Preface

This volume contains the papers presented at WLP 2014, the 28th Workshop on (Constraint) Logic Programming. The workshop was held on September 15-17, 2014 in Wittenberg, at the Leucorea Conference Center of Halle-Wittenberg University.

The Workshops on (Constraint) Logic Programming are the annual meeting of the German Society of Logic Programming (GLP) and bring together researchers interested in logic programming, constraint programming, answer set programming, and related areas like databases and artificial intelligence (not only from Germany). Previous workshops have been held in Germany, Austria, Switzerland and Egypt. The workshops provide a forum for exchanging ideas on declarative logic programming, nonmonotonic reasoning and knowledge representation, and facilitate interactions between research in theoretical foundations and in the design and implementation of logic-based programming systems. The WLP workshop series started 1988 in Berlin (in the first three years there were two workshops per year).

The workshop was jointly organized and located with the 23rd International Workshop on Functional and (Constraint) Logic Programming (WFLP 2014). The WFLP workshops series is running since 1992 and brings together researchers interested in functional programming, logic programming, as well as their integration.

I thank the invited speaker, authors of papers, programme committee members, external reviewers, as well as the Leucorea team and my local organization staff Ramona Vahrenhold, Heike Stephan, and Alexander Hinneburg. I would also like to thank Johannes Waldmann for organizing WFLP 2014. It was a pleasure to work together.

September 12, 2014 Stefan Brass
Program Committee

Andreas Behrend (Universität Bonn)
Christoph Beierle (FernUniversität in Hagen)
Stefan Brass (Martin-Luther-Universität Halle-Wittenberg, chair)
François Bry (Ludwig-Maximilians-Universität München)
Jürgen Dix (TU Clausthal)
Thom Frühwirth (Universität Ulm)
Michael Hanus (Christian-Albrechts-Universität zu Kiel)
Petra Hofstedt (Brandenburgische Technische Universität Cottbus)
Michael Leuschel (Heinrich-Heine-Universität Düsseldorf)
Rainer Manthey (Universität Bonn)
Dietmar Seipel (Julius-Maximilians-Universität Würzburg)
Sibylle Schwarz (HTWK Leipzig)
Hans Tompits (TU Wien)
Armin Wolf (Fraunhofer FOKUS)

Additional Reviewers

Stephan Frank
Gabriele Keller
Ludwig Ostermayer

Local Organization

Stefan Brass
Alexander Hinneburg
Heike Stephan
Ramona Vahrenhold
Johannes Waldmann
# Table of Contents

Embedding Defeasible Logic Programs into Generalized Logic Programs .................................................. 5  
Martin Baláž, Jozef Frtús, Martin Homola, Jan Sefranek and Giorgos Flouris

Describing and Measuring the Complexity of SAT encodings for Constraint Programs ............................... 20  
Alexander Bau and Johannes Waldmann

PPI — A Portable Prolog Interface for Java ................................................................................................. 32  
Ludwig Ostermayer, Frank Flederer and Dietmar Seipel

Declarative Evaluation of Ontologies with Rules ......................................................................................... 47  
Dietmar Seipel, Joachim Baumeister and Klaus Prätor

Automated Exercises for Constraint Programming ..................................................................................... 60  
Johannes Waldmann

Complex Certainty Factors for Rule Based Systems — Detecting Inconsistent Argumentations .................. 75  
Taïeb Mellouli
Embedding Defeasible Logic Programs into Generalized Logic Programs

Martin Baláž¹, Jozef Frtúš¹, Martin Homola¹, Ján Šefránek¹, and Giorgos Flouris²

¹ Comenius University in Bratislava, Slovakia
² FORTH-ICS, Greece

Abstract. A novel argumentation semantics of defeasible logic programs (DeLP) is presented. Our goal is to build a semantics, which respects existing semantics and intuitions of “classical” logic programming. Generalized logic programs (GLP) are selected as an appropriate formalism for studying both undermining and rebutting. Our argumentation semantics is based on a notion of conflict resolution strategy (CRS), in order to achieve an extended flexibility and generality. Our argumentation semantics is defined in the frame of assumption-based framework (ABF), which enables a unified view on different non-monotonic formalisms. We present an embedding of DeLP into an instance of ABF. Consequently, argumentation semantics defined for ABF are applicable to DeLP. Finally, DeLP with CRS is embedded into GLP. This transformation enables to commute argumentation semantics of a DeLP via semantics of the corresponding GLP.

1 Introduction

Defeasible Logic Programs (DeLPs) [8] combine ideas from Defeasible Logic [13,12] and Logic Programming. While classically, logic programs (LPs) feature default negation, which enables to express default assumptions (i.e., propositions which are supposed to hold unless we have some hard evidence against them), DeLPs additionally introduce defeasible rules (i.e., rules which are supposedly applicable unless we have some hard evidence opposing them). Strict rules (i.e., regular LP rules) are denoted by $\rightarrow$ and defeasible rules by $\Rightarrow$. Let us illustrate this with an example.

Example 1. Brazil is the home team, and has a key player injured. Home teams tend to work the hardest, and who works the hardest usually wins. The team who has a key player injured does not usually win. The program is formalized into the DeLP:

$\rightarrow home \ r_1: home \Rightarrow works\_hard \ r_2: works\_hard \Rightarrow wins$

$\rightarrow key\_player\_injured \ r_3: key\_player\_injured \Rightarrow not\ wins$
In the program from Example 1 there are two strict and three defeasible rules. The strict rules are facts, hence home and key_player_injured should always be true. Based on the first fact we are able to derive that Brazil should win, using the two defeasible rules $r_1$ and $r_2$, while based on the second fact we are able to derive that Brazil should not win, again relying on a defeasible rule, in this case $r_3$. Hence there is a conflict which somehow should be resolved.

Various approaches to DeLP typically rely on argumentation theory in order to determine which rules should be upheld and which should be defeated. However, as it can be perceived from Example 1, it is not always immediately apparent how this should be decided.

According to García and Simari [8], both rules immediately causing the conflict ($r_2$ and $r_3$) would be undecided, accepting home, key_player_injured and works_hard as valid derivations while taking both wins and not wins as undecided. ASPIC$^+$ [14,11], on the other hand, allows two additional solutions, one with $r_2$ undefeated, $r_3$ defeated, and wins valid; and the other one with $r_2$ defeated, $r_3$ undefeated, and not wins valid.

Some approaches, like ASPIC$^+$, allow to specify a preference relation on rules. In such a case conflict resolution may take this into account. Specifically, ASPIC$^+$ has two built-in conflict resolution strategies, weakest-link principle by which the rule with smallest preference is defeated among those involved in each conflict, and last-link principle by which only the rules immediately causing the conflict are considered and the least preferred is defeated.

We observe that more ways to resolve conflicts may be needed. This is due to the fact that defeasible rules are domain specific, a different conflict resolution strategy may be needed for a different domain, or in distinct application. We therefore argue that the conflict resolution strategy should be a user-specified parameter of the framework, and any DeLP framework should allow a generic way how to specify it (alongside some predefined strategies).

Some of the semantics proposed for DeLP satisfy the well accepted rationality properties for defeasible reasoning, such as consistency (extensions should be conflict-free) and closure (extensions should be closed w.r.t. the strict rules), as defined by Caminada and Amgoud [4]. While these properties are important, DeLP is an extension of LP, and some attention should be also devoted to keeping it in line with it. Specifically, we would like to have the semantics backward-compatible with the underlying language of logic programs – if no defeasible rules are present, the extensions should be in line with the respective class of models.

In our previous work [2] we have formalized the notion of conflict resolution strategy (CRS) and we have proposed a DeLP framework which allows to use any such strategy. The relationship with the underlying class of LPs was not investigated though. In the current paper we extend this work as follows:

- We rebuild the argumentation semantics (including the notion of conflict resolution strategy) using the Assumption Based Framework (ABF), an argumentation formalism very close in spirit to logic programming.
We show that the semantics satisfies the closure and consistency properties
[4], and we also show two additional properties which govern the handling of defeasible rules.

We provide an alternative transformational semantics, which translates the DeLP and the given CRS into a regular logic program. We show that both semantics are equivalent. Thanks to the transformational semantics we also show full backward compatibility with the underlying class of generalized logic programs. What is more, the semantics of DeLP can now be computed using existing LP solvers.

All proofs can be found in a technical report which appears at http://kedrigern.dcs.fmph.uniba.sk/reports/download.php?id=58.

2 Preliminaries

Generalized logic programs and assumption-based frameworks provide a background for our investigation. We are aiming at a computation of our argumentation semantics of DeLP in the frame of classical logic programs. Generalized logic programs (with default negations in the heads of rules) are selected as a simplest LP-formalism, which enables to consider both undermining and rebutting.

Assumption-based frameworks are used in our paper as a basis for building a semantics of DeLP. ABF is a general and powerful formalism providing a unified view on different non-monotonic formalisms using argumentation semantics.

2.1 Generalized Logic Programs

We will consider propositional generalized logic programs (GLPs) in this paper.

Let \( \mathcal{A} \) be a set of atoms and \( \text{not} \mathcal{A} = \{ \text{not} A \mid A \in \mathcal{A} \} \) be a set of default literals. A literal is an atom or a default literal. The set of all literals is denoted by \( \mathcal{L}_\mathcal{A} \). If \( L = \text{not} A \) and \( A \in \mathcal{A} \), then by \( \text{not} L \) we denote \( A \). If \( S \subseteq \mathcal{L}_\mathcal{A} \), then \( \text{not} S = \{ \text{not} A \mid A \in S \} \).

A rule over \( \mathcal{L}_\mathcal{A} \) is an expression \( r \) of the form \( L_1, \ldots, L_n \rightarrow L_0 \) where \( 0 \leq n \) and \( L_i \in \mathcal{L}_\mathcal{A} \) for each \( 0 \leq i \leq n \). The literal \( \text{head}(r) = L_0 \) is called the head of \( r \) and the set of literals \( \text{body}(r) = \{ L_1, \ldots, L_n \} \) is called the body of \( r \).

A generalized logic program is a finite set of rules. We will often use only the term program. If \( \mathcal{A} \) is the set of all atoms used in a program \( \mathcal{P} \), it is said that \( \mathcal{P} \) is over \( \mathcal{A} \). If heads of all rules of a program \( \mathcal{P} \) are atoms, it is said that \( \mathcal{P} \) is normal. A program \( \mathcal{P} \) is called positive if the head of every rule is an atom and the body of every rule is a set of atoms or propositional constants \( t, u, f \).

Note that a GLP \( \mathcal{P} \) can be viewed as consisting of two parts, a normal logic program \( \mathcal{P}^+ = \{ r \in \mathcal{P} \mid \text{head}(r) \in \mathcal{A} \} \) (also called the positive part of \( \mathcal{P} \)) and a set of “constraints” \( \mathcal{P}^- = \mathcal{P} \setminus \mathcal{P}^+ \) (also called the negative part of \( \mathcal{P} \)).

Our definitions of some basic semantic notions follow the ideas of Przymusinski [17] (see also [6]); however, an adaptation to the case of rules with default negations in head is needed. In our approach we will use the positive part \( \mathcal{P}^+ \) of
the program as a generator of a broad set of candidate models and consecutively we will use the negative part $P^-$ to filter out some of the models.

**Definition 1 (Partial and Total Interpretation).** A set of literals $S$ is consistent, if it does not contain a pair $A, \neg A$ where $A \in At$. A partial interpretation is a consistent set of literals. A total interpretation is a partial interpretation $I$ such that for every $A \in At$ either $A \in I$ or not $A \in I$.

Each interpretation can be viewed as a mapping $I: At \mapsto \{0, \frac{1}{2}, 1\}$ where $I(A) = 0$ if not $A \in I$, $I(A) = \frac{1}{2}$ if $A \not\in I$ and not $A \not\in I$, and $I(A) = 1$ if $A \in I$. A valuation given by an interpretation $I$ is a mapping $\hat{I}: \mathcal{A}_{At} \mapsto \{0, \frac{1}{2}, 1\}$ where $\hat{I}(A) = I(A)$ and $\hat{I}(\neg A) = 1 - I(A)$ for each atom $A \in At$, and $\hat{I}(t) = 1$, $\hat{I}(u) = \frac{1}{2}$, $\hat{I}(f) = 0$. An interpretation $I$ satisfies a rule $r$ (denoted $I \models r$) iff $\hat{I}(\text{head}(r)) \geq \hat{I}(\text{body}(r)) = \min\{\hat{I}(L) \mid L \in \text{body}(r)\}$.

**Definition 2 (Model).** An interpretation $I$ is a model of a generalized logic program $P$ iff $I$ satisfies each rule in $P$.

As usual in logic programming, “classical” model, as defined above, are too broad and a number of more fine-grained semantics, based on certain notion of minimality are used. We proceed by defining these semantics summarily for GLPs. Not all of them were thoroughly investigated in literature, however we use analogy with other classes of logic programs, especially normal logic programs.

The notions of truth ordering and knowledge ordering on partial interpretations will be needed. For a partial interpretation $I$, let $T(I) = \{A \in At \mid I(A) = 1\}$ and $F(I) = \{A \in At \mid I(A) = 0\}$.

**Definition 3 (Truth and Knowledge Ordering).** If $I, J$ are partial interpretations, then

- $I \preceq J$ iff $T(I) \subseteq T(J)$ and $F(I) \supseteq F(J)$,
- $I \preceq_k J$ iff $T(I) \subseteq T(J)$ and $F(I) \subseteq F(J)$.

**Definition 4 (Program Reduct).** Let $I$ be an interpretation. The reduct of a normal logic program $P$ is a positive logic program $P^I$ obtained from $P$ by replacing in every rule of $P$ all default literals which are true (resp. unknown, resp. false) in $I$ by propositional constant $t$ (resp. $u$, resp. $f$).

Finally a fixed-point condition is expressed on a reduced program, which is formally captured by the operator $\Gamma_P$.

**Definition 5 (Operator $\Gamma_P$).** Let $P$ be a normal logic program and $I$ be an interpretation. By $\Gamma_P(I)$ we denote the $t$-least model of $P^I$.

**Definition 6 (Semantics Family for GLPs).** Let $P$ be a generalized logic program and $I$ be a model of $P$. Then

- $I$ is a partial stable model of $P$ iff $\Gamma_P^+(I) = I$
- $I$ is a well-founded model of $P$ iff $I$ is a $k$-minimal partial stable model of $P$
– $I$ is a maximal stable model of $P$ iff $I$ is a $k$-maximal partial stable model of $P$.
– $I$ is a least-undefined stable model of $P$ iff $I$ is a partial stable model of $P$ with subset-minimal \( \{ A \in At \mid I(A) = \frac{1}{2} \} \).
– $I$ is a total stable model of $P$ iff $I$ is a partial stable model of $P$ which is total.

The produced semantics properly generalize existing semantics for normal logic programs.

**Proposition 1.** If $P$ is a normal logic program, the notion of partial stable model in Definition 6 coincides with the definition of partial stable models in [17], the notion of total stable model in Definition 6 coincides with the definition of stable models in [10], the notion of well-founded model in Definition 6 coincides with the definition of well-founded model in [9], and the notions of maximal and least-undefined stable model in Definition 6 coincides with the definition of maximal and least-undefined stable models in [18].

If $P$ is a generalized logic program, the definition of stable models in Definition 6 coincides with the definition of stable models in [1].

### 2.2 Assumption-based Framework

Assumption-based frameworks (ABF) [3] enable to view non-monotonic reasoning as a deduction from assumptions. Argumentation semantics of [7, 5] were applied to sets of assumptions. As a consequence, a variety of semantic characterizations of non-monotonic reasoning has been provided.

