutor and a defense. First, the prosecutor and the defense have knowledge bases $K_1^0 = (P_1, \emptyset)$ and $K_2^0 = (P_2, \emptyset)$, respectively, where

$$\begin{aligned}
P_1 :\ & guilty \leftarrow suspect,\ motive, \\
& evidence \leftarrow witness,\ not\ \neg\, credible, \\
& suspect \leftarrow, \quad motive \leftarrow, \quad witness \leftarrow . \\
P_2 :\ & \neg guilty \leftarrow suspect,\ not\ evidence, \\
& \neg\, credible \leftarrow witness,\ dark, \\
& suspect \leftarrow, \quad dark \leftarrow .
\end{aligned}$$

A debate game proceeds as follows.

– First, the prosecutor K_1^0 makes the initial claim:

  $(\mathtt{in}(X_0), \_)$ with $X_0 = [\,guilty \leftarrow suspect,\ motive;\ suspect \leftarrow;\ motive \leftarrow]$
  ("The suspect is guilty because he has a motive for the crime.")

  where $X_0 \in Args(P_1)$.

– The defense revises K_2^0 into $K_2^1 = rev(K_2^0, X_0) = (P_2^1, O_2^1)$ where $P_2^1 = P_2$ and $O_2^1 = X_0$, and makes a counter-claim:

  $(\mathtt{out}(X_0), \mathtt{in}(Y_1))$ with $Y_1 = [\,\neg guilty \leftarrow suspect,\ not\ evidence;\ suspect \leftarrow\,]$
  ("The suspect is not guilty as there is no evidence.")

  where $Y_1 \in Args(P_2^1 \cup O_2^1)$ and Y_1 rebuts X_0.
– The prosecutor revises K_1^0 into $K_1^1 = rev(K_1^0, Y_1) = (P_1^1, O_1^1)$ where $P_1^1 = P_1$ and $O_1^1 = Y_1$, and makes a counter-claim:

  $(\mathtt{out}(Y_1), \mathtt{in}(X_1))$ with $X_1 = [\,evidence \leftarrow witness,\ not\ \neg\, credible;\ witness \leftarrow\,]$
  ("There is an eyewitness who saw the suspect on the night of the crime.")

  where $X_1 \in Args(P_1^1 \cup O_1^1)$ and X_1 undercuts Y_1.
– The defense revises K_2^1 into $K_2^2 = rev(K_2^1, X_1) = (P_2^2, O_2^2)$ where $P_2^2 = P_2$ and $O_2^2 = X_0 \cup X_1$. (Note that the first rule of P_2 is not removed by the revision because $P_2^1 \cup O_2^1 \cup X_1$ can defeat X_1). Then, the defense makes a counter-claim:

  $(\mathtt{out}(X_1), \mathtt{in}(Y_2))$ with $Y_2 = [\,\neg\, credible \leftarrow witness,\ dark;\ witness \leftarrow;\ dark \leftarrow\,]$
  ("The testimony is incredible because it was dark at night.")

  where $Y_2 \in Args(P_2^2 \cup O_2^2)$ and Y_2 undercuts X_1.
– The prosecutor revises K_1^1 into $K_1^2 = rev(K_1^1, Y_2) = (P_1^2, O_1^2)$ where $P_1^2 = P_1 \setminus \{evidence \leftarrow witness,\ not\ \neg\, credible\}$ and $O_1^2 = Y_1 \cup Y_2$. Since K_1^2 cannot refute the claim by K_2^2, the defense wins the game.

3.2 Dishonest Player

In debate games, each player constructs claims using rules included in its program or rules brought by the opponent. To defeat a claim by the opponent, a player may claim an argument which the player does not believe its conclusion.

Example 3.2. Suppose that the prosecutor in Example 3.1 has the program

$$P_1' = P_1 \cup \{\,\neg\, dark \leftarrow light,\ not\ broken,\quad light \leftarrow,\quad broken \leftarrow\,\}.$$

In response to the last claim $(\mathtt{out}(X_1), \mathtt{in}(Y_2))$ by the defense K_2^2, suppose that the prosecutor $K_1^2 = ({P_1'}^2, O_1^2)$ where ${P_1'}^2 = P_1'$ makes a counter-claim:

$$(\mathtt{out}(Y_2), \mathtt{in}(X_2)) \text{ with } X_2 = [\,\neg\, dark \leftarrow light,\ not\ broken;\ light \leftarrow\,].$$

("It was not dark because the witness saw the suspect under the light of the victim's apartment."). Then, X_2 defeats Y_2.

In Example 3.2, the prosecutor K_1^2 claims the argument X_2 but he/she does not believe its conclusion $concl(X_2)$. In fact, $\neg\, dark$ is included in no answer set of the program ${P_1'}^2 \cup Q$ for any $Q \subseteq O_1^2$. Generally, a player may behave dishonestly by concealing believed facts to justify another fact which the player wants to conclude. We classify different types of claims which may appear in a debate game.

Definition 3.5 (credible, misleading, incredible, incorrect, false claims). Let $\varGamma$ be a claim of either $(\texttt{in}(U), \_)$ or $(\texttt{out}(V), \texttt{in}(U))$ by a player $K_l^i = (P_l^i, O_l^i)$ $(l = 1, 2; i \geq 0)$. Also, let U^S be an argument which consists of rules in the reduct of U with respect to a set S.

- $\varGamma$ is *credible* if $concl(U) \subseteq S$ for every answer set S of $P_l^i \cup Q$ for some $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent and $concl(U) = concl(U^S)$.
- $\varGamma$ is *misleading* if $concl(U) \subseteq S$ for every answer set S of $P_l^i \cup Q$ for some $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent but $concl(U) \neq concl(U^S)$.
- $\varGamma$ is *incredible* if $concl(U) \subseteq S$ for some (but not every) answer set S of $P_l^i \cup Q$ for any $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent.
- $\varGamma$ is *incorrect* if $concl(U) \not\subseteq S$ for any answer set S of $P_l^i \cup Q$ for any $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent, and $concl(U) \cup S$ is consistent for some answer set S of $P_l^i \cup Q$ for some $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent.
- $\varGamma$ is *false* if $concl(U) \cup S$ is inconsistent for any answer set S of $P_l^i \cup Q$ for any $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent.

A claim is called *dishonest* if it is not credible. A player K_l is *honest* in a debate game $\varDelta$ if every claim made by K_l^i $(i \geq 0)$ in $\varDelta$ is credible. Otherwise, K_l is *dishonest*.

During a game, a player K_l^i constructs an argument U using some rules $Q \subseteq O_l^i$. Then, U has the answer set which coincides with $concl(U)$ (Proposition 3.1), but this does not always imply that $concl(U)$ is a subset of an answer set of P_l^i.

Proposition 3.3. *Every claim in a debate game is classified as one of the five types of claims of Definition 3.5.*

Example 3.3.

- Given $K_1 = (\{\, p \leftarrow not\, q \,\}, \emptyset)$, the claim $\varGamma_1 = (\texttt{in}([\, p \leftarrow not\, q \,]), \_)$ is credible.
- Given $K_2 = (\{\, p \leftarrow not\, q,\quad p \leftarrow q,\quad q \leftarrow \}, \emptyset)$, the claim $\varGamma_2 = (\texttt{in}([\, p \leftarrow not\, q \,]), \_)$ is misleading.
- Given $K_3 = (\{\, p \leftarrow not\, q,\quad q \leftarrow not\, p \,\}, \emptyset)$, the claim $\varGamma_3 = (\texttt{in}([\, p \leftarrow not\, q \,]), \_)$ is incredible.
- Given $K_4 = (\{\, p \leftarrow not\, q,\quad q \leftarrow \}, \emptyset)$, the claim $\varGamma_4 = (\texttt{in}([\, p \leftarrow not\, q \,]), \_)$ is incorrect.
- Given $K_5 = (\{\, p \leftarrow not\, \neg p,\quad \neg p \leftarrow \}, \emptyset)$, the claim $\varGamma_5 = (\texttt{in}([\, p \leftarrow not\, \neg p \,]), \_)$ is false.

In Example 3.3, $\varGamma_1$ is credible because $concl([p \leftarrow not\, q]) = \{p\}$ coincides with the answer set of the program $\{p \leftarrow not\, q\}$ in K_1. By contrast, $\varGamma_2$ is misleading because for $U = \{p \leftarrow not\, q\}$ it becomes $U^S = \emptyset$ by the answer set $S = \{p, q\}$ of the program in K_2, so that $concl(U) \neq concl(U^S)$. That is, a misleading claim does not use rules in a proper manner to reach conclusions. $\varGamma_3$ is incredible because p is included in some but not in every answer set of the program in K_3. $\varGamma_4$ is incorrect because p is included in no answer set of the program in K_4. $\varGamma_5$ is false because $\neg p$ is included in every answer set of the program in K_5.

The existence of dishonest claims is due to the nonmonotonic nature of a program. A player $K = (P, O)$ is *monotonic* if P contains no default literal. In this case, the following result holds.

Proposition 3.4. *Let Δ be a debate game between two monotonic players. Then, every claim in Δ is credible.*

Proof. Let Γ be a claim of either $(\mathtt{in}(U), \_)$ or $(\mathtt{out}(V), \mathtt{in}(U))$ by a player $K = (P, O)$. By $U \in Args(P \cup O)$, $U \subseteq P \cup Q$ for some $Q \subseteq O$ such that $P \cup Q$ is consistent. Since U is an argument associated with $P \cup Q$, $concl(U) \subseteq S$ holds for the answer set S of $P \cup Q$ by Proposition 2.1. By $U^S = U$, $concl(U) = concl(U^S)$. Hence, Γ is credible. $\square$

Generally, it is unknown which player wins a debate game. In real life, a player who is more knowledgeable than another player is likely to win a debate. The situation is formulated as follows.

Proposition 3.5. *Let Δ be a debate game between two players $K_1^0 = (P_1, \emptyset)$ and $K_2^0 = (P_2, \emptyset)$. If K_1 (resp. K_2) is honest and $P_2 \subset P_1$ (resp. $P_1 \subset P_2$), then K_1^i (resp. K_2^i) ($i \geq 1$) wins the game.*

Proof. Suppose that K_1 is honest and $P_2 \subset P_1$. Let Γ_m be a honest claim of either $(\mathtt{in}(X_0), \_)$ or $(\mathtt{out}(Y_i), \mathtt{in}(X_i))$ by $K_1^i = (P_1^i, O_1^i)$ $(i \geq 0)$ in Δ. By $O_1^i \subseteq P_1$, $P_1^i \subseteq P_1$ and every rule in $P_1 \setminus P_1^i$ is undercut by some argument $A \in Args(P_1)$ and A is not defeated by any argument in $Args(P_1)$. Then, every rule in $P_1 \setminus P_1^i$ is eliminated in the reduct P_1^S for any answer set S of P_1. Then, P_1^i and P_1 have the same answer sets. Thus, $concl(X_i) \subseteq S$ for every answer set S of P_1. Suppose that K_2^{i+1} makes a counter-claim $\Gamma_{m+1} = (\mathtt{out}(X_i), \mathtt{in}(Y_{i+1}))$, and K_1^{i+1} cannot refute Γ_{m+1} by any honest claim. In this case, P_1 has no rule to defeat Y_{i+1}. By $P_2 \subset P_1$, $Y_{i+1} \subset P_1$. Then, $concl(Y_{i+1}) \subseteq S$ for every answer set S of P_1. Since Y_{i+1} defeats X_i, either (i) Y_{i+1} undercuts X_i or (ii) Y_{i+1} rebuts X_i but X_i does not undercut Y_{i+1}. In either case, $concl(X_i) \not\subseteq S$ for any answer set S of P_1. This contradicts the fact that $concl(X_i) \subseteq S$. Hence, K_1^{i+1} can refute Γ_{m+1} by a honest claim in Δ. As such, every claim by K_2^i is honestly refuted by K_1^i. Hence, K_1^i wins the game. When K_2 is honest and $P_1 \subset P_2$, it is shown in a similar way that K_2^i wins the game. $\square$

Proposition 3.5 presents that if a player K has information more than another player, K has no reason to behave dishonestly to win a debate. In fact, if a more informative player K behaves dishonestly, K may lose a game.

Example 3.4. Consider two players $K_1^0 = (P_1, \emptyset)$ and $K_2^0 = (P_2, \emptyset)$ where $P_1 = \{\, p \leftarrow not\, q, \quad q \leftarrow \}$ and $P_2 = \{\, q \leftarrow \}$. Then, $P_2 \subset P_1$. Suppose a debate game between K_1^0 and K_2^0 such that

$$\begin{aligned} K_1^0 &: \ \Gamma_0 = (\mathtt{in}([\, p \leftarrow not\, q\,]), \_) \\ K_2^1 &: \ \Gamma_1 = (\mathtt{out}([\, p \leftarrow not\, q]), \mathtt{in}([\, q \leftarrow])). \end{aligned}$$

The claim Γ_0 by K_1^0 is incorrect because p is included in no answer set of P_1. Since the player K_1^1 cannot refute Γ_1, K_2^1 wins the game.

A player has an incentive to build a dishonest claim if the player cannot build a honest counter-claim in response to the claim by the opponent. Then, our next question is how a player effectively uses dishonest claims as a tactic to win a debate. We first show that among different types of dishonest claims, misleading claims are useless for the purpose of winning a debate.

Proposition 3.6. *Let $\varDelta$ be a debate game between two players $K_1^0 = (P_1, \emptyset)$ and $K_2^0 = (P_2, \emptyset)$.*

1. *If the initial claim $\varGamma_0 = (\mathtt{in}(X_0), \_)$ by K_1^0 is misleading, there is a credible claim $\varGamma_0' = (\mathtt{in}(X), \_)$ by K_1^0 such that $concl(X) = concl(X_0)$.*
2. *If a claim $\varGamma_k = (\mathtt{out}(V), \mathtt{in}(U))$ by a player K_l^i $(l = 1, 2;\ i \geq 1)$ is misleading, there is a credible claim $\varGamma_k' = (\mathtt{out}(V), \mathtt{in}(W))$ by K_l^i such that $concl(W) = concl(U)$.*

Proof. (1) Since $concl(X_0) \subseteq S$ for every answer set S of P_1, there is a set $X \subseteq P_1$ of rules such that $concl(X_0) = concl(X) = concl(X^S)$. Selecting a minimal set X of rules satisfying the conditions of Definition 2.1, the result holds. (2) Since $concl(U) \subseteq S$ for every answer set S of $P_l^i \cup Q$ for some $Q \subseteq O_l^i$ such that $P_l^i \cup Q$ is consistent, there is a set $W \subseteq P_l^i \cup Q$ of rules such that $concl(U) = concl(W) = concl(W^S)$. Selecting a minimal set W of rules satisfying the conditions of Definition 2.1, the result holds. $\square$

Thus, dishonest claims which are effectively used for the purpose of winning a debate game are either incredible, incorrect or false claims. Once a player makes a dishonest claim in a game, however, it will restrict what the player can claim later in the game. In Example 3.3, the player K_4 who makes the incorrect claim $\varGamma_4$ cannot subsequently use the believed-true fact $q \leftarrow$ which conflicts with $\varGamma_4$ (Definition 3.4(g)). To keep conflict-freeness of a player's claims in a game, dishonest claims would restrict the use of believed-true rules in later claims and may result in a net loss of freedom in playing the game. With this reason, it seems reasonable to select a dishonest claim only if there is no choice among honest claims. Comparing different types of dishonest claims, it is considered that incredible claims are preferred to incorrect claims, and incorrect claims are preferred to false claims. If a claim $\varGamma = (\mathtt{out}(V), \mathtt{in}(U))$ is incredible, the player does not *skeptically* believe the conclusion of U but *credulously* believes the conclusion of U. If $\varGamma$ is incorrect, the player does not credulously believe the conclusion of U but the conclusion is consistent with the player's belief. If $\varGamma$ is false, on the other hand, the conclusion of U is inconsistent with the player's belief. Thus, the degree of truthfulness (against the belief state of a player) decreases from incredible claims to incorrect claims, and from incorrect claims to false claims (Figure 2). Generally, a dishonest claim deviates from the reality as believed by a player, and a claim which increases such deviation is undesirable for a player because it increases a chance of making the player's claims conflict. A player wants to keep claims close to its own belief as much as possible, so the best-practice strategy for a debate game is to firstly use credible claims, secondly use incredible ones, thirdly use incorrect ones, and finally use false ones to refute the opponent.

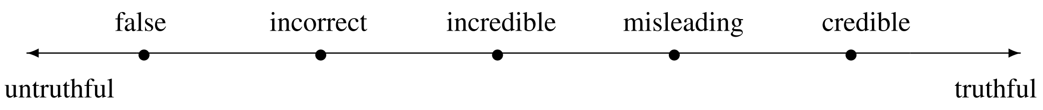


Fig. 2. Degree of truthfulness

4 Discussion

A formal argumentation framework has been used for modelling dialogue games or discussion games ([11, 13, 3]; and references therein). However, most of the studies use abstract argumentation and pay much attention on identifying acceptable arguments based on the topological nature of dialectical graphs associated with dialogues. On the other hand, the content of dialogue is important in human communication. Participants in debates are interested in why one's argument is defeated by the opponent, whether arguments made by the opponent are logically consistent, which arguments made by the opponent are unacceptable, and so on. In debate games proposed in this paper, each player can see the *inside* of the arguments in claims made by the opponent. As a result, a player can judge whether a counter-claim made by the opponent is grounded on evidences, and whether claims made by the opponent are consistent throughout a debate. Moreover, a player can obtain new information from arguments posed by the opponent.

In AI agents are usually assumed to be honest and little attention has been paid for representing and reasoning with dishonesty. In real-life debates, however, it is a common practice for one to misstate their beliefs or opinions [19]. In formal argumentation, [15] characterizes dishonest agents in a game-theoretic argumentation mechanism design and [18] introduces dishonest arguments in a debate game. These studies use the abstract argumentation framework and do not show how to construct dishonest arguments from the underlying knowledge base. In this paper, we show how to build dishonest arguments from a knowledge base represented by a logic program. Using arguments associated with logic programs, we argue that at least four different types of dishonest claims exist. In building dishonest claims, default literals play an important role—concealing known rules or facts could produce conclusions which are not believed by a player. Proposition 3.4 shows an interesting observation that players cannot behave dishonestly without default assumption. Dishonest reasoning in logic programs is introduced by [16] in which the notion of *logic programs with disinformation* is introduced and its computation by abductive logic programming is provided. An application of dishonest reasoning to multiagent negotiation is provided by [17] in which agents represented by abductive logic programs misstate their bargaining positions to gain one's advantage over the other. The current study shows yet another application of dishonest reasoning in argumentation-based logic programming.

Prakken and Sartor [12] introduce *dialogue trees* in order to provide a proof theory of argumentation-based extended logic programs. A dialogue tree consists of nodes representing arguments by the proponent and the opponent, and edges representing attack relations between arguments. Given the initial argument of the proponent at the root node of a dialogue tree, the opponent attacks the argument by a counterargument if any (called a *move*). Two players move in turn and one player wins a dialogue if the other player run out of moves in a tree. Comparing dialogue trees with debate games, a dialogue tree is constructed by arguments associated with a *single* extended logic program. In debate games, on the other hand, two players have different knowledge bases and build arguments associated with them. Dialogue trees are introduced to provide a proof theory of argumentation-based logic programs, and they do not intend to provide a formal theory of dialogues between two players. As a result, dialogue trees do no have mechanisms of revision and dishonest reasoning. Fan and Toni [6] propose a formal model for argumentation-based dialogues between agents. They use *assumption-based argumentation* (ABA) [5] for this purpose. In ABA arguments are built from rules and supported by assumptions, and attacks against arguments are directed at the assumptions supporting the arguments, and are provided by arguments for the contrary of assumptions. In their dialogue model, agents can utter claims to be debated, rules, assumptions, and contraries. A dialogue between the proponent and the opponent constructs a dialectical tree which represents moves by agents during a dialogue and outcomes. In their framework, two agents share a common ABA framework and assumed to have a common background knowledge. With this setting, an agent cannot behave dishonestly as one cannot keep some information from the other.

5 Conclusion

The contributions of this paper are mainly twofold. First, we developed debate games using a non-abstract argumentation framework associated with logic programming. We applied argumentation-based extended logic programs to formal modelling of dialogue games. Second, we showed an application of dishonest reasoning in argumentation-based logic programming. Debate games introduced in this paper realize dishonest reasoning by players using nonmonotonic nature of logic programs. To the best of our knowledge, there is no formal dialogical system which can deal with argumentative reasoning, belief revision and dishonest reasoning in a uniform and concrete manner. The current study contributes to a step toward integrating logic programming and formal argumentation.

The proposed framework will be extended in several ways. In real-life debates, players may use *assumptions* in their arguments. Assumptions are also used for constructing arguments in an assumption-based argumentation framework [5]. Arguments considered in this paper use assumptions in the form of default literals. To realize debate games in which players can also use objective literals as assumptions, we can consider a non-abstract assumption-based argumentation framework associated with *abductive logic programs*. In this framework, an argument associated with an abductive logic program can contain *abducibles* as well as rules in a program. A player can claim an argument containing abducibles whose truthfulness are unknown. This is an another type of dishonest claims called *bullshit* [7]. To realize debate games, we are now implementing a prototype system of debate games based on the abstract argumentation framework [16]. We plan to extend the system to handle non-abstract arguments associated with extended logic programs.

References

1. Bondarenko, A., Dung, P. M., Kowalski, R. and Toni, F. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence* 93(1,2):63–101, 1997.
2. Caminada, M. and Wu, Y. On the limitation of abstract argumentation. In: *Proceedings of the 23rd Benelux Conference on Artificial Intelligence (BNAIC)*, Gent, Belgium, 2011.
3. Caminada, M. Grounded semantics as persuasion dialogue. In: *Proceedings of the 4th International Conference on Computational Models of Argument (COMMA), Frontiers in Artificial Intelligence and Applications* 245, IOS Press, pp. 478–485, 2012.
4. Dung, P. M. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artificial Intelligence* 77(2):321–357, 1995.
5. Dung, P. M., Kowalski, R. A. and Toni, F. Assumption-based argumentation. In: I. Rahwan and G. R.. Simari (eds.), *Argumentation in Artificial Intelligence*, pp. 199–218, Springer, 2009.
6. Fan, X. and Toni, F. Assumption-based argumentation dialogues. In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pp. 198–203, 2011.
7. Frankfurt, H. G. *On Bullshit*. Princeton University Press, 2005.
8. García, A. J. and Simari, G. R. Defeasible logic programming: an argumentative approach. *Theory and Practice of Logic Programming* 4(1,2):95–138, 2004.
9. García, A. J., Dix, J. and Simari, G. R. Argumentation-based logic programming. In: I. Rahwan and G. R.. Simari (eds.), *Argumentation in Artificial Intelligence*, pp. 153–171, Springer, 2009.
10. Gelfond, M. and Lifschitz, V. Logic programs with classical negation. In: *Proceedings of the 7th International Conference on Logic Programming (ICLP)*, MIT Press, pp. 579–597, 1990.
11. Parsons, S., Wooldridge, M. and Amgoud, L. Properties and complexity of some formal inter-agent dialogues. *Journal of Logic and Computation* 13(3):347–376, 2003.
12. Prakken, H. and Sartor, G. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics* 7(1):25–75, 1997.
13. Prakken, H. Coherence and flexibility in dialogue games for argumentation. *Journal of Logic and Computation* 15(6):1009–1040, 2005.
14. Prakken, H. On the nature of argument schemes. In: *Dialectics, Dialogue and Argumentation, An Examination of Douglas Walton's Theories of Reasoning and Argument*, pp. 167–185, College Pub, 2010.
15. Rahwan, I., Larson, K., and Tohmé, F. A characterisation of strategy-proofness for grounded argumentation semantics. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 251–256, 2009.
16. Sakama, C. Dishonest reasoning by abduction. In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI-11)*, pp. 1063–1068, 2011.
17. Sakama, C., Son, T. C. and Pontelli, E. A logical formulation for negotiation among dishonest agents. In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 1069–1074, 2011.
18. Sakama, C. Dishonest arguments in debate games. In: *Proceedings of the 4th International Conference on Computational Models of Argument (COMMA), Frontiers in Artificial Intelligence and Applications* 245, IOS Press, pp. 177–184, 2012.
19. Schopenhauer, A. *The Art of Controversy*. Originally published in 1896 and is translated by T. Bailey Saunders, Cosimo Classics, New York, 2007.
20. Schweimeier, R. and Schroeder, M. A parameterized hierarchy of argumentation semantics for extended logic programming and its application to the well-founded semantics. *Theory and Practice of Logic Programming* 5(1,2), pp. 207–242, 2005.

A Datalog Engine for GPUs

Carlos Alberto Martínez-Angeles[1], Inês Dutra[2], Vítor Santos Costa[2], and Jorge Buenabad-Chávez[1]

[1] Departamento de Computación, CINVESTAV-IPN, Av. Instituto Politécnico Nacional 2508, 07360 D.F., México.
camartinez@cinvestav.mx,jbuenabad@cs.cinvestav.mx

[2] Departmento de Ciência de Computadores, Universidade do Porto, Rua do Campo Alegre, 1021, 4169-007, Porto, Portugal.
{ines,vsc}@dcc.fc.up.pt

Abstract. We present the design and evaluation of a Datalog engine for execution in Graphics Processing Units (GPUs). The engine evaluates recursive and non-recursive Datalog queries using a bottom-up approach based on typical relational operators. It includes a memory management scheme that automatically swaps data between memory in the host platform (a multicore) and memory in the GPU in order to reduce the number of memory transfers.
To evaluate the performance of the engine, three Datalog queries were run on the engine and on a single CPU in the multicore host. One query runs up to 200 times faster on the (GPU) engine than on the CPU.

Keywords: Logic Programming, Datalog, Parallel Computing, GPUs, Relational Databases

1 Introduction

The traditional view of Datalog as a query language for deductive databases is changing as a result of the new applications where Datalog has been in use recently, including declarative networking, program analysis, distributed social networking, security [17] — datalog recursive queries are at the core of these applications. This renewed interest in Datalog has in turn prompted new designs of Datalog targeting computing architectures such as GPUs, Field-programmable Gate Arrays (FPGAs) [17] and cloud computing based on Google's Mapreduce programming model [7]. This paper presents a Datalog engine for GPUs.

GPUs can substantially improve application performance and are thus now being used for general purpose computing in addition to game applications. GPUs are single-instruction-multiple-data (SIMD) [2] machines, particularly suitable for compute-intensive, highly parallel applications. They fit scientific applications that model physical phenomena over time and space, wherein the "compute-intensive" aspect corresponds to the modelling over time, while the "highly parallel" aspect to the modelling at different points in space.

Data-intensive, highly parallel applications such as database relational operations can also benefit from the SIMD model, substantially in many cases[11, 16, 15]. However, the communication-to-computation ratio must be relatively low for applications to show good performance, i.e.: the cost of moving data from host memory to GPU memory and vice versa must be low relative to the cost of the computation performed by the GPU on that data.

The Datalog engine presented here was designed considering various optimisations aimed to reduce the communication-to-computation ratio. Data is preprocessed in the host (a multicore) in order for: i) data transfers between the host and the GPU to take less time, and ii) for data to be processed more efficiently by the GPU. Also, a memory management scheme swaps data between host memory and GPU memory seeking to reduce the number of swaps.

Datalog queries, recursive and non-recursive, are evaluated using typical relational operators, *select, join* and *project*, which are also optimised in various ways in order to capitalise better on the GPU architecture.

Sections 2 and 3 present background material to the GPU architecture and the Datalog language. Section 4 presents the design and implementation of our Datalog Engine as a whole, and Section 5 of its relational operators. Section 6 presents an experimental evaluation of our Datalog engine. Section 7 presents related work and we conclude in Section 8.

2 GPU Architecture and Programming

GPUs are SIMD machines: they consist of many processing elements that all run the *same program* but on distinct data items. This same program, referred to as the *kernel*, can be quite complex including control statements such as *if* and *while* statements. However, a kernel is *synchronised by hardware*, i.e.: each instruction within the kernel is executed across all processing elements running the kernel. Thus, if a kernel has to compare strings, processing elements that compare longer strings will take longer and the other processing elements will wait for them.

Scheduling GPU work is usually as follows. A thread in the host platform (e.g., a multicore) first copies the data to be processed from host memory to GPU memory, and then invokes GPU threads to run the *kernel* to process the data. Each GPU thread has an unique id which is used by each thread to identify what part of the data set it will process. When all GPU threads finish their work, the GPU signals the host thread which will copy the results back from GPU memory to host memory and schedule new work.

GPU memory is organised hierarchically as shown in Figure 1. Each (GPU) thread has its own *per-thread local* memory. Threads are grouped into *blocks*, each block having a memory *shared* by all threads in the block. Finally, thread blocks are grouped into a single *grid* to execute a kernel — different grids can be used to run different kernels. All grids share the *global memory.*

The global memory is the GPU "main memory". All data transfers between the host (CPU) and the GPU are made through reading and writing global memory. It is the slowest memory. A common technique to reducing the number of global memory reads is *coalesced memory access*, which takes place when

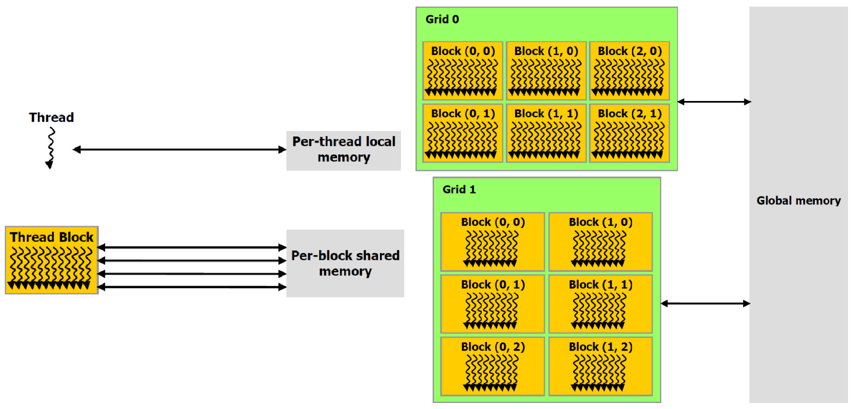


Fig. 1. GPU memory organization.

consecutive threads read consecutive memory locations allowing the hardware to coalesce the reads into a single one.