An ABF is constructed over a deductive system. A **deductive system** is a pair \((L, R)\) where $L$ is a language and $R$ is a set of inference rules over $L$. A **language** is a set $L$ of all well-formed sentences. Each inference rule $r$ over $L$ is of the form $\varphi_1, \ldots, \varphi_n \rightarrow \varphi_0$ where $0 \leq n$ and $\varphi_i \in L$ for each $0 \leq i \leq n$. The sentence $\text{head}(r) = \varphi_0$ is called the **head** of $r$ and the set of sentences $\text{body}(r) = \{\varphi_1, \ldots, \varphi_n\}$ is called the **body** of $r$.

A **theory** is a set $S \subseteq L$ of sentences. A sentence $\varphi$ is an immediate consequence of a theory $S$ iff there exists an inference rule $r \in R$ with $\text{head}(r) = \varphi$ and $\text{body}(r) \subseteq S$. A sentence $\varphi$ is a consequence of a theory $S$ iff there is a sequence $\varphi_1, \ldots, \varphi_n$, $0 < n$, of sentences such that $\varphi = \varphi_n$ and for each $0 < i \leq n$ holds $\varphi_i \in S$ or $\varphi_i$ is an immediate consequence of $\{\varphi_1, \ldots, \varphi_{i-1}\}$. By $Cn_R(S)$ we denote the set of all consequences of $S$.

An **assumption-based framework** is a tuple $F = (L, R, A, \sim)$ where $(L, R)$ is a deductive system, $A \subseteq L$ is a set of assumptions, and $\sim: A \mapsto L$ is a mapping called contrariness function. We say that $\overline{\alpha}$ is the **contrary** of an assumption $\alpha$.

A **context** is a set $\Delta \subseteq A$ of assumptions. We say that $\Delta$ is conflict-free iff $\{\alpha, \overline{\alpha}\} \not\subseteq Cn_R(\Delta)$ for each assumption $\alpha$. A context $\Delta$ is closed iff $\Delta = Cn_R(\Delta) \cap A$, i.e., only such assumptions, which are members of $\Delta$, are derivable from $\Delta$. A context $\Delta$ attacks an assumption $\alpha$ iff $\overline{\alpha} \in Cn_R(\Delta)$. A context $\Delta$ defends an assumption $\alpha$ iff each closed context attacking $\alpha$ contains an assumption attacked by $\Delta$. 


ABFs enable to apply argumentation semantics to sets of assumptions and, consequently, subtle and rich semantic characterizations of sets of assumptions (and of their consequences) can be specified. A closed context $\Delta$ is

- attack-free iff $\Delta$ does not attack an assumption in $\Delta$;
- admissible iff $\Delta$ is attack-free and defends each assumption in $\Delta$;
- complete iff $\Delta$ is admissible and contains all assumptions defended by $\Delta$;
- grounded iff $\Delta$ is a subset-minimal complete context;
- preferred iff $\Delta$ is a subset-maximal admissible context;
- semi-stable iff $\Delta$ is a complete context such that $\Delta \cup \{\alpha \in A \mid \Delta \text{ attacks } \alpha\}$ is subset-maximal;
- stable iff $\Delta$ is attack-free and attacks each assumption which does not belong to $\Delta$.

3 Defeasible Logic Programs

Our knowledge can be divided according to its epistemological status into two categories: on the one hand, one that is gained by deductively valid reasoning and on the other hand, knowledge that is reached by defeasible reasoning [13]. Defeasible logic programs (DeLPs) [15,8,14,11] consider two kinds of rules: strict and defeasible. Strict rules represent deductive reasoning: whenever their preconditions hold, we accept the conclusion. Defeasible rules formalize tentative knowledge that can be defeated and validity of preconditions of a defeasible rule does not necessarily imply the conclusion. Given a set of literals $L_{At}$, a strict (resp. defeasible) rule is an expression $L_1, \ldots, L_n \rightarrow L_0$ (resp. $L_1, \ldots, L_n \Rightarrow L_0$) where $0 \leq n$ and $L_i \in L_{At}$ for each $0 \leq i \leq n$. We will use $\rightsquigarrow$ to denote either a strict or a defeasible rule. Each defeasible rule $r$ has assigned its name $\text{name}(r)$. The name of $r$ is a default literal from separate language $L_N$. The intuitive meaning of $\text{name}(r) = \neg A$ is “by default, the defeasible rule $r$ can be used”, and consequently, the intuitive meaning of not $\text{name}(r) = A$ is “the defeasible rule $r$ can not be used”. In the following, we will denote the defeasible rule $r = L_1, \ldots, L_n \Rightarrow L_0$ with $\text{name}(r) = \neg A$ as $A: L_1, \ldots, L_n \Rightarrow L_0$.

**Definition 7 (Defeasible Logic Program).** Let $A$ be a set of atoms, $N$ be a set of names, and $A \cap N = \emptyset$. A defeasible logic program is a tuple $P = (S,D,\text{name})$ where $S$ is a set of strict rules over $L_{At}$, $D$ is a set of defeasible rules over $L_{At}$, and $\text{name} : D \mapsto \neg N$ is an injective naming function.

3.1 From Arguments to Conflict Resolutions

The argumentation process usually consists of five steps [15,16,8,14,11]. At the beginning, a knowledge base is described in some logical language. The notion of an argument is then defined within this language. Then conflicts between arguments are identified. The resolution of conflicts is captured by an attack relation (also called “defeat relation” in some literature) among conflicting arguments. The status of an argument is then determined by the attack relation. In this
paper, conflicts are not resolved by attacking some of the conflicting arguments, but by attacking some of the weak parts of an argument called *vulnerabilities*. This helps us to satisfy argumentation rationality postulates [4] and to keep the semantics aligned with LP intuitions.

Two kinds of arguments can usually be constructed in the language of defeasible logic programs. Default arguments correspond to default literals. Deductive arguments are constructed by chaining of rules.

We define several functions `prems`, `rules` and `vuls` denoting premises (i.e. default literals) and rules occurring in an argument. Intended meaning of `vuls(A)` is a set of vulnerabilities of an argument `A` (i.e., weak parts which can be defeated) consisting of premises and names of defeasible rules of an argument `A`.

**Definition 8 (Argument).** Let `P = (S, D, name)` be a defeasible logic program. An argument `A` for a literal `L` is

1. a default argument for a literal `L`:
   \[
   \begin{align*}
   \text{prems}(A) &= \{ L \} \\
   \text{rules}(A) &= \emptyset
   \end{align*}
   \]

2. a deductive argument `[A_1, \ldots, A_n \Rightarrow L]` for a literal `L`:
   \[
   \begin{align*}
   \text{prems}(A) &= \text{prems}(A_1) \cup \cdots \cup \text{prems}(A_n) \\
   \text{rules}(A) &= \text{rules}(A_1) \cup \cdots \cup \text{rules}(A_n) \cup \{ r \}
   \end{align*}
   \]

For both kinds of an argument `A`,

\[
\text{vuls}(A) = \text{prems}(A) \cup \text{name}(\text{rules}(A) \cap D)
\]

**Example 2.** Consider a defeasible logic program consisting of the only defeasible rule `r: \neg b \Rightarrow a`. Two default arguments `A_1 = [\neg a]`, `A_2 = [\neg b]` and one deductive argument `A_3 = [A_2 \Rightarrow a]` can be constructed. We can see that `vuls(A_1) = \{\neg a\}`, `vuls(A_2) = \{\neg b\}`, `vuls(A_3) = \{\neg b, \neg r\}`.

The difference between a default and a deductive argument for a literal `\neg A` is in the policy of how the conflict is resolved. Syntactical conflict between arguments is formalized in the following definition. As usual in the literature [14], we distinguish two kinds of conflicts: *undermining* \(^3\) and *rebutting*. While an undermining conflict is about a falsification of a hypothesis (assumed by default), a rebutting conflict identifies a situation where opposite claims are derived.

**Definition 9 (Conflict).** Let `P` be a defeasible logic program. The pair of arguments `C = (A, B)` is called a conflict iff

- `A` is a deductive argument for a default literal `\neg L` and `B` is a deductive argument for the literal `L`; or

\(^3\) Also called *undercutting* in [15].
A is a default argument for a default literal not \( L \) and \( B \) is a deductive argument for the literal \( L \).

The first kind is called a rebutting conflict and the second kind is called an undermining conflict.

The previous definition just identifies the conflict, but does not say how to resolve it; the notion of conflict resolution (to be formalized below) captures a possible way to do so. In our paper, conflicts are not resolved through attack between arguments as in [8,15,14,11], but by attacking some of the vulnerabilities in the conflicting arguments. Since our goal is to define semantics for DeLP respecting existing semantics and intuitions in LP, we assume that all undermining conflicts are resolved in a fixed way as in LP: by attacking the default argument. On the other hand, rebutting conflict is resolved by attacking some defeasible rule. Since, in general, there can be more reasonable ways how to choose which defeasible rules to attack, resolving of all rebutting conflicts is left as domain dependent for the user as an input. Note, that an attack on a defeasible rule \( r \) is formalized as an attack on the default literal \( \text{name}(r) \) which is interpreted as “a defeasible rule \( r \) can be used”.

**Definition 10 (Conflict Resolution).** Let \( \mathcal{P} \) be a defeasible logic program. A conflict resolution is a tuple \( \rho = (A, B, V) \) where \( C = (A, B) \) is a conflict, \( A \) is an argument for not \( L \), and \( V \) is a default literal

- not \( L \) if \( C \) is an undermining conflict; or
- \( \text{name}(r) \) where \( r \) is a defeasible rule in \( \text{rules}(A) \cup \text{rules}(B) \) if \( C \) is a rebutting conflict.

A conflict resolution strategy of \( \mathcal{P} \) is a set \( R \) of conflict resolutions.

Let \( \rho = (A, B, V) \) be a conflict resolution. The contrary of \( V \) is called the resolution of \( \rho \), and denoted by \( \text{res}(\rho) \). The set of vulnerabilities of \( \rho \), denoted by \( \text{vuls}(\rho) \), is defined as:

\[
\text{vuls}(\rho) = \begin{cases} 
(\text{vuls}(A) \cup \text{vuls}(B)) & \text{whenever } V \in \text{vuls}(A) \cap \text{vuls}(B) \\
(\text{vuls}(A) \cup \text{vuls}(B)) \setminus \{V\} & \text{otherwise}
\end{cases}
\]

Usually, there may be more ways how to resolve a conflict and a conflict resolution may resolve other conflicts as well, thus causing other conflict resolutions to be irrelevant or inapplicable. Intuitively, if all vulnerabilities in \( \text{vuls}(\rho) \) are undefeated (i.e. true), then in order to resolve the conflict in \( \rho \), the contrary \( \text{res}(\rho) \) of the chosen vulnerability in \( \rho \) should be concluded (i.e. true). Notions of \( \text{vuls}(\rho) \) and \( \text{res}(\rho) \) will be used for definition of the argumentation semantics in the next subsection.

**Example 3.** Consider the defeasible logic program \( \mathcal{P} = \{\text{not } a \rightarrow a\} \) and undercutting arguments \( A = \{\text{not } a\} \) and \( B = \{\text{not } a \rightarrow a\} \). Then \( \rho = (A, B, \text{not } a) \) is a conflict resolution with \( \text{res}(\rho) = a \) and \( \text{vuls}(\rho) = \{\text{not } a\} \). Please note that although not \( a \) has to be removed to resolve conflict between \( A \) and \( B \), it remains a vulnerability of \( \rho \) since not \( a \) is self-attacking.
**Example 4.** Consider the following defeasible logic program \( \mathcal{P} \)

\[
\begin{align*}
   r_1 &: \Rightarrow a \\
   r_2 &: \Rightarrow b \\
   a &\rightarrow \neg c \\
   b &\rightarrow c
\end{align*}
\]

and arguments \( A = [\Rightarrow a \rightarrow \neg c] \) and \( B = [\Rightarrow b \rightarrow c] \). The rebutting conflict \((A, B)\) can be resolved in two different ways, namely \( \rho_1 = (A, B, \neg r_1) \) is a conflict resolution with \( \text{res}(\rho) = r_1 \) and \( \text{vuls}(\rho) = \{ \neg r_2 \} \), and \( \rho_2 = (A, B, \neg r_2) \) is another conflict resolution with \( \text{res}(\rho) = r_2 \) and \( \text{vuls}(\rho) = \{ \neg r_1 \} \).

The previous example shows that there are more reasonable ways how to resolve rebutting conflicts. We show two examples of different conflict resolution strategies – the weakest-link and the last-link strategy inspired by ASPIC+ [14]. In both strategies, a user-specified preference order \( \prec \) on defeasible rules is assumed. In the last-link strategy, all last-used defeasible rules of conflicting arguments are compared and \( \prec \)-minimal defeasible rules are chosen as resolutions of the conflict. In the weakest-link strategy, each \( \prec \)-minimal defeasible rule of conflicting arguments is a resolution of the conflict.

**Example 5.** Given the defeasible logic program

\[
\begin{align*}
   r_1 &: \Rightarrow b \\
   r_2 &: b \Rightarrow a \\
   r_3 &: \Rightarrow \neg a
\end{align*}
\]

and the preference order \( r_1 \prec r_3, r_1 \prec r_2, r_2 \prec r_3 \), deductive arguments are

\[
A_1 = [\Rightarrow b] \quad A_2 = [A_1 \Rightarrow a] \quad A_3 = [\Rightarrow \neg a]
\]

Then \( R_1 = \{(A_3, A_2, \neg r_3)\} \) is the last-link strategy and \( R_2 = \{(A_3, A_2, \neg r_1)\} \) is the weakest-link strategy.

In the weakest-link strategy from Example 5, a non-last defeasible rule \( r_1 \) is used as a resolution of the conflict. Please note that in [15,8,14,11], such conflict resolutions are not possible, which makes our approach more flexible and general.

### 3.2 Argumentation Semantics

In the previous subsection, definition of an argument structure, conflicts identification and examples of various conflict resolutions were discussed. However, the status of literals and the actual semantics has not been stated.

Argumentation semantics for defeasible logic programs will be formalized within ABF – a general framework, where several existing non-monotonic formalisms have been embedded [3]. In order to use some of the existing argumentation semantics, we need to specify ABF’s language \( \mathcal{L} \), set of inference rules \( \mathcal{R} \), set of assumptions \( \mathcal{A} \), and the contrariness function \( \tilde{-} \). Since ABF provides only one kind of inference rules (i.e. strict), we need to transform defeasible rules into strict. We transform defeasible rule \( r \) by adding a new assumption \( \text{name}(r) \)
into the preconditions of \( r \). Furthermore, chosen conflict resolutions \( R \) determining how rebutting conflicts will be resolved are transformed into new inference rules. Intuitively, given a conflict resolution \( \rho \), if all assumptions in \( \text{vuls}(\rho) \) are accepted, then, in order to resolve the conflict in \( \rho \), the atom \( \text{res}(\rho) \) should be concluded. To achieve this, an inference rule \( \text{vuls}(\rho) \rightarrow \text{res}(\rho) \) for each conflict resolution \( \rho \in R \) is added to the set of inference rules \( \mathcal{R} \).