The most approach to program Nvidia GPUs is by using the CUDA toolkit, a set of developing tools and a compiler that allow programmers to develop GPU applications using a version of the `C` language extended with keywords to specify GPU code. CUDA also includes various libraries with algorithms for GPUs such as the Thrust library [5] which resembles the C++ Standard Template Library (STL) [18]. We use the functions in this library to perform sorting, prefix sums [14] and duplicate elimination as their implementation is very efficient.

3 Datalog

As is well known, Datalog is a language based on first order logic that has been used as a data model for relational databases [22, 23]. A Datalog program consist of *facts* about a subject of interest and *rules* to deduce new facts. Facts can be seen as rows in a relational database table, while rules can be used to specify complex queries. Datalog recursive rules facilitate specifying (querying for) the transitive closure of relations, which is a key concept in many applications [17].

3.1 Datalog Programs

A Datalog program consists of a finite number of facts and rules. Facts and rules are specified using atomic formulas, which consist of predicate symbols with arguments[22], e.g.:

```
FACTS                                  father relational table
                                       -----------------------
father(harry, john).                   harry    john
father(john, david).                   john     david
...                                    ...
```

```
RULE
grandfather(Z, X) :- father(Y, X), father(Z, Y).
```

Traditionally, names beginning with lower case letters are used for predicate names and constants, while names beginning with upper case letters are used for variables; numbers are considered constants. Facts consist of a single atomic formula, and their arguments are constants; facts that have the same name must also have the same arity. Rules consist of two or more atomic formulas with the first one from left to right, the rule *head*, separated from the other atomic formulas by the implication symbol ':-'; the other atomic formulas are *subgoals* separated by ',', which means a logical AND. We will refer to all the subgoals of a rule as the *body* of the rule. Rules, in order to be general, are specified with variables as arguments, but can also have constants.

3.2 Evaluation of Datalog Programs

Datalog programs can be evaluated through a top-down approach or a bottom-up approach. The top-down approach (used by the Prolog language) starts with the goal which is reduced to subgoals, or simpler problems, until a trivial problem is reached. Thus, the solution of larger problems is composed of the solutions of simpler problems until the solution of the original problem is obtained. It is tuple-oriented and hence more difficult to adapt to massive parallelism.

The bottom-up approach works by applying the rules to the given facts, thereby deriving new facts, and repeating this process with the new facts until no more facts are derivable. The query is considered only at the end, when the facts matching the query are selected. Benefits of this approach include the fact that rules can be evaluated in any order and in a highly parallel manner, based on equivalent relational operations as described shortly.

To improve the bottom-up approach, several methods have been proposed such as the magic sets transformation [8] or the subsumptive demand transformation [21]. Basically, these methods transform a set of rules and a query into a new set of rules such that the set of facts that can be inferred from the new set of rules contains only facts that would be inferred during a top-down evaluation.

3.3 Evaluation based on relational algebra operators

Evaluation of Datalog rules can be implemented using the typical relational algebra operators *select*, *join* and *projection*, as outlined in Figure 2. *Selections* are made when constants appear in the body of a rule. Then a *join* is made between two or more subgoals in the body of a rule using the variables as reference. The result of a join can be seen as a temporary subgoal that has to be joined in turn to the rest of the subgoals in the body. Finally, a *projection* is made over the variables in the head of the rule.

For recursive rules, fixed-point evaluation is used. The basic idea is to iterate through the rules in order to derive new facts, and using these new facts to derive even more new facts until no new facts are derived.

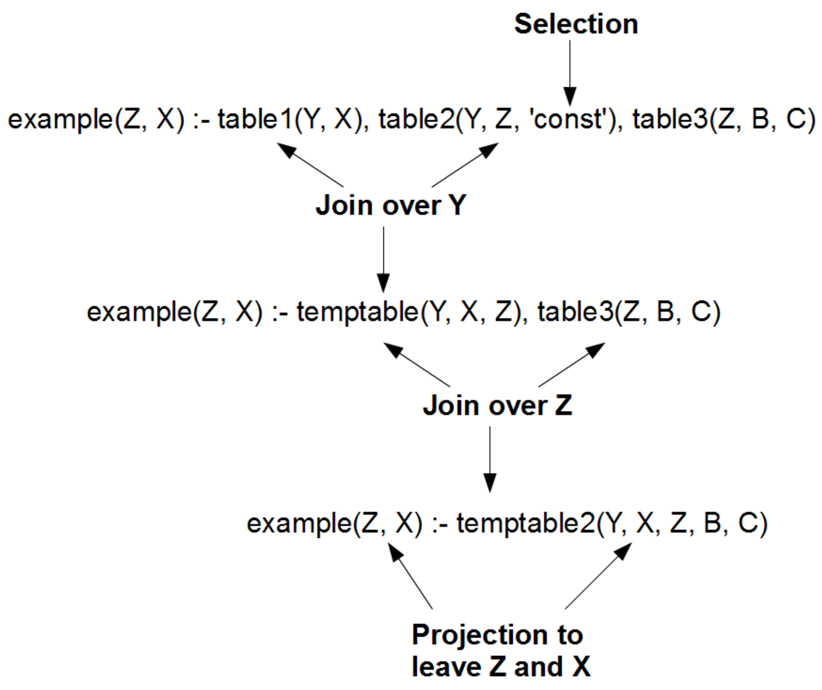

Selection
example(Z, X) :- table1(Y, X), table2(Y, Z, 'const'), table3(Z, B, C)
Join over Y
example(Z, X) :- temptable(Y, X, Z), table3(Z, B, C)
Join over Z
example(Z, X) :- temptable2(Y, X, Z, B, C)
Projection to leave Z and X

Fig. 2. Evaluation of a Datalog rule based on relational algebra operations.

4 Our Datalog Engine for GPUs

This section presents the design of our Datalog engine for GPUs.

4.1 Architecture

Figure 3 shows the main components of our Datalog engine. There is a single *host* thread that runs in the host platform (a multi-core in our evaluation). In addition to scheduling GPU work as outlined in Section 2, the host thread preprocesses the data to send to the GPU so that GPUs can process the data more efficiently, as described in Section 4.2.

The data sent to the GPU is organized into arrays that are stored in global memory. The results of rule evaluations are also stored in global memory.

Our Datalog (GPU) engine is organized into various GPU kernels. When evaluating rules, for each pair of subgoals in a rule, selection and selfjoin kernels are applied first in order to eliminate irrelevant tuples as soon as possible, followed by join and projection kernels. At the end of each rule evaluation, the duplicate elimination kernels are applied. Figure 3, right-hand side, shows these steps.

The memory management module helps to identify the most recently used data within the GPU in order to maintain it in global memory and discard sections of data that are no longer necessary.

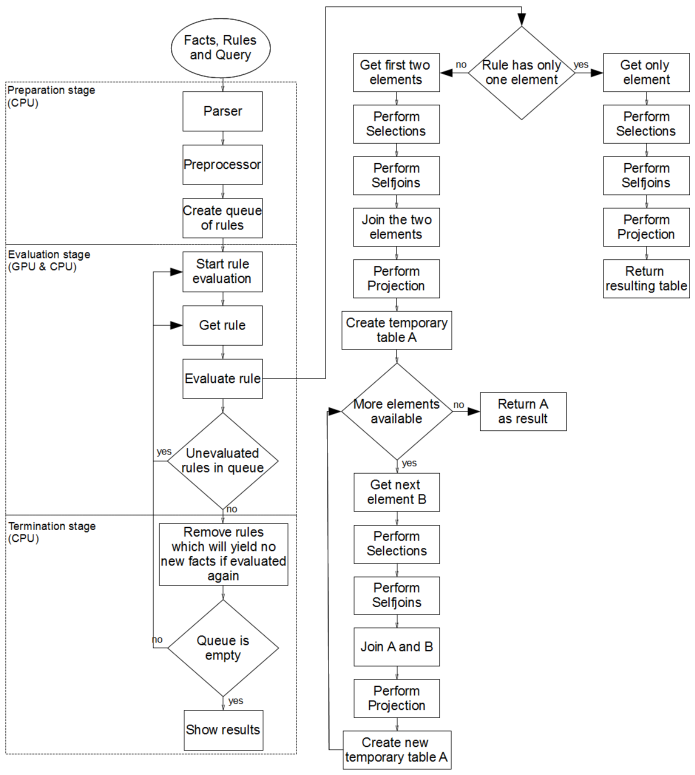


Fig. 3. GPU Datalog engine organisation.

4.2 Host Thread Tasks

Parsing. To capitalise on the GPU capacity to process numbers and to have short and constant processing time for each tuple (strings variable size entails varying processing time), we identify and use facts and rules with/as numbers, keeping their corresponding strings in a hashed dictionary. Each unique string is assigned a unique id, equal strings are assigned the same id. The GPU thus works with numbers only; the dictionary is used at the very end when the final results are to be displayed.

Preprocessing. A key factor for good performance is preprocessing data before sending it to the GPU. As mentioned before, Datalog rules are evaluated through a series of relational algebra operations: selections, joins and projections. For the evaluation of each rule, the specification of what operations to perform, including constants, variables, facts and other rules involved, is carried out in the host (as opposed to be carried out in the GPU by each kernel thread), and sent to the GPU for all GPU threads to use. Examples:

– **Selection** is specified with two values, column number to search and the constant value to search; the two values are sent as an array which can include more than one selection (more than one pair of values), as in the following example, where columns 0, 2, and 5 will be searched for the constants a, b and c, respectively:

```
fact1('a',X,'b',Y,Z,'c'). -> [0, 'a', 2, 'b', 5, 'c']
```

– **Join** is specified with two values, column number in the first relation to join and column number in the second relation to join; the two values are sent as an array which can include more than one join, as in the following example where the following columns are joined in pairs: column 1 in fact1 (X) with column 1 in fact 2, column 2 in fact1 with column 4 in fact2, and column 3 in fact1 with column 0 in fact2.

```
fact1(A,X,Y,Z), fact2(Z,X,B,C,Y). -> [1, 1, 2, 4, 3, 0]
```

Other operations are specified similarly with arrays of numbers. These arrays are stored in GPU shared memory (as opposed to global memory) because they are small and the shared memory is faster.

4.3 Memory Management

Data transfers between GPU memory and host memory are costly in all CUDA applications [1]. We designed a memory management scheme that tries to minimize the number of such transfers. Its purpose is to maintain facts and rule results in GPU memory for as long as possible so that, if they are used more than once, they may often be reused from GPU memory. To do so, we keep track of GPU memory available and GPU memory used, and maintain a list with information about each fact and rule result that is resident in GPU memory. When data (facts or rule results) is requested to be loaded into GPU memory, it is first looked up in that list. If found, its entry in the list is moved to the beginning of the list; otherwise, memory is allocated for the data and a list entry is created at the beginning of the list for it. In either case, its address in memory is returned. If allocating memory for the data requires deallocating other facts and rule results, those at the end of the list are deallocated first until enough memory is obtained — rule results are written to CPU memory before deallocating them. By so doing, most recently used fact and rule results are kept in GPU memory.

5 GPU Relational Algebra Operators

This section presents the design decisions we made for the relational algebra operations we use in our Datalog engine: select, join and project operations for GPUs. The GPU kernels that implement these operations access (read/write) tables from GPU global memory.

5.1 Selection

Selection has two main issues when designed for running in GPUs. The first issue is that the size of the result is not known beforehand, and increasing the size of the results buffer is not convenient performance-wise because it may involve reallocating its contents. The other issue is that, for efficiency, each GPU thread must know onto which global memory location it will write its result without communicating with other GPU threads.

To avoid those issues, our selection uses three different kernel executions. The first kernel marks all the rows that satisfy the selection predicate with a value one. The second kernel performs a prefix sum on the marks to determine the size of the results buffer and the location where each GPU thread must write the results. The last kernel writes the results.

5.2 Projection

Projection requires little computation, as it simply involves taking all the elements of each required column and storing them in a new memory location. While it may seem pointless to use the GPU to move memory, the higher memory bandwidth of the GPU, compared to that of the host CPU/s, and the fact that the results remain in GPU memory for further processing, make projection a suitable operation for GPU processing.

5.3 Join

Our Datalog engine uses these types of join: Single join, Multijoin and Selfjoin. A single join is used when only two columns are to be joined, e.g.: $table_1(X,Y) \bowtie table_2(Y,Z)$. A multijoin is used when more than two columns are to be joined: $table_1(X,Y) \bowtie table_2(X,Y)$. A selfjoin is used when two columns have the same variable in the same predicate: $table_1(X,X)$.

Single join. We use a modified version of the Indexed Nested Loop Join described in [16], which is as follows:

```
Make an array for each of the two columns to be joined
Sort one of them
Create a CSS-Tree for the sorted column
Search the tree to determine the join positions
Do a first join to determine the size of the result
Do a second join to write the result
```

The CSS-Tree [19] (Cache Sensitive Search Tree) is very adequate for GPUs because it can be quickly constructed in parallel and because tree traversal is performed via address arithmetic instead of the traditional memory pointers.

While the tree allows us to know the location of an element, it does not tell us how many times each element is going to be joined with other elements nor in which memory location must each thread write the result, so we must perform a “preliminary” join. This join counts the number of times each element has to be joined and returns an array that, as in the select operation, allows us to determine the size of the result and write locations when a prefix sum is applied to it. With the size and write locations known, a second join writes the results.

Multijoin. To perform a join over more than two columns, e.g., $table_1(X, Y) \bowtie table_2(X, Y)$, first we take a pair of columns say (X, X) to create and search on the CSS-Tree as described in the single join algorithm. Then, as we are doing the first join, we also check if the values of the remaining columns are equal (in our example we check if $Y = Y$) and discard the rows that do not comply.

Selfjoin. The selfjoin operation is very similar to the selection operation. The main difference is that instead of each thread checking a constant value on its corresponding row, it checks if the values of the columns affected by the self join match.

5.4 Optimisations

Our relational algebra operations make use of the following optimisations in order to improve performance. The purpose of these optimisations is to reduce memory use and in principle processing time — the cost of the optimisations themselves is not yet evaluated.

Duplicate Elimination. Duplicate elimination uses the *unique* function of the Thrust library. It takes an array and a function to compare two elements in the array, and returns the same array with the unique elements at the beginning. We apply duplicate elimination to the result of each rule: when a rule is finished, its result is sorted and the *unique* function is applied.

Optimising projections. Running a *projection* at the end of each join, as described below, allows us to discard unnecessary columns earlier in the computation of a rule. For example, consider the following rule:

```
rule1(Y, W) :- fact1(X, Y), fact2(Y, Z), fact3(Z,W).
```

The evaluation of the first join, $fact_1 \bowtie_Y fact_2$, generates a temporary table with columns (X, Y, Y, Z), not all of which are necessary. One of the two Y columns can be discarded; and column X can also be discarded because it is not used again in the body nor in the head of the rule.

Fusing operations. Fusing operations consists of applying two or more operations to a data set in a single read of the data set, as opposed to applying only one operation, which involves as many reads of the data set as the number of operations to be applied. We fuse the following operations.

- All selections required by constant arguments in a subgoal of a rule are performed at the same time.
- All selfjoins are also performed at the same time.
- Join and projection are always performed together at the same time.

To illustrate these fusings consider the following rule:

```
rule1(X,Z):- fact1(X,'const1',Y,'const2'),fact2(Y,'const3',Y,Z,Z).
```

This rule will be evaluated as follows. $fact_1$ is processed first: the selections required by $const_1$ and $const_2$ are performed at the same time — $fact_1$ does not require selfjoins. $fact_2$ is processed second: a) the selection required by $const_3$ is performed, and then b) the selfjoins between Ys and Zs are performed at the same time. Finally, a join is performed between the third column of $fact_1$ and the first column of $fact_2$ and, at the same time, a projection is made (as required by the arguments in the rule head) to leave only the first column of $fact_1$ and the fourth column of $fact_2$.

6 Experimental Evaluation

This section describes our platform, applications and experiments to evaluate the performance of our Datalog engine. We are at this stage interested in the performance benefit of using GPUs for the evaluation of Datalog queries, as opposed to using a CPU only. Hence we present results that show the performance of 3 Datalog queries running on our engine compared to the performance of the same queries running on a single CPU in the host platform. (We plan to compare our Datalog engine to similar GPU work discussed in Section 7, Related Work, in further work).

On a single CPU in the host platform, the 3 queries were run with the Prolog systems YAP [9] and XSB [20], and the Datalog system from the MITRE Corporation [3]. As the 3 queries showed the best performance with YAP, our results plots below show the performance of the queries with YAP and with our Datalog engine only. YAP is a high-performance Prolog compiler developed at LIACC/Universidade do Porto and at COPPE Sistemas/UFRJ. Its Prolog engine is based on the WAM (Warren Abstract Machine) [9], extended with some optimizations to improve performance. The queries were run on this platform:

Hardware. *Host platform*: Intel Core 2 Quad CPU Q9400 2.66GHz (4 cores in total), Kingston RAM DDR2 6GB 800 MHz. *GPU platform*: Fermi GeForce GTX 580 - 512 cores - 1536 MB GDDR5 memory.

Software. Ubuntu 12.04.1 LTS 64bits. CUDA 5.0 Production Release, gcc 4.5, g++ 4.5. YAP 6.3.3 Development Version, Datalog 2.4, XSB 3.4.0.

For each query, in each subsection below, we describe first the query, and then discuss the results. Our results show the evaluation of each query once all data has been preprocessed and in CPU memory, i.e.: I/O, parsing and preprocessing costs are not included in the evaluation.

6.1 Join over four big tables.

Four tables, all with the same number of rows filled with random numbers, are joined together to test all the different operations of our Datalog engine. The rule and query used are:

```
join(X,Z) :- table1(X), table2(X,4,Y), table3(Y,Z,Z), table4(Y,Z).
join(X,Z)?
```

Fig. 4 shows the performance of the join with YAP and our engine, in both normal and logarithmic scales to better appreciate details. Our engine is clearly faster, roughly 200 times. Both YAP and our engine take proportionally more time as the size of the tables grows. Our engine took just above two seconds to process tables with five million rows each, while YAP took about two minutes process tables with one million rows each.

The time taken by each operation was as follows: joins were the most costly operations with the Multijoin alone taking more than 70% of the total time; the duplicate elimination and the sorting operations were also time consuming but within acceptable values; prefix sums and selections were the fastest operations.

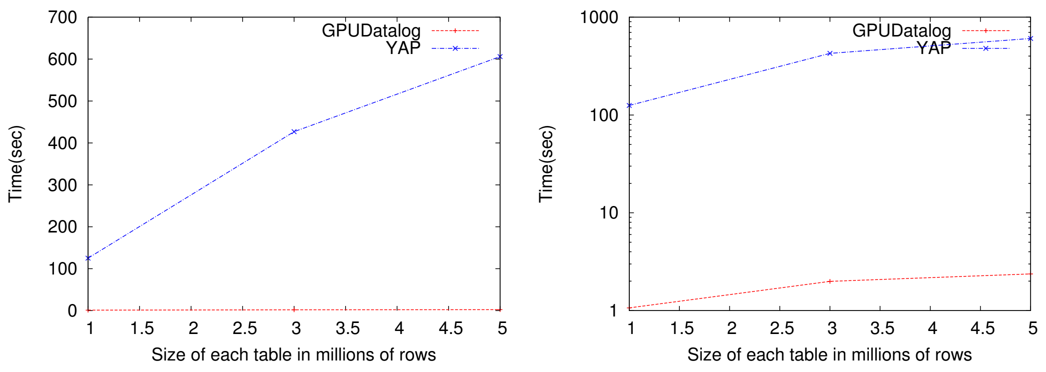


Fig. 4. Performance of join over four big tables (log. scale on the right).

6.2 Transitive closure of a graph.

The transitive closure of a graph (TCG) is a recursive query. We use a table with two columns filled with random numbers that represent the edges of a graph [12]. The idea is to find all the nodes that can be reached if we start from a particular node. This query is very demanding because recursive queries involve various iterations over the relational operations that solve the query. The rules and the query are:

```
path(X,Y) :- edge(X,Y).
path(X,Z) :- edge(X,Y), path(Y,Z).
path(X,Y)?
```

Fig. 5 shows the performance of TCG with YAP and our engine. Similar observations can be made as for the previous experiment. Our engine is 40x times faster than YAP for TCG. Our engine took less than a second to process a table of 10 million rows while YAP took 3.5 seconds to process 1 million rows.

For the first few iterations, duplicate elimination was the most costly operation of each iteration, and the join second but closely. As the number of rows to process in each iteration decreased, the join became by far the most costly operation.

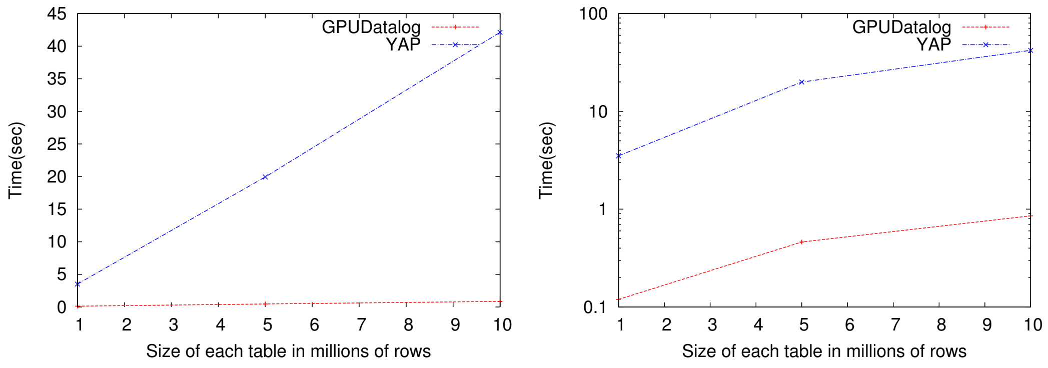


Fig. 5. Performance of transitive closure of a graph (log. scale on the right).

6.3 Same-Generation program.

This is a well-known program in the Datalog literature, and there are various versions. We use the version described in [6]. Because of the initial tables and the way the rules are written, it generates lots of new tuples in each iteration. The three required tables are created with the following equations:

$$up = \{(a, b_i)|i\epsilon[1, n]\} \cup \{(b_i, c_j)|i, j\epsilon[1, n]\}. \quad (1)$$

$$flat = \{(c_i, d_j)|i, j\epsilon[1, n]\}. \quad (2)$$

$$down = \{(d_i, e_j)|i, j\epsilon[1, n]\} \cup \{(e_i, f)|i\epsilon[1, n]\}. \quad (3)$$

Where a and f are two known numbers and b, c, d and e are series of n random numbers. The rules and query are as follows:

```
sg(X,Y) :- flat(X,Y).
sg(X,Y) :- up(X,X1), sg(X1,Y1), down(Y1,Y).
sg(a,Y)?
```

The results show (Fig. 6) very little gain in performance, with our engine taking an average of 827ms and YAP 1600ms for $n = 75$. Furthermore, our engine cannot process this application for n > 90 due to lack of memory.

The analysis of each operation revealed that duplicate elimination takes more than 80% of the total time and is also the cause of the memory problem. The reason of this behaviour is that the join creates far too many new tuples, but most of these tuples are duplicates (as an example, for $n = 75$ the first join creates some 30 million rows and, after duplicate elimination, less than 10 thousand rows remain).

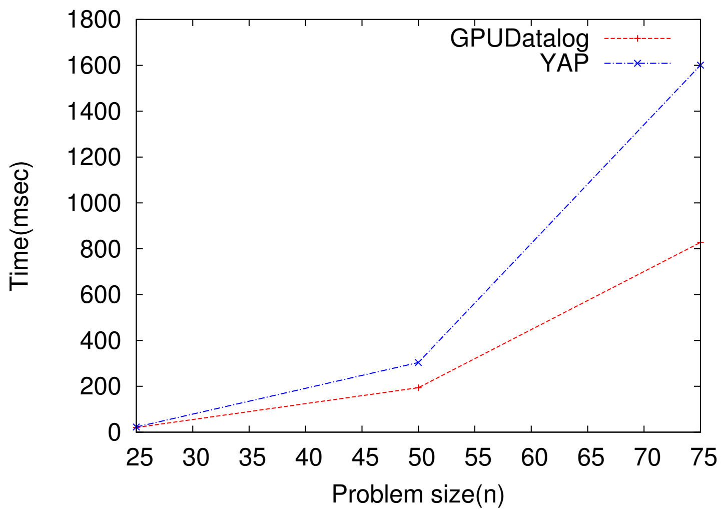


Fig. 6. Same-Generation program.

7 Related Work

He *et. al* [15] have designed, implemented and evaluated GDB, an in-memory relational query coprocessing system for execution on both CPUs and GPUs. GDB consists of various primitive operations (scan, sort, prefix sum, etc.) and relational algebra operators built upon those primitives.

We modified the Indexed Nested Loop Join (INLJ) of GDB for our single join and multijoin, so that more than two columns can be joined, and a projection performed, at the same time. Their selection operation and ours are similar too; ours takes advantage of GPU shared memory and uses the Prefix Sum of the Thrust Library. Our projection is fused into the join and does not perform duplicate elimination, while they do not use fusion at all.

Diamos *et. al* [10, 11, 24–26] have also developed relational operators for GPUs, which are being integrated into the Red Fox [4] platform, an extended Datalog developed by LogicBlox [13] for multiple-GPU systems [26]. Their relational operators partition and process data in blocks using algorithmic skeletons. Their join algorithm, compared to that of GDB, shows 1.69 performance improvement [11]. Their selection performs two prefix sums and the result is written and then moved to eliminate gaps; our selection performs only one prefix sum and writes the result once. They discuss kernel fusion and fission in [25]. We applied fusion (e.g., simultaneous selections, selection then join, etc.) at source code, while they implement it automatically through the compiler. Kernel fission, the parallel execution of kernels and memory transfers, is not yet adopted in our work. We plan to compare our relational operators to those of GDB and Red Fox, and extending them for multiple-GPU systems too.

8 Conclusions

Our Datalog engine for GPUs evaluates queries based on the relational operators select, join and projection. Our evaluation using 3 queries shows a dramatic performance improvement for two of the queries, up to 200 times for one of them. The performance of the same-generation problem is improved twice only, but we believe it can be improved more. We will work on the following extensions to our engine.

– Extended syntax to accept built-in predicates and negation [6].
– Evaluation based on tabling [21] or magic sets [8] methods.
– Managing tables larger than the total amount of GPU memory.
– Mixed processing of rules both on the GPU and on the host multicore.
– Improved join operations to eliminate duplicates before writing final results.

Acknowledgements

Carlos A. thanks the University of Porto and its Departmento de Ciência de Computadores, CINVESTAV-IPN (Centre for Research and Postgraduate Studies of the National Polytechnic Institute) and CONACyT (Mexican Council of Science and Technology) for their support throughout his postgraduate studies. ICD and VSC were partially supported by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project LEAP (FCOMP-01-0124-FEDER-015008).

References

1. CUDA C Best Practices Guide. `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`.
2. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.
3. Datalog by the MITRE Corporation. `http://datalog.sourceforge.net/`.
4. Red Fox: A Compilation Environment for Data Warehousing. `http://gpuocelot.gatech.edu/projects/red-fox-a-compilation-environment-for-data-warehousing/`.
5. Thrust: A Parallel Template Library. `http://thrust.github.io/`.
6. Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
7. Foto N. Afrati, Vinayak R. Borkar, Michael J. Carey, Neoklis Polyzotis, and Jeffrey D. Ullman. Cluster computing, recursion and datalog. In *Datalog*, pages 120–144, 2010.
8. Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *J. Log. Program.*, 10(3&4):255–299, 1991.
9. Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. *TPLP*, 12(1-2):5–34, 2012.

10. Gregory Diamos, Haicheng Wu, Jin Wang, Ashwin Lele, and Sudhaka Yalamanchili. Relational algorithms for multi-bulk-synchronous processors. In *PPoPP-13: The 18th Symposium on Principles and Practice of Parallel Programming*, 2013.
11. Gregory F. Diamos, Haicheng Wu, Ashwin Lele, Jin Wang, and Sudhakar Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. Technical report, Georgia Institute of Technology, 2012.
12. Guozhu Dong, Jianwen Su, and Rodney W. Topor. Nonrecursive incremental evaluation of datalog queries. *Ann. Math. Artif. Intell.*, 14(2-4):187–223, 1995.
13. Todd J. Green, Molham Aref, and Grigoris Karvounarakis. Logicblox, platform and language: a tutorial. In *Proceedings of the Second international conference on Datalog in Academia and Industry*, Datalog 2.0'12, pages 1–8, Berlin, Heidelberg, 2012. Springer-Verlag.
14. Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.
15. Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst. (TODS)*, 34(4), 2009.
16. Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. In *SIGMOD Conference*, pages 511–524, 2008.
17. Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and emerging applications: an interactive tutorial. In *SIGMOD Conference*, pages 1213–1216, 2011.
18. David R. Musser, Gilmer J. Derge, and Atul Saini. *STL tutorial and reference guide: C++ programming with the standard template library, 2nd Ed.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
19. Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
20. Terrance Swift and David Scott Warren. Xsb: Extending prolog with tabled logic programming. *TPLP*, 12(1-2):157–187, 2012.
21. K. Tuncay Tekle and Yanhong A. Liu. More efficient datalog queries: subsumptive tabling beats magic sets. In *SIGMOD Conference*, pages 661–672, 2011.
22. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
23. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II.* Computer Science Press, 1989.
24. Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *Micro-12: 45th International Symposium on Microarchitecture*, 2012.
25. Haicheng Wu, Gregory Diamos, Jin Wang, Srihari Cadambi, Sudhakar Yalamanchili, and Srimat Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *IPDPSW-12: IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, 2012.
26. Jeffrey Young, Haicheng Wu, and Sudhakar Yalamanchili. Satisfying data-intensive queries using GPU clusters. In *HPCDB-12: 2nd Annual Workshop on High-Performance Computing meets Databases*, 2012.

Heuristic Search Over Program Transformations

Claus Zinn

Department of Computer Science
University of Konstanz
`claus.zinn@uni-konstanz.de`

Abstract. In prior work, we have developed a method for the automatic reconstruction of buggy Prolog programs from correct programs to model learners' incorrect reasoning in a tutoring context. The method combines an innovative variant of algorithmic debugging with program transformations. Algorithmic debugging is used to indicate a learner's error and its type; this informs a program transformation that "repairs" the expert program into a buggy variant that is closer at replicating a learner's behaviour. In this paper, we improve our method by using heuristic search. To search the space of program transformations, we estimate the distance between programs. Instead of only returning the first irreducible disagreement between program and Oracle, the algorithmic debugger now traverses the entire program. In the process, all irreducible agreements and disagreements are counted to compute the distance metrics, which also includes the cost of transformations. Overall, the heuristic approach is a significant improvement to our existing blind method.

1 Introduction

Typically, programs have bugs. We are interested in runtime bugs where the program terminates with output that the programmer judges incorrect. In these cases, Shapiro's algorithmic debugging technique can be used to pinpoint the location of the error. A dialogue between the debugger and the programmer unfolds until the meta-interpretation of the program reaches a statement that captures the cause of disagreement between the program's actual behaviour and the programmer's intent of how the program should behave. Once the bug has been located, it is the programmer's task to repair the program, and then, to start another test-debugging-repair cycle. Let us make the following assumption: there exists an Oracle that relieves the programmer from answering any of the questions during the debugging cycle; the Oracle "knows" the programmer's intent for each and every piece of code. With the mechanisation of the Oracle to locate the program's bugs, we now seek to automate the programmer's task to repair the bug, and thus, to fully automate the test-debug-repair cycle.

In the tutoring context, Oracles can be mechanised: for a given domain of instruction, there is always a reference model that defines expert problem solving behaviour. Moreover, a learner's problem solving behaviour is judged with regard to this model; a learner commits a mistake whenever the learner deviates from the expert problem solving path. Algorithmic debugging can be used to identify the location of learners' erroneous behaviour. For this, we have to turn Shapiro's method on its head: we take the expert program to take the role of the buggy program, and the learner to take the role of the programmer, that is, the Oracle. As in the traditional method, any disagreement between the two parties indicates the location of the bug. Moreover, we can relieve the learner from answering Oracle questions. Answers to all questions can be reconstructed from the learner's answer to a given problem, using the expert model [9].

With the ability to locate a learner's error, we now seek to "repair" the expert program (assumed buggy) in such a way that is reproduces the learner's erroneous (assumed expert) behaviour. The resulting program acts as symbolic artifact of a deep diagnosis of a learner's problem solving process; it can be used to inform effective remediation, helping learners to realize and correct their mistakes. Ideally, repair operators shall mirror typical learner errors. This is feasible indeed. There is a small set of error types, and many of them can be formally described in a domain-independent manner.

With the identification of an error's location, and a small, effective set of mutation operators for program repair, we strive to fully automate the test-debug-repair cycle in the tutoring context. Our approach is applicable for a wider context, given the specification of an ideal program and a theory of error.

Main contributions. To address an important issue in intelligent tutoring, the deep diagnosis of learner input, we cast the problem of automatically deriving one (erroneous) program from another (expert) program as a heuristic search problem. We define a metric that quantifies the distance of two given programs with regard to an input/output pair. We define a number of domain-independent code perturbation operators whose execution transforms a given program into its mutated variant. Most mutation operators encode typical actions that learners perform when encountering an impasse during problem solving. We show the effectiveness of our approach for the most frequent learner bugs in the domain of multi-column subtraction. Erroneous procedures are automatically derived to reproduce these errors. This work extends and generalises our previous work in this area [9], [10] with regard to the heuristic search approach, which is novel.

Overview. Sect. 2 gives a very brief review on student errors in tutoring. It presents multi-column subtraction as domain of instruction and gives an encoding of the expert model in Prolog. For each of the top-eight learner errors in this domain, we demonstrate how the expert model needs to be perturbated to reproduce them. We show that most perturbations are based on a small but effective set of mutation operators. Also, we briefly review our existing method of error diagnosis in the tutoring context. In Sect. 3, we improve and generalise our method. The problem of deriving one program from another is cast in terms of a heuristic search problem. We introduce a distance metrics between programs that is based on algorithmic debugging, and use a best-first search algorithm to illustrate and evaluate the effectiveness of our approach. Sect. 4 discusses our approach and relates it to existing work. Sect. 5 concludes with future work.

2 Background

2.1 Human Error in Tutoring

When learning something new, one is bound to make mistakes. Effective teaching depends on deep cognitive analyses to diagnose learners' problem solving path, and subsequently to repair the incorrect parts. Good teacher are thus capable to reconstruct students' erroneous procedures and use this information to inform their remediation. In the area of elementary school mathematics, our chosen tutoring domain, the seminal works of Brown and Burton [1, 2], O'Shea and Young [8], and VanLehn [7], among others, extensively studied the subtraction errors of large populations of pupils. Their research included a computational account of errors by manually constructing cognitive models that reproduced learners' most frequent errors. The main insight of this research is that student errors are seldom random. There are two main causes. The first cause is that student errors may result from *correctly* executing an erroneous procedure; for some reasons, the erroneous rather than the expert procedure has been acquired. The second cause is based on VanLehn's theory of *impasses* and *repairs*. Following VanLehn, learners "know" the correct procedure, but face difficulties executing it. They "treat the impasse as a problem, solve it, and continue executing the procedure" [7, p. 42]. The repair strategies to address an impasse are known to be common across student populations and domains. Typical repairs include executing only the steps known to the learner and to skip all other steps, or to adapt the situation to prevent the impasse from happening.

2.2 Expert Model for Multi-Column Subtraction

Fig. 1 depicts the entire cognitive model for multi-column subtraction using the decomposition method. The Prolog code represents a subtraction problem as a list of column terms `(M, S, R)` consisting of a minuend `M`, a subtrahend `S`, and a result cell `R`. The main predicate `subtract/2` determines the number of columns and passes its arguments to `mc_subtract/3`.[1] This predicate processes columns from right to left until all columns have been processed and the recursion terminates. The predicate `process_column/3` receives a partial sum, and processes its right-most column (extracted by `last/2`). There are two cases. Either the column's subtrahend is larger than its minuend, when a borrowing operation is required, or the subtrahend is not larger than the minuend, in which case we can subtract the former from the latter (calling `take_difference/4`). In the first case, we add ten to the minuend (`add_ten_to_minuend/3`) by borrowing from the left (calling `decrement/3`). The decrement operation also consists of two clauses, with the second clause being the easier case. Here, the minuend of the column left to the current column is not zero, so we simply reduce the minuend by one. If the minuend is zero, we need to borrow again, and hence `decrement/3` is called recursively. When we return from recursion, we first add ten to the minuend, and then reduce it by one.

[1] The argument `CurrentColumn` is passed onto most other predicates; it is only used to help automating the Oracle.

```
subtract(PartialSum, Sum) ←
        length(PartialSum, LSum),
        mc_subtract(LSum, PartialSum, Sum).

mc_subtract(_, [], []).
mc_subtract(CurrentColumn, Sum, NewSum) ←
        process_column(CurrentColumn, Sum, Sum1),
        shift_left(CurrentColumn, Sum1, Sum2, ProcessedColumn),
        CurrentColumn1 is CurrentColumn − 1,
        mc_subtract(CurrentColumn1, Sum2, SumFinal),
        append(SumFinal, [ProcessedColumn], NewSum).

process_column(CurrentColumn, Sum, NewSum) ←
        last(Sum, LastColumn), allbutlast(Sum, RestSum),
        minuend(LastColumn, M), subtrahend(LastColumn, S),
        S > M, !,
        add_ten_to_minuend(CurrentColumn, M, M10),
        CurrentColumn1 is CurrentColumn − 1,
        decrement(CurrentColumn1, RestSum, NewRestSum),
        take_difference(CurrentColumn, M10, S, R),
        append(NewRestSum, [(M10, S, R)], NewSum).

process_column(CurrentColumn, Sum, NewSum) ←
        last(Sum, LastColumn), allbutlast(Sum, RestSum),
        minuend(LastColumn, M), subtrahend(LastColumn, S),
        % S =< M,
        take_difference(CurrentColumn, M, S, R),
        append(RestSum, [(M, S, R)], NewSum).

shift_left( _CurrentColumn, SumList, RestSumList, Item ) ←
        allbutlast(SumList, RestSumList), last(SumList, Item).

decrement(CurrentColumn, Sum, NewSum ) ←
        irreducible,
        last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
        M == 0, !,
        CurrentColumn1 is CurrentColumn − 1,
        decrement(CurrentColumn1, RestSum, NewRestSum ),
        NM is M + 10,
        NM1 is NM − 1,
        append( NewRestSum, [(NM1, S, R)], NewSum),

decrement(CurrentColumn, Sum, NewSum) ←
        irreducible,
        last( Sum, (M, S, R) ), allbutlast( Sum, RestSum),
        % \+ (M == 0),
        M1 is M − 1,
        append( RestSum, [(M1, S, R)], NewSum ).

add_ten_to_minuend( _CC, M, M10) ← irreducible, M10 is M + 10.
take_difference(_CC, M, S, R) ← irreducible, R is M − S.

minuend( (M, _S, _R), M).
subtrahend( (_M, S, _R), S).

allbutlast([], []).
allbutlast([_H], []).
allbutlast([H1|[H2|T]], [H1|T1]) ← allbutlast([H2|T], T1).

irreducible.
```

Fig. 1. The Decomposition Method for Subtraction in Prolog

2.3 Buggy Sets in Multi-column Subtraction

$$\begin{array}{cccc} & & 9 & \\ & 3 & \cancel{10} & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 2 & 0 & 2 \end{array}$$

(a) correct solution

$$\begin{array}{cccc} & 4 & 0 & 1 \\ - & 1 & 9 & 9 \\ \hline = & 3 & 9 & 8 \end{array}$$

(b) smaller-from-larger

$$\begin{array}{cccc} & 3 & 10 & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 2 & 1 & 2 \end{array}$$

(c) stops-borrow-at-zero

$$\begin{array}{cccc} & 2 & & \\ & \cancel{3} & 10 & 11 \\ & \cancel{4} & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 1 & 1 & 2 \end{array}$$

(d) borrow-across-zero

$$\begin{array}{cccc} & & 9 & 11 \\ & 4 & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 0 & 2 \end{array}$$

(e) borrow-from-zero

$$\begin{array}{cccc} & & 10 & 11 \\ & 4 & \cancel{0} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 1 & 2 \end{array}$$

(f) borrow-no-decrement

$$\begin{array}{cccc} & & & 11 \\ & 4 & 0 & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 3 & 9 & 2 \end{array}$$

(g) stops-borrow-at-zero diff-0-N=N

$$\begin{array}{cccc} & 2 & & \\ & \cancel{3} & 11 & 11 \\ & \cancel{4} & \cancel{1} & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 1 & 2 & 2 \end{array}$$

(h) always-borrow-left

$$\begin{array}{cccc} & 3 & & 11 \\ & \cancel{4} & 0 & \cancel{1} \\ - & 1 & 9 & 9 \\ \hline = & 1 & 9 & 2 \end{array}$$

(i) borrow-across-zero diff-0-N=N

Fig. 2. A correct solution, and the top-eight bugs sets, see [7, p. 195].

Fig. 2(a) depicts the correct solution to the subtraction problem 401 − 199, the figures 2(b)–2(i) show how the top-eight bug sets from the DEBUGGY study [7, p. 195, p. 235] manifest themselves in the same task. All erroneous answers are rooted in learners' difficulty to borrow: the errors in Fig. 2(b) and 2(f) result from the learners' more general impasse "does not know how to borrow", and the errors in Fig. 2(c)–Fig. 2(e) results from the learners' more specific impasse "does not know how to borrow from zero". All other errors, but Fig. 2(h), are variations of the previous error types. Fig. 2(h) is better explained by the incorrect acquisition of knowledge rather than within the impasse-repair theory.

We now describe how the expert procedure given in Fig. 1 needs to be "repaired" to reproduce each of the top-eight bugs.

smaller-from-larger: *the student does not borrow, but in each column subtracts the smaller digit from the larger one [7, p. 228].* The impasse "learner does not know how to borrow" is overcome by not letting borrowing to happen. The expert model is perturbated at the level of `process_column/3`. In its first clause, we delete the calls to `add_ten_to_minuend/3` (line 15) and `decrement/3` (line 17). As a consequence, we replace all remaining occurences of `M10` and `NewRestSum` with `M` and `RestSum`, respectively. Moreover, we swap the arguments for `M` and `S` when taking differences (line 18).

borrow-no-decrement: *when borrowing, the student adds ten correctly, but does not change any column to the left [7, p. 223].* The learner addresses the impasse "does not know how to borrow" with a partial skipping of steps. In the first clause of `process_column/3`, the subgoal `decrement/3` (line 17) is deleted; the remaining occurrence of `NewRestSum` is subsequently by `RestSum` (line 19).

stops-borrow-at-zero: *instead of borrowing across a zero, the student adds ten to the column he is doing, but does not change any column to the left [7, p. 229].* The impasse "learner does not know how to borrow from zero" is overcome by not performing *complete* borrowing when the minuend in question is zero. The recursive call to `decrement/3` (line 33) and the goals producing `NM1` and `NM` (lines 34, 35) are removed, and the remaining occurrence of `NM1` replaced by `M` (line 36).

borrow-across-zero: *when borrowing across a 0, the student skips over the* 0 *to borrow from the next column. If this causes him to have to borrow twice, he decrements the same number both times [7, p. 114, p.221].* Same impasse, different repair. The clauses that produce `NM1` and `NM` (lines 34, 35) are removed; the remaining occurrence of `NM1` in `append/3` replaced by `M` (line 36).

borrow-from-zero: *instead of borrowing across a zero, the student changes the zero to nine, but does not continue borrowing from the column to the left [7, p. 223].* Same impasse, yet another repair: the assignments `NM` and `NM1` stay in place, but the recursive call to `decrement/3` (line 33) is deleted; the occurrence of `NewRestSum` is replaced by `RestSum` (line 36).

stops-borrow-at-zero diff0-N=N: *when the student encounters a column of the form $0 - N$, he does not borrow, but instead writes N as the answer, possibly combined with* `stops-borrow-at-zero`*.* For `diff-0-N=N`, we shadow the existing clause for taking differences with `take_difference( M, S, R):- M == 0, R = S.` To ensure that no borrowing operation is performed in case the minuend is zero, the first clause of `process_column/3` is modified. The constraint `S` > `M` (line 14) is complemented with \+ `(M == 0)`; line 23 is changed to `(S =< M) ; (M == 0)`.

always-borrow-left: *the student borrows from the left-most digit instead of borrowing from the digit immediately to the left [7, p. 225].* This error is best explained by the incorrect acquisition of knowledge rather than within the impasse-repair theory. To reproduce it, we shadow the existing clauses for `decrement/3` with `decrement([(M,S,R)|OtherC], [(M1,S,R)|OtherC]) :- !, M1 is M - 1.`

borrow-across-zero diff-0-N=N: *see above.* With both errors already been dealt with, we combine the respective perturbations to reproduce this error.

Summary. All error types except `always-borrow-left` require the deletion of one or more subgoals, in which case a post-processing phase must replace all occurrences of the subgoals' output argument with their respective input argument.

The error `diff0-N=N` co-occurs with `stops-borrow-at-zero` and `borrow-across-zero`. It requires a new clause for `take_difference/4`, shadowing the existing clause when the minuend in question is zero. Also, two constraints need to be added in each of the two definitions for `process_column/3`. Note that the perturbations for `diff0-N=N` are not in conflict with the perturbations for the other errors. For `smaller-than-larger`, the swapping of arguments was necessary. To reproduce the error type `always-borrow-left`, we shadowed the existing clauses for `decrement/3` with a new clause. While the top five errors can be reproduced by syntactic means, the last three errors seem to require constructive action.

2.4 Existing Method

In [10], we have presented a method that interleaves algorithmic debugging with program transformations for the automatic reconstruction of learners' erroneous procedure, see Fig. 3. The function `ReconstructErroneousProcedure/3` is recursively called until a program is obtained that reproduces learner behaviour, in which case there are no further disagreements. Note that multiple perturbations may be required to reproduce single bugs, and that multiple bugs are tackled by iterative applications of algorithmic debugging and code perturbation.

```
1: function ReconstructErroneousProcedure($Program, Problem, Solution$)
2:     ($Disagr, Cause$) ← $AlgorithmicDebugging(Program, Problem, Solution)$
3:     if $Disagr = nil$ then
4:         return $Program$
5:     else
6:         $NewProgram$ ← Perturbation($Program, Disagr, Cause$)
7:         ReconstructErroneousProcedure($NewProgram, Problem, Solution$)
8:     end if
9: end function

10: function Perturbation($Program, Clause, Cause$)
11:     return chooseOneOf(Cause)
12:         DeleteCallToClause($Program, Clause$)
13:         DeleteSubgoalsOfClause($Program, Clause$)
14:         SwapClauseArguments($Program, Clause$)
15:         ShadowClause($Program, Clause$)
16: end function
```

Fig. 3. Pseudo-code: compute variant of *Program* to reproduce a learner's *Solution*.

The irreducible disagreement resulting from the algorithmic debugging phase locates the code pieces where perturbations must take place; its cause determines the kind of perturbation. The function `Perturbation/3` can invoke various kinds of transformations: the deletion of a call to the clause in question, or the deletion of one of its subgoals, or the shadowing of the clause in question by a more specialized instance, or the swapping of the clause' arguments. These perturbations reflect the repair strategies learners use when encountering an impasse.

Our algorithm for clause call deletion, *e.g.*, traverses a given program until it identifies a clause whose body contains the clause in question, `Clause`; once identified, it removes `Clause` from the body and replaces all occurrences of its output argument by its input argument in the adjacent subgoals as well as in the clause's head, if present. Then, the modified program is returned.

There are many choice points as an action can materialise in many different ways. Our original method uses Prolog's built-in depth-first mechanism to *blindly* search the space of program transformations. Our new method uses a heuristics to make *informed* decisions during search.

3 Heuristic Search over Program Transformations

The problem of automatically reconstructing a Prolog program to model a learner's incorrect reasoning can be cast as a heuristic search problem. The initial state holds a Prolog program that solves arbitrary multi-column subtraction tasks in a expert manner. The goal state holds the program's perturbated variant whose execution reproduces the learner's erroneous behaviour. For each state s, a successor state s' can be obtained by the application of a single perturbation operator op_i. We seek a sequence of perturbation actions $op_1, op_2, ...op_n$ to define a path between start and goal state, with minimal costs.

3.1 Heuristic Function

Best-first search depends on a heuristic function to evaluate a node's distance to the goal node. Our method [10] can be extended to serve as a heuristic function. A heuristic score can be obtained by counting the number of agreements until the first irreducible disagreement is found; however, when errors occur early in the problem solving process, this simple scoring performs poorly. A better score is yielded by the following: the debugger is modified to always traverse the entire program and count all irreducible agreements and disagreements during traversal.

Fig. 4 depicts the algorithmic debugger in pseudo-code; it extends a simple meta-interpreter. Before start, both counters are initialised, and the references set for `Goal`, `Problem`, `Solution` to hold the top-level goal, the task to be solved and the learner's `Solution` to the task, respectively. There are four main cases. The meta-interpreter encounters either (i) a conjunction of goals, (ii) a goal that is a system predicate, (iii) a goal that does not need to be inspected, or (iv) a goal that needs to be inspected. For (i), algorithmic debugging is called recursively on each of the goals of the conjunctions; for (ii), the goal is called; and for (iii), we obtain the goal's body and ask the meta-interpreter to inspect it. The interesting aspect is case (iv) for goals marked relevant. Here, the goal is evaluated by both the expert program (using `call/1`) and the Oracle. The Oracle retrieves the learner's solution for the given `Problem` and reconstructs

```
1: $NumberAgreements \leftarrow 0$, $NumberDisagreements \leftarrow 0$
2: $Problem \leftarrow$ current task to be solved, $Solution \leftarrow$ learner input to task
3: $Goal \leftarrow$ top-clause of routine, with input $Problem$ and output $Solution$
4: procedure ALGORITHMICDEBUGGING($Goal$)
5:     if $Goal$ is conjunction of goals ($Goal1, Goal2$) then
6:         $\leftarrow$ algorithmicDebugging($Goal1$)
7:         $\leftarrow$ algorithmicDebugging($Goal2$)
8:     end if
9:     if $Goal$ is system predicate then
10:        $\leftarrow$ call($Goal$)
11:    end if
12:    if $Goal$ is not on the list of goals to be discussed with learners then
13:        $Body \leftarrow$ getClauseSubgoals($Goal$)
14:        $\leftarrow$ algorithmicDebugging($Body$)
15:    end if
16:    if $Goal$ is on the list of goals to be discussed with learners then
17:        $SystemResult \leftarrow$ call($Goal$)
18:        $OracleResult \leftarrow$ oracle($Goal$)
19:        if results agree on $Goal$ then
20:            $Weight \leftarrow$ computeWeight($Goal$)    ▷ compute # of skills in proof tree
21:            $NumberAgreements \leftarrow NumberAgreements + Weight$
22:        else
23:            if $Goal$ is leaf node (or marked as irreducible) then
24:                $NumberDisagreements \leftarrow NumberDisagreements + 1$
25:            else
26:                $Body \leftarrow$ getClauseSubgoals($Goal$)
27:                $\leftarrow$ algorithmicDebugging($Body$)
28:            end if
29:        end if
30:    end if
31: end procedure
32: $Score \leftarrow NumberDisagreements - NumberAgreements$
```

Fig. 4. Pseudo-code: Top-Down Traversal, Keeping Track of (Dis-)agreements.

from it the learner's answer to the goal under discussion. Now, there are two cases. If system and learner agree on the goal's result, then the goal's weight is determined and added to the number of agreements; if they disagree, the goal must be inspected further to identify the exact location of the disagreement. If the goal is a leaf node, the irreducible disagreement has been identified and the disagreement counter is incremented by one; otherwise, the goal's body is retrieved and subjected to algorithmic debugging. The heuristic score is obtained by subtracting the number of agreements from the number of disagreements.

3.2 Best-First Search: Guiding Program Transformations with A^*

Typically, a search method maintains two lists of states: an *open list* of all states still to be investigated for the goal property, and a *closed list* for all states that were already checked for the goal property but where the check failed. Among all the open states, *greedy* best-first search always selects the most promising candidate, *i.e.*, the candidate that is most likely the closest to a given goal state [5, Chapt. 4]. Our approach also takes into account the cost of program transformations. With the heuristic function defined as $f(n) = g(n) + h(n)$, we implement the A^*-algorithm. The cost function $g(n)$ returns the cost of producing state n. The function $h(n)$ estimates the distance between the program in state n and the goal state; it is described as `Score` in Fig. 4.

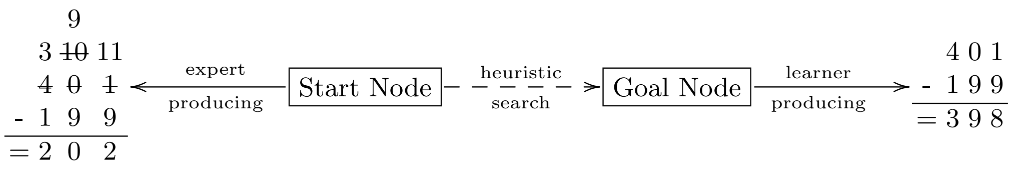
We discuss our approach by example, using the task $401 - 199$ and the learner's solution as depicted in Fig. 2(b). The start state is given by the program in Fig. 1, and we aim for a goal state with a Prolog program that reproduces the `smaller-from-larger` error. We start with representational issues.

Representation. Each state in the search tree is represented by the term

`(Algorithm,IrreducibleDisagreement,Path)`,

encoding a reified version of a Prolog program, the *first* irreducible agreement between program and learner behavior, and the path of prior perturbation actions to reach the current state. Moreover, each state n is associated with a numerical value $f(n)$ that quantifies its production cost as well as the `Algorithm`'s distance to the algorithm of the goal state. A successor state of a given state results from applying a perturbation action. The action obtains a Prolog program in list term representation, performs some sort of mutation, and returns a modified program.

Initialisation. The start node holds the expert program that produces the correct solution. Sought is a mutated variant of the expert program to produce the learner's erroneous solution, here the error `smaller-from-larger`:



Best first search starts with initialising a heap data structure. For this, the start node's distance to the goal node is estimated, using the algorithm given in Fig. 4. There is no single agreement between expert program solving behaviour and learner behaviour, *i.e.*, no single subtraction cell has been filled out the same way. There are six disagreements, yielding a heuristic score of $6 - 0 = 6$.

To inform the generation of the node's children, the first of the six irreducible disagreements – `add_ten_to_minuend(3,1,1)` – (1 instead of 11) is attached to the node's second component. The third component is initialised with the empty path `[]` (cost 0). The node and its estimate is then added to the empty heap.

Checking for Goal State. A state is a goal state when its associated program passes algorithmic debugging with zero disagreements. In this case, best-first search terminates with the goal state, returning the node's algorithm and its path, *i.e.*, a list of actions that were applied to reach the goal state. Here, the initial node, with a non-zero number of disagreements is not the goal node.

Generation of Successor Nodes. If a given state is not the goal state, the state's successors are computed. Given the state's algorithm and the first irreducible disagreement that indicates the location of the "error", Prolog is asked to `findall` applicable perturbation actions, see Fig. 3. For the initial state, the following three successor nodes can be generated:

n_1 `DeleteCallToClause/2`: deletion of the call to `add_ten_to_minuend/3` in the first program clause `process_column/3` (line 15).

n_2 `ShadowClause/2`: addition of the irreducible disagreement (learner's view) `add_ten_to_minuend(3,1,1):-irreducible.`

n_3 `DeleteSubgoalsOfClause/2`: deletion of subgoals from the definition of the predicate `add_ten_to_minuend/3`. As the goal `irreducible/0` cannot be deleted as it is needed by the Oracle, the only permissible action is to delete the subgoal `M10 is M + 10`, and to replace `M10` by `M` in the clause' header.

To add a successor node to the heap, the existing path is extended with the respective action taken. Also, for each node's algorithm, its first irreducible disagreement with the learner must be identified, and the distance to the goal node must be determined. For all successor nodes, we get the irreducible disagreement `decrement(2,[(4,1,S1),(0,9,S2)],[(3,1,S1),(9,9,S2)])`.

Now, consider the nodes' (dis-) agreement scores. Each of the nodes has five disagreements, one less than in the parent node; the second and third child now also feature one agreement. In node n_2, there is an agreement with the new clause `add_ten_to_minuend/3` added; in node n_3, there is an agreement with the perturbated clause `add_ten_to_minuend/3`.

Action Cost. Some program transformations are better than others. Consider the `ShadowClause` action, yielding mutations that are specific to a given input/output pair. The resulting program will, thus, reproduce the learner's error only for the given subtraction task, not for other input. The action's lack of generality is acknowledged by giving it a high cost, namely 5. Hence, the action will only be used when more general and less costlier actions are not applicable.

The action `DeleteSubgoalsOfClause` can delete more than a single subgoal from the predicate indicated by the disagreement. This adds a notion of focus to the perturbation and mirrors the fact that learners often address an impasse, *i.e.*, some difficulty about a skill, with skipping one or more steps of the skill in question. Its cost is defined by the number of subgoals deleted. The mutations performed by `DeleteCallToClause` and `SwapClauseArguments` have a cost of 1.

Score. We obtain the scores $f(n_1) = 1 + (5 - 0) = 6$, $f(n_2) = 5 + (5 - 1) = 9$, and $f(n_3) = 3 + (5 - 1) = 7$. Best-first search selects the child with the lowest valuation, n_1. Its irreducible disagreement on the `decrement/3` operation in column 2 can be addressed by any of the following repairs:

n_{11} The action `DeleteCallToClause` deletes the call to `decrement/3` in the first clause of `process_column/3`. Consequently, two disagreements disappear

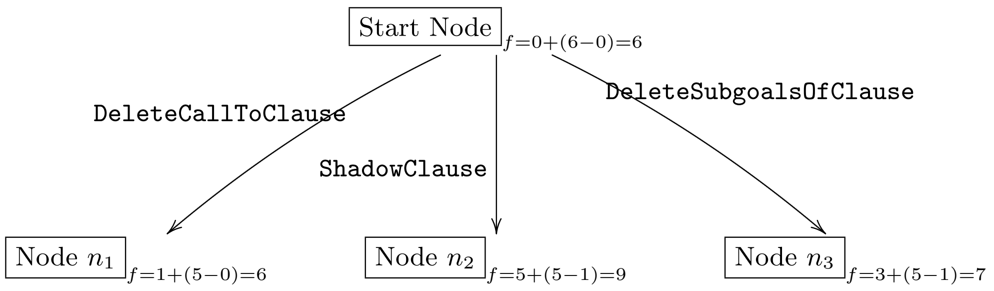


Fig. 5. Best-first Search over Program Transformations: First Expansion.

because `decrement/3` was called twice; also, there is an agreement on `take_difference/4` at the left-most column, because its minuend is not "falsely" decremented. We obtain $f(n_{11}) = (1 + 1) + (2 - 1) = 3$.

n_{12} The action `DeleteSubgoalsOfClause` can remove one or more subgoals in any of the two clause definitions for `decrement/3`.