**Definition 11 (Instantiation).** Let \( \mathcal{P} = (\mathcal{S}, \mathcal{D}, \text{name}) \) be a defeasible logic program built over the language \( \mathcal{L}_{\text{At}} \) and \( R \) be a set of conflict resolutions. An assumption based framework respective to \( \mathcal{P} \) and \( R \) is \((\mathcal{L}, \mathcal{R}, \mathcal{A}, \neg)\) where

\[
\mathcal{L} = \mathcal{L}_{\text{At}} \cup \mathcal{L}_{\neg}, \\
\mathcal{R} = \mathcal{S} \cup \{\text{body}(r) \cup \{\text{name}(r)\} \rightarrow \text{head}(r) \mid r \in \mathcal{D}\} \cup \{\text{vuls}(\rho) \rightarrow \text{res}(\rho) \mid \rho \in R\}, \\
\mathcal{A} = \text{not At} \cup \text{not } \mathcal{N}, \\
\neg \text{not } A = A \text{ for each atom } A \in \text{At} \cup \mathcal{N}.
\]

**Example 6.** Consider the defeasible logic program \( \mathcal{P} \) and the conflict resolution strategy \( R = \{\rho_1, \rho_2\} \) from Example 4. Assumption-based framework respective to \( \mathcal{P} \) and \( R \) is following:

\[
\mathcal{L} = \{a, \text{not } a, b, \text{not } b, c, \text{not } c\} \cup \{r_1, \text{not } r_1, r_2, \text{not } r_2\} \\
\mathcal{R} = \{b \rightarrow \text{not } c, a \rightarrow c\} \cup \{\text{not } r_1 \rightarrow b, \text{not } r_2 \rightarrow a\} \cup \{\text{not } r_1 \rightarrow r_2, \text{not } r_2 \rightarrow r_1\} \\
\mathcal{A} = \{\text{not } a, \text{not } b, \text{not } c\} \cup \{\text{not } r_1, \text{not } r_2\} \\
\neg \text{not } A = A \text{ for each } A \in \{a, b, c\} \cup \{r_1, r_2\}
\]

Now we define the actual semantics for defeasible logic programs. Given an ABF \( \mathcal{F} = (\mathcal{L}, \mathcal{R}, \mathcal{A}, \neg) \), by \( \mathcal{F}^+ \) we denote its flattening – that is, \( \mathcal{F}^+ \) is the ABF \((\mathcal{L}, \{r \in \mathcal{R} \mid \text{head}(r) \notin \mathcal{A}\}, \mathcal{A}, \neg)\).

**Definition 12 (Extension).** Let \( \mathcal{P} = (\mathcal{S}, \mathcal{D}, \text{name}) \) be a defeasible logic program, \( R \) be a set of conflict resolutions, and \( \mathcal{F} = (\mathcal{L}, \mathcal{R}, \mathcal{A}, \neg) \) an assumption-based framework respective to \( \mathcal{P} \) and \( R \). A set of literals \( E \subseteq \mathcal{L} \) is

1. a complete extension of \( \mathcal{P} \) with respect to \( R \) iff \( E \) is a complete extension of \( \mathcal{F}^+ \) with \( \text{Cn}_\mathcal{R}(E) \subseteq E \) and \( \text{Cn}_\mathcal{R}(E') \subseteq E' \);
2. a grounded extension of \( \mathcal{P} \) with respect to \( R \) iff \( E \) is a subset-minimal complete extension of \( \mathcal{P} \) with respect to \( R \);
3. a preferred extension of \( \mathcal{P} \) with respect to \( R \) iff \( E \) is a subset-maximal complete extension of \( \mathcal{P} \) with respect to \( R \);
4. a semi-stable extension of \( \mathcal{P} \) with respect to \( R \) iff \( E \) is a complete extension of \( \mathcal{P} \) with respect to \( R \) with subset-minimal \( E' \setminus E \);
5. a stable extension of \( \mathcal{P} \) with respect to \( R \) iff \( E \) is a complete extension of \( \mathcal{P} \) with respect to \( R \) and \( E' = E \).

where \( E' = \mathcal{L} \setminus \text{not } E \).
Example 7. Consider the assumption-based framework from Example 6. Then $E_1 = \emptyset$, $E_2 = \{\text{not } r_1, r_2, \text{not } a, b, \text{not } c\}$, and $E_3 = \{r_1, \text{not } r_2, a, \text{not } b, c\}$ are complete extensions of $\mathcal{P}$ with respect to $R$. Furthermore, $E_1$ is the grounded extension and $E_2$, $E_3$ are preferred, semi-stable and stable extensions of $\mathcal{P}$ with respect to $R$.

3.3 Transformational Semantics

The argumentation semantics defined above allows us to deal with conflicting rules and to identify the extensions of a DeLP, given a CRS, and hence it constitutes a reference semantics. This semantics is comparable to existing argumentation-based semantics for DeLP, and, as we show below, it satisfies the expected desired properties of defeasible reasoning. In this section we investigate on the relation of the argumentation-based semantics and classical logic programming. As we show, an equivalent semantics can be obtained by transforming the DeLP and the given CRS into a classical logic program, and computing the respective class of models.

In fact the transformation that is required is essentially the same which we used to embed DeLPs with CRS into ABFs. The names of rules become new literals in the language, intuitively if $\text{name}(r)$ becomes true it means that the respective defeasible rule is defeated. By default $\text{name}(r)$ holds and so all defeasible rules can be used unless the program proves otherwise. The conflict resolution strategy $R$ to be used is encoded by adding rules of the form $\text{vuls}(\rho) \rightarrow \text{res}(\rho)$ for each conflict resolution $\rho \in R$, where the head of such rules is always an atom and the body is a set of default literals.

Formally the transformation is defined as follows:

Definition 13 (Transformation). Let $\mathcal{P} = (\mathcal{S}, \mathcal{D}, \text{name})$ be a defeasible logic program and $R$ be a set of conflict resolutions. Transformation of $\mathcal{P}$ with respect to $R$ into a generalized logic program, denoted as $T(\mathcal{P}, R)$, is defined as

$$T(\mathcal{P}, R) = \mathcal{S} \cup \{\text{body}(r) \cup \{\text{name}(r)\} \rightarrow \text{head}(r) \mid r \in \mathcal{D}\} \cup \{\text{vuls}(\rho) \rightarrow \text{res}(\rho) \mid \rho \in R\}$$

Thanks to the transformation, we can now compute the semantics of each DeLP, relying on the semantics of generalized logic programs. Given a DeLP $\mathcal{P}$ and the assumed CRS $R$, the extensions of $\mathcal{P}$ w.r.t. $R$ corresponds to the respective class of models. Complete extensions correspond to partial stable models, the grounded extension to the well-founded model, preferred extensions to maximal stable models, semi-stable extensions to least-undefined stable models, and stable extensions to total stable models.

Proposition 2. Let $\mathcal{P}$ be a defeasible logic program and $R$ be a set of conflict resolutions. Then

1. $E$ is a complete extension of $\mathcal{P}$ with respect to $R$ iff $E$ is a partial stable model of $T(\mathcal{P}, R)$. 

15
2. $E$ is a grounded extension of $P$ with respect to $R$ iff $E$ is a well-founded model of $T(P, R)$.
3. $E$ is a preferred extension of $P$ with respect to $R$ iff $E$ is a maximal partial stable model of $T(P, R)$.
4. $E$ is a semi-stable extension of $P$ with respect to $R$ iff $E$ is a least-undefined stable model of $T(P, R)$.
5. $E$ is a stable extension of $P$ with respect to $R$ iff $E$ is a total stable model of $T(P, R)$.

A remarkable special case happens when the input program $P$ does not contain defeasible rules, and hence it is a regular GLP. In such a case our argumentation-based semantics exactly corresponds to the respective class of models for the GLP. This shows complete backward compatibility of our semantics with the underlying class of logic programs.

**Proposition 3.** Let $S$ be a generalized logic program and $P = (S, \emptyset, \emptyset)$ a defeasible logic program with the empty set of conflict resolutions. Then

1. $E$ is a complete extension of $P$ iff $E$ is a partial stable model of $S$.
2. $E$ is a grounded extension of $P$ iff $E$ is a well-founded model of $S$.
3. $E$ is a preferred extension of $P$ iff $E$ is a maximal partial stable model of $S$.
4. $E$ is a semi-stable extension of $P$ iff $E$ is a least-undefined partial stable model of $S$.
5. $E$ is a stable extension of $P$ iff $E$ is a total stable model of $S$.

4 Properties

In this section we will have a look on desired properties for defeasible reasoning, and show that our semantics satisfies these properties. The properties are formulated in general, that is, they should be satisfied for any a defeasible logic program $P$, any set of conflict resolutions $R$, and any extension $E$ of $P$ w.r.t. $R$.

The first two properties formulated below are well known, they were proposed by Caminada and Amgoud [4]. The closure property originally requires that nothing new can be derived from the extension using strict rules. We use a slightly more general formulation, nothing should be derived using the consequence operator $Cn$ which applies all the strict rules and in addition also all the defeasible rules which are not defeated according to $R$. The original property [4] is a straightforward consequence of this.

**Property 1 (Closure).** Let $R' = S \cup \{\text{body}(r) \cup \{\text{name}(r)\} \rightarrow \text{head}(r) \mid r \in D\}$. Then $Cn_{R'}(E) \subseteq E$.

The consistency property [4] formally requires that all conflicts must be resolved in any extension.

**Property 2 (Consistency).** $E$ is consistent.
In addition we propose two new desired properties which are concerned with
handling of the default assumptions. Reasoning with default assumptions is a
fixed part of the semantics of GLPs (and most other classes of logic programs),
and therefore in DeLPs it should be governed by similar principles. The first
property (dubbed \textit{positive defeat}) requires that for each default literal \( \neg A \),
this literal should be always defeated in any extension \( E \) such that there is a
conflict resolution \( \rho \in R \) whose assumptions are all upheld by \( E \); and, in such a
case the opposite literal \( A \) should be part of the extension \( E \).

\textbf{Property 3 (Positive Defeat).} For each atom \( A \in \mathcal{L}_N \), if there exists a conflict
resolution \( \rho \in R \) with \( \text{res}(\rho) = A \) and \( \text{vuls}(\rho) \subseteq E \) then \( A \in E \).

The previous property handles all cases when there is an undefeated evidence
against not \( A \) and requires that \( A \) should hold. The next property (dubbed \textit{nega-
tive defeat}) handles the opposite case. If there is no undefeated evidence against
not \( A \), in terms of a conflict resolution \( \rho \in R \) whose assumptions are all upheld
by \( E \), then not \( A \) should be part of the extension \( E \).

\textbf{Property 4 (Negative Defeat).} For each default literal \( \neg A \in \mathcal{L}_N \), if for each
conflict resolution \( \rho \in R \) with \( \text{res}(\rho) = A \) holds \( \text{not vuls}(\rho) \cap E \neq \emptyset \) then \( \neg A \in E \).

Closure and consistency trivially hold for our semantics, as the semantics
was designed with these properties in mind. They are assured by the definition
of complete extension of a DeLP (Definition 12).

\textbf{Proposition 4.} Each complete extension \( E \) of a defeasible logic program \( \mathcal{P} \) with
respect to a set of conflict resolutions \( R \) has the property of closure.

\textbf{Proposition 5.} Each complete extension \( E \) of a defeasible logic program \( \mathcal{P} \) with
respect to a set of conflict resolutions \( R \) is consistent.

Satisfaction of a positive and a negative defeat properties follow from the
instantiation of an ABF (Definition 11), where an inference rule \( \text{vuls}(\rho) \rightarrow \text{res}(\rho) \)
is added for every conflict resolution \( \rho \in R \).

\textbf{Proposition 6.} Each complete extension \( E \) of a defeasible logic program \( \mathcal{P} \) with
respect to a set of conflict resolutions \( R \) has the property of positive defeat.

\textbf{Proposition 7.} Each complete extension \( E \) of a defeasible logic program \( \mathcal{P} \) with
respect to a set of conflict resolutions \( R \) has the property of negative defeat.

5 Related Work

There are two well-known argumentation-based formalisms with defeasible in-
ference rules – defeasible logic programs [8] and ASPIC\textsuperscript{+} [14,11]. It is known
that the semantics in [8] does not satisfy rationality postulates formalized in [4],
especially the closure property. Although ASPIC\textsuperscript{+} satisfies all postulates in [4],
it uses transposition or contraposition which violate directionality of inference rules [2] and thus violating LP intuitions. It also does not satisfy positive or negative defeat property introduced in this paper.

Francesca Toni’s paper [19] describes a mapping of defeasible reasoning into assumption-based argumentation framework. The work takes into account the properties [4] that we also consider (closedness and consistency), and it is formally proven that the constructed assumption-based argumentation framework’s semantics is closed and consistent. However no explicit connection with existing LP semantics is discussed in [19].

The paper [20] does not deal with DeLP, but on how to encode defeasible semantics inside logic programs. The main objective is on explicating a preference ordering on defeasible rules inside a logic program, so that defeats (between defeasible logic rules) are properly encoded in LP. This is achieved with a special predicate defeated with special semantics.

Caminada et al. [6] investigated how abstract argumentation semantics and semantics for normal logic programs are related. Authors found out that abstract argumentation is about minimizing/maximizing argument labellings, whereas logic programming is about minimizing/maximizing conclusion labellings. Further, they proved that abstract argumentation semantics cannot capture the least-undefined stable semantics for normal logic programs.

6 Conclusion

In this paper we investigated the question of how to define semantics for defeasible logic programs, which satisfies both the existing rationality postulates from the area of structured argumentation and is also aligned with LP semantics and intuitions. To achieve these objectives, we diverged from the usual argumentation process methodology. Most importantly, conflicts are not resolved by attacking some of the conflicting arguments, but by attacking some of the weak parts of an argument called vulnerabilities. Main contributions are as follows:

- We presented an argumentation semantics of defeasible logic programs, based on conflict resolution strategies within assumption-based frameworks, whose semantics satisfies desired properties like consistency and closedness under the set of strict rules.
- We constructed a transformational semantics, which takes a defeasible logic program and a conflict resolution strategy as an input, and generates a corresponding generalized logic program. As a consequence, a computation of argumentation semantics of DeLP in the frame of GLP is enabled.
- Equivalence of a transformational and an argumentation semantics is provided.

Acknowledgements

This work resulted from the Slovak–Greek bilateral project “Multi-context Reasoning in Heterogeneous environments”, registered on the Slovak side under no.
SK-GR-0070-11 with the APVV agency and co-financed by the Greek General Secretariat of Science and Technology and the European Union. It was further supported from the Slovak national VEGA project no. 1/1333/12. Martin Baláž and Martin Homola are also supported from APVV project no. APVV-0513-10.

References

Describing and Measuring the Complexity of SAT encodings for Constraint Programs

Alexander Bau⋆ and Johannes Waldmann

HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany
abau|waldmann@imn.htwk-leipzig.de

Abstract. The CO⁴ language is a Haskell-like language for specifying constraint systems over structured finite domains. A CO⁴ constraint system is solved by an automatic transformation into a satisfiability problem in propositional logic that is handed to an external SAT solver. We investigate the problem of predicting the size of formulas produced by the CO⁴ compiler. The goal is to help the programmer in understanding the resource consumption of CO⁴ on his program. We present a basic cost model, with some experimental data, and discuss ongoing work towards static analysis. It turns out that analysis steps will use constraint systems as well.

1 Introduction

CO⁴ is a constraint programming language that allows to write a constraint problem as declarative specification. The CO⁴ compiler solves it by transforming the constraint to a propositional satisfiability problem, so that a SAT solver can be applied. Syntactically, the language is a subset of the purely functional programming language Haskell [3] that includes user-defined algebraic data types and recursive functions defined by pattern matching, as well as higher-order polymorphic types.

In CO⁴, a constraint system over elements of set U is specified by a parametrized predicate constraint : P × U → Bool, where P denotes the parameter domain. Thus, constraint does not denote a single constraint, but a family of constraints. For a given constraint and parameter p ∈ P, u ∈ U is a solution if constraint (p, u) = True.

For the CO⁴ compiler to generate a propositional encoding, the input constraint is transformed into an abstract program constraint' that operates on abstract values. An abstract value represents an undetermined value of the input program by encoding the constructor’s index using propositional formulas. Evaluating the abstract program generates the final formula that is passed to the external SAT solver.

It is desirable to predict the runtime of the SAT solver for a generated propositional encoding. Such a prediction is hard because as it depends on a lot of design and implementation decisions of the SAT solver. Therefore we take the

⋆ This author is supported by ESF grant 100088525
size of the SAT encoding as a reasonable indicator for its hardness. To estimate the size of the encoding, we introduce a cost model for abstract values and abstract programs. This cost model captures two important facts: the size of intermediate abstract values and the costs to evaluate them. Especially the evaluation of case distinctions on abstract values is not obvious, and often they cannot be evaluated in a straightforward manner.

This paper has three parts. The first part illustrates the syntax and semantics of CO⁴ (Section 2) and gives an overview on of the propositional encoding (Section 3). This is a summary of material that has already been published[1]. The second part presents current work on cost analysis: in Section 4 we present our cost model, and in Section 5 we analyze the cost of the merge operation, which is a basic operator in our translation scheme. Section 6 illustrates how the current CO⁴ implementation measures concrete costs of SAT-compiled function calls. The third part outlines future work in static analysis of CO⁴ programs. Section 7 describes moded types and their inference, which will allow a more efficient propositional encoding of case distinctions. Section 8 describes an approach to bound function costs by resource types.

## 2 Syntax and Semantics of CO⁴

Syntactically, CO⁴ is a subset of Haskell. Domains are specified by algebraic data types (ADT), where constructors enumerate the values of the type.

```haskell
1 data Bool = False | True
2 data Color = Red  | Green | Blue
3 data Monochrome = Black | White
4 data Pixel    = Colored Color | Background Monochrome
```

CO⁴ supports recursive ADTs as well, but recursions must be restricted while generating a propositional encoding. We do not deal with recursions in the scope of this paper.

A constructor may be parametrized either by types or type variables.

```haskell
1 data Pair a b = Pair a b
2 data Either a b = Left a  | Right b
```

Inspecting the constructor of a value d of some ADT is done by a case distinction on d (the discriminant of the case distinction):

```haskell
1 case color of Blue   -> True
2                otherwise -> False
```

Case distinctions provides conditional branching of the control-flow. Other kinds of expressions in CO⁴ are constructor calls, applications, abstractions and local bindings. CO⁴ provides restricted support of higher-order, polymorphic functions. Besides type definitions, constraint systems in CO⁴ contain global function bindings with constraint being the top-level function:
data Bool = False | True
data Color = Red | Green | Blue
data Monochrome = Black | White
data Pixel = Colored Color | Background Monochrome

constraint :: Bool → Pixel → Bool
constraint p u = case p of
    False → case u of Background m → True
    otherwise → False
    True → isBlue u

isBlue :: Pixel → Bool
isBlue u = case u of
    Colored color → case color of Blue → True
    otherwise → False
    Background m → False

Listing 1.1. A trivial constraint over pixels

Semantically, a constraint system in CO⁴ over elements of set \( U \) is a binary predicate \( \text{constraint} : P \times U \rightarrow \text{Bool} \) on \( U \) and some parameter domain \( P \). In Listing 1.1, \( P = \text{Bool} \) and \( U = \text{Pixel} \).

For a given parameter \( p \in P \), \( u \in U \) is a solution if \( \text{constraint} \ (p, u) = \text{True} \). One advantage of specifying constraint systems in a functional language like CO⁴ is that a solution can be tested against the constraint simply by evaluating \( \text{constraint} \ (p, u) \). Note that CO⁴ expressions are evaluated strictly, while Haskell features a non-strict evaluation strategy.

3 Propositional Encoding of CO⁴ constraints

In the following, we call the source constraint a \textit{concrete program}. Concrete programs operate on \textit{concrete values}, e.g., the concrete program in Listing 1.1 operates on concrete values like \textit{False}, \textit{White} or \textit{Colored Red}.

To find a solution \( u \in U \) for a constraint \( \text{constraint} : P \times U \rightarrow \text{Bool} \) and a parameter \( p \in P \), CO⁴ performs the following steps:

1. The concrete program is transformed into an \textit{abstract program}. An abstract program doesn’t operate on concrete values, but on \textit{abstract values}.
2. Evaluating the abstract program for an abstract value that represents parameter \( p \) gives a formula \( f \in \mathcal{F} \) in propositional logic.
3. An external SAT solver is called to find a satisfying assignment \( \sigma \in \Sigma \) for \( f \).
4. If there is a satisfying assignment, the solution \( u \in U \) is constructed from \( \sigma \). Optionally, testing whether \( \text{constraint} \ p \ u = \text{True} \) ensures that there are no implementation errors. This check must always succeed if there is a solution.

In the following we briefly illustrate the first two steps of this process. Firstly, an abstract program is generated from a given concrete program. This transformation not only modifies the program structure, the domain is changed as well.
Data Transformation

An abstract program is an untyped, first-order and imperative program on abstract values.

Definition 1. Assume \( \mathcal{F} \) being the set of propositional formulas. Then, the set of abstract values \( \mathcal{A} \) is the smallest set with \( \mathcal{A} = \mathcal{F}^* \times \mathcal{A}^* \) where \( \mathcal{F}^* \) denotes the set of sequences with elements from \( \mathcal{F} \). An abstract value \( a \in \mathcal{A} \) is a tuple \((\overrightarrow{f}, \overrightarrow{a})\) of flags \( \overrightarrow{f} \) and arguments \( \overrightarrow{a} \).

An abstract value \( a \in \mathcal{A} \) represents a (maybe unknown) value of a concrete type \( T \). The flags of an abstract value \( a \in \mathcal{A} \) encode the indices of \( T \)’s constructors in binary code using propositional formulas.

Example 1. For an abstract value \( a_1 \in \mathcal{A} \) to represent a value of the ADT `data Color = Red | Green | Blue | Purple` it must contain two flags \( f_1, f_2 \in \mathcal{F} \) because `Color` has four constructors. Thus, \( a_1 = ((f_1, f_2), ()) \). \( a_1 \) has no arguments because none of `Color`’s constructors has any arguments.

Consider an ADT `data Maybe a = Nothing | Just a` and an abstract value \( a_2 \in \mathcal{A} \) that is supposed to represent a value of type `Maybe Color`. As `Maybe` consists of two constructors, one flag \( f_3 \in \mathcal{F} \) is needed to discriminate both. Thus, \((f_3, a_1)\) is a proper value for \( a_2 \). Note that \( a_2 \) has a single argument \( a_1 \) that encodes the constructor argument of type `Color` of `Maybe`’s `Just` constructor.

As the flags of an abstract value \( a \in \mathcal{A} \) may contain propositional variables, \( a \) can be decoded to different values according to the Boolean values that are assigned to these variables. By \( \text{decode}_T : \mathcal{A} \times \Sigma \to T \) we denote a mapping from abstract values and propositional assignments \( \Sigma \) to concrete values.

If the flags of an abstract value \( a \in \mathcal{A} \) don’t contain propositional variables, then the flags of \( a \) index a particular constructor and \( a \) can only be decoded to a single concrete value. By \( \text{encode}_T : T \to \mathcal{A} \) we denote a mapping from concrete values to abstract values that represent a fixed value.

Example 2. Recall the ADTs defined in Example 1 and assume the flags of an abstract value reference a constructor’s index using binary code where the first flag encodes the most significant bit. Then:

\[
\text{encode}_{\text{Color}}(\text{Blue}) = ((\text{True}, \text{False}), ()) \\
\text{encode}_{\text{Maybe Color}}(\text{Just Blue}) = (\text{True}, ((\text{True}, \text{False}), ()))
\]

As we’ve omitted details about abstract values we don’t provide definitions for \( \text{encode} \) and \( \text{decode} \).

Program Transformation

The program structure of abstract programs resembles the structure of their concrete counterparts. The most important difference concerns case distinctions: while concrete values may be examined by matching on their constructor, this is often not possible for abstract values. That’s because an abstract value’s flags may contain propositional variables. Therefore, it
is undetermined which constructor is indexed by the flags and there is no way to known which branch to evaluate. Thus, all branches must be evaluated and their result is merged according to the discriminant of the case distinction.

Example 3. The following case distinction matches on a Boolean value \( x \) in a concrete program:

\[
    r = \text{case } x \text{ of } \{ \text{False} \rightarrow g \ ; \text{True} \rightarrow h \}
\]

In the abstract counterpart of this expression, the abstract values \( g' \) and \( h' \) of both branches are evaluated and merged according to \( x \)

\[
    r' = \text{let } \_1 = g' \\
    \_2 = h' \\
    \text{in } \text{merge}_{x'}(\_1, \_2)
\]

where \( r' \) (resp. \( x', g', h' \)) denote the abstract counterpart of \( r \) (resp. \( x, g, h \)).

The function \( \text{merge}_x : \mathbb{A}^n \rightarrow \mathbb{A} \) encodes a case distinction on a value \( x \in \mathbb{A} \) using the flags of \( x \) and the abstract values of all evaluated branches. We don’t give a definition for \( \text{merge} \), but illustrate its semantics by the following example.

Example 4. Recall the case distinction in Example 3 and assume \( r' \) (resp. \( x', g', h' \)) denotes the abstract counterpart of \( r \) (resp. \( x, g, h \)). The following two clauses are emitted when evaluating \( \text{merge}_{x'}(g', h') \):

\[
    (x' \equiv \text{encode}_\text{Bool}(\text{False}) \implies r' \equiv g') \\
    \land (x' \equiv \text{encode}_\text{Bool}(\text{True}) \implies r' \equiv h')
\]

Informally, both clauses encode the semantics of the original case distinction in terms of abstract values: \( r' \) equals \( g' \) if \( x' \) equals \( \text{encode}_\text{Bool}(\text{False}) \), otherwise \( r' \) equals \( h' \).

However, if none of the flags of an abstract value contain any propositional variables, then the constructor that is indexed by these flags can be determined and the associated branch can be evaluated. In this case it is not necessary to evaluate the other branches.

Evaluation of Abstract Programs The constraint : \( P \times U \rightarrow \text{Bool} \) function in a concrete program has a counterpart constraint' : \( \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{A} \) in the abstract program of the same arity. Evaluating constraint' \( p' \ u' \) on

- \( p' = \text{encode}_P(p) \) for some parameter \( p \in P \), and
- \( u' \in \mathbb{A} \), which represents a undetermined value in \( U \),

gives a value \( a \in \mathbb{A} \) that represents a Boolean value, i.e., \( a \) contains a single flag \( f \in \mathbb{F} \). Solving \( f \) using an external SAT solver gives a satisfying assignment \( \sigma \in \Sigma \) for all variables in \( f \) if there is such an assignment. The final solution \( u \in U \) can be constructed by decode_U(\( u' \), \( \sigma \)).

We refer to [1] for more technical details on the transformation process.
4 Cost Model

We illustrate an approach for formalizing the costs associated with a function in a CO\(^4\) program. For readability we stick to unary functions and omit details about functions of higher arities.

We measure the cost of a function \( f : A \rightarrow B \) in a concrete program by analyzing its counterpart \( f' : A \rightarrow A \) in the corresponding abstract program. The costs of \( f' \) depend on the size of its argument. Thus, we introduce a function \( \text{size} : A \rightarrow \mathbb{N} \) to measure the size of an abstract value.

Example 5. There are at least two naive definitions for \( \text{size} \): one that counts the number of nested abstract values

\[
\text{size}_1(f, (a_1, \ldots, a_n)) = 1 + \sum_{i=1}^{n} \text{size}_1(a_i)
\]

and one that counts the number of flags in an abstract value

\[
\text{size}_2((f_1, \ldots, f_m), (a_1, \ldots, a_n)) = m + \sum_{i=1}^{n} \text{size}_2(a_i)
\]

Fixing a particular implementation for \( \text{size} \), the cost of the abstract function \( f' \) is described by a pair of functions \( s_f, c_f : \mathbb{N} \rightarrow \mathbb{N} \).

Definition 2. \( s_f(n) \) gives the maximal output size for all arguments of \( f' \) with size \( n \) or smaller:

\[
s_f(n) = \max\{\text{size}(f(\text{encode}_A(x))) \mid x \in A \land \text{size}(\text{encode}_A(x)) \leq n\}
\]

Whereas \( s_f \) quantifies the size of a function’s result, \( c_f \) measures the evaluation costs of \( f' \).

Definition 3. \( c_f(n) \) gives the evaluation costs for all arguments of \( f' \) with size \( n \) or smaller

\[
c_f(n) = \max\{\text{work}(f, \text{encode}_A(x)) \mid x \in A \land \text{size}(\text{encode}_A(x)) \leq n\}
\]

where \( \text{work}(f, x) \) equals the cost of evaluating \( f'(\text{encode}_A(x)) \) in the abstract program.

We can instantiate this scheme in several ways: for example, \( \text{work}(f, x) \) could give the number of propositional variables or clauses that are allocated while computing the abstract value \( f'(\text{encode}_A(x)) \). Other techniques may include additional characteristics about the propositional encoding, like the number of literals or the depth of the formula.
5 Cost of merge

Example 4 illustrated the semantics of the merge operation on abstract values. Now we quantify the cost of merge in terms of the cost model in Section 4.

Assume the following case distinction with $n$ branches $b_1, \ldots, b_n$ in a function $f$:

$$f(x) = \text{case } x \text{ of } C_1 \rightarrow b_1 \ldots \text{ } C_n \rightarrow b_n$$

Listing 1.2. A case distinction over $n$ branches

where $f' : A \rightarrow A$ denotes the abstract counterpart of $f$. In order to evaluate $f'(x')$ for some abstract argument $x' \in A$ we need to evaluate all abstract branches $b'_i \in A$ for $i \in [1, n]$ and merge the results by $\text{merge}_{x'}(b'_1, \ldots, b'_n)$. We denote the result of this merge by $r' \in A$.

A first cost measure determines the numbers of variables that are needed to represent the result of an application of merge (variable-cost).

**Definition 4.** $\text{work}_V(f, x)$ denotes the variable-cost of function $f$ in Listing 1.2 and equals the maximum number of flags in the branches, i.e., if $m_i$ denotes the number of flags in branch $b'_i$ for $i \in [1, n]$, then

$$\text{work}_V(f, x) = \max\{m_i \mid 1 \leq i \leq n\}$$

As the result of $f'(x)$ must equal one of the branches $b'_i$ (c.f. Example 4) it is reasonable for $\text{work}_V$ to assume that $r'$ must consist of the maximum number of flags that are present in the abstract branches $b'_i \in A$ for $1 \leq i \leq n$.

Furthermore, we define the clause-cost of an application of merge. Example 4 illustrated that the flags in a result of merge encode the case distinction in terms of abstract values. The clause-costs represent the number of clauses in a propositional formula that are needed to encode a case distinction.

**Definition 5.** $\text{work}_C(f, x)$ denotes the clause-cost of function $f$ in Listing 1.2 where $n$ denotes the number of branches:

$$\text{work}_C(f, x) = 2 \times \text{work}_V(f, x) \times n$$

$\text{work}_C$ is reasonable because two clauses are emitted for each of the $\text{work}_V(f, x)$ flags in $r'$ and each of the $n$ branches.

6 Profiling CO4

In the following we compare the profiling output of CO4 for some examples and show the relation to the previously defined cost-model.