(a) In the first clause of `decrement/3`, we delete the subgoals `NM1 is NM-1`, `NM is M+10`, and `decrement(CurrentColumn1, RestSum, NewRestSum)`. The clause becomes a null operation, and hence, has the same effect than the first action when the minuend is zero. When the minuend is not zero, the second clause of `decrement/3` is called, generating two disagreements with regards to `decrement/3`, and subsequently, `take_difference/4` (left-most column). We obtain $f(n_{12(a)}) = (1 + 3) + (4 - 1) = 7$.

(b) When we delete the two goals `NM is M + 10` and `NM1 is NM-1`, we obtain four disagreements and one agreement: $f(n_{12(b)}) = (1 + 2) + 4 - 1 = 6$.

(c) The deletion of the single goal `NM1 is NM-1` in the same clause has no positive effect. There are still 5 disagreements, and there is no agreement: $f(n_{12(c)}) = (1 + 1) + 5 - 0 = 7$.

(d) Deleting the recursive call to decrement/3 in `decrement/3` yields three disagreements and one agreement: $f(n_{12(d)}) = (1 + 1) + 3 - 1 = 4$.

n_{13} If we add the disagreement clause to the program

```
decrement(2,[(4,1,S1),(0,9,S2)],[(4,1,3),(0,9,9)]):-irreducible.
```

we also have four disagreements and 1 agreement: $f(n_{13}) = (1{+}5){+}4{-}1 = 9$.

The node n_{11} has the lowest overall estimate; since it is not the goal node, we continue search on its successors.

Generation of Successors (next level). To attack the irreducible disagreement at `goal(take_difference(3,1,9,8)` (8 *vs.* −8) the choices are:

n_{111} `DeleteCallToClause/2` to delete calls to `take_difference/4` in the first or second clause of `process_column/3`; this is not fruitful, as the result cell of each column must obtain a value.

n_{112} `DeleteSubgoalsOfClause/2` to delete the single subgoal in the definition of `take_difference/4`; this mutation produces incorrect result cells.

n_{113} `ShadowClause/2` the insertion of the clause `take_difference(3,1,9,8)`. This removes the disagreement with regard to this goal in the right-most column, but does not address the incorrect result cell in the middle column.

n_{114} `SwapClauseArguments/2` to swap the arguments of `take_difference/4` in the first or second clause of `process_column/3`; Swapping the arguments in the first clause of yields the intended effect, a program with zero disagreements.

The following sequence of actions to produce a goal node with a program that reproduces the `smaller-from-larger` error was found: (1) deletion of the call to the clause `add_ten_to_minuend/3` (line 15), deletion of the call to the clause `decrement/3` (line 17), and the swapping of arguments in `take_difference/4` (line 18). This is the same sequence of actions as advertised in Sect. 2.3.

3.3 Evaluation

We have tested our heuristically-driven algorithm against the eight most frequent bugs of VanLehn's study (Fig. 2). For the top-five bugs, our method is capable of reproducing the perturbations as described in Sect. 2.3. For each case, the goal node found has the same path than the one described. Also, alternative, costlier, goal nodes were found with variations of the perturbations.

Our method is also capable of reproducing programs for the other errors. In all three cases, however, they contain `ShadowClause` perturbations that are specific to the subtraction task. The reconstruction of `always-borrow-left`, *e.g.*, makes the second clause of `decrement/3` a null operation (deleting line 41) – for the task 411-199, the first clause of `decrement/3` is never needed – and then shadows this clause for the left-most column with `decrement(1,[(4,1, _R)],[(2,1,1)]):-irreducible`, a rather surprising but effective permutation. For the reproduction of the error `diff-0-N=N` the task-specific clause `take_difference(2,0,9,9):-irreducible` is added to accommodate the error.

The perturbation action `ShadowClause` acts as a fallback mechanism that ensures that our informed search is complete and always terminates with a mutation that completely reproduces a learner's erroneous answer to a given subtraction task. In this case, the resulting mutation is task-specific and usually fails to reproduce a learner's consistent erroneous behaviour across other tasks.

4 Related Work

Our research has an interesting link to program testing and the design and reliability of test data [3]. The theory of program testing rests on the *competent programmer hypothesis*, which states that programmers "create programs that are *close* to being correct" [3]. In other words, if a program is buggy, then it differs from the correct program only by a combination of simple errors. Moreover, programmers have a rough idea of the kind of error that are likely to occur, and they have the ability to examine their programs in detail. Program testing is also thought to be aided by the *coupling effect*: test cases that detect simple types of faults are sensitive enough to detect more complex types of faults. The analogy to VanLehn's theory of impasses and repairs is striking. When learners encounter an impasse in executing a correct procedure, they address the impasse by a local repair, which often can be explained in terms of simple errors. Also, teachers have a rough idea of the kind of errors learners are likely to make (and learners might be aware of their repairs, too). Good teachers are able to reconstruct the erroneous procedure a learner is executing, and learners are able to correct their mistakes either themselves or under teacher supervision.

In program testing, the technique of *mutation testing* aims at identifying deficiencies in test suites, and to increase the programmer's confidence in the tests' fault detection power. A mutated variant p' of a program p is created only to evaluate the test suite designed for p on p'. If the behaviour between p and p' on test t is different, then the mutant p' is said to be dead, and the test suite "good enough" wrt. the mutation. If they are equal, then p and p' are equivalent, or the test set is not good enough. In this case, the programs' equivalence must be examined by the programmer; if they are not equivalent, the test suite must be extended to cover the critical test. This relates to our approach. When a given program is unable to reproduce a learner's solution, we create a set of perturbated variants, or mutants. If one of them reproduces the learner's solution, it passes the test, and we are done. Otherwise, we choose the best mutant, given the heuristic function f, and continue with the perturbations. The originality of our approach is due to our systematic search for mutations and the use of f to measure the distance between mutants wrt. a given input/output.

In [4], Kilperäinen & Mannila describe a general method for producing complete sets of test data for simple Prolog programs. Their method is based on the competent programmer hypothesis, and works by mutating list processing programs with a small class of suitable modifications. In [6], the authors give a wide range of mutation operators for Prolog. At the clause level, they have operators for the removal of subgoals, for changing the order of subgoals, and for the insertion, removal, or permutation of cuts. At the operator level, they propose mutations that change one arithmetic or relational operator by another one. Moreover, they propose mutations that act on Prolog variables or constants, *e.g.,* the changing of one variable by another variable, an anonymous variable, or a constant, or the changing of one constant into another one. All mutations are syntactic, and aim at capturing typical programmer errors. So far, our approach makes use of a subset of the aforementioned mutation operators. It is surprising that the top-five bugs, accounting for nearly 50% of all learner errors, can be explained by learners skipping steps, that is, mostly in terms of clause deletions.

5 Conclusion and Future Work

In this paper, we propose a method to automatically transform an initial Prolog program into another program capable of producing a given input/output behaviour. The method depends on a heuristic function that estimates the distance between programs. It shows that the test-debug-repair cycle can be mechanised in the tutoring context. Here, there is always a reference model to encode ideal behaviour; moreover, many learner errors can be captured and reproduced by a combination of simple, syntactically-driven program transformation actions.

In the near future, we would like to include more mutation operators (see [6]), investigate their interaction with our existing ones, fine-tune the cost function, and study whether erroneous procedures can be obtained that better reflect learners' incorrect reasoning. Ideally, the new operators can be used to "unemploy" the costly `ShadowClause` operator, whose primary purpose is to serve as a fallback action when all other actions fail. With the aforementioned improvements in place, we will also conduct an experimental evaluation to determine the computational benefits of using heuristic search when compared to the blind (depth-first) search we have used in [10]. Moreover, we are currently working on a web-based interface for multi-column subtraction tasks that we want to give to learners, and where we plan an evaluation in terms of pedagogical benefits.

In the long term, we would like to take-on another domain of instruction to underline the generality of our approach. The domain of learning programming in Prolog is particularly interesting. In the subtraction domain discussed in this paper, we are systematically modifying an expert program into a buggy program to model a learner's erroneous behaviour. In the "learning Prolog domain", we can re-use our program distance measure in a more traditional sense. When learners do specify an executable Prolog program, we compare its behaviour with the prescribed expert program, identify their (dis-)agreement score, and then repair the learner's program, step by step, to become the expert program.

References

1. J. S. Brown and R. R. Burton. Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155–192, 1978.
2. R. R. Burton. Debuggy: Diagnosis of errors in basic mathematical skills. In D. Sherman and J.S. Brown, editors, *Intelligent tutoring systems*. Academic Press, 1982.
3. R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
4. P. Kilperäinen and H. Mannila. Generation of test cases for simple Prolog programs. *Acta Cybern.*, 9(3):235–246, November 1990.
5. S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, first edition, 1995.
6. J. R. Toaldo and S. R. Vergilio. Applying mutation testing in Prolog programs. See `http://www.lbd.dcc.ufmg.br/colecoes/wtf/2006/st2_1.pdf`.
7. K. VanLehn. *Mind Bugs: the origins of proc. misconceptions.* MIT Press, 1990.
8. R. M. Young and T. O'Shea. Errors in children's subtraction. *Cognitive Science*, 5(2):153 – 177, 1981.
9. C. Zinn. Algorithmic debugging to support cognitive diagnosis in tutoring systems. In J. Bach and S. Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *LNAI*, pages 357–368. Springer, 2011.
10. C. Zinn. Program analysis and manipulation to reproduce learner's erroneous reasoning. In E. Albert, editor, *22nd Intl. Symp. of Logic-Based Program Synthesis and Transformation*, volume 7844 of *LNCS*, pages 228–243. Springer, 2013.

HEX-Programs with Existential Quantification*

Thomas Eiter, Michael Fink, Thomas Krennwallner, and Christoph Redl

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter,fink,tkren,redl}@kr.tuwien.ac.at

Abstract. HEX-programs extend ASP by external sources. In this paper, we present *domain-specific existential quantifiers* on top of HEX-programs, i.e., ASP programs with external access which may introduce new values that also show up in the answer sets. Pure logical existential quantification corresponds to a specific instance of our approach. Programs with existential quantifiers may have infinite groundings in general, but for specific reasoning tasks a finite subset of the grounding can suffice. We introduce a generalized grounding algorithm for such problems, which exploits domain-specific termination criteria in order to generate a finite grounding for bounded model generation. As an application we consider query answering over existential rules. In contrast to other approaches, several extensions can be naturally integrated into our approach. We further show how terms with *function symbols* can be handled by HEX-programs, which in fact can be seen as a specific form of existential quantification.

1 Introduction

Answer Set Programming (ASP) is a declarative programming approach which due to expressive and efficient systems like SMODELS, DLV and CLASP, has been gaining popularity for many applications [3]. Current trends in computing, such as context awareness or distributed systems, raised the need for access to external sources in a program, which, e.g., on the Web ranges from light-weight data access (e.g., XML, RDF, or data bases) to knowledge-intensive formalisms (e.g., description logics).

To cater for this need, HEX-programs [7] extend ASP with so-called external atoms, through which the user can couple any external data source with a logic program. Roughly, such atoms pass information from the program, given by predicate extensions, into an external source which returns output values of an (abstract) function that it computes. This convenient extension has been exploited for many different applications, including querying data and ontologies on the Web, multi-context reasoning, or e-government, to mention a few; however, it can also be used to realize built-in functions. The extension is highly expressive as also recursive data access is possible.

A particular feature of external atoms is *value invention*, i.e., that they introduce new values that do not occur in the program. Such values may also occur in an answer set of a HEX-program, e.g., if we have a rule like

$$\mathit{lookup}(X, Y) \leftarrow p(X), \&\mathit{do\_hash}[X](Y)$$

* This research has been supported by the Austrian Science Fund (FWF) project P20840, P20841, P24090, and by the Vienna Science and Technology Fund (WWTF) project ICT08-020.

where intuitively, the external atom $\&do\_hash[X](Y)$ generates a hash key Y for the input X and records it in the fact $lookup(X,Y)$. Here, the variable Y can be seen under existential quantification, i.e., as $\exists Y$, where the quantifier is externally evaluated, by taking domain-specific information into account; in the example above, this would be a procedure to calculate the hashkey. Such domain-specific quantification occurs frequently in applications, be it e.g. for built-in functions (just think of arithmetic), the successor of a current situation in situation calculus, retrieving the social security number of a person etc. To handle such quantifiers in ordinary ASP is cumbersome; they amount to interpreted functions and require proper encoding and/or special solvers.

HEX-programs however provide a uniform approach to represent such domain-specific existentials. The external treatment allows to deal elegantly with datatypes (e.g., the social security number, or an IBAN of bank account, or strings and numbers like reals), to respect parameters, and to realize partial or domain-restricted quantification of the form $\exists Y.\phi(X) \supset p(X,Y)$ where $\phi(X)$ is a formula that specifies the domain of elements X for which an existential value needs to exist; clearly, also range-restricted quantification $\exists Y.\psi(Y) \supset p(X,Y)$ that limits the value of Y to elements that satisfy ψ can be conveniently realized.

In general, such value invention on an infinite domain (e.g.,for strings) leads to infinite models, which can not be finitely generated. Under suitable restrictions on a program Π, this can be excluded, in particular if a finite portion of the grounding of Π is equivalent to its full, infinite grounding. This is exploited by various notions of *safety* of HEX-programs that generalize safety of logic programs.

In particular, *liberal domain-expansion safety (de-safety)* [6] is a recent notion based on term-bounding functions, which makes it modular and flexible; various well-known notions of safety are subsumed by it. For example, consider the program

$$\Pi = \{\ s(a); \quad t(Y) \leftarrow s(X), \&concat[X,a](Y); \quad s(X) \leftarrow t(X), d(X)\ \}, \qquad (1)$$

where $\&concat[X,a](Y)$ is true iff Y is the string concatenation of X and a. Program Π is safe (in the usual sense) but $\&concat[X,a](Y)$ could hold for infinitely many Y, if one disregards the semantics of *concat*; however, if this is done by a term bounding function in abstract form, then the program is found to be liberally de-safe and thus a finite part of Π's grounding is sufficient to evaluate it.

Building on a grounding algorithm for liberally de-safe programs [5], we can effectively evaluate HEX-programs with domain-specific existentials that fall in this class. Moreover, we in fact generalize this algorithm with *domain specific termination*, such that for non-safe programs, a finitely *bounded grounding* is generated. Roughly speaking, such a bounded grounding amounts to domain-restricted quantification $\exists Y.\phi(X) \supset p(X,Y)$ where the domain condition $\phi(X)$ is dynamically evaluated during the grounding, and information about the grounding process may be also considered. Thus, domain-specific termination leads to a partial (bounded) grounding of the program, Π', yielding *bounded models* of the program Π; the idea is that the grounding is faithful in the sense that every answer set of Π' can be extended to a (possibly infinite) answer set of Π, and considering bounded models is sufficient for an application. This may be fruitfully exploited for applications like query answering over existential rules, reasoning about actions, or to evaluate classes of logic programs with function symbols like FDNC programs [8]. Furthermore, even if bounded models are not faithful (i.e., may not be extendible to models of the full grounding), they might be convenient e.g. to provide strings, arithmetic, recursive data structures like lists, trees etc, or action sequences of bounded length resp. depth. The point is that the bound does not have to be "coded" in the program (like `maxint` in `DLV` to bound the integer range), but can be provided via termination criteria in the grounding, which gives greater flexibility.

Organization. After the preliminaries we proceed as follows.

• We introduce domain-specific existential quantification in HEX-programs and considers its realization (Section 3). To this end, we introduce a generalized grounding algorithm with *hooks* for termination criteria, which enables bounded grounding. Notably, its output for de-safe programs (using trivial criteria) is equivalent to the original program, i.e., it has the same answer sets.
We illustrate some advantages of our approach, which cannot easily be integrated into direct implementations of existential quantifiers.

• As an example, we consider the realization of *null values* (which are customary in databases) as a domain-specific existential quantifier, leading to HEX$^{\exists}$-programs (Section 4); they include existential rules of form $\forall \mathbf{X} \forall \mathbf{Z} \exists \mathbf{Y}.\psi(\mathbf{Z},\mathbf{Y}) \leftarrow \phi(\mathbf{X},\mathbf{Y},\mathbf{Z})$ (also known as tuple-generating dependencies), where $\psi(\mathbf{Z},\mathbf{Y})$ is an atom[1] and $\phi(\mathbf{X},\mathbf{Y},\mathbf{Z})$ is a conjunction of atoms. Our framework can be thus exploited for bounded grounding, and in combination with a HEX-solver for bounded model generation of such programs.

• As an immediate application, we consider query answering over existential rules (Section 5), which reduces for prominent settings to query answering over a universal model. Under de-safety, a finite such model can be generated using our framework; this allows to cover a range of acyclic existential rules, including the very general notion of model-faithful acyclicity [14]. For non-de safe programs, a bounded universal model may be generated under suitable conditions; we illustrate this for Shy-programs - a class of programs with existential rules for which query answering is decidable, cf. [17].

• Furthermore, we show how terms with function symbols can be processed using an encoding as a HEX-program (Section 6). To this end, we use dedicated external atoms to construct and decompose functional terms; bounded grounding enables us here to elegantly restrict the term depth, which is useful for applications such as reasoning with actions in situation calculus under bounded horizon, or reasoning from FDNC programs.

We conclude with a discussion and an outlook on future work in Section 7. Our prototype system is available at http:/www.kr.tuwien.ac.at/research/systems/dlvhex.

2 Preliminaries

HEX-Program Syntax. HEX-programs generalize (disjunctive) logic programs under the answer set semantics [13] with external source access; for details and background see [7]. They are built over mutually disjoint sets $\mathcal{P}$, $\mathcal{X}$, $\mathcal{C}$, and $\mathcal{V}$ of ordinary predicates, external predicates, constants, and variables, respectively. Every $p \in \mathcal{P}$ has an arity $ar(p) \geq 0$, and every external predicate $\&g \in \mathcal{X}$ has an input arity $ar_i(\&g) \geq 0$ of input parameters and an output arity $ar_o(\&g) \geq 0$ of output arguments.

[1] In general, $\psi(\mathbf{Z},\mathbf{Y})$ might be a conjunction of atoms but this may be normalized.

An *external atom* is of the form $\&g[\mathbf{X}](\mathbf{Y})$, where $\&g \in \mathcal{X}$, $\mathbf{X} = X_1, \dots, X_\ell$ ($\ell = ar_i(\&g)$) are input parameters with $X_i \in \mathcal{P} \cup \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, and $\mathbf{Y} = Y_1, \dots, Y_m$ ($m = ar_o(\&g)$) are output terms with $Y_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq m$; we use lower case $\mathbf{x} = x_1, \dots, x_\ell$ resp. $\mathbf{y} = y_1, \dots, y_m$ if $\mathbf{X}$ resp. $\mathbf{Y}$ is variable-free. We assume the input parameters of $\&g$ are typed by $type(\&g, i) \in \{\mathtt{const}, \mathtt{pred}\}$ for $1 \leq i \leq ar_i(\&g)$, and that $X_i \in \mathcal{P}$ if $type(\&g, i) = \mathtt{pred}$ and $X_i \in \mathcal{C} \cup \mathcal{V}$ otherwise.

A HEX-program consists of rules

$$a_1 \vee \dots \vee a_k \leftarrow b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n \ , \tag{2}$$

where each a_i is an (ordinary) atom $p(X_1, \dots, X_\ell)$ with $X_i \in \mathcal{C} \cup \mathcal{V}$ for all $1 \leq i \leq \ell$, each b_j is either an ordinary atom or an external atom, and $k + n > 0$.

The *head* of a rule r is $H(r) = \{a_1, \dots, a_n\}$ and the *body* is $B(r) = \{b_1, \dots, b_m, \text{not } b_{m+1}, \dots, \text{not } b_n\}$. We call b or not b in a rule body a *default literal*; $B^+(r) = \{b_1, \dots, b_m\}$ is the *positive body*, $B^-(r) = \{b_{m+1}, \dots, b_n\}$ is the *negative body*. For a program Π (rule r), let $A(\Pi)$ ($A(r)$) be the set of all ordinary atoms and $EA(\Pi)$ ($EA(r)$) be the set of all external atoms occurring in Π (in r).

HEX-Program Semantics. Following [11], a *(signed) ground literal* is a positive or a negative formula $\mathbf{T}a$ resp. $\mathbf{F}a$, where a is a ground ordinary atom. For a ground literal $\sigma = \mathbf{T}a$ or $\sigma = \mathbf{F}a$, let $\overline{\sigma}$ denote its opposite, i.e., $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. An *assignment* $\mathbf{A}$ is a consistent set of literals $\mathbf{T}a$ or $\mathbf{F}a$, where $\mathbf{T}a$ expresses that a is true and $\mathbf{F}a$ that a is false. We also identify a complete assignment $\mathbf{A}$ with its true atoms, i.e., $\mathbf{T}(\mathbf{A}) = \{a \mid \mathbf{T}a \in \mathbf{A}\}$. The semantics of a ground external atom $\&g[\mathbf{x}](\mathbf{y})$ wrt. a complete assignment $\mathbf{A}$ is given by a $1{+}k{+}l$-ary Boolean-valued *oracle function*, $f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y})$. Parameter x_i with $type(\&g, i) = \mathtt{pred}$ is *monotonic* (*antimonotonic*), if $f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y}) \leq f_{\&g}(\mathbf{A}', \mathbf{x}, \mathbf{y})$ ($f_{\&g}(\mathbf{A}', \mathbf{x}, \mathbf{y}) \leq f_{\&g}(\mathbf{A}, \mathbf{x}, \mathbf{y})$) whenever $\mathbf{A}'$ increases $\mathbf{A}$ only by literals $\mathbf{T}a$, where a has predicate x_i; otherwise, x_i is called *nonmonotonic*.

Non-ground programs are handled by grounding as usual. The set of constants appearing in a program Π is denoted C_Π. The *grounding* $grnd_C(r)$ of a rule r wrt. $C \subseteq \mathcal{C}$ is the set of all rules $\{\sigma(r) \mid \sigma : \mathcal{V} \mapsto C\}$, where σ is a *grounding substitution*, and $\sigma(r)$ results if each variable X in r is replaced by $\sigma(X)$. The *grounding of a program* Π wrt. C is defined as $grnd_C(\Pi) = \bigcup_{r \in \Pi} grnd_C(r)$.

Satisfaction of rules and programs [13] is extended to HEX-rules r and programs Π in the obvious way. The *FLP-reduct* is defined as $f grnd_{\mathcal{C}}(\Pi)^{\mathbf{A}} = \{r \in grnd_{\mathcal{C}}(\Pi) \mid \mathbf{A} \models B(r)\}$. An *answer set* of a program Π is a model of $f grnd_{\mathcal{C}}(\Pi)^{\mathbf{A}}$ that is subset-minimal in its positive part [9]. We denote by $\mathcal{AS}(\Pi)$ the set of all answer sets of Π.

Take as an example the program $\Pi = \{str(N) \leftarrow str(L), \&head[L](N); str(N) \leftarrow str(L), \&tail[L](N)\}$, where $\&head[L](N)$ ($\&tail[L](N)$) is true iff string N is string L without the last (first) character. For $str(x)$, Π computes all substrings of string x.

Safety. In general, $\mathcal{C}$ has constants that do not occur in Π and can even be infinite (e.g., the set of all strings). Safety criteria guarantee that a finite portion $\Pi' \subseteq grnd_{\mathcal{C}}(\Pi)$ (also called *finite grounding* of Π; usually by restricting to a finite $C \subseteq \mathcal{C}$) has the same answer sets as Π. Ordinary safety requires that every variable in a rule r occurs either in an ordinary atom in $B^+(r)$, or in the output list $\mathbf{Y}$ of an external atom $\&g[\mathbf{X}](\mathbf{Y})$ in $B^+(r)$ where all variables in $\mathbf{X}$ are safe. However, this notion is not sufficient.

Example 1. Let $\Pi = \{s(a); t(Y) \leftarrow s(X), \&concat[X,a](Y); s(X) \leftarrow t(X), d(X)\}$, where $\&concat[X,a](Y)$ is true iff Y is the string concatenation of X and a. Then Π is safe but $\&concat[X,a](Y)$ can introduce infinitely many values.

The general notion of *(liberal) domain-expansion safety (de-safety)* subsumes a range of other well-known notions and can be easily extended in a modular fashion [6]. It is based on *term bounding functions* (TBFs), which intuitively declare terms in rules as *bounded*, if there are only finitely many substitutions for this term in a *canonical grounding* $CG(\Pi)$ of Π.[2] The latter is infinite in general but finite for de-safe programs.

More specifically we consider *attributes* and *ranges*. For an ordinary predicate $p \in \mathcal{P}$, let $p{\restriction}i$ be the *i-th attribute of p* for all $1 \leq i \leq ar(p)$. For an external predicate $\&g \in \mathcal{X}$ with input list $\mathbf{X}$ in rule r, let $\&g[\mathbf{X}]_r{\restriction}_T i$ with $T \in \{\mathrm{I}, \mathrm{O}\}$ be the *i-th input resp. output attribute of $\&g[\mathbf{X}]$ in r* for all $1 \leq i \leq ar_T(\&g)$. For a ground program Π, an attribute *range* is, intuitively, the set of ground terms which occur in the position of the attribute. Formally, for an attribute $p{\restriction}i$ we have $range(p{\restriction}i, \Pi) = \{t_i \mid p(t_1, \dots, t_{ar(p)}) \in A(\Pi)\}$; for $\&g[\mathbf{X}]_r{\restriction}_T i$ it is $range(\&g[\mathbf{X}]_r{\restriction}_T i, \Pi) = \{x_i^T \mid \&g[\mathbf{x}^{\mathrm{I}}](\mathbf{x}^{\mathrm{O}}) \in EA(\Pi)\}$, where $\mathbf{x}^s = x_1^s, \dots, x_{ar_s(\&g)}^s$. Now term bounding functions are introduced as follows:

Definition 1 (Term Bounding Function (TBF)). *A TBF $b(\Pi, r, S, B)$ maps a program Π, a rule $r \in \Pi$, a set S of already safe attributes, and a set B of already bounded terms in r to an enlarged set $b(\Pi, r, S, B) \supseteq B$ of bounded terms, s.t. every $t \in b(\Pi, r, S, B)$ has finitely many substitutions in $CG(\Pi)$ if (i) the attributes S have a finite range in $CG(\Pi)$ and (ii) each term in $terms(r) \cap B$ has finitely many substitutions in $CG(\Pi)$.*

Liberal domain-expansion safety of programs is then parameterized with a term bounding function, such that concrete syntactic and/or semantic properties can be plugged in; concrete term bounding functions are described in [6]. The concept is defined in terms of domain-expansion safe attributes $S_\infty(\Pi)$, which are stepwise identified as $S_n(\Pi)$ in mutual recursion with bounded terms $B_n(r, \Pi, b)$ of rules r in Π.

Definition 2 ((Liberal) Domain-expansion Safety). *Given a TBF b, the set of* bounded terms *$B_n(r, \Pi, b)$ in step $n \geq 1$ in a rule $r \in \Pi$ is $B_n(r, \Pi, b) = \bigcup_{j \geq 0} B_{n,j}(r, \Pi, b)$ where $B_{n,0}(r, \Pi, b) = \emptyset$ and for $j \geq 0$, $B_{n,j+1}(r, \Pi, b) = b(\Pi, r, S_{n-1}(\Pi), B_{n,j})$.*

The set of domain-expansion safe attributes *$S_\infty(\Pi) = \bigcup_{i \geq 0} S_i(\Pi)$ of a program Π is iteratively constructed with $S_0(\Pi) = \emptyset$ and for $n \geq 0$:*

- *$p{\restriction}i \in S_{n+1}(\Pi)$ if for each $r \in \Pi$ and atom $p(t_1, \dots, t_{ar(p)}) \in H(r)$, it holds that $t_i \in B_{n+1}(r, \Pi, b)$, i.e., t_i is* bounded*;*
- *$\&g[\mathbf{X}]_r{\restriction}_{\mathrm{I}} i \in S_{n+1}(\Pi)$ if each $\mathbf{X}_i$ is a* bounded *variable, or $\mathbf{X}_i$ is a predicate input parameter p and $p{\restriction}1, \dots, p{\restriction}ar(p) \in S_n(\Pi)$;*
- *$\&g[\mathbf{X}]_r{\restriction}_{\mathrm{O}} i \in S_{n+1}(\Pi)$ if and only if r contains an external atom $\&g[\mathbf{X}](\mathbf{Y})$ such that $\mathbf{Y}_i$ is bounded, or $\&g[\mathbf{X}]_r{\restriction}_{\mathrm{I}} 1, \dots, \&g[\mathbf{X}]_r{\restriction}_{\mathrm{I}} ar_{\mathrm{I}}(\&g) \in S_n(\Pi)$.*

A program Π is (liberally) de-safe*, if it is safe and all its attributes are de-safe.*

Example 2. The program Π from Example 1 is liberally de-safe using the TBF b_{synsem} from [6] as the generation of infinitely many values is prevented by $d(X)$ in the last rule.

Every de-safe HEX-program has a finite grounding that preserves all answer sets [6].

[2] $CG(\Pi)$ is the least fixed point $G_\Pi^\infty(\emptyset)$ of a monotone operator $G_\Pi(\Pi') = \bigcup_{r \in \Pi}\{r\theta \mid r\theta \in grnd_{\mathcal{C}}(r), \exists \mathbf{A} \subseteq \mathcal{A}(\Pi'), \mathbf{A} \not\models \bot, \mathbf{A} \models B^+(r\theta)\}$ on programs Π' [6].

3 HEX-Programs with Existential Quantification

In this section, we consider HEX-programs with *domain-specific existential quantifiers*. This term refers to the introduction of new values in rule bodies which are propagated to the head such that they may appear in the answer sets of a program. Logical existential quantification is a special case of our approach (used in Section 4 to illustrate a specific instance), where just the existence but not the structure of values is of interest. Instead, in our work also the structure of introduced values may be relevant and can be controlled by external atoms.

Instantiating, i.e., applying, our approach builds on an extension of the grounding algorithm for HEX-programs in [5] by additional *hooks*. They support the insertion of application-specific termination criteria, and thus can be exploited for computing a finite subset of the grounding in case of non-de-safe HEX-programs. The latter may be sufficient to consider a certain reasoning task, e.g., for bounded model building. For instance, we discuss *queries* over (positive) programs with (logical) existential quantifiers in Section 5, which can be answered by computing a finite part of a canonical model.

HEX-Program Grounding. For introducing our *bounded grounding algorithm* BGroundHEX, we make use of so-called input auxiliary rules. We say that an external atom $\&g[\mathbf{Y}](\mathbf{X})$ *joins* an atom b, if some variable from $\mathbf{Y}$ occurs in b, where in case b is an external atom the occurrence is in the output list of b.

Definition 3 (Input Auxiliary Rule). *Let Π be a HEX-program. Then for each external atom $\&g[\mathbf{Y}](\mathbf{X})$ occurring in rule $r \in \Pi$, a rule $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ is composed as follows:*
- *The head is $H(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}) = \{g_{inp}(\mathbf{Y})\}$, where g_{inp} is a fresh predicate; and*
- *The body $B(r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})})$ contains all $b \in B^+(r) \backslash \{\&g[\mathbf{Y}](\mathbf{X})\}$ which join $\&g[\mathbf{Y}](\mathbf{X})$.*

Intuitively, input auxiliary rules are used to derive all ground input tuples $\mathbf{y}$, under which the external atom needs to be evaluated.

Our grounding approach is based on a grounder for ordinary ASP programs. Compared to the naive grounding $grnd_{\mathcal{C}}(\Pi)$, we allow the ASP grounder GroundASP to eliminate rules if their body is always false, and ordinary body literals from the grounding that are always true, as long as this does not change the answer sets. More formally, a rule r' is an *o-strengthening* (ordinary-strengthening) of a rule r, if $H(r') = H(r)$, $B(r') \subseteq B(r)$ and $B(r) \setminus B(r')$ contains only ordinary literals, i.e., no external atom replacements.

Definition 4. *An algorithm GroundASP that takes as input a program Π and outputs a ground program Π' is a* faithful ASP grounder *for a safe program Π, if:*
- *$\mathcal{AS}(\Pi') = \mathcal{AS}(grnd_{C_\Pi}(\Pi))$;*
- *Π' consists of o-strengthenings of rules in $grnd_{C_\Pi}(\Pi)$;*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has no o-strengthening in Π', then every answer set of $grnd_{C_\Pi}(\Pi)$ falsifies some ordinary literal in $B(r)$; and*
- *if $r \in grnd_{C_\Pi}(\Pi)$ has some o-strengthening $r' \in \Pi'$, then every answer set of $grnd_{C_\Pi}(\Pi)$ satisfies $B(r) \setminus B(r')$.*

Intuitively, the bounded grounding algorithm BGroundHEX can be explained as follows. Program Π is the non-ground input program. Program Π_p is the non-ground

Algorithm BGroundHEX

```
    Input: A HEX-program $\Pi$
    Output: A ground HEX-program $\Pi_g$
(a) $\Pi_p = \Pi \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})} \mid \&g[\mathbf{Y}](\mathbf{X}) \text{ in } r \in \Pi\}$
    Replace all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules $r$ in $\Pi_p$ by $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$
    $i \leftarrow 0$
(b) while Repeat() do
        $i \leftarrow i + 1$ // Remember already processed input tuples at iteration $i$
(c)     Set $NewInputTuples \leftarrow \emptyset$ and $PIT_i \leftarrow \emptyset$
        repeat
            $\Pi_{pg} \leftarrow$ GroundASP($\Pi_p$) // partial grounding
(d)         for $\&g[\mathbf{Y}](\mathbf{X})$ in a rule $r \in \Pi$ do // evaluate all external atoms
(e)             // do this under all relevant assignments
                $\mathbf{A}_{ma} = \{\mathbf{T}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_m\} \cup \{\mathbf{F}p(\mathbf{c}) \mid a(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_a\}$
                for $\mathbf{A}_{nm} \subseteq \{\mathbf{T}p(\mathbf{c}), \mathbf{F}p(\mathbf{c}) \mid p(\mathbf{c}) \in A(\Pi_{pg}), p \in \mathbf{Y}_n\}$ s.t. $\nexists a : \mathbf{T}a, \mathbf{F}a \in \mathbf{A}_{nm}$ do
                    $\mathbf{A} = (\mathbf{A}_{ma} \cup \mathbf{A}_{nm} \cup \{\mathbf{T}a \mid a \leftarrow \in \Pi_{pg}\}) \setminus \{\mathbf{F}a \mid a \leftarrow \in \Pi_{pg}\}$
(f)                 for $\mathbf{y} \in \{\mathbf{c} \mid r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c}) \in A(\Pi_{pg})$ s.t. Evaluate($r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})) = true$ do
(g)                     // add ground guessing rules and remember $\mathbf{y}$-evaluation
                        $\Pi_p \leftarrow \Pi_p \cup \{e_{r,\&g[\mathbf{y}]}(\mathbf{x}) \vee ne_{r,\&g[\mathbf{y}]}(\mathbf{x}) \leftarrow \mid f_{\&g}(\mathbf{A}, \mathbf{y}, \mathbf{x}) = 1\}$
                        $NewInputTuples \leftarrow NewInputTuples \cup \{r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{y})\}$

            $PIT_i \leftarrow PIT_i \cup NewInputTuples$
        until $\Pi_{pg}$ did not change
(h) Remove input auxiliary rules and external atom guessing rules from $\Pi_{pg}$
    Replace all $e_{\&g[\mathbf{y}]}(\mathbf{x})$ in $\Pi_{pg}$ by $\&g[\mathbf{y}](\mathbf{x})$
    return $\Pi_{pg}$
```

ordinary ASP *prototype program*, which is an iteratively updated variant of Π enriched with additional rules. In each step, the *preliminary ground program* Π_{pg} is produced by grounding Π_p using a standard ASP grounding algorithm. Program Π_{pg} is intended to converge against a fixpoint, i.e., a final *ground* HEX-*program* Π_g. For this purpose, the loop at (b) and the abortion check at (f) introduce two *hooks* (Repeat and Evaluate) which allow for realizing application-specific termination criteria. They need to be substituted by concrete program fragments depending on the reasoning task; for now we assume that the loop at (f) runs exactly once and the check at (f) is always true (which is sound and complete for model computation of de-safe programs, cf. Proposition 1).

The algorithm first introduces input auxiliary rules $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}$ for every external atom $\&g[\mathbf{Y}](\mathbf{X})$ in a rule r in Π in Part (a). Then, all external atoms $\&g[\mathbf{Y}](\mathbf{X})$ in all rules r in Π_p are replaced by ordinary *replacement atoms* $e_{r,\&g[\mathbf{Y}]}(\mathbf{X})$. This allows the algorithm to use an ordinary ASP grounder GroundASP in the main loop at (b). After the grounding step, it is checked whether the grounding contains all relevant constants. For this, the algorithm checks, for all external atoms (d) and all relevant input interpretations (e), potential output tuples at (f), if they contain any new value that was not yet respected in the grounding. (Note that, $\mathbf{Y}_m, \mathbf{Y}_a, \mathbf{Y}_n$ denote the sets of *monotonic*, *antimonotonic*, and *nonmonotonic* predicate input parameters in $\mathbf{Y}$, respectively.) It adds the relevant constants in form of guessing rules at (g) to Π_p (this may also be expressed by unstratified negation). Then the main loop starts over again. Eventually, the algorithm is intended to find a program respecting all relevant constants. Then at (h), auxiliary input rules are removed and replacement atoms are translated to external atoms.

Let us illustrate the grounding algorithm with the following example.

Example 3. Let Π be the following program:

$$f : d(a).\ d(b).\ d(c).\quad r_1 : s(Y) \leftarrow d(X), \&\mathit{diff}[d,n](Y), d(Y).$$
$$r_2 : n(Y) \leftarrow d(X), \&\mathit{diff}[d,s](Y), d(Y).$$
$$r_3 : c(Z) \leftarrow \&\mathit{count}[s](Z).$$

Here, $\&\mathit{diff}[s_1, s_2](x)$ is true for all elements x, which are in the extension of s_1 but not in that of s_2, and $\&\mathit{count}[s](i)$ is true for the integer i corresponding to the number of elements in the extension of s. The program first partitions the domain (extension of d) into two sets (extensions of s and n) and then computes the size of s. Program Π_p at the beginning of the first iteration is as follows, where $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ are shorthands for $e_{r_1,\&\mathit{diff}[d,n]}(Y)$, $e_{r_2,\&\mathit{diff}[d,s]}(Y)$, and $e_{r_3,\&\mathit{count}[s]}(Z)$, respectively.

$$f : d(a).\ d(b).\ d(c).\quad r_1 : s(Y) \leftarrow d(X), e_1(Y), d(Y).$$
$$r_2 : n(Y) \leftarrow d(X), e_2(Y), d(Y).$$
$$r_3 : c(Z) \leftarrow e_3(Z).$$

Program Π_{pg} contains no instances of r_1, r_2 and r_3 because the optimizer recognizes that $e_1(Y)$, $e_2(Y)$ and $e_3(Z)$ occur in no rule head and no ground instance can be true in any answer set. Then the algorithm moves to the checking phase. It evaluates the external atoms in r_1 and r_2 under $\mathbf{A} = \{d(a), d(b), d(c)\}$ (note that $\&\mathit{diff}[s_1, s_2](x)$ is monotonic in s_1 and antimonotonic in s_2) and adds the rules $\{e_i(Z) \lor ne_i(Z) \leftarrow \mid\ Z \in \{a, b, c\}, i \in \{1, 2\}\}$ to Π_p. Then it evaluates $\&\mathit{count}[s](Z)$ under all $\mathbf{A} \subseteq \{s(a), s(b), s(c)\}$ because it is nonmonotonic in s, and adds the rules $\{e_3(Z) \lor ne_3(Z) \leftarrow \mid\ Z \in \{0, 1, 2, 3\}\}$. It terminates after the second iteration. $\square$

The main difference to the algorithm from [5] is the addition of the two hooks at (c) (Repeat) and at (f) (Evaluate), that need to be defined for a concrete instance of the algorithm (which we do in the following). We assume that the hooks are substituted by code fragments with access to all local variables. Moreover, the set PIT_i contains the input atoms for which the corresponding external atoms have been evaluated in iteration i. Evaluate decides for a given input atom $r_{inp}^{\&g[\mathbf{Y}](\mathbf{X})}(\mathbf{c})$ if the corresponding external atom shall be evaluated under $\mathbf{c}$. This allows for abortion of the grounding even if it is incomplete, which can be exploited for reasoning tasks over programs with infinite groundings where a finite subset of the grounding is sufficient. The second hook Repeat allows for repeating the core algorithm multiple times such that Evaluate can distinguish between input tuples processed in different iterations. Naturally, soundness and completeness of the algorithm cannot be shown in general, but depends on concrete instances for (c) and (f) which in turn may vary for different reasoning tasks.

Domain-specific Existential Quantification in HEX-Programs. We can realize domain-specific existential quantification naturally in HEX-programs by appropriate external atoms that introduce new values to the program. The realization exploits *value invention* as supported by HEX-programs, i.e., external atoms which return constants that do not show up in the input program. Realizing existentials by external atoms also allows to use constants different from Skolem terms, i.e., datatypes with a specific semantics. The values introduced may depend on input parameters passed to the external atom.

Example 4. Consider the following rule:

$$\mathit{iban}(B, I) \leftarrow \mathit{country}(B, C), \mathit{bank}(B, N), \&\mathit{iban}[C, B, N](I).$$

Suppose $\mathit{bank}(b, n)$ models financial institutions b with their associated national number n, and $\mathit{country}(b, c)$ holds for an institution b and its home country c. Then one can use $\&\mathit{iban}[C, B, N](I)$ to generate an IBAN (*International Bank Account Number*) from the country, the bank name and account number.

Here, the structure of the introduced value is relevant, but an algorithm which computes it can be hidden from the user. The introduction of new values may also be subject to additional conditions which cannot easily be expressed in the program.

Example 5. Consider the following rule:

$$\mathit{lifetime}(M, L) \leftarrow \mathit{machine}(M, C), \&\mathit{lifetime}[M, C](L).$$

It expresses that each purchased machine m with cost c ($\mathit{machine}(m, c)$) higher than a given limit has assigned an expected lifetime l ($\mathit{lifetime}(m, l)$) used for fiscal purposes, whereas purchases below that limit are fully tax deductible in the year of acquirement. Then testing for exceedance of the limit might involve real numbers and cannot easily be done in the logic program. However, the external atom can easily be extended in such a way that a value is only introduced if this side constraint holds.

Counting quantifiers may be realized in this way, i.e., expressing that there exist *exactly* k or *at least* k elements, which is used e.g. in description logics.While a direct implementation of existentials requires changes in the reasoner, a simulation using external atoms is easily extensible.

4 HEX$^{\exists}$-Programs

We now realize the logical existential quantifier as a specific instance of our approach, which can also be written in the usual syntax; a rewriting then simulates it by using external atoms which return dedicated *null values* to represent a representative for the unnamed values introduced by existential quantifiers. We start by introducing a language for HEX-programs with logical existential quantifiers, called HEX$^{\exists}$*-programs*.

A HEX$^{\exists}$*-program* is a finite set of rules of form

$$\forall \mathbf{X} \exists \mathbf{Y} : p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \qquad (3)$$

where $\mathbf{X}$ and $\mathbf{Y}$ are disjoint sets of variables, $\mathbf{X}' \subseteq \mathbf{X}$, $p(\mathbf{X}', \mathbf{Y})$ is an atom, and $\mathbf{conj}[\mathbf{X}]$ is a conjunction of default literals or default external literals containing all and only the variables $\mathbf{X}$; without confusion, we also omit $\forall \mathbf{X}$.

Intuitively speaking, whenever $\mathbf{conj}[\mathbf{X}]$ holds for some vector of constants $\mathbf{X}$, then there should exist a vector $\mathbf{Y}$ of (unnamed) individuals such that $p(\mathbf{X}', \mathbf{Y})$ holds. Existential quantifiers are simulated by using *new* null values which represent the introduced unnamed individuals. Formally, we assume that $\mathcal{N} \subseteq \mathcal{C}$ is a set of dedicated null values, denoted by ω_i with $i \in \mathbb{N}$, which do not appear in the program.

We transform HEX$^{\exists}$-programs to HEX-programs as follows. For a HEX$^{\exists}$-program Π, let $T_{\exists}(\Pi)$ be the HEX-program with each rule r of form (3) replaced by

$$p(\mathbf{X}', \mathbf{Y}) \leftarrow \mathbf{conj}[\mathbf{X}], \&\mathit{exists}^{|\mathbf{X}'|, |\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y}),$$

where $f_{\&\mathit{exists}^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 1$ iff $\mathbf{y} = \omega_1, \ldots, \omega_m$ is a vector of *fresh and unique null values* for $r, \mathbf{x}$, and $f_{\&\mathit{exists}^{n,m}}(\mathbf{A}, r, \mathbf{x}, \mathbf{y}) = 0$ otherwise.

Each existential quantifier is replaced by an external atom $\&exists^{|\mathbf{X}'|,|\mathbf{Y}|}[r, \mathbf{X}'](\mathbf{Y})$ of appropriate input and output arity which exploits value invention for simulating the logical existential quantifier similar to the *chase* algorithm.

We call a HEX$^{\exists}$-program Π liberally de-safe iff $T_{\exists}(\Pi)$ is liberally de-safe. Various notions of cyclicity have been introduced, e.g., in [14]; here we use the one from [6].

Example 6. The following set of rules is a HEX$^{\exists}$-program Π:

$$employee(john). \quad employee(joe).$$
$$r_1 : \exists Y : office(X,Y) \leftarrow employee(X). \quad r_2 : room(Y) \leftarrow office(X,Y)$$

Then $T_{\exists}(\Pi)$ is the following de-safe program:

$$\begin{array}{ll} & employee(john). \quad employee(joe). \\ r_1' : & office(X,Y) \leftarrow employee(X), \&exists^{1,1}[r_1, X](Y). \\ r_2 : & room(Y) \leftarrow office(X,Y) \end{array}$$

Intuitively, each employee X has some unnamed office Y of X, which is a room. The unique answer set of $T_{\exists}(\Pi)$ is $\{employee(john), employee(joe), office(john, \omega_1), office(joe, \omega_2), room(\omega_1), room(\omega_2)\}$.

For grounding de-safe programs, we simply let Repeat test for $i < 1$ and Evaluate return $true$. Explicit model computation is in general infeasible for non-de-safe programs. However, the resulting algorithm GroundDESafeHEX always terminates for de-safe programs. For non-de-safe programs, we can support bounded model generation by other hook instantiations. This is exploited e.g. for query answering over cyclic programs (described next). One can show that the algorithm computes all models of the program.

Proposition 1. *For de-safe programs Π, $\mathcal{AS}(\text{GroundDESafeHEX}(\Pi)) \equiv^{pos} \mathcal{AS}(\Pi)$, where $\equiv^{pos}$ denotes equivalence of the answer sets on positive atoms.*

5 Query Answering over Positive HEX$^{\exists}$-Programs

The basic idea for query answering over programs with possibly infinite models is to compute a ground program with a single answer set that can be used for answering the query. Positive programs with existential variables are essentially grounded by simulating the *parsimonious chase procedure* from [17], which uses null values for each existential quantification. However, for termination of BGroundHEX we need to provide specific instances of the hooks in the grounding algorithm.

We start by restricting the discussion to a fragment of HEX$^{\exists}$-programs, called *Datalog*$^{\exists}$-programs [17]. A *Datalog*$^{\exists}$-program is a HEX$^{\exists}$-program where every rule body $\mathbf{conj}[\mathbf{X}]$ consists of positive ordinary atoms. Thus compared to HEX$^{\exists}$-programs, default negation and external atoms are excluded.

As an example, the following set of rules is a *Datalog*$^{\exists}$-program:

$$\begin{array}{l} person(john). \quad person(joe). \\ r_1 : \exists Y : father(X,Y) \leftarrow person(X). \quad r_2 : person(Y) \leftarrow father(X,Y). \end{array} \tag{4}$$

Next, we recall *homomorphisms* as used for defining *Datalog*$^{\exists}$-semantics and query answering over *Datalog*$^{\exists}$-programs. A *homomorphism* is a mapping $h \colon \mathcal{N} \cup \mathcal{V} \to \mathcal{C} \cup \mathcal{V}$. For a homomorphism h, let $h|_S$ be its restriction to $S \subseteq \mathcal{N} \cup \mathcal{V}$, i.e., $h|_S(X) = h(X)$ if $X \in S$ and is undefined otherwise. For any atom a, let $h(a)$ be the atom where each variable and null value V in a is replaced by $h(V)$; this is likewise extended to $h(S)$ for sets S of atoms and/or vectors of terms. A homomorphism h is a *substitution*, if $h(N) = N$ for all $N \in \mathcal{N}$. An atom a is *homomorphic* (*substitutive*) to atom b, if some homomorphism (substitution) h exists such that $h(a) = b$. An isomorphism between two atoms a and b is a bijective homomorphism h s.t. $h(a) = b$ and $h^{-1}(b) = a$.

A set M of atoms is a model of a $\mathit{Datalog}^{\exists}$-program Π, denoted $M \models \Pi$, if $h(B(r)) \subseteq M$ for some substitution h and $r \in \Pi$ of form (3) implies that $h|_{\mathbf{X}}(H(r))$ is substitutive to some atom in M; the set of all models of Π is denoted by $\mathit{mods}(\Pi)$.

Next, we can introduce queries over $\mathit{Datalog}^{\exists}$-programs. A *conjunctive query* (CQ) q is an expression of form $\exists \mathbf{Y} : \leftarrow \mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$, where $\mathbf{Y}$ and $\mathbf{X}$ (the free variables) are disjoint sets of variables and $\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]$ is a conjunction of ordinary atoms containing all and only the variables $\mathbf{X} \cup \mathbf{Y}$.

The answer of a CQ q with free variables $\mathbf{X}$ wrt. a model M is defined as follows:

$$\mathit{ans}(q, M) = \{h|_{\mathbf{X}} \mid h \text{ is a substitution and } h(\mathbf{conj}[\mathbf{X} \cup \mathbf{Y}]) \subseteq M\}.$$

Intuitively, this is the set of assignments to the free variables such that the query holds wrt. the model. The answer of a CQ q wrt. a program Π is then defined as the set $\mathit{ans}(q, \Pi) = \bigcap_{M \in \mathit{mods}(\Pi)} \mathit{ans}(q, M)$.

Query answering can be carried out over some *universal model* U of the program that is embeddable into each of its models by applying a suitable homomorphism. Formally, a model U of a program Π is called *universal* if, for each $M \in \mathit{mods}(\Pi)$, there is a homomorphism h s.t. $h(U) \subseteq M$. Thus, a universal model may be obtained using null values for unnamed individuals introduced by existential quantifiers. Moreover, it can be used to answer any query according to the following proposition [10]:

Proposition 2 ([10]). *Let U be a universal model of $\mathit{Datalog}^{\exists}$-program Π. Then, for any CQ q, it holds that $h \in \mathit{ans}(q, \Pi)$ iff $h \in \mathit{ans}(q, U)$ and $h : \mathcal{V} \to \mathcal{C} \setminus \mathcal{N}$.*

Intuitively, the set of all answers to q wrt. U which map all variables to non-null constants is exactly the set of answers to q wrt. Π.

Example 7. Let Π be the program consisting of rules (4). The CQ $\exists Y :\leftarrow \mathit{person}(X), \mathit{father}(X, Y)$ asks for all persons who have a father. The model $U = \{\mathit{person}(\mathit{john}), \mathit{person}(\mathit{joe}), \mathit{father}(\mathit{john}, \omega_1), \mathit{father}(\mathit{joe}, \omega_2), \mathit{person}(\omega_1), \mathit{person}(\omega_2), \ldots\}$ is a universal model of Π. Hence, $\mathit{ans}(q, \Pi)$ contains answers $h_1(X) = \mathit{john}$ and $h_2(X) = \mathit{joe}$.

Thus, computing a universal model is a key issue for query answering. A common approach for this step is the chase procedure. Intuitively, it starts from an empty interpretation and iteratively adds the head atoms of all rules with satisfied bodies, where existentially quantified variables are substituted by fresh nulls. However, in general this procedure does not terminate. Thus, a restricted *parsimonious chase procedure* was introduced in [17], which derives less atoms, and which is guaranteed to terminate for the class of *Shy-programs*. Moreover, it was shown that the interpretation computed by the parsimonious chase procedure is, although not a model of the program in general, still sound and complete for query answering and a *bounded model* in our view.

For query answering over $\mathit{Datalog}^{\exists}$-programs we reuse the translation in Section 4.

Example 8. Consider the $\mathit{Datalog}^{\exists}$-program Π and its HEX translation $T_{\exists}(\Pi)$:

$$\begin{array}{ll}
\Pi: & T_{\exists}(\Pi): \\
\quad person(john). \quad person(joe). & \quad person(john). \quad person(joe). \\
\quad \exists Y: father(X,Y) \leftarrow person(X). & \quad father(X,Y) \leftarrow person(X), \\
\quad person(Y) \leftarrow father(X,Y). & \qquad\qquad \&exists^{1,1}[r_1, X](Y). \\
 & \quad person(Y) \leftarrow father(X,Y).
\end{array}$$

Intuitively, each person X has some unnamed father Y of X which is also a person.

Note that $T_{\exists}(\Pi)$ is *not* de-safe in general. However, with the hooks in Algorithm BGroundHEX one can still guarantee termination. Let $\mathsf{GroundDatalog}^{\exists}(\Pi, k) = \mathsf{BGroundHEX}(T_{\exists}(\Pi))$ where Repeat tests for $i < k + 1$ where k is the number of existentially quantified variables in the query, and $\mathsf{Evaluate}(PIT_i, x) = true$ iff atom x is *not* homomorphic to any $a \in PIT_i$.

The produced program has a single answer set, which essentially coincides with the result of $pChase$ [17] that can be used for query answering. Thus, query answering over Shy-programs is reduced to grounding and solving of a HEX-program.

Proposition 3. *For a Shy-program Π, $\mathsf{GroundDatalog}^{\exists}(\Pi, k)$ has a unique answer set which is sound and complete for answering CQs with up to k existential variables.*

The main difference to $pChase$ in [17] is essentially due to the homomorphism check. Actually, $pChase$ instantiates existential variables in rules with satisfied body to new null values only if the resulting head atom is not homomorphic to an already derived atom. In contrast, our algorithm performs the homomorphism check for the input to $\&exists^{n,m}$ atoms. Thus, homomorphisms are detected when constants are cyclically sent to the external atom. Consequently, our approach may need one iteration more than $pChase$, but allows for a more elegant integration into our algorithm.

Example 9. For the program and query from Example 8, the algorithm computes a program with answer set $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2), person(\omega_1), person(\omega_2)\}$. In contrast, $pChase$ would stop already earlier with the interpretation $\{person(john), person(joe), father(john, \omega_1), father(joe, \omega_2)\}$ because $person(\omega_1), person(\omega_2)$ are homomorphic to $person(john), person(joe)$.

More formally, one can show that $\mathsf{GroundDatalog}^{\exists}(\Pi, k)$ yields, for a Shy-program Π, a program with a single answer set that is equivalent to $pChase(\Pi, k + 1)$ in [17]. Lemma 4.9 in [17] implies that the resulting answer set can be used for answering queries with k different existentially quantified variables, which proves Proposition 3.

While $pChase$ intermingles grounding and computing a universal model, our algorithm cleanly separates the two stages; modularized program evaluation by the solver will however also effect such intermingling. We nevertheless expect the more clean separation to be advantagagoues for extending Shy-programs to programs that involve existential quantifiers and other external atoms, which we leave for future work.

6 HEX-Programs with Function Symbols

In this section we show how to process terms with function symbols by a rewriting to de-safe HEX-programs. We will briefly discuss advantages of our approach compared to a direct implementation of function symbols.

We consider HEX-programs, where the arguments X_i for $1 \leq i \leq \ell$ of ordinary atoms $p(X_1, \dots, X_\ell)$, and the constant input arguments in $\mathbf{X}$ and the output $\mathbf{Y}$ of an external atom $\&g[\mathbf{X}](\mathbf{Y})$ are from a set of *terms* $\mathcal{T}$, that is the least set $\mathcal{T} \supseteq \mathcal{V} \cup \mathcal{C}$ such that $f \in \mathcal{C}$ (constant symbols are also used as function symbols) and $t_1, \dots, t_n \in \mathcal{T}$ imply $f(t_1, \dots, t_n) \in \mathcal{T}$.

Following [4], we introduce for every $k \geq 0$ two external predicates $\&comp_k$ and $\&decomp_k$ with $ar_{\mathrm{I}}(\&comp_k) = 1 + k$, $ar_{\mathrm{O}}(\&comp_k) = 1$, $ar_{\mathrm{I}}(\&decomp_k) = 1$, and $ar_{\mathrm{O}}(\&decomp_k) = 1 + k$. We define

$$f_{\&comp_k}(\mathbf{A}, f, X_1, \dots, X_k, T) = f_{\&decomp_k}(\mathbf{A}, T, f, X_1, \dots, X_k) = 1,$$

iff $T = f(X_1, \dots, X_k)$.

Composition and decomposition of function terms can be simulated using these external predicates. Function terms are replaced by new variables and appropriate additional external atoms with predicate $\&comp_k$ or $\&decomp_k$ in rule bodies to compute their values. More formally, we introduce the following rewriting.

For any HEX-program Π with function symbols, let $T_f(\Pi)$ be the HEX-program where each occurrence of a term $t = f(t_1, \dots, t_n)$ in a rule r such that $B(r) \neq \emptyset$ is recursively replaced by a new variable V, and if V occurs afterwards in $H(r)$ or the input list of an external atom in $B(r)$, we add $\&comp_n[f, t_1, \dots, t_n](V)$ to $B(r)$; otherwise (i.e., V occurs afterwards in some ordinary body atom or the output list of an external atom), we add $\&decomp_n[V](f, t_1, \dots, t_n)$ to $B(r)$.

Intuitively, $\&comp_n$ is used to construct a nested term from a function symbol and arguments, which might be nested terms themselves, and $\&decomp_n$ is used to extract the function symbol and the arguments from a nested term. The translation can be optimized wrt. evaluation efficiency, but we disregard this here for space reasons.

Example 10. Consider the HEX-program Π with function symbols and its translation:

$$\begin{array}{rl@{\qquad}rl}
\Pi: & q(z).\ q(y). & T_f(\Pi): & q(z).\ q(y). \\
p(f(f(X))) & \leftarrow q(X). & p(V) & \leftarrow q(X), \&comp_1[f, X](U), \\
 & & & \quad \&comp_1[f, U](V). \\
r(X) & \leftarrow p(X). & r(X) & \leftarrow p(X). \\
r(X) & \leftarrow r(f(X)). & r(X) & \leftarrow r(V), \&decomp_1[V](f, X).
\end{array}$$

Intuitively, $T_f(\Pi)$ builds $f(f(X))$ for any X on which q holds using two atoms over $\&comp_1$, and it extracts terms X from derived $r(f(X))$ facts using a $\&decomp_1$-atom.

Note that $\&decomp_n$ supports a well-ordering on term depth such that its output has always a strictly smaller depth than its inputs. This is an important property for proving finite groundability of a program by exploiting the TBFs introduced in [6].

Example 11. The program $\Pi = \{q(f(f(a)));\ q(X) \leftarrow q(f(X))\}$ is translated to $T_f(\Pi) = \{q(f(f(a)));\ q(X) \leftarrow q(V), \&decomp_1[V](f, X)\}$. Since $\&decomp_1$ supports a well-ordering, the cycle is *benign* [6], i.e., it cannot introduce infinitely many values because the nesting depth of terms is strictly decreasing with each iteration.