The first example illustrates the difference between the cost of evaluating a concrete program and an abstract program.
data Bool = False | True deriving Show
data T = T1 | T2 | T3 deriving Show

g :: T -> Bool
g t = case t of
  T1 -> True
  T2 -> False
  T3 -> False

f1 :: Bool -> Bool
f1 b = case b of
  False -> g T1
  True -> g T2

f2 :: Bool -> Bool
f2 b = g (case b of
  False -> T1
  True -> T2)

Listing 1.3. Profiling two semantically equivalent functions

Listing 1.3 defines two functions f1,f2 with the same concrete semantics. Assume f1’ (resp. f2’,g’) being the abstract counterpart of f1 (resp. f2,g). Further assume that b ∈ A is an abstract value that represents an undetermined value of type Bool. Then, evaluating f1’ b gives

("f1’", {numCalls = 1, numVariables = 1, numClauses = 4})
("g’", {numCalls = 2, numVariables = 0, numClauses = 0})

g’ does not allocate any variables nor clauses as its argument is constant in both calls g T1 and g T2 in the concrete program. Thus, the case distinction in g’ can be evaluated straightforwardly without applying merge.

f1’ is called once and allocates one variable (resp. four clauses). That matches the work\_V (resp. work\_C) cost function, because

- work\_V(f1, b) = max\{1, 1\} = 1 as each branch in f1’ is represented by an abstract value with one flag (because Bool has two constructors)
- work\_C(f1, b) = 2 * work\_V(f1, b) * 2 = 4 as there are n = 2 branches in f1’

On the other hand, evaluating f2’ b gives

("f2’", {numCalls = 1, numVariables = 2, numClauses = 8})
("g’", {numCalls = 1, numVariables = 1, numClauses = 6})

Again, the profiling information matches with the cost functions work\_V and work\_C defined in Section 5, because

- work\_V(f2, b) = max\{2, 2\} = 2 as each branch in f2’ is represented by an abstract value with two flags (because T has three constructors)
- work\_C(f2, b) = 2 * work\_V(f2, b) * 2 = 8 as there are n = 2 branches in f2’
- work\_V(g, t) = max\{1, 1, 1\} = 1 as each branch in g’ is represented by an abstract value with one flag (because Bool has two constructors)
- work\_C(g, t) = 2 * work\_V(g, t) * 3 = 6 as there are n = 3 branches in g’
Note that $f_2'$ allocates more variables than $f_1'$ because it merges branches of type $T$, which has more constructors than $\text{Bool}$. In the second case, $g'$ is only called once, but this time with an unknown argument: its argument indirectly depends on the unknown $b$. Thus, $g'$ allocates variables and emits clauses.

We give a more complex example: CO$^4$ has been applied to problems of termination analysis of term rewriting systems. One exemplary problem is the specification of a lexicographic path order (LPO) that proves the termination of a given term rewriting system$^1$.

**Listing 1.4. Exemplary inner-under-profiling**

Listing 1.4 shows the inner-under-profiling for a LPO constraint. For each function $f$ in the abstract program, inner-under-profiling associates the number of variables and clauses to $f$ that has been allocated by $f$ and by all functions transitively called in $f$. Unsurprisingly, $\text{constraint}'$ allocates the most resources according to inner-under-profiling as it is the top-level function in every abstract program.

**Listing 1.5. Exemplary under-profiling**

Listing 1.5 shows the under-profiling for a LPO constraint. For each function $f$ in the abstract program, under-profiling only associates the number of variables and clauses to $f$ that has been allocated by $f$. Listing 1.5 shows that for the LPO constraint the abstract function $\text{gtNat}'$ allocates the most propositional variables.

CO$^4$ also provides information about the number of variables and clauses allocated in the abstract program as a whole:

#variables: 167, #clauses: 517, #literals: 1365

---

$^1$ available at https://github.com/apunktbau/co4/blob/master/test/CO4/Example/LPO.hs
We give one more example: Listing 1.6 shows the profiling data for a CO\(^4\) specification of the \(n\)-queens problem (with \(n = 8\))\(^2\).

Profiling (inner-under):

- \("\text{constraint}'", \{\text{numCalls} = 1, \text{numVariables} = 2324, \text{numClauses} = 6447\}\)
- \("\text{allSafe}'", \{\text{numCalls} = 9, \text{numVariables} = 2251, \text{numClauses} = 6237\}\)
- \("\text{safe}'", \{\text{numCalls} = 36, \text{numVariables} = 2244, \text{numClauses} = 6217\}\)
- \("\text{noAttack}'", \{\text{numCalls} = 28, \text{numVariables} = 2216, \text{numClauses} = 6140\}\)
- \("\text{equal}'", \{\text{numCalls} = 717, \text{numVariables} = 1724, \text{numClauses} = 5100\}\)
- \("\text{noDiagon}'", \{\text{numCalls} = 28, \text{numVariables} = 1488, \text{numClauses} = 4012\}\)
- \("\text{noStraight}'", \{\text{numCalls} = 28, \text{numVariables} = 700, \text{numClauses} = 2044\}\)

Profiling (inner):

- \("\text{equal}'", \{\text{numCalls} = 717, \text{numVariables} = 1724, \text{numClauses} = 5100\}\)
- \("\text{add}'", \{\text{numCalls} = 359, \text{numVariables} = 352, \text{numClauses} = 704\}\)
- \("\text{and2}'", \{\text{numCalls} = 101, \text{numVariables} = 100, \text{numClauses} = 291\}\)
- \("\text{not}'", \{\text{numCalls} = 84, \text{numVariables} = 84, \text{numClauses} = 168\}\)
- \("\text{less}'", \{\text{numCalls} = 65, \text{numVariables} = 64, \text{numClauses} = 184\}\)

#variables: 2397, #clauses: 6522, #literals: 16697

**Listing 1.6.** Exemplary profiling for the \(n\)-queens problem (with \(n = 8\))

Here, inner-profiling reveals that the \text{equal}' function allocates the most resources. This is reasonable because the \(n\)-queens constraint pair-wisely compares the position of all queens in order to exclude all possibilities for two queens to attack each other.

### 7 Moded Types and Mode Inference

For the future work on CO\(^4\), we plan to develop a mode inference system that allows the generation of propositional encodings with fewer variables and clauses. That is desirable as smaller formulas are often solved in less time by a SAT solver.

Moded types allow the differentiation between expressions that are constant during abstract evaluation and expressions that are not. This information would allow the CO\(^4\) compiler to determine case distinctions that have a constant discriminant, i.e., that can be evaluated during abstract evaluation without allocating any propositional variables.

In this context, a mode is either ! or ?. Mode ! states that the constructor of a value is known during abstract evaluation, while mode ? states that the constructor of a value is not known during abstract evaluation. A moded type is a type that has been annotated by modes. For example, \text{List}\(^!\) \text{Bool}\(^?\) denotes a list type, where each of list constructor is known, but each element of type \text{Bool} has an unknown constructor. Thus, such a type encodes a list of known length with unknown Boolean elements.

We consider a moded program to be a typed program where each type is annotated by modes. For a moded program to be dynamically well-moded, it is required that the constructor of all case distinctions' discriminants must be constant that have mode $\text{!}$, i.e., their flags are constant.

We plan to develop a static mode analysis as a safe approximation for dynamically well-moded programs. One possible approach for a mode inference algorithm is the construction of a Boolean constraint (because there are two different modes) that can be solved by a SAT solver.

A similar approach has been successfully applied to infer modes in for the Mercury language[4].

8 Resource Types and Resource Inference

Mode analysis allows a more strict analysis on the estimated cost for a $\text{CO}^4$ constraint system.

A possible approach to predict the resource cost is to annotate each function in a $\text{CO}^4$ constraint with a resource type, where a resource type for function $f$ according to the cost model introduced in Section 4 is a pair of functions $s_f, c_f : \mathbb{N} \rightarrow \mathbb{N}$.

A dynamically well-resource-typed program is a program where each function $f$ has a resource type annotation, so that for each call of $f$ with argument $x$ the actual cost $\text{work}(f, \text{encode}(x))$ is less or equal to $c_f(\text{size}(\text{encode}(x)))$ for some cost function $\text{work}$ (see Section 4).

A resource-typed program is considered statically well-typed, if all resource annotations are consistent with some sound set of rules for cost of case distinctions, merge operations and function compositions.

These rules should guarantee that the static resource type is a safe approximation for actual costs. We are especially interested in polynomial upper bounds.

Related work consists of amortized resource analysis in Resource Aware ML (RAML)[2], where polynomial potential functions are used as costs functions. The coefficients for these polynomial are determined by a constraint system.

We want to emphasize again that this approach is ongoing work and there are currently no results nor experimental data to verify it. We plan to extend this approach into a reasonable formalism to capture the resource constraints of $\text{CO}^4$ programs in order to estimate the size of the propositional encodings.

References


PPI- A Portable PROLOG Interface for JAVA

Ludwig Ostermayer, Frank Flederer, Dietmar Seipel
University of Würzburg, Department of Computer Science
Am Hubland, D – 97074 Würzburg, Germany
{ludwig.ostermayer,dietmar.seipel}@uni-wuerzburg.de

Abstract. As a first step to combine the two programming paradigms – object-oriented programming and logic programming – we have introduced a generic default mapping for JAVA objects and PROLOG terms. This mapping can be used without any modification to the JAVA classes that stand behind the objects. We also can generate automatically JAVA classes from predicates in PROLOG that map to each other. Apart from the default mapping it is further possible to customise the mapping by JAVA annotations. This allows for different Prolog-Views on a given class in JAVA. The data exchange format between JAVA and PROLOG is a simple textual representation of the JAVA objects and of the terms in PROLOG. Because this textual representation already conforms PROLOG’s syntax, it can be directly used within PROLOG.

In a second step we have to develop the link between JAVA and PROLOG that executes the mapping and communication. We already have presented a connector architecture for PROLOG and JAVA and an example interface that successfully uses our object term mapping with high performance. However, this interface depends heavily on a single PROLOG implementation: SWI-PROLOG. But as we have a generic mapping between JAVA objects and PROLOG terms, we also strive for an interface that also is generic and can be used independently of the PROLOG implementation.

In this paper, we present the Portable Prolog Interface (PPI) for JAVA that uses the standard streams stdin, stdout and stderr to communicate with a PROLOG instance. Because these standard streams are available for all popular operating systems and are used by most of the PROLOG implementations for the user interaction, the PPI works for a broad range of PROLOG implementations. We evaluate our new generic interface PPI with different PROLOG engines and without changing the underlying JAVA or PROLOG source code of our tests.

Keywords. Multi-Paradigm Programming, Logic Programming, Prolog, Java.

1 Introduction

The object-oriented software engineering concept is one of the most used in the field of software development. However, there are other programming paradigms which are eminently suitable for particular problem domains. One of those programming concepts is the logic programming paradigm. The best known logical programming language is PROLOG. PROLOG programs consist of a collection of rules that describe horn clauses.

In PROLOG programs, these rules are used by the inference mechanism to find solutions
for which the rules are true. Due to the simple definition of those rules, PROLOG is eligible, for instance, for a simple definition of business rules.

It is desirable to combine different programming paradigms in order to use the strength of each individual concept for suitable parts in a piece of software. Several approaches for an interaction between JAVA and PROLOG have been proposed in the past. But most of those concepts are specialized to specific PROLOG implementations. The benefit of such a strong binding to a PROLOG implementation leads to a good performance, but it lacks of portability across several PROLOG implementations. A well known interface between JAVA and PROLOG is JPL. JPL, however, makes use of the foreign language interface (FLI) of SWI-PROLOG. Because of this strong binding to SWI-PROLOG it is not easily usable for other PROLOG implementations as XSB- or YAP-PROLOG. Another interface for PROLOG and JAVA is Interprolog. Also this interface is heavily dependent of a single PROLOG implementation: XSB-PROLOG. Although support for SWI- and YAP-PROLOG are announced, they have to put a lot of effort into porting Interprolog to those other PROLOG implementations. There are other implementations of interfaces between JAVA and PROLOG that attach importance to the portability regarding PROLOG implementations. One of those interfaces is JPC [5]. But also for this interface much porting effort has to be expended.

The main contribution of this paper is for our connector a new generic Portable Prolog Interface (PPI) which uses our object term mapping [14] for a smooth communication between JAVA and PROLOG. The PPI extends our connector architecture for PROLOG and JAVA as presented in [15] and allows the usage of the connector with several PROLOG implementations and operating systems. The PPI uses the standard streams stdin, stdout and stderr to communicate with a PROLOG instance. These standard streams are part of the operating systems and are also used by most of the PROLOG implementations for the interaction with users.

The rest of the paper is structured as follows: In Section 2 we discuss work that is related to the results presented in this paper. In Section 3 we recap our object term mapping and how it is realised for JAVA and PROLOG. This is followed by the presentation of the PPI in Section 4 which we evaluate in Section 5. Finally in Section 6, we conclude and discuss future work.

2 Related Work

The results in this paper are the continuation of our work with a portable connector architecture for JAVA and PROLOG that allows a smooth communication between both programming languages.

In [13], we have presented the framework PBR4J (PROLOG Business Rules for JAVA) that allows to request a given set of PROLOG rules from a JAVA application. To overcome the interoperability problems, a JAVA archive (JAR) has been generated containing methods to query the set of PROLOG rules. PBR4J uses XML Schema to describe the data exchange format. From the XML Schema description, we have generated JAVA classes for the JAVA archive. For our connector the mapping information for JAVA objects and PROLOG terms is not saved to an intermediate, external layer. It is part of the JAVA class we want to map and though we can get rid of the XML Schema as used in
Either the mapping is given indirectly by the structure of the class or directly by
annotations. While BR4J just provides with every JAR only a single PROLOG query,
we are now able to use every object as goal in PROLOG and depending on which vari-
ables are bound different queries are possible. BR4J transmitted along with a request
facts in form of a knowledge base. The result of the request was encapsulated in a result
set. With our connector we do not need any more wrapper classes for a knowledge base
and the result set as it was with BR4J. That means that we have to write less code in
JAVA. We either assert facts from a file or persist objects with JAVA methods directly to
PROLOG’s database.

The generic mapping mechanism as described in Section 3 was introduced first
in [14]. Using the default mechanism, nearly every class in JAVA can be mapped to
a PROLOG term without any modification to the class’ source code. Additionally, if
a customised mapping is needed, we provide JAVA annotations in order to realise the
modification easily. These annotations do not affect the original program code in JAVA.
Apart from the mapping, we have proposed the Prolog-View-Notation (PVN) to de-
scribe existing PROLOG terms and to create JAVA classes that map under the default
mapping to the described terms. Because there are nested references of different JAVA
objects, the mapping is done recursively. However, we observed a mapping anomaly
that we call Reference Cycle. A Reference Cycle occurs if a JAVA object is referenced
by itself. Using our mapping mechanism as proposed this leads to a cyclic term. We
want to avoid cyclic terms because our mapping mechanism in this case leads to infi-
nite long strings in JAVA. We have proposed a solution that replaces an attribute which
is a member of the Reference Cycle by a list that contains all referenced objects. If any
of those objects is referenced, a reference identifier is used instead of the nested term
representations. Because the list is restricted to the referenced objects, we avoid the
cyclic term problem and are able to map objects that have a self-reference.

In [15] we have introduced our general connector architecture for JAVA and PRO-
LOG. The presented implementation of the connector is lightweight. The object term
mapping is realised with only two classes. As communication interface we have pre-
sented the Prolog-Interface (PI) for SWI-PROLOG. The PI uses the Foreign Language
Interface (FLI) of SWI-PROLOG and is therefore only applicable for this single PRO-
LOG implementation. An evaluation of the performance has shown that the PI in com-
bination with our object term mapping is as fast as JPL [16], a highly optimized JAVA
interface for SWI-PROLOG. However, the implementations for the evaluation with our
connector have proven simpler, clearer and shorter as with the reference JPL. We re-
quired with our connector 25% less lines of code than with JPL.

There are other approaches how to establish a communication between JAVA and
PROLOG like [3,4,5,7,10] that we already have discussed in [14,15]. In contrast to our
work, all these approaches are limited to single PROLOG implementations and none of
these approaches allow the mapping of already existing classes to terms in PROLOG,
especially without any modifications to the underlying source code. The implementa-
tion of our connector itself is much more lightweight and programs using our connector
need less lines of code than with the other approaches.
3 The Object Term Mapping between JAVA and PROLOG

In [14] we have proposed a customisable mapping between JAVA objects and PROLOG terms. The mapping provides a default mapping of JAVA classes to PROLOG terms. This makes it possible to use almost any already existing JAVA class without any modification to the classes in JAVA. Thus, a JAVA developer can make use of PROLOG functionalities with minimal effort. If the default mapping of JAVA objects to PROLOG terms does not match already existing PROLOG predicates or any needed data structure on the PROLOG side, we also have proposed a customisable mapping of JAVA objects to PROLOG terms. This customisation allows the user to change the functor as well as the composition and the type of a term’s arguments. In this section we want to recap the default and the customisable mapping of JAVA objects to PROLOG terms with some examples.

3.1 Default Mapping

To provide JAVA developers an easy way to use PROLOG, our approach implements a smart default mapping. For the default mapping JAVA classes can be used without any modifications. The JAVA programmer does not need to know the syntax and only little of the functionalities in PROLOG in order to establish a connection.

The mapping target of an object is a term in PROLOG, also referred as target term in this paper. The default mapping from a JAVA object to a PROLOG term uses the object’s class name as the target term’s functor. Class names in JAVA are usually written in Upper Camel Case notation. But upper first characters in predicate names are not allowed in PROLOG because functors are atoms. Atoms in PROLOG usually begin with lowercase characters, otherwise the predicate’s name must be escaped by surrounding single quotes, e.g. 'Book'.

For the default mapping, we have decided to convert the in JAVA common Camel Case notation to the in PROLOG common Snake Case notation. This is done, by replacing uppercase characters by their lowercase equivalent and add an underscore prefix, if the character is not the first one. For instance, the class’ name MyBook maps to the atom my_book.

Classes usually contain member variables. The default mapping just maps every member variable to an argument of the target term. This is done, by getting all these variables via Java Reflections. The order of the arguments in the target term is given by the getDeclaredFields() method in JAVA. According to JavaDoc, there is no assured order but Oracle’s JVM (JAVA Virtual Machine) returns an array of fields that is sorted by the position of the variables’ declarations in the JAVA class files.

Another aspect of our default mapping is the fix conversion of some types in JAVA to certain types/structures in PROLOG. A natural mapping of JAVA types (to PROLOG) is as follows: short (integer), int (integer), long (integer), float (float), double (float), String (atom), Array (PROLOG list), List (PROLOG list) and Object (compound PROLOG term). For other data types, only existing in certain PROLOG implementations like string in SWI-PROLOG [18], the default mapping can be further extended or changed. It is possible to save these changes to the default mapping to a configuration file.

A special part in logical programming are logical variables. The inference mechanism of PROLOG tries to assign valid values to them for which the rules of the PROLOG
program are true. Because different values for a logical variable might be true, several solutions can be found for a single request made available through backtracking.

The inference mechanism and the unification of logical variables to valid values is a strong feature of PROLOG. In order to make this unification process available in JAVA we have introduced the concept of Object Unification.

When transforming JAVA objects to PROLOG terms, we transform specific variables of an object into a logical variable in PROLOG. We map null values in JAVA to variables in PROLOG. More precise, every time we map an object to PROLOG, all member variables with a null value are substituted in PROLOG with different variables. Then, these variables can be unified within the inference process of PROLOG. Finally, a in PROLOG unified term leads to a substitution of the initial null values in JAVA by the values of the in PROLOG unified variables.

To illustrate the default mapping mechanism, different implementations of a Book class follow. In each step, further details are implemented in order to show another detail of the default mapping mechanism. The first Book class does not implement any member variable at all:

```java
class Book {
}
```

When we create a instance of the Book class and transform it via the default mapping to a term in PROLOG, any instance will result in the same term with an arity of zero:

```prolog
book
```

The name of the class Book is transformed to a snake case notation which in this case just leads to the lowercase book.

We extend the Book class of the previous implementation by the title of the book:

```java
class Book {
    private String title;
    // ... constructor\ getter\ setter
}
```

For lack of space, we omit the implementation details of a class’ constructor, getter and setter methods. Now, we create again an instance:

```java
Book b = new Book();
b.setTitle("Sophie's World");
```

The target term in PROLOG then looks like in the following listing:

```prolog
book('Sophie\'s World')
```

The single member variable title is used for the single argument of the book term. Because the member variable is of the data type String in JAVA, it is transformed by the default mapping into an atom in PROLOG.

In the next step we extend the Book class by another member variable, the amount of pages. This time, we use the int data type in JAVA:
class Book {
    private String title;
    private int pages;
    // ... constructor\ getter\ setter
}

We create again an instance of the Book class:

Book b = new Book();
b.setTitle("Sophie's World");
b.setPages(518);

The instance b of Book then is transformed via the default mapping to:

book('Sophie's World', 518)

The resulting term in PROLOG contains the values of both member variables, because the default mapping just transforms all of the member variables of a JAVA class. The pages variable, however, is not set within quotes, as the default mapping transforms a JAVA int to an integer in PROLOG. The order of the two member variables within the PROLOG term is defined by the return of the getDeclaredFields() method of the JAVA reflection API. In this case, the sorting of the member variables is the declaration order of them within the Book class.

We give now an example where one of the member variables is set to null. For this, we instantiate a Book object and do not set the pages member variable:

Book b = new Book();
b.setTitle("Sophie's World");

Not setting the pages amount leads to an uninitialized variable that is set to null. According to the default mapping, all member variables that have a value of null are transformed to variables in PROLOG:

book('Sophie's World', Book@123_pages)

The name of the logical variable is composed of the object’s reference in JAVA (Book@123) and the name of the member variable (pages). Generating the name of logical variables this way we obtain an unique variable name. Its uniqueness results from the unique object reference within a JAVA program and the unique name of the member variable within a JAVA class. Even if the same JAVA object is transformed multiple times to PROLOG, it is sufficient. Because we need just one single unification for a member variable of an object, the multiple occurrence of a member variable of the same object, leads to a single unification on the PROLOG side.

But we are not limited by transforming a single JAVA object to PROLOG. Referenced objects are transformed recursively. To show this, we extend the book class again and create a new class named Author:

class Book {
    private String title;
    private Author author;
}
As one can see, we now have a reference from the `Book` class to the `Author` class.

```java
private int pages;
// ... constructor\ getter\ setter
}
class Author {
private String name;
// ... constructor\ getter\ setters
}
```

Author a = new Author();
a.setName("Jostein Gaarder");

Book b = new Book();
b.setTitle("Sophie's World");
b.setAuthor(a);

The target term in PROLOG of the book `b` now is a complex compound term:

```prolog
book('Sophie\'s World', author('Jostein Gaarder'),
    Book@123_pages)
```

The `Author` class just contains a single member variable, so does the resulting target term in PROLOG. The resulting term for the author object `a` is contained as argument within the target term for the book `b`.

### 3.2 Customised Mapping

If the default mapping does not map an object to a desired term structure, the user is able to modify the mapping with a special purpose annotation layer in JAVA. With JAVA annotations we can add the necessary meta-data of a desired mapping to the source code in JAVA. Note, that annotations are not part of a JAVA program, i.e. they do usually not affect the code itself they annotate. Annotations are parsed in JAVA with the methods of the Reflection API. To customise the mapping between objects and terms we only need three annotations in a nested way: `@PlView`, `@Arg` and `@PlViews`.

`@PlView` is used to describe a single Prolog-View on a given class in JAVA. With Prolog-View we mean single mapping of a given class in JAVA to term in PROLOG. It is possible to define different Prolog-Views on the same class. We achieve this with different `@PlView` annotations which are collected within a `@PlViews` annotation in the given class. A `@PlView` annotations consists of several elements:

- `viewId` is a mandatory element and identifies the Prolog-View. The predicate name normally set by the default mapping can be written over by the element `functor`. There are three remaining elements of a `@PlView` annotation that are lists consisting of strings: `orderArgs`, `ignoreArgs` and `modifyArgs`. These lists are used to manipulate the structure of the target term corresponding to the desired Prolog-View.

- `orderArgs` determines which member variable values, defined by their JAVA names, are used within the textual term representation. As the name `orderArgs` suggests the order of members in this list matters. The order of the resulting term arguments corresponds to the order of the member variable names in this list.
ignoreArgs removes one or a few member variables from the default mapping. The user simply can add the names of the ignored member variables in this list instead of writing all the other names into the orderArgs list. As ignoreArgs contains all the missing arguments, there is no order information that describes the order of the arguments left over to the mapping. Therefore, the relative order of the arguments within the default mapping is unaffected. The user is told not to use orderArgs and ignoreArgs together within the same @PlView annotation, as this could lead to anomalies like member variable names that are in both or in none of the two lists. To prevent an accidental wrong use, an exception is raised if both elements are used together within an @PlView annotation. The orderArgs and ignoreArgs are just to define which member variables are considered for the mapping and which not, as well as the order of those parameters.

modifyArgs modifies the mappings of single member variables to arguments of the target terms. It is an array consisting of @Arg annotations.

@Arg has three elements for the modifications: valueOf, type and viewId. As long as there is no @PlArg annotation in an @PlView annotation, the default mapping is applied for all mapped member variables.

valueOf references the name of the member variable whose mapping is going to be manipulated by the @Arg annotation. In a single @PlView annotation only one @Arg annotation is allowed for every member variable referenced by valueOf.

type defines the PROLOG type which will be used within the target term in PROLOG. Options for type are elements of an enumeration representing certain PROLOG types like atom, float, integer or structures like compound term, list. In case of compound term and list, it is possible that again several Prolog-Views in a referenced class exist. Though, the user again can select a Prolog-View for the referenced class via an according viewId. The type compound is always a reference to another object. Because an object can be mapped to a list, the type list can also be a reference.

viewId is used for member variables that reference a class for which different Prolog-Views are defined.

To illustrate the meaning of the different annotations and their arguments, we give an example of a Book class with three different Prolog-Views defined on it by @PlView annotations:
The first Prolog-View, identified by `book1`, just selects the member variable `title` to be part of the resulting PROLOG term. Therefore, the resulting term just contains the title of the book.

The second Prolog-View `book2` uses the selection list `ignoreArgs`. The only entry in this list is the member variable `author`. That means all the other member variables, namely `title` and `pages`, are still contained in the resulting target term in PROLOG.

The last Prolog-View `book3` has no restriction on the included member variables at all. Thus, all member variables are mapped to PROLOG. However, two modifications to the mapping are done within the annotation: the functor of the target term is changed from `book` to `tome`; the type in PROLOG of the mapping of the member variable `pages` is changed in PROLOG from integer as in the default mapping to atom. Therefore, the resulting term in PROLOG has a single quoted value for the pages of the book.

The resulting three target terms in PROLOG are summarised in the following listing:

```
book('Sophie\'s World').
book('Sophie\'s World', 518).
tome('Sophie\'s World', author('Jostein Gaarder'), '518').
```

3.3 Creating Textual Term Representations

All the information needed for the creation of textual term representations can be derived from the classes involved in the mapping. The default mapping uses the information of the class structure itself. The customised mapping uses the information contained in the JAVA annotations `@PlView` that are identified by the string `viewId`. As in [15] shown the object to term conversion as well as the parsing is implemented in a wrapper class called `OTT` (Object-Term-Transformer). An example for the usage of

![Fig. 1: A tree of OTT Objects](image)

OTT instances is shown in Figure 1. The object `o1` is destined to be unified in PROLOG. It has references to two other objects `o2` and `o3` which lead to a nested term structure in PROLOG. The class `Query` is a wrapper for a call to PROLOG. To its constructor `o1` is passed and an instance of `OTT` is created, here `ott1`. For all the other references in
instances of OTT are created in a nested way, namely ott2 for o2 and ott3 for o3. In order to create the textual term representation of o1, the instance query causes ott1 to call its toTerm() method that triggers a recursive call of toTerm() in all involved instances of OTT. In doing so, the first operation is to determine which fields have to be mapped. Depending on a requested Prolog-View or the default mapping an array of Field references is created that contains all the needed member variables for the particular view in the corresponding order. The information about the Fields is retrieved with help of the Reflection API in JAVA. The same way, additional information like PROLOG types and viewIds for particular member variables are saved within such arrays. As the information of a Prolog-View on a class is solid and does not change with the instances, this field information is just created once and cached for further use. For the creation of the textual term representation, the functor is determined either from a customised functor element of an @PlView annotation or from the class name in the default case. After that, the Field array is iterated and the string representation for its elements are created. The pattern of those strings depend on the PROLOG type that is defined for a member. If a member is a reference to another object, the toTerm() method for the reference is called recursively.

3.4 Parsing Textual Term Representations

After query has received the textual representation of the unified term from PROLOG, it is parsed to set the unified values to the appropriate member variables of the JAVA objects involved. The parsing uses again the structure of nested OTT objects as shown in Figure 1. The class OTT has the method fromTerm(String term). This method splits the passed string into functor and arguments. The string that contains all the arguments is split into single arguments. This is done under consideration of nested term structures. According to the previously generated Field array the arguments are parsed. This parsing happens in dependence of the defined PROLOG type of an argument. For instance, an atom either has single quotes around its value or, if the first character is lowercase, there are no quotes at all. If there is a quote detected, it is removed from the string before assigning it as a value for the appropriate member variable. Assignments for referenced objects in o1 are derived recursively by calling the fromTerm(String term) method of the appropriate instances of OTT, in our example ott2 and ott3.

4 PPI - The Portable PROLOG Interface

In a previous paper [15] we already have presented our connector architecture for PROLOG and JAVA. Figure 2 gives an overview of the components and how the connector works. An object is transformed to a PROLOG term in string format via the object term mapping (OTM) as described in Section 3. This string is transmitted via a Prolog-Interface (PI). Then the string is parsed in PROLOG. Because the string already conforms to PROLOG’s syntax, this is an easy task. The resulting term is unified and sent back again in string format which is finally processed by a parser in JAVA. The used PI
can be any Prolog Interface for JAVa that is available for the considered PROLOG implementation. We already have implemented a high performance PI for SWI-PROLOG based on its Foreign Language Interface. The combination of our mapping and the PI for SWI has been optimized to the point where our connector works as fast as an implementation with JPL [16] which is the standard JAVA interface bundled with SWI. It is more complex to operate with JPL than with our connector. We have this quantified by the amount of lines of code necessary to call PROLOG from JAVA: an implementation with our connector needs 25% less lines of code. In addition, a user developing with JPL must have a deeper understanding for PROLOG and its structures.

However, the PI is only applicable with SWI-PROLOG. In order to conserve an independence regarding the PROLOG implementations we introduce in this paper a generic interface suitable for almost every PROLOG implementation and operating systems, the Portable Prolog Interface (PPI). Instead of a specialized interface between JAVA and a certain PROLOG implementation, we use standard streams of every operating system to connect to a PROLOG process: the standard input (stdin), the standard output (stdout) and the standard error (stderr). Every PROLOG implementation usually provides user interaction via these streams. To write as user a request directly to PROLOG the stdin stream is used. The output is channelled via the stdout or stderr stream to a user interface. The output contains the resulting bindings of the variables that have been unified by PROLOG’s inference engine.

Our object term mapping can be combined with the PPI in a native way because the textual term representation that we transmit already conforms to PROLOG’s syntax. Our connector using the PPI now is deployable for a broad range of operating systems and PROLOG implementations. Normally, these streams are used for writing and calling goals as well as for getting the variable bindings of a solution. In addition, the user is able to kick off features in most PROLOG systems like backtracking by typing the character semicolon. This is just an input for stdin and therefore our interface is also able to use such meta commands.
Another difference of the PPI and the interface PI in [15] is that the PI returned the unified term as a whole. Using the standard streams PROLOG returns only the binding of the variables, e.g. X=4. In order to make this separate bindings usable for our mapping process, the variables in the initial term are replaced by the appropriate bindings. Fig. 4 shows the schematic flow between the individual pipes that process the information flow. When opening a connection from a JAVA program to a PROLOG engine two classes are initialised in JAVA: OutPipe and InPipe. The class OutPipe shares the main thread of the underlying JAVA program and has the job to write text to PROLOG’s stdin. When the unify method is called within the JAVA program, OutPipe writes the textual term representation, which should be unified, to PROLOG’s stdin. The class InPipe runs a separate thread and receives the results from PROLOG by reading the stdout or stderr stream.

To avoid unnecessary overhead the two threads are paused if they are not needed. Because we want the result as return of the method unify, we pause the calling thread in order to wait for the result from PROLOG. We have sent a request to PROLOG and await the result to be written to PROLOG’s stdout or stderr. Therefore, after writing the text to stdin, the InPipe thread is resumed in order to collect the unification result. As soon as the resulting data has arrived via stdout or stderr, the InPipe returns it to OutPipe which resumes its computations and thread of InPipe is paused. After the result, in form of variable bindings, is converted back to the textual representation of the in PROLOG unified term, the resulting string is returned by unify.