The realization of function symbols via external atoms (which can in fact also be seen as domain-specific existential quantifiers) has the advantage that their processing can be controlled. For instance, the introduction of new nested terms may be restricted by additional conditions which can be integrated in the semantics of the external predicates

$\&comp_k$ and $\&decomp_k$. A concrete example is *data type checking*, i.e., testing whether the arguments of a function term are from a certain domain. In particular, values might also be rejected, e.g., bounded generation up to a maximal term depth is possible. Another example is to compute some of the term arguments automatically from others, e.g., constructing the functional term $num(7, vii)$ from 7, where the second argument is the Roman representation of the first one.

Another advantage is that the use of external atoms for functional term processing allows for exploiting de-safety of HEX-programs to guarantee finiteness of the grounding. An expressive framework for handling domain-expansion safe programs [6] can be reused without the need to enforce safety criteria specific for function terms.

7 Discussion and Conclusion

We presented model computation and query answering over HEX-programs with domain-specific existential quantifiers, based on external atoms and a new grounding algorithm. In contrast to usual handling of existential quantifiers, ours especially allows for an easy integration of extensions such as additional constraints (even of non-logical nature) or data types. This is useful e.g. for model building applications where particular data is needed for existential values, and gives one the possibility to implement domain-restricted quantifiers and introduce null values, as in databases. The new grounding algorithm allows for controlled bounded grounding; this can be exploited for *bounded model generation*, which might be sufficient (or convenient) for applications. Natural candidates are configuration or, at an abstract level, generating finite models of general first-order formulas as in [12], where an incremental computation of finite models is provided by a translation into incremental ASP. There, grounding and solving is interleaved by continously increasing the bound on the number of elements in the domain. (Note that, although not designed for interleaved evaluation, our approach is flexible enough to also mimic exactly this technique with suitable external atoms.) The work in [1] aims at grounding first-order sentences with complex terms such as functions and aggregates for model expansion tasks. Similar to ours, it is based on bottom-up computation, but we do not restrict to finite structures and allow for potentially infinite domains. As a show case, we considered purely logical existentials (null values), for which our grounding algorithm amounts to a simulation of the one in [17] for $Datalog^{\exists}$-programs. However, while [17] combine grounding and model building, our approach clearly separates the two steps; this may ease possible extensions.

We then realized function symbol processing as in [4], by using external atoms to manipulate nested terms. In contrast to other approaches, no extension of the reasoner is needed for this. Furthermore, using external atoms has the advantage that nested terms can be subject to (even non-logical) constraints given by the semantics of the external atoms, and that finiteness of the grounding follows from de-safety of HEX-programs.

In model-building over HEX$^{\exists}$-programs, we can combine existentials with function symbols, as HEX$^{\exists}$-programs can have external atoms in rule bodies. To allow this for query answering over $Datalog^{\exists}$-programs remains to be considered. More generally, also combining existentials with arbitrary external atoms and the use of default-negation in presence of existentials is an interesting issue for future research. This leads to *non-monotonic existential rules*, which most recently are considered in [18] and in [15], which equips the $Datalog^{\pm}$formalism, which is tailored to ontological knowledge representation and tractable query answering, with well-founded negation. Another line for future research is to allow disjunctive rules and existential quantification as in $Datalog^{\exists,\vee}$ [2], leading to a generalization of the class of Shy-programs. Continuing on the work on guardedness conditions as in open answer set programming [16], $Datalog^{\exists}$, and $Datalog^{\pm}$ should prove useful to find important techniques for constructing more expressive variants of HEX-programs with domain-specific existential quantifiers. The separation of grounding and solving in our approach should be an advantage for such enhancements.

References

1. Aavani, A., Wu, X.N., Ternovska, E., Mitchell, D.: Grounding formulas with complex terms. In: Canadian AI. pp. 13–25. Springer (2011)
2. Alviano, M., Faber, W., Leone, N., Manna, M.: Disjunctive datalog with existential quantifiers: semantics, decidability, and complexity issues. TPLP 12(4-5), 701–718 (2012)
3. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Commun. ACM 54(12), 92–103 (2011)
4. Calimeri, F., Cozza, S., Ianni, G.: External Sources of Knowledge and Value Invention in Logic Programming. Ann. Math. Artif. Intell. 50(3–4), 333–361 (2007)
5. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Grounding HEX-Programs with Expanding Domains (2013), Manuscript, submitted
6. Eiter, T., Fink, M., Krennwallner, T., Redl, C.: Liberal safety for answer set programs with external sources. In: AAAI 2013, pp. 267–275. AAAI Press (2013)
7. Eiter, T., Ianni, G., Schindlauer, R., Tompits, H.: A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In: IJCAI. pp. 90–96. (2005)
8. Eiter, T., Simkus, M.: FDNC: Decidable nonmonotonic disjunctive logic programs with function symbols. ACM Trans. Comput. Log. 11(2), 14:1–14:50 (2010)
9. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artif. Intell. 175(1), 278–298 (2011)
10. Fagin, R., Kolaitis, P., Miller, R., Popa, L.: Data Exchange: Semantics and Query Answering. Theor. Comput. Sci. 336(1), 89–124 (2005)
11. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187–188, 52–89 (2012)
12. Gebser, M., Sabuncu, O., Schaub, T.: An incremental answer set programming based system for finite model computation. AI Commun. 24(2), 195–212 (2011)
13. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generat. Comput. 9(3–4), 365–386 (1991)
14. Grau, B.C., Horrocks, I., Krötzsch, M., Kupke, C., Magka, D., Motik, B., Wang, Z.: Acyclicity conditions and their application to query answering in description logics. In: KR 2012.
15. Hernich, A., Kupke, C., Lukasiewicz, T., Gottlob, G.: Well-founded semantics for extended datalog and ontological reasoning. In: PODS 2013, pp. 225–236. ACM (2013)
16. Heymans, S., Nieuwenborgh, D.V., Vermeir, D.: Open answer set programming with guarded programs. ACM Trans. Comput. Logic 9(4), 26:1–26:53 (2008)
17. Leone, N., Manna, M., Terracina, G., Veltri, P.: Efficiently computable datalog$^{\exists}$ programs. In: KR 2012. AAAI Press.
18. Magka, D., Krötzsch, M., Horrocks, I.: Computing Stable Models for Nonmonotonic Existential Rules. In: IJCAI 2013. AAAI Press, to appear

An Abductive Paraconsistent Semantics – MH_P

Mário Abrantes[1]* and Luís Moniz Pereira[2]

[1] Escola Superior de Tecnologia e de Gestão
Instituto Politécnico de Bragança
Campus de Santa Apolónia, 5300-253 Bragança, Portugal
mar@ipb.pt
[2] Centro de Inteligência Artificial (CENTRIA), Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal
lmp@fct.unl.pt

Abstract. In this paper we present a paraconsistent abdutive semantics for extended normal logic programs, the *paraconsistent minimal hypotheses* semantics MH_P. The MH_P is a semantics of *total paraconsistent models* wich combines the merits of two already existing semantics: it inherits the existence property of the abductive *minimal hypotheses* semantics MH [1], which is a semantics of *total models*, and the property of detection of support on contradiction of the paraconsistent *well-founded semantics with explicit negation* $WFSX_P$ [2], which is a semantics of *partial paraconsistent models*. The MH_P enjoys also the property of *simple relevance*, which permits top-down query answering for brave reasoning purposes. Besides, the MH_P lends itself to various types of skeptical and brave reasoning, which include the possibility of drawing conclusions from inconsistent models in a nontrivial way. The MH_P coincides with the MH on normal logic programs, and with the $WFSX_P$ on stratified extended programs.

Keywords: Hypotheses, Semantics, Abduction, Total Paraconsistent Model, Partial Paraconsistent Model, Paraconsistency.

1 Introduction

In this work we present an abduction based paraconsistent semantics, for extended logic programs, the MH_P, wich combines the merits of two already existing semantics: an abductive semantics, the *minimal hypotheses* semantics MH [1], and a paraconsistent semantics, the *well-founded semantics with explicit negation* $WFSX_P$ [2]. We first expound on the general merits of abductive semantics, and then highlight the advantages of paraconsistent semantics.

Abductive logic programming is an extension of logic programming to perform abductive reasoning [3]. The following example gives a glimpse on how it can be used to get a 2-valued semantics for every normal logic program.

* Partially supported by Fundação para a Ciência e Tecnologia and Instituto Politécnico de Bragança grant PROTEC : SFHR/49747/2009.

Example 1. In order for program $P = \{a \leftarrow not\ b,\ b \leftarrow not\ c,\ c \leftarrow not\ a\}$, that has no stable models, to have a 2-valued model, a subset of $\{a, b, c\}$ must be a part of the positive literals of the model. This can be achieved by considering *abductive extensions* [3] $P \cup H$ of program P, where H is a subset of $\{a, b, c\}$ such that $P \cup H$ has a single stable model [4]. For example, if we consider the explanation $H = \{a\}$, the stable model obtained for $P \cup H$ is $\{a, b\}$. In case we take, for instance, the explanation $\{b\}$ (resp. $\{c\}$), we get the model $\{b, c\}$ (resp. $\{c, a\}$). Each set H is called *hypotheses set* [1] for the corresponding model, and is an *abductive explanation* [3] for program P.

Abduction allows us to envisage loops in normal logic programs as semantic choice devices. This is one of the main features of the *minimal hypotheses* semantics MH, presented in section 3. MH takes as assumable hypotheses of a normal logic program P the atoms that appear default negated in the *layered remainder* $\mathring{P}$, which is a transformed of the original program P. In the example above, as $P = \mathring{P}$, the MH models of program P are $\{a, b\}$, $\{b, c\}$, $\{c, a\}$, with hypotheses sets respectively, $\{a\}$, $\{b\}$, $\{c\}$. The hypotheses sets are minimal with respect to set inclusion. We next highlight the advantages of having paraconsistent semantics. Several authors, [5–10], have stressed the need to endow normal logic programs with a second kind of negation operator, the explicit negation '$\neg$', for representing contradictory knowledge, in addition to the negation-as-failure (or default negation) operator, '*not*', used for representing incomplete information. There are plenty of examples that display the need for explicitly negated literals in logic programming, both in the heads and in the bodies of the rules. The following is a typical example.

Example 2. (adapted from [11]) Consider the statement "Penguins do not fly". This statement may be represented within logic programming by $no\_fly(X) \leftarrow penguin(X)$. Meanwhile, if additionally we wish to represent the statement "Birds fly", $fly(X) \leftarrow birds(X)$, no connection results between the predicates $no\_fly(X)$ and $fly(X)$, although the intention of the programmer is to set them as contradictory. In this case it is suitable to have the rule $\neg fly(X) \leftarrow penguin(X)$ instead of $no\_fly(X) \leftarrow penguin(X)$, since a semantics that deals with the operator $\neg$ will by definition consider predicates $fly(X)$ and $\neg fly(X)$ as contradictory opposites.

As a consequence of the need for an explicit negation operator, a number of semantics that interpret this type of operator have been proposed – those for *extended normal logic programs*, *ELPs*. Among them are the *paraconsistent* semantics [2, 12–15], i.e. semantics that admit non trivial models containing contradictory literals, say l and $\neg l$. This has been shown an important property for frameworks of knowledge and reasoning representation. The *well-founded semantics with explicit negation* $WFSX_P$, is a paraconsistent semantics for extended normal logic programs that envisages default negation and explicit negation necessarily related through the *coherence principle* [7]: if $\neg l$ holds, then $not\ l$ should also hold (similarly, if l then $not\ \neg l$). In section 5 we define the *paraconsistent minimal hypotheses* semantics MH_P. As exposed there, the MH_P semantics endows $WFSX_P$ with choice mechanisms, in the fashion of MH, that allow reasoning by cases. MH_P models are total paraconsistent models, meaning that no MH_P model of an extended normal logic program contains undefined literals. MH_P inherits the advantages of the MH semantics, being existential, using loops as choice mechanisms and being *simple relevant* (see subsection 5.1). It also inherits the advantages of the $WFSX_P$ by not enforcing default consistency and hence producing models that allow to spot support on contradiction.

The rest of this paper proceeds as follows. In section 2 we define the language of extended normal logic programs and the terminology to be used in the sequel. For self-containment we present in sections 3 and 4 the definitions of the MH semantics and $WFSX_P$ semantics. In section 5 we exhibit in technical detail the definiton and characterization of the MH_P semantics. Section 6 is dedicated to conclusions and future work.

2 Language and Terminology of Logic Programs

An *extended normal logic program* is a finite set of ground rules, each one of the form $l_0 \leftarrow l_1, \cdots, l_m, not\ l_{m+1}, \cdots, not\ l_n$, where l_i, $0 \leq i \leq n$, is an *objective literal* (either an atom b or its explicit negation, $\neg b$, where $\neg\neg b = b$); m, n are natural numbers, the operator ',' stands for the classical conjunctive connective and the operator *not* stands for default negation (*not l* is called a *default literal*). The set of all atoms that appear in an extended normal logic program is named the *Herbrand base* of P, denoted $\mathcal{H}_P$. If $m = n = 0$ a rule is called *fact*. If $\mathcal{H}_P$ contains no explicitly negated literals, the program is called *normal*; a rule with no explicitly negated literals is also called normal. Given a program P, program Q is a *subprogram* of P if $Q \subseteq P$, where Q and P are envisaged as sets of rules. Given a rule $r = l_0 \leftarrow l_1, \cdots, l_m, not\ l_{m+1}, \cdots, not\ l_n$, the objective literal l_0 is the *head* of the rule and $l_1, \cdots, l_m, not\ l_{m+1}, \cdots, not\ l_n$ is the *body* of the rule.[3] Some terminology used in the sequel is now established, concerning the dependencies among the elements (atoms and rules) of ground normal logic programs, triggered by the dependency operator '$\leftarrow$'.

Complete Rule Graph. (adapted from [4]) The *complete rule graph* of an extended normal logic program P, denoted by $CRG(P)$, is the directed graph whose vertices are the rules of P. Two vertices representing rules r_1 and r_2

[3] For ease of exposition, we henceforth use the following abbreviations: $Atoms(E)$, is the set of all *atoms* that appear in the ground structure E, where E can be a rule, a set of rules, a set of logic expressions, etc; $Bodies(E)$, is the set of all bodies that appear in the set of rules E; if E is unitary, we may use '$Body$' instead of '$Bodies$'; $Heads(E)$, is the set of all atoms that appear in the heads of the set of rules E; if E is unitary, we may use '$Head$' instead of '$Heads$'; $Facts(E)$, is the set of all facts that appear in the set of rules E; $Loops(E)$, is the set of all rules that appear involved in some loop contained in the set of rules E (see definition of *loop* in the sequel). We may compound some of these abbreviations, as for instance $Atoms(Bodies(P))$ whose meaning is immediate.

define an arc from r_1 to r_2, iff $Head(r_1) \in Atoms(Body(r_2))$, or $Head(r_1) = \neg Head(r_2)$, or $Head(r_2) = \neg Head(r_1)$.
Subprogram Relevant to an Objective Literal. (adapted from [4]) Let P be an extended normal logic program. We say that a rule $r \in P$ is *relevant* to an objective literal $l \in \mathcal{H}_P$, iff there is a rule s such that $Heads(s) = l$, and there is a direct path from r to s in the complete rule graph of P. In particular, rule s is relevant to l. The set of all rules of P relevant to l is represented by $Rel_P(l)$, and is named *subprogram relevant* to l.
Loop. (adapted from [16]) We say that a finite set of normal ground rules P forms a loop, iff P is of the form $\{h_1 \leftarrow l_2, B_1,\ h_2 \leftarrow l_3, B_2,\ \cdots,\ h_n \leftarrow l_1, B_n\}$, where $l_i = h_i$, or $l_i = not\ h_i$, and each B_i stands for a conjunction (possibly empty) of a finite number of literals. We say that each rule $h_i \leftarrow l_{i+1}, B_i$ is *involved* in the loop *through the literal* l_{i+1} or *through the atom* involved in the literal l_{i+1}, where $i+1$ is replaced by 1 if $i = n$.

Given a 3-valued interpretation I of an extended normal logic program, we represent by I^+ (resp. I^-) the set of its positive objective literals (resp. objective literals whose default negation is true with respect to I), and by I^u the set of undefined objective literals with respect to I. We represent I by the 3-tuple $I = \langle I^+, I^u, I^- \rangle$.[4]

The following operator shall be used in the sequel.

Definition 1. $\triangle$ operator. Given a normal logic program Q, we denote by $\triangle Q$ the 3-valued interpretation that can be read from Q in the following way: $b \in (\triangle Q)^+$ iff $(b \leftarrow) \in Q$; $b \in (\triangle Q)^u$ iff $(b \leftarrow) \notin Q$ and there is a rule r in Q such that $Head(r) = b$; $b \in (\triangle Q)^-$ iff b has no rule in Q.

3 The MH Semantics

In [17] the authors propose a reduction system comprised of the following five operators, each of which transforming normal logic programs into normal logic programs while keeping invariant the *well-founded model* [18] of the involved programs: *positive reduction*, $\mapsto_P$, *negative reduction*, $\mapsto_N$, *success*, $\mapsto_S$, *failure*, $\mapsto_F$, and *loop detection*, $\mapsto_L$ [5]. We here represent this reduction system by $\mapsto_{WFS} := \mapsto_P \cup \mapsto_N \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$. Given a normal logic program P, the transformation $P \mapsto^*_{WFS} \widehat{P}$ [6] is such that $WFM(\widehat{P}) = WFM(P)$. Program $\widehat{P}$ will be here called the WFS *remainder* of P or *remainder* of P [7]. The system $\mapsto_{WFS}$ is both *terminating* and *confluent*, meaning that for any finite ground normal logic program the number of operations needed to reach $\widehat{P}$ is finite,

[4] We also write $b = +$, $b = u$, $b = -$, to mean respectively, $b \in I^+$, $b \in I^u$, $b \in I^-$.
[5] See INAP 11 paper [1], defs. 8, 9, 11, 12, 13, for definitions of these operators.
[6] Where $\mapsto^*_{WFS}$ means the nondeterministic performing of operations of the system $\mapsto_{WFS}$, until the resulting program becomes invariant.
[7] See INAP 11 paper [1], def. 16.

and the order in which the operations are executed is irrelevant, as long as the preconditions for the application of each one of them are verified. To compute the MH semantics of a normal logic program, we need the variant $\mapsto_{LWFS}$ of the reduction system $\mapsto_{WFS}$, which results from $\mapsto_{WFS}$ by substituting the negative reduction operator, $\mapsto_N$, by the *layered negative reduction* operator, $\mapsto_{LN}$ – see [1], def. 10. Given a normal logic program P, the transformation $P \mapsto^*_{LWFS} \mathring{P}$ [8] is such that $\triangle\mathring{P} = LWFS(P)$, where $LWFS(P)$ [4] stands for the *layered well-founded model* of P – the transformed program $\mathring{P}$ is called *$LWFS$ remainder* or *layered remainder* of P. The system $\mapsto_{LWFS}$ is terminating and confluent when applied to finite ground normal logic programs.

Example 3. The layered remainder of program P (left column) is program $\mathring{P}$ (center column), and the remainder of P is program $\widehat{P}$ (right column) – rules and literals stripped out, are eliminated during $\mathring{P}$ and $\widehat{P}$ computations.[9]

$b \leftarrow h$	$b \leftarrow h$	$b \leftarrow$ ~~h~~
$h \leftarrow not\ p, b$	$h \leftarrow not\ p,$ ~~b~~	$h \leftarrow$ ~~$not\ p, b$~~
$p \leftarrow not\ b$	$p \leftarrow not\ b$	~~$p \leftarrow not\ b$~~
$a \leftarrow not\ c, b$	$a \leftarrow$ ~~$not\ c, b$~~	$a \leftarrow$ ~~$not\ c, b$~~
$d \leftarrow not\ b$	~~$d \leftarrow not\ b$~~	~~$d \leftarrow not\ b$~~
$b \leftarrow$	$b \leftarrow$	$b \leftarrow$

The layered well-founded model of P is thus $LWFM(P) = \triangle\mathring{P} = \langle\{a, b\}^+, \{h, p\}^u, \{c, d\}^-\rangle$, and the well-founded model of P is $WFM(P) = \triangle\widehat{P} = \langle\{a, b, h\}^+, \{\}^u, \{c, d, p\}^-\rangle$.

MH being an abductive semantics, we shall now define the *assumable hypotheses set* and the *minimal hypotheses* model of a normal logic program.

Definition 2. Assumable Hypotheses Set of a Program. (adapted from [1]) Let P be a finite ground normal logic program. We write $Hyps(P)$ to denote the *assumable hypotheses set* of P: those atoms that appear default negated in the bodies of rules of $\mathring{P}$ and which are not facts in $\mathring{P}$, i.e. they belong to $(\triangle\mathring{P})^u$.

[8] $\mapsto^*_{LWFS}$ means the nondeterministic performing of operations of the system $\mapsto_{LWFS}$ until the resulting program becomes invariant.

[9] Rule $d \leftarrow not\ b$ in P is eliminated, both in $\mathring{P}$ and $\widehat{P}$, by layered negative reduction in the first case and by negative redution in the second – the two operations coincide here, since the rule is not in loop via literal *not* b; the body of rule $a \leftarrow not\ c, b$ in P, becomes empty by success (which eliminates b) and by positive reduction (which eliminates *not* c), both in $\mathring{P}$ and $\widehat{P}$; rule $p \leftarrow not\ b$ in is eliminated in $\widehat{P}$ by negative reduction, but not in $\mathring{P}$, since the rule is in loop through literal *not* b and layered negative redution does nothing is such cases; b is eliminated from the body of rule $h \leftarrow not\ p, b$ by success, both in $\mathring{P}$ and $\widehat{P}$, whilst positive reduction eliminates also the literal *not* p in $\widehat{P}$, because p is an atom without rule due to the elimination of rule $p \leftarrow not\ b$, turning h into a fact; h is eliminated from the body of rule $b \leftarrow h$, by success, in $\widehat{P}$.

The purpose of computing $\overset{\circ}{P}$ is to find the set of assumable hypotheses of P, which are then used to compute the minimal hypoteses models of the program.

Definition 3. Minimal Hypotheses Model. (adapted from [1]) Let P be a finite ground normal logic program. Let $Hyps(P)$ be the assumable hypotheses set of P (cf. def. 2), and H a subset of $Hyps(P)$. A 2-valued interpretation M of P is a *minimal hypotheses* model of P, iff $WFM^u(P \cup H) = \emptyset$, where $H = \emptyset$ or H is a nonempty set that is minimal with respect to set inclusion (set inclusion minimality disregards any empty H). I.e. the hypotheses set H is minimal but sufficient to determine (via the well-founded model) the truth-value of all literals in the program.

Theorem 1. Every normal logic program has at least one MH model.

Every stable model of a normal logic program is also a minimal hypotheses model of the program. This justifies the catering for whole models with empty hypotheses set, which are stable models of programs whose layered remainders are stratified programs. The reason hypotheses minimization does not contemplate empty hypotheses set models, is to allow loops to be taken as choice devices, also in these cases. For instance, program P in example 3 has the assumable hypotheses set $Hyps(P) = \{p\}$, since $not\ p$ appears in $\overset{\circ}{P}$ and p is not a fact of this program [10]. The MH models of P are $\{a, b, not\ c, h, not\ p\}$ with hypotheses set $\emptyset$, and $\{a, b, not\ c, not\ h, p\}$ with hypotheses set $\{p\}$. If the empty hypotheses set were allowed in the hypotheses minimization, the non-empty hypotheses set model would be discarded and we would be left with just the stable model $\{a, b, not\ c, h, not\ p\}$.

Example 4. Consider the following variation of the *vacation problem*[11] [1], $P = \{a \leftarrow not\ b,\ b \leftarrow not\ a, not\ c,\ c \leftarrow not\ d,\ d \leftarrow not\ e, not\ a,\ e \leftarrow not\ a, not\ c\}$, where $P = \overset{\circ}{P}$. The hypotheses set of P is $Hyps(P) = \{a, b, c, d, e\}$. The MH models of P are: $\{a, not\ b, c, not\ d, not\ e\}$ with hypotheses set $\{a\}$, $\{not\ a, b, not\ c, d, e\}$ with hypotheses set $\{b, d\}$[12] and $\{a, not\ b, c, not\ d, e\}$ with hypotheses set $\{e\}$.

This example shows this type of problem is not solvable by resorting to answer sets semantics [19], if we stick to the set of rules of P, since models may not be minimal (e.g. $\{a, not\ b, c,\ not\ d, e\}$ above). Should there be a transformation on normal logic programs, let it be $\mapsto_Y$, such that $P \mapsto_Y P^*$, where the MH models of P could be extracted from the stable models of P^*, then P^* would have a different set of rules and/or a different language, with respect to P, which

[10] Notice that although $not\ b$ appears in $\overset{\circ}{P}$, b is not an assumable hypothesis of P since it is a fact of $\overset{\circ}{P}$.

[11] Five friends are planning a joint vacation. First friend says "If we don't go to place b, then we should go to place a", which corresponds to rule $a \leftarrow not\ b$; the same rationale for the remaining rules.

[12] There are no MH models with hypotheses sets $H = \{b\}$ or $H = \{d\}$, since in these cases $WFM^u(P \cup H) \neq \emptyset$.

means that this type of problem is specified in a more elegant way if the solution is to be obtained via the MH semantics. Yet, it is an open problem whether such a transformation exists. [13]

4 The $\mathbf{WFSX_P}$ Semantics

The $WFSX_P$ model of an extended normal logic program may be computed by means of a dedicated fixpoint operator [2]. In this section, however, we instead present a definition of the $WFSX_P$ by means of a program transformation for extended normal logic programs, dubbed *$t-o$ transformation*[14] [20], which embeds the $WFSX_P$ into the WFS. This means that the $WFSX_P$ model of an extended normal logic program P, denoted by $WFM_P(P)$, may be extracted from the well-founded model of the transformed program P^{t-o}.

Definition 4. $\mathbf{t-o}$ Transformation. (adapted from [20]) The *$t-o$ transformation*, maps an extended normal logic program P into a normal logic program P^{t-o}, by means of the two following steps:
1. Every explicitly negated literal in P, say $\neg b$, appears also in the transformed program, where it must be read as a new atom, say $\neg\_b$. Let P^* be the program resulting from making these transformations on P.
2. Every rule $r = (Head(r) \leftarrow Body(r))$ in P^* is substituted by the following pair of rules: (i) A rule also designated r, for simplicity, obtained from $r \in P^*$ by placing the superscript 'o' in the default negated atoms of $Body(r)$; (ii) A rule r^o, obtained from $r \in P^*$ by adding to $Body(r)$ the literal $not\ \neg Head(r)$ (where $\neg\neg l = l$)[15], and by placing the superscript 'o' in $Head(r)$ and in every positive literal of $Body(r)$.
We call *co-rules* to each pair r, r^o of rules in P^{t-o} (each rule is the *co-rule* of the other one), and *co-atoms* to each pair b, b^o of atoms in P^{t-o} (each atom is the *co-atom* of the other one). To each objective literal l of the language of P^*, there corresponds the pair of co-atoms l, l^o of the language of P^{t-o} and vice-versa.[16]

[13] The solution proposed by MH semantics for the vacation problem has the following rationale: rule $a \leftarrow not\ b$, for example, states that the first friend prefers place b to place a, because a is suggested in case b fails; each model of MH tries to satisfy the first options of the friends, by considering them as hypotheses. The answer set solution to this type of problem, when it exists, retrives models that statisfy all the friends' demands (rules) with the smallest (with respect to set inclusion) possible number of places to visit (due to the minimality of models).

[14] The original designation is *$T-TU$ transformation*. Our definition alters the notation, for simplicity purposes, while keeping the meaning of the original one.

[15] The literal $not\ \neg Head(r)$ is added to enforce the coherence principle.

[16] The rules with superscript 'o' in the heads are used to derive the literals that are true or undefined in the $WFSXp$ model, and the rules with no superscript in the head are used to derive the true literals of the $WFSXp$ model.

Example 5. (adapted from [20]) Program P (left column) has the $t-o$ transformed P^{t-o} (two columns on the right):

$a \leftarrow$	$a \leftarrow$	$a^o \leftarrow not\ \neg a$
$\neg a \leftarrow$	$\neg a \leftarrow$	$\neg a^o \leftarrow not\ a$
$b \leftarrow a$	$b \leftarrow a$	$b^o \leftarrow a^o, not\ \neg b$
$c \leftarrow not\ b$	$c \leftarrow not\ b^o$	$c^o \leftarrow not\ b, not\ \neg c$
$d \leftarrow not\ d$	$d \leftarrow not\ d^o$	$d^o \leftarrow not\ d, not\ \neg d$

The following theorem states how to read the $WFM_P(P)$ model from the $WFM(P^{t-o})$, where P is any extended normal logic program.

Theorem 2. WFSX$_{\mathbf{P}}$ is embeddable into WFS. (adapted from [20]) Let P be an extended normal logic program. Then the following equivalences hold for an arbitrary atom b of the language of P: $b \in WFMp(P)$, iff $b \in WFM(P^{t-o})$; $not\ b \in WFMp(P)$, iff $not\ b^o \in WFM(P^{t-o})$; $\neg b \in WFMp(P)$, iff $\neg b \in WFM(P^{t-o})$; $not\ \neg b \in WFMp(P)$, iff $not\ \neg b^o \in WFM(P^{t-o})$, where the symbols $\neg b, \neg b^o$ on the right sides of the 'iffs' must be taken as names of atoms (they are not explicitly negated literals), in accordance with definition 4.

Definition 5. $\triangledown$ operator. Given a 3-valued interpretation $I = \langle I^+, I^u, I^- \rangle$, where I^+, I^u, I^- may contain 'o' superscript or otherwise nonsuperscript atoms, we denote by $\triangledown I$ the 3-valued interpretation obtained by means of the lexical correspondences stated in theorem 2.