5 Evaluation

In this section we evaluate the combination of the generic PPI with our connector architecture for PROLOG and JAVA. We have implemented three tests for the evaluation. For the computations, we have successfully used the following freely available PROLOG implementations: B-, CIAO-, GNU-, SWI-, YAP- and XSB-PROLOG. In doing so, no modifications to the original program files in JAVA and PROLOG have been neces-
ecessary. We have tested consecutively 50000 calls. The resulting average execution time for the different PROLOG implementations are presented in the tables that follow the short descriptions of the tests. Note, that the resulting execution times in tables always include the time necessary to process the goal on the different PROLOG engines.

In the first test the goal, that we send from JAVA to PROLOG, is just the atom true which is always true and needs no unification. We do this, to better estimate the time that only is spent for establishing a connection and the transmission of the data.

<table>
<thead>
<tr>
<th></th>
<th>B-P</th>
<th>Ciao</th>
<th>GNU</th>
<th>SWI</th>
<th>XSB</th>
<th>YAP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.0 sec</td>
<td>15.5 sec</td>
<td>3.0 sec</td>
<td>7.4 sec</td>
<td>3.7 sec</td>
<td>2.9 sec</td>
</tr>
</tbody>
</table>

The second test has two different implementations. The first implementation sends as goal to PROLOG a variable assignment to an atom consisting of 100 characters. The second implementation has an increased character count of 1000 for the assigned atom. The purpose of this test is to analyse the influence of the length of a goal, measured in characters, on the execution time.

<table>
<thead>
<tr>
<th>Characters</th>
<th>B-P</th>
<th>Ciao</th>
<th>GNU</th>
<th>SWI</th>
<th>XSB</th>
<th>YAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4.4 sec</td>
<td>15.5 sec</td>
<td>4.2 sec</td>
<td>12.5 sec</td>
<td>5.5 sec</td>
<td>8.6 sec</td>
</tr>
<tr>
<td>1000</td>
<td>18.4 sec</td>
<td>33.4 sec</td>
<td>18.5 sec</td>
<td>40.5 sec</td>
<td>18.6 sec</td>
<td>61.7 sec</td>
</tr>
</tbody>
</table>

Third test considers underground railway networks, as in [15] the London Underground. These networks are represented as undirected graphs with stations as nodes and lines as edges connecting the individual stations. In PROLOG this is simply realised via the facts connected. The first and the second argument of a connected fact is a station. The third argument is the line connecting the two stations. The next listing gives some examples for connected facts for the London Underground:

```prolog
connected(station(green_park), station(charing_cross),
           line(jubilee)).
connected(station(bond_street), station(green_park),
           line(jubilee)).
...
```

In this third test we request for a station adjacent to a given station and line. We process the graphs for the London Underground, Sydney and Vienna. The number of edges in these graphs decreases from London with 412 edges over Sydney with 284 edges to Vienna with only 90 edges.

<table>
<thead>
<tr>
<th></th>
<th>B-P</th>
<th>Ciao</th>
<th>GNU</th>
<th>SWI</th>
<th>XSB</th>
<th>YAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>London</td>
<td>7.8 sec</td>
<td>25.5 sec</td>
<td>4.6 sec</td>
<td>15.3 sec</td>
<td>8.5 sec</td>
<td>5.0 sec</td>
</tr>
<tr>
<td>Sydney</td>
<td>6.3 sec</td>
<td>22.8 sec</td>
<td>4.4 sec</td>
<td>15.2 sec</td>
<td>7.7 sec</td>
<td>4.9 sec</td>
</tr>
<tr>
<td>Vienna</td>
<td>5.2 sec</td>
<td>22.0 sec</td>
<td>3.8 sec</td>
<td>13.7 sec</td>
<td>7.6 sec</td>
<td>4.7 sec</td>
</tr>
</tbody>
</table>

\[on Core i5 2x2.4 GHz, 6 GB RAM, Ubuntu 14.04\]
As one can see in the second table the amount of data which is transmitted to and from PROLOG has a huge influence on the execution time. All the times of the execution with 1000 characters are up to 7 times slower than the execution with 100 characters.

In the last table, one can see that the decrement of execution time for the PROLOG inference mechanism has only a slight effect on the complete execution time including the input and output operations. The slowest execution in this evaluation is the 1000 character benchmark for YAP. But 61.7 seconds for 50000 executions means an execution time of 1.234 milliseconds for a single execution. Because in real world applications 50000 executions in a row are unusual, this delay of about one millisecond is still a good value.

6 Conclusions and Future Work

In this paper we have presented the portable PROLOG interface (PPI) for our connector architecture. The PPI is based on standard streams that are part of nearly every operating system and are used by most PROLOG implementations for the user interaction. In a first evaluation we could verify the applicability of the PPI within our connector for several PROLOG implementations, and that with a decent performance. This way, we have improved the portability of our connector architecture for PROLOG and JAVA.

In a future work, we want to extend our tests with the PPI to other PROLOG implementations, maybe to commercial PROLOG systems, too. In addition, we currently work on the integration of existing high performance interfaces into our connector. In doing this, we expect to get better execution times for our mapping technique between JAVA and PROLOG.

References


   [http://www.swi-prolog.org](http://www.swi-prolog.org)
Abstract. Currently, the extension of ontologies by a rule representation is a very popular research issue. A rule language increases the expressiveness of the underlying knowledge in many ways. Likewise, the integration creates new challenges for the design process of such ontologies, but also existing evaluation methodologies have to cope with the extension of ontologies by rules. In this work, we introduce supplements to existing verification techniques to support the design of ontologies with rule enhancements, and we focus on the detection of anomalies that can especially occur due to the combined use of rules and ontological definitions.

Keywords. evaluation, anomalies, OWL, SWRL, RULEML, PROLOG, DATALOG

1 Introduction

The use of ontologies has shown its benefits in many applications of intelligent systems in the last years. It is not only a fantasy of computer scientists, but it corresponds to real needs. E.g., in scholarly editions, the lack of semantic search is very obvious. The works of the poet Stifter, e.g., are full of geological metaphors but the word geological is never mentioned. The philosopher Wittgenstein is dealing with philosophical problems, but does not use traditional philosophical terminology. So the editors are considering an ontology for the work of Wittgenstein. In the context of [PZW13], Pichler and Zöllner–Weber were also exploring the potential of PROLOG ontologies and logic reasoning as tools in the Humanities [Zoe09].

Whereas, the implementation of lower parts of the semantic web stack has successfully led to standardizations, the upper parts, especially rules and the logic framework, are still heavily discussed in the research community, e.g., see Horrocks et al. [3]. This insight has led to many proposals for rule languages compatible with the semantic web stack, e.g., the definition of SWRL (semantic web rule language) originating from RULEML and similar approaches [4]. It is well agreed that the combination of ontologies with rule–based knowledge is essential for many interesting semantic web tasks, e.g., the realization of semantic web agents and services. SWRL allows for the combination of a high–level abstract syntax for Horn–like rules with OWL, and a model theoretic semantics is given for the combination of OWL with SWRL rules. An XML syntax derived from RULEML allows for a syntactical compatibility with OWL. However, with the increased expressiveness of such ontologies, new demands for the development
and for maintenance guidelines arise. Thus, conventional approaches for evaluating and
maintaining ontologies need to be extended and revised in the light of rules, and new
measures need to be defined to cover the implied aspects of rules and their combination
with conceptual knowledge in the ontology.

Concerning the expressiveness of the ontology language we focus on the basic sub-
set of OWL DL (that should make the work transferable to ontology languages other
than OWL) and we mostly describe syntactic methods for the analysis of the considered
ontology. We also focus on the basic features of SWRL: we consider Horn clauses with
class or property descriptions as literals, and we omit a discussion of SWRL built–ins.
Due to the use of rules with OWL DL the detection of all anomalies is an undecidable
task, cf. [4].

Here, the term verification denotes the syntactic analysis of ontologies for detecting
anomalies. On one hand, the discussed issues of the presented work originate from the
evaluation of taxonomic structures in ontologies introduced by Gómez–Pérez [5]. On
the other hand, in the context of rule ontologies classical work on the verification of
rule–based knowledge has to be reconsidered as done, e.g., by Preece and Shinghal [6,
7]. In their works, the verification of ontologies (mostly taxonomies) and rules (based on
predicate logic), respectively, has been investigated separately. However, the combina-
tion of taxonomic and other ontological knowledge with a rule extension leads to new
evaluation metrics that can cause redundant or even inconsistent behavior. The main
contribution of our work is the extension of these measures by novel anomalies that
are emerging from the combination of rule–based and ontological knowledge. Here, the
concept of dependency graphs from deductive databases can be used [8]. Of course, the
collection of possible anomalies may always be incomplete, since additional elements
of the ontology language may also introduce new possibilities of occurring anomalies.

In detail, we investigate the implications and problems that can be drawn from rule
definitions in combination with some of the following ontological descriptions: 1. class
relations like subclass of, complement of, disjointness 2. basic property characteristics
like transitivity, ranges and domains, and cardinality restrictions. We distinguish the
following classes of anomalies:

- **Circularity** in taxonomies and rule definitions.
- **Redundancy** due to duplicate or subsuming knowledge.
- **Inconsistency** because of contradicting definitions.
- **Deficiency** as a category comprising subtle issues describing questionable design in
  an ontology.

The presented work is different from the evaluation of an ontology with respect
to the intended semantic meaning: the OntoClean methodology [9] is an example for
semantic checks of taxonomic decisions made in an ontology. We also do not consider
common errors that can be implemented due to the incorrect understanding of logical
implications of OWL descriptions as described by Rector et al. [12].

This paper is organized as follows: The next section gives basic definitions and
describes the expressiveness of the underlying knowledge representation; in the context
of this work a subset of OWL DL is used. Then, the four main classes of anomalies are
discussed. In Section 3, we present a case study with anomalies in OWL ontologies. The
paper is concluded with a discussion.
2 Expressiveness and Basic Notions

For the analysis of ontologies with rules, we restrict the range of the considered constructs to a subset of OWL DL: we investigate the implications of rules that are mixed with subclass relations and/or the property characteristics transitivity, cardinality restrictions, complement, and disjointness.

Given a class \( C \) and a property \( P \). When used in rules, we call \( C(x) \) a class atom and \( P(x, y) \) a property atom. For the following it will be useful to extend the relations on classes and properties to relations on class and property atoms. Given two atoms \( A, A' \), we write \( \circ(A, A') \), if both atoms have the same argument tuple, and their predicate symbols are related by \( \circ \), i.e., if \( A \) and \( A' \) both are

- class atoms, such that \( A = C(x), A' = C'(x) \), and \( \circ(C, C') \), or
- property atoms, such that \( A = P(x, y), A' = P'(x, y) \), and \( \circ(P, P') \).

E.g., the relation \( \circ \) can be sub_class, isa, disjoint, complement, etc. From a relationship \( \circ(A, A') \) it follows that \( A \) and \( A' \) are of the same type.

2.1 Implementation in DATALOG*

The detection of anomalies has been done using a PROLOG meta-interpreter DATALOG*, which we have implemented in SWI PROLOG [13]. Due to their compactness and conciseness, we give the corresponding formal definitions for the anomalies, which are evaluated using a mixed bottom-up/top-down approach based on DATALOG and PROLOG concepts, respectively.

Variables such \( A, B, C, \ldots, A' \), or \( B_i \) can denote both class atoms and property atoms, whereas \( A_s, B_s, \ldots \), denote sets of class atoms and property atoms. We denote a relationship \( A \) is-a \( A' \) by \( \text{isa}(A, A') \). SWRL rules \( B_1 \land \cdots \land B_n \Rightarrow A \) are represented as non-ground DATALOG* facts \( \text{rule}(A-B_s) \) (with variable symbols), where \( B_s = [B_1, \ldots, B_n] \) is the list of body atoms and \( A \) is the head atom. Since SWRL rules with conjunctive rule heads can be split into several rules, we can – without loss of generality – assume rule heads are atomic. In DATALOG* and PROLOG rules, conjunction (and) is denoted by \( \land \), disjunction (or) is denoted by \( \lor \), and negation by \( \neg \).

**Incompatible Classes: Complements and Disjointness.** For classes, there exists the construct \( \text{complementOf} \) to point to instances that do not belong to a specified class. In DATALOG*, the complement relation between two classes \( C_1 \) and \( C_2 \) is denoted by \( \text{complement}(C_1, C_2) \). In OWL, the disjointness between two classes is defined by the \( \text{disjointWith} \) constructor; with \( \text{disjoint}(C_1, C_2) \) we denote the disjointness between two classes \( C_1 \) and \( C_2 \). We call two classes \( C_1 \) and \( C_2 \) **incompatible**, if there exists a disjoint or a complement relation between them. This is detected by the following PROLOG predicate:

\[
\text{incompatible}(C_1, C_2) :- \\
( \text{complement}(C_1, C_2) \\
; \text{disjoint}(C_1, C_2) ).
\]
Taxonomic Relationships and Rules  An obvious equivalence exists between the relationships $B$ is-a $A$ – where $A$ and $B$ are both class atoms or both property atoms with the same arguments – and rules of the form $B \Rightarrow A$ with a single atom $B$ in the body having the same argument as $A$. Thus, we combine them into the single formalism derives in DATALOG*:

\[
\text{derives}(C_1, C_2) :-
\begin{align*}
\text{isa}(C_1, C_2) & ; \text{rule}(A-[B]) \ B ==.. \ [C_1, X_1], \ A ==.. \ [C_2, X_2], \\
\text{var}(X_1), \ X_1 == X_2 
\end{align*}
\]

\[
\text{isa}(C_1, C_2) :- \\
\text{sub_class}(C_1, C_2).
\]

\[
\text{isa}(C_1, C_3) :- \\
\text{isa}(C_1, C_2), \text{sub_class}(C_2, C_3).
\]

Observe, that the call \text{var}(X_1), \ X_1 == X_2 tests if $X_1$ and $X_2$ are bound to the same variable.

With the existence of equivalence definitions $E_1 \equiv E_2$ in an ontology language, e.g., the OWL definitions \text{equivalent_class} and \text{equivalent_property}, we can further extend the definition of \text{derives}: an element $E_1$ is derived by an element $E_2$, if the elements are equivalent classes or properties. Since such an equivalence is symmetrical, the predicate \text{derives}/2 always creates cyclic derivations of equivalent elements with length 1.

\[
\text{derives}(E_1, E_2) :-
\begin{align*}
\text{equivalent_class}(E_1, E_2) & ; \text{equivalent_property}(E_1, E_2) 
\end{align*}
\]

We compute the transitive closure \text{tc_derives} of \text{derives} using the following simple, standard DATALOG* scheme:

\[
\text{tc_derives}(E_1, E_2) :- \\
\text{derives}(E_1, E_2).
\]

\[
\text{tc_derives}(E_1, E_3) :- \\
\text{derives}(E_1, E_2), \text{tc_derives}(E_2, E_3).
\]

Subsequently, the reflexive transitive closure \text{tcr_derives} of \text{derives} is computed using the following PROLOG predicate:

\[
\text{tcr_derives}(E_1, E_2) :- \\
\begin{align*}
E_1 == E_2 & ; \text{tc_derives}(E_1, E_2) 
\end{align*}
\]

Remark on Examples. In the following we give examples for most of the described anomalies. For this task, we use a printer domain, because to its popularity and intuitive understanding.
2.2 Mixing DATALOG and PROLOG: Forward and Backward Chaining

The detection of anomalies in SWRL ontologies could not be formulated using PROLOG backward chaining or DATALOG forward chaining alone, since we may need recursion on cyclic data, function symbols (mainly for representing lists), non–ground facts, negation and disjunction in rule bodies, aggregation, and stratification.