Using this operator we may write, for example, $WFM_P(P) = \triangledown WFM(P^{t-o})$. The next proposition and corollary characterize the valuations of the pairs of co-atoms b, b^o of the language of P^{t-o}, with respect to the $WFM(P^{t-o})$: it is the case that $b^o \leq_t b$ for any such pair (b^o, b standing here for their valuations with respect to the $WFM(P^{t-o})$), where $\leq_t$ is the truth ordering[17].

Proposition 1. Let P^{t-o} be the $t-o$ transformed of a finite ground extended normal logic program P. Let $\Gamma^n_{P^{t-o}}(\emptyset)$ be the result of the n-th self composition of the Gelfond-Lifschitz Γ operator [21] on program P^{t-o}, with argument $\emptyset$. Then, for every $n \in \mathbb{N}$ and for every atom b of the Herbrand base of P we have $b^o \in \Gamma^n_{P^{t-o}}(\emptyset) \Rightarrow b \in \Gamma^n_{P^{t-o}}(\emptyset)$.

Corollary 1. (of proposition 1) Let b, b^o be two co-atoms of the Herbrand base of P^{t-o}. Then it is not possible to have any of the following three types of valuations with respect to the $WFM(P^{t-o})$: $(b, b^o) = (-, +)$, $(b, b^o) = (-, u)$, $(b, b^o) = (u, +)$. The only possible valuations are $(b, b^o) = (-, -)$, $(b, b^o) = (u, -)$, $(b, b^o) = (+, -)$, $(b, b^o) = (u, u)$, $(b, b^o) = (+, u)$, $(b, b^o) = (+, +)$.

[17] Given the logic values f (false), u (undefined) and t (true), their *truth ordering* is defined by: $f \leq_t u \leq_t t$, [11].

The $WFM(P^{t-o})$ in example 5 is
$\langle\{a, \neg a, b, c\}^+, \{d, d^o\}^u, \{a^o, \neg a^o, b^o, \neg b, \neg b^o, c^o, \neg c, \neg c^o, \neg d, \neg d^o\}^-\rangle$. Using theorem 2 we obtain the $WFSX_P$ model $\langle\{a, \neg a, b, c\}^+, \{d\}^u, \{a, \neg a, b, \neg b, c, \neg c, \neg d\}^-\rangle$. To get the meaning of this model, notice that $WFSX_P$ does not enforce default consistency, i.e. l and *not* l can be simultaneously true in a paraconsistent model, in contradistinction to all other paraconsistent semantics [20], which allows to detect dependence on contradictory information. This is a consequence of adopting the *coherence principle* [22]. In the above example, for any $l \in \{a, \neg a, b, c\}$, both l and *not* l belong to the $WFM_P(P)$, thus revealing the valuations of $\{a, b, c\}$ as inconsistent (the valuation of a is also contradictory with respect to explicit negation, since a and $\neg a$ are both true; is due to this contradictory valuation that the inconsistency of a, b, c occurs). The valuation of atom d is in turn consistent. Table 1 (see appendix) presents in column $NINE$[18] the correspondence between each possible 4-tuple of $WFSX_P$ valuations of a literal $(l^o, l, \neg l^o, \neg l)$ and the nine possible logic values literal l may assume with respect to logic $NINE$. According to table 1, the logic values of the atoms in the $NINE$ model corresponding to the $WFM_P(P)$ above, are as follow: a is contradictory true (logic value I); b, c are true with contradictory belief (logic value II); d is default true (logic value dt).

Definition 6. Total/Partial Paraconsistent Model. (adapted from [23]) Let P be a finite ground extended normal logic program and SEM a semantics for extended normal logic programs. A SEM model $M = \langle M^+, M^u, M^-\rangle$ of P is a *total paraconsistent* model, iff $M^u = \emptyset$. M is called a *partial paraconsistent model*, iff $M^u \neq \emptyset$.

Proposition 2. The $WFSX_P$ semantics is a partial paraconsistent models semantics, meaning that the $WFSX_P$ models of some extended normal logic programs contain undefined literals.

Theorem 3. (adapted from [2]) If P is a normal logic program, then the models $WFM(P)$ and $WFM_P(P)$ are equal, if we neglect the explicitly negated literals in $WFM_P(P)$.

5 The MH_P Semantics

The MH_P semantics of an extended normal logic program P is computed via the following steps: 1) Compute the *balanced layered remainder* bP^{t-o} of P (see definition 9); 2) Compute the *assumable hypotheses set* of P (see definition 10); 3) Compute the MH_P models of P (see definition 11).

The balanced layered remainder bP^{t-o} of an extended normal logic program P, is the outcame of the *balanced layered reduction* transformation of P, $P \mapsto^*_{bLWFM}$

[18] A logic dubbed $NINE$ [20], presented in the appendix, provides a truth-functional model theory for the $WFSXp$ semantics.

bP^{t-o}, where (i) $\mapsto_{bLWFM}$ is obtained from $\mapsto_{LWFM}$ replacing the layered negative reduction operator, $\mapsto_{LN}$, by the *balanced layered negative reduction* operator, $\mapsto_{bLN}$, defined below; (ii) $\mapsto^*_{bLWFM}$ means the nondeterministic performing of operations of the system $\mapsto_{bLWFM}$ until the resulting program becomes invariant under any further operation.

Definition 7. Balanced[19] Layered Negative Reduction. Let P_1 and P_2 be two ground normal logic programs, whose Herbrand bases contain 'o' superscript and nonsuperscript atoms. We say that P_2 results from P_1 by a *balanced layered negative reduction* operation, $P_1 \mapsto_{bLN} P_2$, iff one of the next two cases occur: (1) There is a fact b^o in P_1 and a rule r in P_1 whose body contains the literal *not* b^o, where neither r is involved in a loop through the literal *not* b^o nor is its co-rule r^o involved in a loop through the literal *not* b, and $P_2 = P_1 \setminus \{r\}$; (2) There is a fact b in P_1 and a rule r^o in P_1 whose body contains the literal *not* b, where neither r^o is involved in a loop through the literal *not* b, nor is its co-rule r involved in a loop through the literal *not* b^o, and $P_2 = P_1 \setminus \{r^o\}$.[20]

Definition 8. Balanced Layered Reduction. The *balanced layered reduction* system is the system $\mapsto_{bLWFM} := \mapsto_P \cup \mapsto_{bLN} \cup \mapsto_S \cup \mapsto_F \cup \mapsto_L$.

Theorem 4. Termination and Confluency. The system $\mapsto_{bLWFM}$, when applied to finite ground programs, is both terminating and confluent.

Definition 9. Balanced Layered Remainder. Let P be a finite ground extended normal logic program and P^{t-o} its $t-o$ transformed. We call *balanced layered remainder* of P to the program bP^{t-o} such that $P^{t-o} \mapsto^*_{bLWFM} bP^{t-o}$.

Example 6. The balanced layered remainder of the program P (left column) is bP^{t-o} (two columns on the right) – rules and literals stripped out, are eliminated during bP^{t-o} computation.[21]

$b \leftarrow h$	$b \leftarrow h$	~~$b^o \leftarrow h^o, not\ \neg b$~~
$h \leftarrow not\ p$	$h \leftarrow not\ p^o$	~~$h^o \leftarrow not\ p, not\ \neg h$~~
$p \leftarrow not\ b$	$p \leftarrow not\ b^o$	$p^o \leftarrow not\ b,$ ~~$not\ \neg p$~~
$b \leftarrow$	$b \leftarrow$	$b^o \leftarrow$ ~~$not\ \neg b$~~
$\neg h \leftarrow$	$\neg h \leftarrow$	$\neg h^o \leftarrow not\ h$

[19] The expression 'balanced' refers to the consideration of pairs of co-rules in this definition.

[20] This operation is weaker than layered negative reduction, meaning that where the former is applicable so is the latter.

[21] Notice that rule $h^o \leftarrow not\ p, not\ \neg h$ is eliminated by balanced negative reduction, because $\neg h$ is a fact of bP^{t-o}, and neither is the rule involved in a loop through literal $\neg h$, nor is its co-rule $h \leftarrow not\ p^o$ involved in a loop through literal $\neg h^o$ – balanced layered negative reduction has, in this case, the same effect as negative reduction; rule $b^o \leftarrow h^o, not\ \neg b$ is eliminated by failure, because h^o does not have a rule after elimination of $h^o \leftarrow not\ p, not\ \neg h$; literals $not\ \neg b, not\ \neg p$ are eliminated by positive reduction, since $\neg b, \neg p$ do not have a rule in bP^{t-o}.

Proposition 3. Let P be an extended normal logic program and $M = \triangle bP^{t-o}$. Then the valuation with respect to M of any pair of co-atoms, say l, l^o, of the Herbrand base of P^{t-o}, agrees with the statement of corollary 1.

Definition 10. Assumable Hypotheses Set of a Program. Let P be a finite ground extended normal logic program and bP^{t-o} its balanced layered remainder. We say that $Hyps(P) \subseteq \mathcal{H}_P$ is the *assumable hypotheses set* of P, iff for all $h \in Hyps$ it is the case that the default literal *not* h^o appears in program bP^{t-o}, and h is not a fact in bP^{t-o}, i.e. $h \in (\triangle bP^{t-o})^u$.[22]

Definition 11. Paraconsistent Minimal Hypotheses Semantics, MH$_\mathbf{P}$. Let P be a finite ground extended normal logic program, bP^{t-o} its balanced layered remainder and $Hyps$ the set of assumable hypotheses of P. Then the *paraconsistent minimal hypotheses* semantics of P, denoted $MHp(P)$, is defined by the paraconsistent minimal hypotheses models M of P, which are computed as follows:

1. $M = WFM_P(P \cup H) = \bigtriangledown WFM(P^{t-o} \cup H \cup \{h^o \leftarrow \textit{not}\ \neg h : h \in H\})$, for all $H \subseteq Hyps$, $H \neq \emptyset$, H is inclusion-minimal and $WFM_P^u(P \cup H) = \emptyset$.
2. $M = WFM_P(P) = \bigtriangledown WFM(P^{t-o})$, where $WFM_P^u(P) = \emptyset$.

Each hypothesis, say h, is added to bP^{t-o} as a pair of rules $\{h \leftarrow,\ h^o \leftarrow \textit{not}\ \neg h\}$. No MH_P model of an extended normal logic program has undefined literals. Table 1 (see appendix) presents in column SIX[23] the correspondence between each possible 4-tuple of MH_P valuations of a literal $(l^o, l, \neg l^o, \neg l)$ and the six possible logic values literal l may assume with respect to SIX.

Example 7. Consider the program in example 6. The assumable hypotheses set of P is $\{p\}$, since although both *not* b^o, *not* p^o appear in bP^{t-o}, p is not a fact in the program bP^{t-o}, whilst b is. The MH_P models are:[24]

$$\begin{aligned}
M_1 &= \bigtriangledown WFM(P^{t-o}) \\
&= \bigtriangledown \langle \{b, b^o, h, \neg h\}^+, \{\}^u, \{\neg b, \neg b^o, h^o, \neg h^o, p, p^o, \neg p, \neg p^o\}^- \rangle \\
&= \langle \{b, h, \neg h\}^+, \{\}^u, \{\neg b, h, \neg h, p, \neg p\}^- \rangle, \text{ with hypotheses set } \emptyset \\
M_2 &= \bigtriangledown WFM(P^{t-o} \cup \{p\} \cup \{p^o \leftarrow \textit{not}\ \neg p\}) \\
&= \bigtriangledown \langle \{b, b^o, \neg h, \neg h^o, p, p^o\}^+, \{\}^u, \{\neg b, \neg b^o, h, h^o, \neg p, \neg p^o, \}^- \rangle \\
&= \langle \{b, \neg h, p\}^+, \{\}^u, \{\neg b, h, \neg p\}^- \rangle, \text{ with hypotheses set } \{p\}
\end{aligned}$$

[22] This is equivalent to saying that $h \in (\bigtriangledown \triangle bP^{t-o})^u$. Notice that the purpose of computing bP^{t-o}, is to find the set of assumable hypotheses of P.

[23] Logic SIX provides a truth-functional model theory for the MH_P semantics.

[24] According to column SIX of table 1, the interpretations of the valuations of the literals in the two MH_P models above, are as follows: with respect to model M_1, b is true (logic value t), h is contradictory true (logic value I), p has contradictory belief (logic value IV); with respect to model M_2, b, p are true (logic value t) and h is false (logic value f).

We see that the first model is inconsistent while the second is not. The first model coincides with the $WFM_P(P)$. The second model arises because the MH_P uses the loop $\{b \leftarrow h,\ h \leftarrow not\ p,\ p \leftarrow not\ b\}$ of program P as a choice device.

The following results show that MH_P is a *total paraconsistent models* semantics (cf. def. 6) and that MH and MH_P coincide for normal logic programs, if we discard the default literals involving explicitly negated atoms from the MH_P models.

Proposition 4. The MHp semantics is a total paraconsistent models semantics, meaning that for any extended normal logic program P all the MH_P models of P are total paraconsistent models.

Theorem 5. MHp and MH coincide on Normal Logic Programs. Let P be a normal logic program and P^{t-o} the $t-o$ transformed of P. Let $MH(P)$ be the set of minimal hypotheses models of P and $MH_P(P)$ the set of paraconsistent minimal hypotheses models of P. Then $M \in MH(P)$, iff there is a model $M_p \in MH_P(P)$, such that $M^+ = M_p^+$ and $M^- = (M_p^- \cap \mathcal{H}_p)$, where $\mathcal{H}_p$ is the Herbrand base of P.

5.1 Formal Properties

The MH_P semantics enjoys the properties of *existence* and *simple relevance*[25].

5.2 Complexity

[26]The theorem below shows that a *brave reasoning* task with MH_P semantics, i.e. finding an MH_P model satisfying some particular set of literals (a query), is in Σ_2^P.

Theorem 6. Brave reasoning with MH_P semantics is in Σ_2^P.

Proof. Let us show that finding a MH_P model of an extended normal logic program P is in Σ_2^P. Computing P^{t-o} is fulfilled in linear time. The balanced layered remainder $\dot{b}P^{t-o}$ is computed in polynomial time, by the following reasoning: the calculus of the remainder of a normal logic program is kown to be of polynomial time complexity [17]; the difference between $\mapsto_{WFS}$ and $\mapsto_{\dot{b}LWFS}$ lies on the operator $\mapsto_N$ of the former being replaced by the operator $\mapsto_{\dot{b}LN}$ of the latter; to perform $\mapsto_{\dot{b}LN}$ the rule layering must be computed; the rule layering can be calculated in polynomial time since it is equivalent to identifying the *strongly connected components* [24] in a graph, $SCCs$, in this case in the complete rule graph of P^{t-o}; once the $SCCs$ are found, one collects their heads in sets, one set for each SCC, and searches the bodies of each SCC rules for

[25] We say that SEM has the property of *simple relevance*, iff for any extended normal logic program P, whenever there is a SEM model of $Rel_P(l)$ such that $l \in M_l$, there is also a SEM model M of P such that $l \in M$.

[26] This subsection follows closely subsection 4.6 of [1].

default literals, collecting in a set per SCC the corresponding atoms that also pertain to the set of heads of that SCC – this is all linear time; when verifying the preconditions to perform a negative reduction operation (existence of a fact, say b, and a rule with *not* b in the body), it is linear time to check if a rule is in loop (check if it belongs to a SCC) and if it is in loop through literal *not* b (check if b belongs to the heads of the SCC) – the same for cheking if the co-rule is in loop through *not* b^o; therefore, balanced layered negative reduction adds only polynomial time complexity operations over negative reduction. Once bP^{t-o} is computed, nondeterministically guess a set H of hypotheses – the assumable hypotheses set, is the set of all atoms involved in default negations in bP^{t-o}, that are not facts. Check if $WFM_P^u(P \cup H) = \emptyset$ – this is polynomial time. Checking that H is empty or non-empty minimal, requires another nondeterministic guess of a strict subset H' of H and then a polynomial check if $WFM_P^u(P \cup H') = \emptyset$.□

The theorem below shows that a *cautious reasoning* task with MH_P semantics, i.e. guaranteeing that every MH_P model satisfies some particular set of literals (a query), is in Π_2^P.

Theorem 7. Cautious reasoning with MH_P semantics is in Π_2^P.

Proof. Cautious reasoning is the complement of brave reasoning, and since the latter is in Σ_2^P the former must necessarily be in Π_2^P. □

6 Conclusions and Future Work

We have presented an abductive paraconsistent semantics, MH_P, that inherits the advantages of the abductive semantics MH, which renders the MH_P existential and simple relevant, and of the paraconsistent semantics $WFSX_P$, due to which the MH_P is endowed with the ability to detect support based on contradiction through non-trivial inconsistent models. MH_P explores a new way to envisage logic programming semantics through abduction, by dealing with paraconsistency using total paraconsistent models. The MH_P semantics may be used to perform reasoning in the following ways (let Q be a query and P an extended normal logic program in the role of a database): **Skeptical Consistent Reasoning:** Query Q succeeds, iff it succeeds for all consistent MH_P models of P (it fails if there are no consistent models); **Brave Consistent Reasoning:** Query Q succeeds, iff it succeeds for at least one non-contradictory MH_P model of P; **Skeptical Paraconsistent Reasoning:** Query Q succeeds, iff it succeeds for all MH_P models of P, such that none of the atoms in $Atoms(Q)$ has support on contradiction for any of these models; (it fails if there are no such models); **Brave Paraconsistent Reasoning:** Query Q succeeds, iff it succeeds for at least one MH_P model of P, such that none of the atoms in $Atoms(Q)$ has support on contradiction in this model; **Skeptical Liberal Reasoning:** Query Q succeeds, iff it succeeds for all MH_P models of P; **Brave Liberal Reasoning:** Query Q succeeds, iff it succeeds for at least one MH_P model of P. As future work, the MH_P may be extended in order to obtain a framework for representing and integrating knowledge updates from external sources and also inner source knowledge updates (or self updates), in line with the proposal in [25].

Acknowledgments. We are grateful to three anonymous referees whose observations were useful to improve the paper.

References

1. Pinto, A.M., Pereira, L.M.: Each normal logic program has a 2-valued minimal hypotheses semantics. INAP 2011, CoRR **abs/1108.5766** (2011)
2. Alferes, J.J., Damásio, C.V., Pereira, L.M.: A logic programming system for non-monotonic reasoning. J. Autom. Reasoning **14**(1) (1995) 93–147
3. Denecker, M., Kakas, A.C.: Abduction in logic programming. In: Computational Logic: Logic Programming and Beyond'02. (2002) 402–436
4. Pinto, A.M.: Every normal logic program has a 2-valued semantics: theory, extensions, applications, implementations. PhD thesis, Universidade Nova de Lisboa (July 2011) Published in the December issue of the ALP newsletter:http://www.cs.nmsu.edu/ALP/2012/12/doctoral-dissertation-every-normal-logic-program-has-a-2-valued-semantics-theory-extensions-applications-implementations/.
5. Gelfond, M., Lifschitz, V.: Logic programs with classical negation. In: ICLP. (1990) 579–597
6. Kowalski, R.A., Sadri, F.: Logic programs with exceptions. In: ICLP'90. (1990) 598–613
7. Pereira, L.M., Alferes, J.J.: Well founded semantics for logic programs with explicit negation. In: ECAI'92. (1992) 102–106
8. Przymusinski, T.: Well-founded Semantics Coincides With Three-Valued Stable Semantics. Fundamenta Informaticae **XIII** (1990) 445–463
9. Przymusinski, T.C.: Extended stable semantics for normal and disjunctive programs. In: ICLP'90. (1990) 459–477
10. Wagner, G.: A database needs two kinds of negation. In: MFDBS. (1991) 357–371
11. Alferes, J.J., Pereira, L.M.: Reasoning with Logic Programming. Volume 1111 of Lecture Notes in Computer Science. Springer (1996)
12. Arieli, O.: Paraconsistent declarative semantics for extended logic programs. Ann. Math. Artif. Intell **36**(4) (2002) 381–417
13. Blair, H.A., Subrahmanian, V.S.: Paraconsistent logic programming. Theor. Comput. Sci. (1989) 135–154
14. Pearce, D., Wagner, G.: Logic programming with strong negation. In Schroeder-Heister, P., ed.: ELP. Volume 475 of Lecture Notes in Computer Science., Springer (1989) 311–326
15. Sakama, C.: Extended well-founded semantics for paraconsistent logic programs. In: FGCS. (1992) 592–599
16. Costantini, S.: Contributions to the stable model semantics of logic programs with negation. Theoretical Computer Science **149**(2) (2 October 1995) 231–255
17. Brass, S., Dix, J., Freitag, B., Zukowski, U.: Transformation-based bottom-up computation of the well-founded model. TPLP (2001) 497–538
18. Gelder, A.V.: The alternating fixpoint of logic programs with negation. J. of Comp. System Sciences **47**(1) (1993) 185–221

19. Lifschitz, V.: What is answer set programming? In: AAAI'08. (2008) 1594–1597
20. Damásio, C.V., Pereira, L.M.: A survey of paraconsistent semantics for logic programs. Handbook of defeasible reasoning and uncertainty management systems: volume 2: reasoning with actual and potential contradictions (1998) 241–320
21. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP, MIT Press (1988) 1070–1080
22. Alferes, J.J., Damásio, C.V., Pereira, L.M.: Top-down query evaluation for well-founded semantics with explicit negation. In: ECAI'94. (1994) 140–144
23. Saccà, D., Zaniolo, C.: Partial models and three-valued models in logic programs with negation. In: LPNMR'91. (1991) 87–101
24. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. (1972) 146–160
25. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In: JELIA'02. (2002) 50–61

Appendix

Logics *NINE* and *SIX*

In [20] the author presents a logic, there dubbed $NINE$, that provides a truth-functional model theory for the $WFSXp$, based on Ginsberg's bilattices concept. The truth-space of $NINE$ comprises nine logic values, here presented together with their meanings: '**t**' and '**f**' are the classical values for truth and falsity; '**I**', is the contradictory truth value; '**II**', is understood as truth with contradictory belief; '**III**', is understood as falsity with contradictory belief; '**IV**', is understood as contradictory belief; '⊥', is understood as undefinedness; '**df**', is understood as default falsity; '**dt**', is understood as default truth. Using the ideas seth forth in the definition of $NINE$ we define SIX, a truth-functional model theory for the MHp, whose truth-space is $\{t, f, I, II, III, IV\}$. Table 1 represents all the possible 4-tuple valuations of a literal $(l^o, l, \neg l^o, \neg l)$ with the corresponding $NINE$ and SIX logic values. There are 20 possible 4-tuple literal valuations, as a consequence of corollary 1 and the coherence principle.

Table 1. NINE and SIX valuations

l^o	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	$-$	u	u	u	$-$	$-$	$-$	u	u	$+$
l	$-$	$-$	$-$	$-$	$-$	$-$	u	u	u	u	u	u	u	u	$+$	$+$	$+$	$+$	$+$	$+$
$\neg l^0$	$-$	$-$	u	$-$	u	$+$	$-$	$-$	u	$-$	u	$-$	$-$	u	$-$	$-$	$-$	$-$	$-$	$-$
$\neg l$	$-$	u	u	$+$	$+$	$+$	$-$	u	u	$+$	$+$	$-$	u	u	$-$	u	$+$	$-$	u	$-$
$NINE$	IV	IV	df	III	f	f	IV	IV	df	III	f	dt	dt	$\perp$	II	II	I	t	t	t
SIX	IV	IV		III	f	f	IV	IV		III	f				II	II	I	t	t	t

Meta-set calculus as mathematical basis for creating abstract, structured data store querying technology

Mikus Vanags[1,2], Arturs Licis[2], Janis Justs[2]

[1] Latvia University of Agriculture, 2 Liela Street, Jelgava, LV-3001, Latvia
[2] Logics Research Centre, Sterstu Street 7-6, Riga, LV-1004, Latvia
{mikus.vanags, arturs.licis, janis.justs}@logicsresearchcentre.com

Abstract: Any logic programming engine needs an access to the data stores (knowledge-bases) to perform operations on data (facts). As logic programming engine does not know, which facts it will need in order to process questions (queries) correctly, it has to load in memory and process all necessary data store content which is one of the most useless examples of the database usages. Such limitation can be eliminated by introducing an abstraction: meta-set, which is able to describe one or more facts (becoming meta-facts). This way it is possible to separate the data store from the logic programming engine turning it into meta-set calculus (kind of second order predicate logic) engine, which combines meta-sets (meta-facts and rules) to build resulting meta-set list to be used for auto-generating queries to data store. The proposed solution allows running deduction process in distributed environments where deduction can be performed on the client tier, and queries for real objects will be sent to the centralized data store servers. The meta-set calculi can be used for substituting standard database querying languages with querying language syntax similar to Prolog, therefore improving type safety, rule reuse and reducing the querying syntax dependence on data store. Such concept is suitable for querying the NoSQL data stores which store the property-value pairs in structured groups as well as for querying relational databases.

Keywords: meta-set calculus, logic programming, second order predicate logic, deduction.

1 Introduction

Many different expert systems and deductive databases exist, but most of them use their own syntax which is not comfortable for using it together with the modern object oriented languages. Not only expert system shells use different syntax, but each data store uses more or less different querying syntax. In different database querying can help abstractions like LINQ – Language Integrated Query [1]. But LINQ cannot help in cases where deduction is needed, because such system would request to load in memory all database content to work correctly. Problem can be solved with another abstraction: meta-set. Classic predicate logic as it is implemented in logic programming language Prolog [7] does not work with meta-sets. Meta-sets require modifications in logic programming engine and such modifications in predicate logic leads to the definition of second order predicate logic.

In this paper first order predicate logic is compared with one of the versions of second order predicate logic: meta-set calculus. This is the main reason why this paper does not cover detailed meta-set calculus comparison to Prolog, DataLog and other first order predicate logic systems. First order predicate logic systems can use different approaches to simulate sets, for example, using lists, but lists are not mathematical abstractions. Lists are data structures, which are restricted to contain finite number of objects, while sets as mathematical abstractions can describe unknown or even infinite number of objects.

2 Foundation of meta-set calculus

More abstract thinking leads to a better design [6]. There are many existing abstractions'[1], logics[2], formal calculus[3], approaches, e.g., MDA[4] [2]. Examples of abstractions with its meta, meta-meta, and further models lead to the question how to classify a system which will work with meta-sets. As proposed, the system is based on a modified predicate logic, it is not only approach-based on axiomatic rules; it is logic or at least mathematical formalism. Expressions written in a formal calculus can be transformed into underlying logic; one example is lambda calculus transformations to combinatory logic and vice versa. The proposed system is based on a predicate logic, but meta-set processing operations cannot be described by simple predicate logic expressions, so the meta-set system proposed is not a traditional predicate logic system and cannot be transformed into it. First order predicate logic uses variables that range over individual objects, but second order logic involves variables which range over sets of individual objects [3, 8]. Meta-set is an object set abstraction and meta-set calculi is based on the second order predicate logic where meta-sets are used instead of object sets. Meta-sets contain neither real objects, nor references to real objects; meta-sets can contain only constraints which can be used to generate database query and retrieve set of objects. Meta-set calculus engine can be implemented by improving a basic logic programming engine to work with meta-sets. Meta-set description and explanation, which operators are needed for meta-set calculus to be descriptive and how these operators work, will be given in the following sections.

The difference between predicate logic and meta-set calculus, which works with meta-sets, is in the way of how facts (business objects) are processed. When working with facts directly, programming engine needs to load all necessary database content

[1] Interfaces, inheritance, type systems etc.

[2] Combinatory logic, propositional logic, predicate logic etc.

[3] For example lambda calculus which extends combinatory logic with mathematical function abstraction. [4]

[4] MDA defines only axiomatic rules and gives model and meta-model description syntax; however, development of transformation logic is left up to developers so such approaches can't be used unambiguously.

into memory to complete a deduction process. This problem is solved in a meta-set programming engine by introducing meta-set (abstraction of business objects). One meta-fact can describe many facts, so the number of meta-facts will be significantly smaller than a number of total facts in business object database. Therefore meta-set programming engine would perform much faster and, more importantly; it will be independent from the business object database content allowing decentralization of the deduction process.

Meta-set calculi can be interpreted as similar to idea of constraint logic programming [15], but in addition containing type information and set abstraction.

2.1 Meta-set logical structure

Each meta-set can contain 3 different constraint lists: type constraints, property-value constraints and object set constraints. The proposed syntax for meta-set declaration:

```
<type constraints> [property-value constraints] {object
set constraints} | bound_Variable
```

The mandatory part consists of type constraints which always come before other constraints. Property-value constraints and object set constraints are optional and can be omitted. Each constraint can be declared together with NOT operator, but that does not change type of the constraint. If meta-set contains more than one constraint of the same type, they are separated by commas. Bound variable is optional and is used only to represent meta-set processing operations during logical deduction process; bound variable is not used in any other way – it should not be stored in a database or used to generate database query.

Type constraints are used to indicate types of objects that meta-set represented set of objects should contain. When type constraints are used together with NOT operator, the resulting constraints represent types of objects that meta-set represented set of objects should not contain. Meta-set calculus is designed for use in object oriented environments and meta-set type constraints support such abstractions as type inheritance and interface implementation. For better illustration of type constraint processing in meta-set calculus, including meta-set matching and unification, most of meta-set examples provided in this article are with the class diagrams as shown in Figure 1.

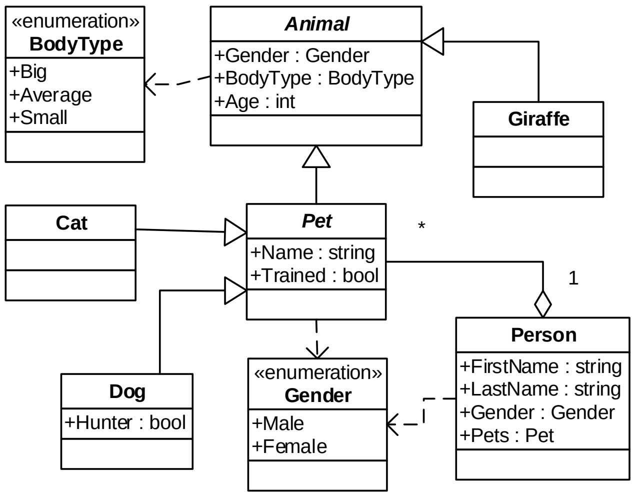
Example of meta-set, which restricts object types to be subclass of Animal and, at the same time, not subclass of Dog:

```
<Animal, not(Dog)>
```

According to Figure 1 class diagram, query generated from the previously declared meta-set would return all instances of the class Cat and the class Giraffe objects, because these classes inherit from the class Animal and does not inherit the class Dog.

When using NOT operator for type constraint declaration, it is advised to provide at least one type constraint which is not used together with NOT operator, because query building system in each database is implemented in a different way and such unambiguous queries can lead to the unexpected results (for example, when type constraint is provided in NOT context without usual type constraint, database db4o 8.1 [12]. interprets it as type constraint ignoring NOT context). In object oriented environment it is useful to use interfaces for constraining objects to be of specific types.

Fig. 1. Physical class structure used in examples of meta-set calculus



Property-value constraints provide desired value range limitation for specified property. The proposed syntax for property-value constraint description is following:

```
[PropertyName EvaluationMode ConstrainedValue]
```

In a pure meta-set calculus 4 different property-value constraint EvaluationMode forms are supported: Equal, Greater, Smaller and Contains. This example demonstrates meta-set which represents dogs younger than 3 years and not trained:

```
<Dog>[Age<3,not(Trained=True)]
```

Property-value constraints can be not only simple value constraints, but also complex constraints indicating type of property returning object and values of its properties. These complex constraints can be built as another meta-set containing desired type of constraints, property-value constraints and even object set constraints. The next example shows how to declare complex meta-set which represents set of persons having[5] at least one trained dog.

```
<Person>[Pets contains <Dog>[_trained=True]]
```

Object set constraints are unique for the second order predicate calculus; by using object set constraints it is possible to define relationships between two different meta-sets. Meta-set can be interpreted as a query to database which returns set of objects. Sometimes it is necessary to define relationships between different sets of objects. Object set constraint can be represented by the following syntax:

[5] meaning: property Pets contains

```
{SetsRelationship(meta-set)}
```

Relationships of object sets are always defined between two sets represented by meta-sets: first is meta-set containing object set constraints list (context meta-set) and second is meta-set used as an object set constraint. In cases where it is necessary to define relationships between more than two meta-sets, it is possible to add more object set constraints.
SetsRelationship can be one of 8 supported meta-set relationship forms:

1. Different – two sets of objects does not contain common objects.
2. Intersect – two sets of objects have at least one common object.
3. Equal – two sets of objects are equal – contain equal objects.
4. NotEqual - two sets are not equal. It does not definitely mean that they are different, it means that two object sets are either different or intersect (including cases when one set is subset of another set).
5. Subset1 – the first set (represented by context meta-set) is subset of the second set (declared as object set constraint).
6. NotSubset1 – the first set (represented by context meta-set) is not subset of the second set (declared as object set constraint). It means that two object sets are either different, intersect or the second set is subset of the first set.
7. Subset2 – the second set (declared as object set constraint) is subset of the first set (represented by context meta-set).
8. NotSubset2 – the second set (declared as object set constraint) is not subset of the first set (represented by context meta-set). It means that two object sets are either different, intersect or the first set is subset of the second set.

Next example shows how to declare meta-set which represents all dogs that are not trained dogs:

```
<Dog>{different(<Dog>[Trained=True])}
```

Equivalent SQL query would be as follows:

```
SELECT * FROM Dogs
EXCEPT
SELECT * FROM Dogs WHERE Trained='True';
```

2.2 Meta-set matching and unification

Logic programming engine works with objects, but logic programming engine can be modified to work with meta-sets as well. In this case predicate terms could be meta-sets. For example:

```
dog(<Dog>). trained(<Pet>[Trained=True]).
```

As meta-sets do not change behavior of variables, their usage does not change rule declaration syntax. Here is an example of rule which uses fact declared using meta-sets:

```
trainedDog(x) :- dog(x), trained(x).
```

In logic programming syntactically identical objects match, but unlike objects do not match. As far as matching is concerned, there is no difference in the treatment of atoms and terms. The difference that is crucial is between variables and non-variables. For example:

female(Ineta) matches with female(Ineta)
female(Ineta) matches with female(x) binding variable x to object Ineta
female(Ineta) does not match with female(Alice).

Meta-set matching differs from object matching, because meta-sets are like small parts of larger query that is being built and not all differences in meta-sets are considered as failures in matching. For example:

something(<Dog>) matches with something(<Dog>)
something(<Dog>) matches with something(<Pet>)
something(<Dog>) matches with something(<Animal>)
something(<Dog>) does not match with something(<Cat>)
something(<Dog>) does not match with something(<Person>)
something(<Dog>) matches with something(x). Matching does not reference variables with meta-set instances, but unification does.

In unification, when meta-set type constraints matches and if variable was used in matching , the meta-set, to which the variable references, will contain updated list with the most specific type constraints from both meta-sets, merged lists of both meta-set property-value constraints and set-constraints and also variable name which references to meta-set. For example, if knowledge base contains facts and rule:

```
dog(<Dog>). trained(<Pet>[Trained=True]).
trainedDog(x) :- dog(x), trained(x).
```

In this case asking question trainedDog(n)? to meta-set calculus engine will lead to unifying trainedDog(n) with trainedDog(x) and in result variable x will be bound to variable n and new list of goals would be dog(n) and trained(n). Dog(n) will match with dog(<Dog>) from knowledge base leading to reference variable n with copy of meta-set <Dog>|n [6] and remaining goal will be updated to: trained(<Dog>|n). Trained(<Dog>|n) would match with trained(<Pet>[Trained=True]) leading to updating variable n referenced meta-set <Dog>|n to <Dog>[Trained=True]|n. Now list of goals would be empty and variable n points to meta-set <Dog>[Trained=True]|n which is the answer to question and reference to variable n is no more needed.

Unification using NOT operator works similarly, with the difference that constraints under NOT operator context are marked with NOT marks and constraints which already were marked with NOT marks are released from NOT marks. For example, if rule base would contain rule notTrainedDog:

```
notTrainedDog(x) :- dog(x), not(trained(x)).
```

[6] Here meta-set <Dog>|n contains reference back to variable n. Such reference is needed for meta-set calculus engine to work correctly and it is not necessary to be stored to database.

Then meta-set calculus engine to the question notTrainedDog(n)? would give an answer: <Dog>[not(Trained=True)]|n.

2.3 New operators to deal with meta-sets

All the necessary meta-sets can be created prior to using them in deduction process, but in complex cases it is more type-safe if meta-sets are built dynamically during deduction process. Dynamic meta-set creation is possible using special operators.

Type constraint operator: IsOfType adds type constraint to meta-set. IsOfType can be used together with NOT operator. For example:

```
animal(<Animal>).
neededAnimal(x) :- animal(x),not(isOfType<Dog>(x)).
```

Processing rule neededAnimal during deduction process will lead to creation of the following meta-set:

```
<Animal,not(Dog)>|x.
```

Pure meta-set calculus supports 4 different property-value comparison operators: equalTo, greater, smaller, contains. All property-value comparison operators can be used together with NOT operator. Property-value comparison operator usage syntax:

```
operator(meta-set, “propertyName”, propertyValue).
```

As a result of using the property-value comparison operator, a new constraint will be added in a meta-set property-value constraint list. For example:

```
person(<Person>).
adultPerson(x):-person(x),greater(x, "Age", 18).
```

Processing rule adultPerson during deduction process will lead to creation of the following meta-set:

```
<Person>[Age>18]|x.
```

Pure meta-set calculus supports 5 different object set operators: different, intersect, equal, subset1, subset2 which together with NOT operator forms 3 additional set relationships: NotEqual, NotSubset1, NotSubset2. Set operator usage syntax:

```
setOperator(metaset1, metaset2).
```

For example, operator different can be used to declare predicate dogDifferentThanTrainedDog:

```
dog(<Dog>). trained(<Pet>[Trained=True]).
dogDifferentThanTrainedDog(x) :- dog(x), dog(y),
trained(y), different(x, y).
```

Processing rule dogDifferentThanTrainedDog during deduction process will lead to creation of the following meta-set:

```
<Dog>{different(<Dog>[Trained=True])}|x.
```

2.4 NOT operator support for meta-sets

In logic programming expression NOT(P) detects presence or absence of the theorem P proof in rule base. In such case operator NOT stops search when the first proof supporting P is found, but such behavior is not correct for meta-set calculus, because meta-sets are not real objects and fact NOT(P) in meta-set case should be persisted in meta-set P as constraint. Meta-set calculus also requires searching knowledge base for all operator NOT argument proofs, only this way it is possible to get correct NOT constraints for all meta-sets involved in the deduction process.

NOT operator can be used together with all the previously mentioned operators except operator “is implied by”. When NOT is used together with AND or OR operators, De Morgan’s laws are applied to simplify expressions.

2.5 Running multiple queries in one question to meta-set calculus engine

In all examples given previously questions were asked using only one variable, but logic programming allows asking questions with more than one variable, for example:

```
father(x, y)?
```

In such case logic programming engine will return all father-son pairs, in form:

```
x=Father1, y=Son1;
x=Father2, y=Son2.
…
```

Meta-set calculus also supports this feature. But as meta-sets are a set of abstractions and represent queries to database, in cases when questions with many variables will be asked to meta-set calculus engine, it will result in a number of queries equal to the number of variables present in question. Theoretically each query generated from resulting meta-sets could return resulting objects ordered in different ways causing inconsistence according to the question asked. Such behavior is natural because meta-sets by definition represent object sets and in sets objects are not ordered.

3 Solution architecture

Content of meta-set calculus implementation sections depends on experiments that have been done. Experimentation platform was .NET and chosen programming language C# [10, 11], because these technologies in combination with object database db4o 12 inspired foundation of meta-set calculus and its first implementation - DDE.

Figure 2 shows generalized meta-set physical structure, where property TypeConstraints is intended to contain type constraints for business objects located in database and TypeNotConstraints should contain type constraints with applied logical operation NOT. Relations contain property-value constraints and are the basic aspect of meta-set abstraction. E.g., if business object database contain 1000 people and 200 of them are younger than 20 years, then all these 200 people (facts) can be described with one meta-set which contains the following single relation in its Relations collection: Age < 20. Similarly is processed NotRelation collection which describes property-value constraints the desired object set should not contain.

Methods GetVariable and SetVariable are used by meta-set calculus engine to hold temporarily bound variable. This information is needed only in deduction process and does not require to be stored into database; for this reason in meta-set implementation for database db4o (Database For Objects) field "variable" is marked with attribute [Transient].

Class Type is used to generalize approach of defining type constraints. In most cases it would be enough to define string representations of types so that they could be persisted (in case of db4o, for type constraint storing and retrieving object translators are used which translate Type objects to String instances and vice versa. Type, CultureInfo and some other system type instances depend on .NET internal behavior and are difficult to store in database).

Fig. 2. Generalized meta-set physical structure

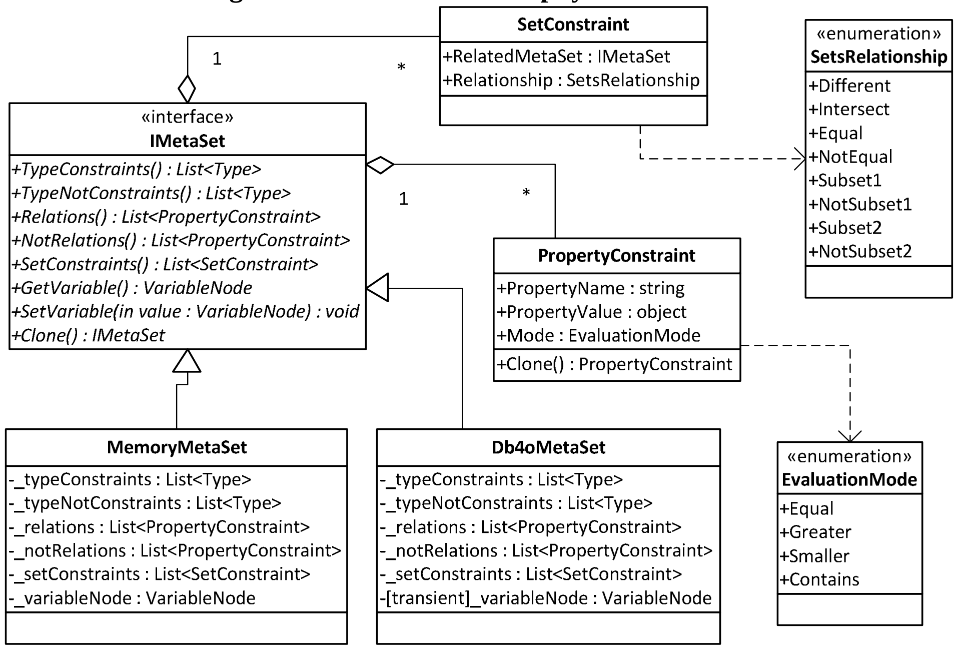
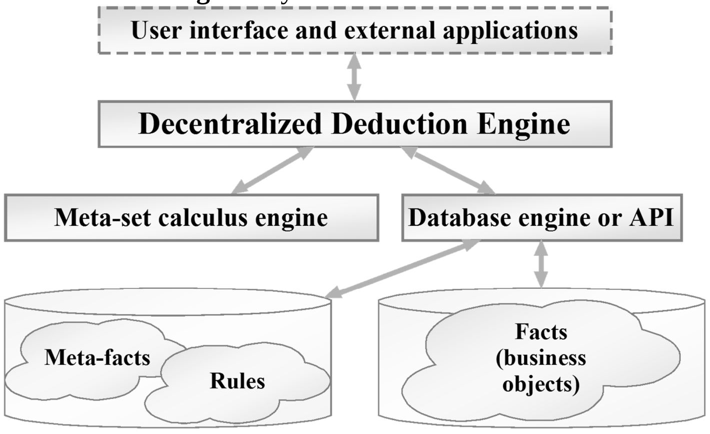

SetConstraint
+RelatedMetaSet : IMetaSet
+Relationship : SetsRelationship
«enumeration»
SetsRelationship
+Different
+Intersect
+Equal
+NotEqual
+Subset1
+NotSubset1
+Subset2
+NotSubset2
1
*
«interface»
IMetaSet
+TypeConstraints() : List<Type>
+TypeNotConstraints() : List<Type>
+Relations() : List<PropertyConstraint>
+NotRelations() : List<PropertyConstraint>
+SetConstraints() : List<SetConstraint>
+GetVariable() : VariableNode
+SetVariable(in value : VariableNode) : void
+Clone() : IMetaSet
1
*
PropertyConstraint
+PropertyName : string
+PropertyValue : object
+Mode : EvaluationMode
+Clone() : PropertyConstraint
MemoryMetaSet
-_typeConstraints : List<Type>
-_typeNotConstraints : List<Type>
-_relations : List<PropertyConstraint>
-_notRelations : List<PropertyConstraint>
-_setConstraints : List<SetConstraint>
-_variableNode : VariableNode
Db4oMetaSet
-_typeConstraints : List<Type>
-_typeNotConstraints : List<Type>
-_relations : List<PropertyConstraint>
-_notRelations : List<PropertyConstraint>
-_setConstraints : List<SetConstraint>
-[transient]_variableNode : VariableNode
«enumeration»
EvaluationMode
+Equal
+Greater
+Smaller
+Contains

Figure 3 depicts the architecture of DDE where it interacts with meta-set calculus engine and data store querying API to provide interface for managing structured data deductive database. The benefit of such architecture is business object database separation from meta-sets (meta-facts) and rules database where rules are similar to meta-facts with the only difference that meta-facts describe facts, but rules combine meta-facts. This separation improves meta-fact reusability and allows keeping centralized only business objects, while deduction specific information can be located on client computers and each client can have different meta-facts and rules based on their specific needs.

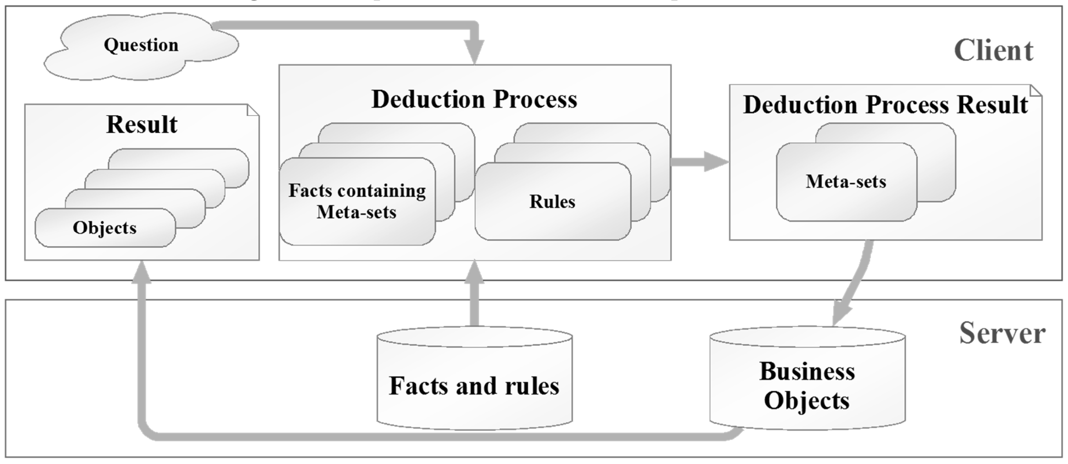
Fig. 3. Layered structure of DDE



3.1 Decentralized Deduction Engine

DDE connects user interface or external code calls through DDE API calls with meta-set calculus engine and data store. Figure 4 shows generalized schema of operations what DDE does. The result of deduction process is a collection of meta-sets; and if database contains more than one instance of each type (for relation databases it means more than one row in each table), then the number of resulting meta-sets would be significantly smaller than number or querying objects in database. The structure of meta-set is similar to simple object data access (SODA) [5] query structure used in object databases, thus collection of meta-sets is used to automatically generated object database SODA query to retrieve business objects from business object database. For this reason DDE does not need to load all database content into memory to work correctly.

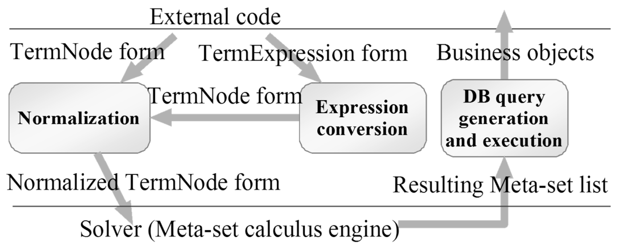
Fig. 4. Conceptual schema of deduction process in DDE



One of project goals was to integrate meta-set calculus syntax with existing programming languages. Practical experiments were mainly concentrated on .NET Framework and programming language C#, but that does not make conceptual restrictions for meta-set calculus technology implementation in other programming languages.

Figure 5 represents question processing workflow in DDE showing two different forms of questions DDE accepts and how these question forms are processed.

Fig. 5. Question processing workflow in DDE



Solver works with TermNode expressions translated in conjunctive normal form. DDE handles all necessary normalizations, but still main data structure for DDE is TermNode expressions, because TermNode instances can be stored in data stores and TermNode instances are platform independent. Here is an example of defining TermNode instance in C# code[7]:

```
var youngPerson = new TermNode("youngPerson", new
object[] { metaYoungPerson });
```

Expressions in TermNode form are neither short, nor type safe and user-friendly. To solve these problems function types[8] are employed in DDE which create TermNode instances. Here is an example of defining fact that person is young using DDE built-in delegates:

```
var youngPerson = Func.Arity1("youngPerson");
TermNode fact = youngPerson(metaYoungPerson);
```

Delegates can help in fact declaration, but with delegate provided syntax is not enough to declare rules and questions with variables in a nicely looking way. Therefore TermExpression form is introduced in DDE. TermExpression form is based on expression tree[9] usage similarly as does LINQ to SQL [13] and other data store LINQ providers. Variables in TermExpression form are declared as lambda expression parameters and it results in prolog like syntax in native C# code:

7 Meta-set metaYoungPerson was defined in chapter 2.
8 In .NET and C# they are delegates.
9 Dependance on platform .NET.

```
var youngPerson = Func.Arity1("youngPerson");
var male = Func.Arity1("male");
var youngMalePerson = Func.Arity1("youngMalePerson");
Expression<Func<object, TermNode>> rule = a =>
youngMalePerson(a) == youngPerson(a) & male(a));
```

Here C#'s overloaded operator "==" represents operator "is implied by" in meta-set calculus.

Idea about prolog like syntax in native C# code was firstly introduced by Microsoft in LINQ CTP in May 2006 as LogicProgramming sample project [10]. LINQ project became part of .NET 3.5, but LogicProgramming sample project quietly disappeared. In authors opinion that happened not because Microsoft was trying to hide some potentially valuable ideas, but because project was difficult to understand for average programmers and prolog like syntax without meta-set calculus is useless. To work properly, logic programming engine requires access to all database content in worst case leading to loading in memory all database content that is not the case with meta-set calculus engine.

Conversion from TermExpression form to TermNode form suitable for Solver

TermExpression form is based on expression trees [14] which are not trivial to persist in data stores. TermExpressions can contain references to function pointers and user created objects located on heap. Such things are specific for physical computer on which application is running. Expression tree instances can be serialized, but, when deserialized on a different computer, function pointers and pointers to user object instances would point to memory locations where objects are on initial computer and that would lead to an exception. TermExpression form uses .NET standard delegate Func overloaded versions and that means arity restriction which does not exceed 17 terms in each predicate. In this way rules and questions declared using TermExpression syntax are nice-looking, type safe, but they still are not platform independent and therefore need to be converted to TermNode form for Solver to work.

TermExpressions are converted by recursively inspecting expression tree instances and creating resulting TermNode objects. To avoid code duplications in standard operators like AND, OR, NOT, Equal and standard operator expressions in expression trees, only parameters are taken from the expression trees and supplied in an appropriate standard operator calls this way building resulting TermNode instance.

Results returned from execution of all standard operators[10], user defined predicates and built-in predicates are represented as TermNode instances. This means that AND, OR, NOT, Contains, Different, Equals, other built-in predicates and even operator "is implied by" are represented as TermNode instances. This feature will be useful in communication between different computers, because every expression can be converted in single TermNode instance.

Meta-set calculus engine

[10] C# operators, but predicates in predicate logic and meta-set calculus.

Meta-set calculus engine works only with normalized TermNode instances in conjunctive normal form. However, the following actions are performed prior to conjunctive normalization execution: reduction of nested NOT operators using De Morgan's laws and other normalizations.

Meta-set calculus engine is based on modified unification algorithm which combines constraints of meta-sets as described in section 2.2. Modified unification algorithm supports both: sets-objects and business objects, but such hybrid deduction engine is complex and depends on further research about meta-set calculus technology use. Unification algorithm improvements could probably progress according to two different scenarios: unification, which supports only meta-sets, and hybrid unification supporting both: meta-sets and business objects. Meta-set-only unification is suitable for database querying and deduction support, which is more attractive to the business environment, while hybrid unification could be useful in expert system shells.

NOT operator implementation in meta-set calculus engine is simpler than described in pure meta-set calculus description. Physical meta-set model contains two lists for type constraints shown in Figure 2: one list for constraints without applied NOT operator and one list for type constraints with applied NOT operator. When meta-set calculus engine detects meta-set in unifying predicate from knowledge base, type constraints from unifying predicate are added to the respective type constraint lists of resulting meta-set. But when meta-set calculus engine detects NOT predicate, content of list TypeNotConstraints of unifying meta-set located in knowledge base is copied to list TypeConstraints of resulting meta-set and content of list TypeConstraints from unifying meta-set is copied to list TypeNotConstraints of resulting meta-set. Property-value constraints in NOT operator context are processed similarly. These constraint lists for constraints without applied NOT operator and with applied NOT operator are useful when considering future changes. For example, invention of new constraints would not require changes in whole constraint handling system and the same constraints will continue to work correctly also in combination with NOT operator. Only object set constraints are stored in one constraint list, because object set constraints are more complex and therefore cannot be classified only in two different groups. Object set constraints change SetsRelationship enumeration value depending on object set constraint, namely, if it is or is not used together with NOT operator.

Built-in predicate processing is performed by meta-set calculus engine during deduction process. For this reason NOT operator and build-in predicates like IsTypeOf, Greater, Smaller, EqualTo, Contains, Different, Equal, Subset1, Subset2 and Intersect, which are in introduced section 2.4, build the resulting meta-sets dynamically. If possible, it is recommended to declare facts providing full desired meta-set specification instead of using dynamic meta-set building. Dynamic meta-set building performs slower and is less type safe.

Db query generation from deduced meta-set list.

Database query generation is the most important part in DDE after meta-set calculus engine. In current DDE prototype only SODA query generation is supported to the database db4o. Next example demonstrates how query is generated from deduced meta-set assuming that the knowledge base contains following facts:

```
dog(<Dog>). trained(<Pet>[Trained=True]).
person(<Person>).
```

Also it has to be assumed that knowledge base contains rule:

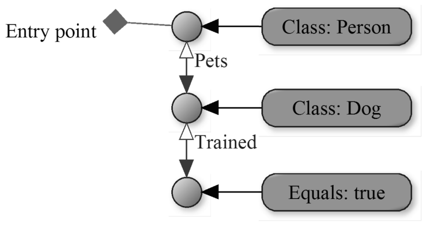
```
personWhoOwnsDog(x) :- person(x), dog(y), trained(y),
contains(x, "Pets", y).
```

Having these assumptions question personWhoOwnsDog(n)? would lead to answer:

```
<Person>[Pets contains <Dog>[Trained=True]]|n.
```

And from that meta-set DDE auto generates SODA query shown in Figure 6 (idea of SODA query representation is taken from db4o supporting documentation [12]). After successful query generation, query is executed and gathered objects are returned to the user who asked question to DDE system.

Fig. 6. SODA query generated from meta-set per-sonWhoOwnsDog



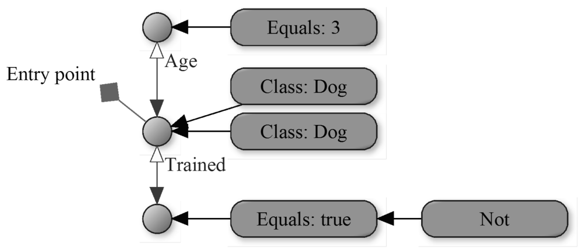
Query generation from meta-sets containing only type constraints and property-value constraints is simple. Query generation from meta-sets containing set constraints poses difficulty, because there are databases which do not support set operators. In cases where the required database system does not support set operators, for example, set operator subset1(metaset1, metaset2), DDE generates separate queries for metaset1 and metaset2, executes them and processes returned objects from queries in a way they comply with used set operator, for example – subset1.

While not all set operations are supported in db4o, it is possible to generate SODA queries from meta-sets containing set constraint: different. For example, from:

```
<Dog>[Age=3]{different(<Dog>[Trained=True])}|x
```

will be generated SODA query shown in Figure 7, where all meta-sets, which set relationship: different, are processed as questions to deduction system with applied NOT operator. Generated SODA query contains two equal type constraints (type of Dog) and query generation can be improved to contain only one instance of the most specific type constraints from all involved meta-sets.

Fig. 7. SODA query generated from <Dog>[Age=3]{different(<Dog>[Trained=True])}



Current implementation of DDE supports database query generation from meta-sets for database db4o, but in similar way it is possible to add query generation support from meta-sets for other key-value stores which store key-value pairs in structured way, for example, XML files. It is also possible to add support for relational database query generation from meta-sets, but in this field more research is needed, because SQL queries can contain join operations which are problematic to interpret in object oriented world. Join operator allows selecting parts from different tables, but object-oriented world requests all objects and not only parts of many objects. This problem could be solved by interpreting queries with join operations as requests for some dynamically generated objects[11], but it needs further research.

4 References

1. LINQ – .NET Language Integrated Query. DOI=http://msdn.microsoft.com/en-us/netframework/aa904594.
2. MDA - Model Driven Architecture. DOI= http://www.omg.org/mda/.
3. Second-order logic. DOI= http://en.wikipedia.org/wiki/Second-order_logic.
4. Raul Rojas. A Tutorial Introduction to the Lambda Calculus. DOI=http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf.
5. SODA - Simple Object Database Access. DOI= http://sourceforge.net/projects/sodaquery/.
6. Pierce B.C. 2002. *Types and programming languages*. MIT Press, London, England.
7. Bratko I. 2000. Prolog Programming for Artificial Intelligence. Pearson, UK.
8. S. Marc Cohen. 2004. *Second Order Logic*. DOI=http://faculty.washington.edu/smcohen/120/SecondOrder.pdf.
9. Stansifer R. 1994.*The Study of Programming Languages*. Prentice Hall, New Jersey, USA.
10. Microsoft Visual Studio Code Name “Orcas” Language-Integrated Query, May 2006 CTP. DOI=http://www.microsoft.com/download/en/details.aspx?displaylang=en&id=11289.
11. Skeet J. 2010. *C# in Depth*. Manning Publications Company.
12. Db4o – Database For Objects. DOI= http://www.db4o.com/.
13. LINQ TO SQL. DOI= http://msdn.microsoft.com/en-us/library/bb386976.aspx.
14. DLR Expression Tree Specification. DOI= http://dlr.codeplex.com/wikipage?title=Docs%20and%20specs&referringTitle=Documentation.
15. Joxan Jaffar, Michael J. Maher. Constraint Logic Programming: A Survey. DOI= http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.102.4288&rep=rep1&type=pdf

[11] LINQ to SQL works in similar way.

Author Index