Thus we have developed a new approach that extends the DATALOG paradigm to DATALOG* and mixes in with PROLOG. However, an intuitive understanding of the presented, mixed rule sets is possible without understanding the new inference method. The interested reader can run the analysis using our DisLog system [2].

DATALOG*. We distinguish between DATALOG* rules and PROLOG rules. DATALOG* rules are forward chaining rules (not necessarily range–restricted) that may contain function symbols (in rule heads and bodies) as well as negation, disjunction, and PROLOG predicates in rule bodies. DATALOG* rules are evaluated bottom–up, and all possible conclusions are derived.

The supporting PROLOG rules are evaluated top–down, and for efficiency reasons only on demand, and they can in turn refer to DATALOG* facts. The PROLOG rules are also necessary for expressivity reasons: they are used for some computations on complex terms, and more importantly for computing very general aggregations of DATALOG* facts.

Ontology Evaluation in DATALOG*. For ontology evaluation, we have implemented two layers $D_1$ and $D_2$ of DATALOG* rules:

- The upper layer $D_2$ consists of the rules for the predicate anomaly/2 and some DATALOG* rules that are stated together with them.
- The lower layer $D_1$ consists of all other DATALOG* rules. E.g., the rules for predicates derives and tc_derives are in $D_1$.

$D_1$ is applied to the DATALOG* facts for the following basic predicates, which have to be derived from the underlying SWRL document:

rule, class, sub_class, complement, incompatible, equivalent_class, equivalent_property, transitive_property, symmetric_property, property_restriction, min_cardinality_restriction, max_cardinality_restriction, class_has_property.

The resulting DATALOG* facts are the input for $D_2$. The stratification into two layers is necessary, because $D_2$ refers to $D_1$ through negation and aggregation. Most PROLOG predicates in this paper support the layer $D_2$.

E.g., the following predicates with calls to DATALOG* facts generalize tc_derives and incompatible to atoms:

```prolog
tc_derives_atom(A1, A2) :-
    tc_derives(P1, P2), A1 =.. [P1|Xs], A2 =.. [P2|Xs].

incompatible_atoms(A1, A2) :-
    incompatible(P1, P2), A1 =.. [P1|Xs], A2 =.. [P2|Xs].
```
We cannot evaluate these rules using forward chaining, since $Xs$ is a unknown list.

The head and body predicates of a rule can be determined using the following pure PROLOG predicates:

$$
\text{head\_predicate}(A-\_, P) :- \\
\quad \text{functor}(A, P, \_).
$$

$$
\text{body\_predicate}(\_\_Bs, P) :- \\
\quad \text{member}(B, Bs), \text{functor}(B, P, \_).
$$

The following PROLOG rules define siblings and aggregate the siblings $Z$ of a class $X$ to a list $Xs$ using the well–known meta–predicate $\text{findall}$, respectively:

$$
\text{sibling}(X, Y) :- \\
\quad \text{sub\_class}(X, Z), \text{sub\_class}(Y, Z), X \neq Y.
$$

$$
\text{siblings}(Xs) :- \\
\quad \text{sibling}(X, \_), \\
\quad \text{findall}( Z, \\
\quad \quad \text{sibling}(X, Z), \\
\quad \quad Xs ).
$$

These rules could also be evaluated in DATALOG* using forward chaining. But, since we need siblings only for certain lists $Xs$, this would be far too inefficient.

**Evaluation of DATALOG**. DATALOG* rules cannot be evaluated in PROLOG or DATALOG alone for the following reasons: Current DATALOG engines cannot handle function symbols and non–ground facts, and they do not allow for the embedded computations, which we need. Standard PROLOG systems cannot easily handle recursion with cycles, because of non–termination, and are inefficient, because of subqueries that are posed and answered multiply. Thus, they have to be extended by some DATALOG* facilities (our approach) or memoing/tabling facilities (the approach of the PROLOG extension XSB). Since we wanted to use SWI PROLOG – because of its publicly available graphical API – we have implemented a new inference machine that can handle mixed, stratified DATALOG*/PROLOG rule systems.

### 3 Case Study

Knowledge representation in the Semantic Web is based on ontologies and logic. The reasoning tasks require search (query answering) and knowledge engineering / modeling (analysis of the structure of the ontologies for anomalies). Knowledge engineering and reasoning in the Semantic Web is based on ontology editors and specialized databases. It can further be supported by deductive databases and logic programming techniques.

In the Semantic Web, it is possible to reason about the ontology / taxonomy (i.e., the schema) and the instances. This is called terminological or assertional (T–Box or A–Box) reasoning, respectively. This makes search in the Semantic Web more effective.
In the following printer ontology, we could search for a printer from HP, and the result could be a laser–jet printer from HP, since the system knows that hpLaserJetPrinter is a sub–class of hpPrinter.

It can also be derived, that all laser–jet printers from HP are no laser writers from Apple; in this case, this is very easy, since it is explicitly stored in the ontology.

Moreover, we will show in the following how to support knowledge engineering by detecting anomalies in OWL ontologies. In the Web Ontology Language (OWL), we can mix concepts from rdf (Resource Description Framework) for defining instances and
rdfs (rdf Schema) for defining the schema of an application. Moreover, tags with the namespace owl are allowed. The Semantic Web Rule Language (SWRL) incorporates logic programming rules into OWL ontologies. There exist well-known, powerful tools for asking queries on and for reasoning with OWL ontologies.

3.1 The Printer Ontology in OWL

The following examples are given in Turtle syntax [15] using the namespace p for resources of the printer ontology. First of all, every laserJetPrinter is a printer, and every hpPrinter is an hpProduct:

```
p:printer       rdf:type owl:Class .
p:hpProduct     rdf:type owl:Class .
hpPrinter       rdfs:subClassOf p:hpProduct .
```

The following owl:Class element defines the class appleLaserWriter:

```
p:appleLaserWriter rdf:type owl:Class ;
   rdfs:comment "Apple laser writers are laser jet printers" ;
   rdfs:subClassOf p:laserJetPrinter ;
```

The rdfs:subClassOf sub-element states that appleLaserWriter is a sub-class of laserJetPrinter. The owl:disjointWith sub-element states that appleLaserWriter is disjoint from hpLaserJetPrinter.

The following owl:Class element defines a class of printers from a joint venture of HP and Apple:

```
p:hpApplePrinter
   rdfs:comment "Printers from a joint venture of HP and Apple" ;
```

The existence of such printers would contradict the disjointWith restriction between the classes hpLaserJetPrinter and appleLaserWriter. The emptiness of the class hpApplePrinter can be detected by reasoners used, for instance, by ontology editors like Protégé.

**Redundant subClassOf Relation.** Since hpLaserJetPrinter is a sub-class of the class hpPrinter, and hpPrinter is a sub-class of hpProduct, it is redundant to explicitly state that hpLaserJetPrinter is a sub-class of hpProduct.

```
p:hpLaserJetPrinter
```

This redundancy is not an error. We could simply consider it as an anomaly, that should be reported to the knowledge engineer. This anomaly is usually not reported by reasoners in standard ontology editors.
Instances. Finally, we have some instances of the defined classes:

\[
\begin{align*}
p:1001 & \text{ rdf:type } p:\text{appleLaserWriter} . \\
p:1002 & \text{ rdf:type } p:\text{appleLaserWriter} . \\
p:1003 & \text{ rdf:type } p:\text{hpLaserJetPrinter} . \\
p:1004 & \text{ rdf:type } p:\text{hpLaserJetPrinter} .
\end{align*}
\]

As mentioned before, there cannot exist instances of the class \text{hpApplePrinter}.

The ontology editor Protégé offers plugged--in reasoners, such as FaCT++, HermiT, and Racer. The ontology reasoner FaCT++ can infer that the class \text{hpApplePrinter} is EquivalentTo the empty class Nothing. By clicking the question mark, an explanation can be shown. There are also databases for handling \text{rdf} data, so called triple stores, such as Sesame or Jena. They use extensions of SQL—most notably SPARQL—as a query language.

Please note, that for the presented Turtle syntax the corresponding XML syntax can be generated. For instance, the definition of the joint HP and Apple printer would read as follows:

\[
<\text{rdf:Description } \text{rdf:about}="printer#hpLaserJetPrinter">
<\text{rdfs:subClassOf } \text{rdf:resource}="printer#hpPrinter"/>
<\text{rdfs:subClassOf } \text{rdf:resource}="printer#laserJetPrinter"/>
</\text{rdf:Description}>
\]

\text{Protégé.} Figure 1 shows the printer ontology in the standard ontology editor Protégé.

3.2 Declarative Queries in \text{FnQuery}

In PROLOG, an XML element can be represented as a term structure \text{T:As:C}, called \text{FN-triple}. \text{T} is the tag of the element, \text{As} is the list of the attribute/value pairs \text{A:V} of the element, and \text{C} is a list of \text{FN-triples} for the sub--elements.

\[
\begin{align*}
\text{\'owl:Class\':[\'rdf:ID\':\'appleLaserWriter\']:[} \\
\text{\'rdfs:comment\':[\'Apple laser ...\']}, \\
\text{\'rdfs:subClassOf\':[} \\
\text{\'rdf:resource\':\'#laserJetPrinter\'][:],} \\
\text{\'owl:disjointWith\':[} \\
\text{\'rdf:resource\':\'#hpLaserJetPrinter\'][:]} \]
\end{align*}
\]

In an \text{OWL} knowledge base \text{Owl}, there exists an \text{isa} relation between two classes \text{C1} and \text{C2}, if a \text{subclassOf} relation is stated explicitely, or if \text{C1} was defined as the intersection of \text{C2} and some other classes:

\[
\% \text{isa}(\text{+Owl, ?C1, ?C2) <-}
\]

\[
\text{isa(Owl, C1, C2) :-} \\
\text{C := Owl/\'owl:Class\':[@\'rdf:ID\'=C1],} \\
( \text{R2 := C/\'rdfs:subClassOf\':@\'rdf:resource\'} \\
\text{; R2 := C/\'owl:intersectionOf\'/\'owl:Class\':@\'rdf:about\'} ), \\
\text{owl_reference_to_id(R2, C2).}
\]

55
Fig. 1. The Printer Ontology in Protégé

% owl_reference_to_id(+Reference, ?Id) <-

owl_reference_to_id(Reference, Id) :-
   ( concat('#', Id, Reference)
   ; Id = Reference).

Disjointness of Classes.

% disjointWith(+Owl, ?C1, ?C2) <-

disjointWith(Owl, C1, C2) :-
   R2 := Ow1/’owl:Class’::[@’rdf:about’=R1]
      /’owl:disjointWith’@’rdf:resource’,
   owl_reference_to_id(R1, C1),
   owl_reference_to_id(R2, C2).

In the following, we often suppress the ontology argument Owl.
Transitive Closure of isa.

```prolog
% subClassOf(?C1, ?C2) <-
subClassOf(C1, C2) :-
    isa(C1, C2).
subClassOf(C1, C2) :-
    isa(C1, C), subClassOf(C, C2).
```

3.3 Anomalies in Ontologies

**Cycle.**

?- isa(C1, C2), subClassOf(C2, C1).

C1 = personalPrinter,
C2 = printer

**Partition Error.** The class C is a sub–class of two disjoint classes C1 and C2.

?- disjointWith(C1, C2),
   subClassOf(C, C1), subClassOf(C, C2).

C = hpApplePrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter

**Incompleteness.** The class C has three sub–classes C1, C2 and C3, from which only the two sub–classes C1 and C2 are declared as disjoint in the knowledge base.

?- isa(C1, C), isa(C2, C), isa(C3, C),
   disjointWith(C1, C2), not(disjointWith(C2, C3)).

C = laserJetPrinter,
C1 = hpLaserJetPrinter,
C2 = appleLaserWriter,
C3 = ibmLaserPrinter

The fact that C2 and C3 are disjoint and that C1 and C3 are disjoint as well, possibly was forgotten by the knowledge engineer during the creation of the knowledge base.

**Redundant subClassOf/instanceOf Relations.** The sub–class relation between C1 and C3 can be derived by transitivity over the class C2.

```prolog
% redundant_isa(?Chain) <-
redundant_isa(C1->C2->C3) :-
    isa(C1, C2), subClassOf(C2, C3),
    isa(C1, C3).
?- redundant_isa(Chain).
```

Chain = hpLaserJetPrinter -> hpPrinter -> hpProduct

Here, isa(C1, C2), subClassOf(C2, C3), requires that this deduction is done over at least two levels.
Undefined Reference. During the development of an ontology in OWL, it is possible that we reference a class that we have not yet defined.

% undefined_reference(+Owl, ?Ref) <-

undefined_reference(Owl, Ref) :-
    rdf_reference(Owl, Ref),
    not(owl_class(Owl, Ref)).

rdf_reference(Owl, Ref) :-
    ( R := Owl/descendant_or_self::*@'rdf:resource'
    ; R := Owl/descendant_or_self::*@'rdf:about' ),
    owl_reference_to_id(R, Ref).

owl_class(Owl, Ref) :-
    Ref := Owl/'owl:Class'@'rdf:ID'.

If we load such an ontology into Protégé, then the ontology reasoners may produce wrong results, even for unrelated parts of the ontology.

4 Discussion

In the last years ontologies have played a major role for building intelligent systems. Currently, standard ontology languages like OWL are extended by rule–based elements, e.g., RULEML and the semantic web rule language SWRL.

We have shown that with the increased expressiveness of ontologies – now also including rules – a number of new evaluation issues have to be considered. In this paper, we have presented a framework for verifying ontologies with rules comprising a collection of anomalies, that verify the represented knowledge in a combined methodology. For all anomalies, we have described a DATALOG⋆ implementation which is used in a prototype for ontology verification. Due to its declarative nature, new methods for anomaly detection can be easily added to the existing work. From our point of view, the declarative approach is crucial because of the incompleteness of the presented anomalies: in principle, an entire overview of possible anomalies is not possible, since the number of anomalies depends on the used expressiveness of the ontology and the rule representation, respectively.

The actual frequency of the introduced anomalies is an interesting issue. However, only a small number of ontologies (mostly toy examples) is available that actually use a rule extension. A sound review of anomaly occurrences would require a reasonable number of ontologies having a significant size.

For many real–world applications, we expect a more expressive rule language to be used than SWRL. With SWRL FOL, an extension of SWRL to first–order logic is currently discussed as a proposal. Furthermore, larger systems may also include parts of a non–monotonic rule base. Here, some work has been done on the verification of non–monotonic rule bases [16], that has to be re–considered in the presence of an ontological layer.
References

Automated Exercises for Constraint Programming

Johannes Waldmann
HTWK Leipzig, Fakultät IMN, 04277 Leipzig, Germany

Abstract. We describe the design, implementation, and empirical evaluation of some automated exercises that we are using in a lecture on Constraint Programming. Topics are propositional satisfiability, resolution, the DPLL algorithm, with extension to DPLL(T), and FD solving with arc consistency. The automation consists of a program for grading student answers, and in most cases also a program for generating random problem instances. The exercises are part of the autotool E-assessment framework. The implementation language is Haskell. You can try them at https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=199.

1 Introduction

I lecture on Constraint Programming [Wal14a], as an optional subject for master students of computer science. The lecture is based on the books Principles of Constraint Programming by Apt[Apt03] and Decision Procedures by Kroening and Strichman [KS08]. Topics include propositional satisfiability (SAT) and resolution, the DPLL algorithm for deciding SAT, with extension to DPLL(T) for solving satisfiability modulo theory (SMT), and FD solving with arc consistency.

I do want to assign homework problems, but I do not have a teaching assistant for grading. This is one motivation for writing software that automates the grading of solutions, and also the generation of random (but reasonable) problem instances. Another motivation is that the software is much more available, reliable and patient than a human would be, so I can pose homework problems that would require super-human teaching assistants.

This software is part of the autotool E-assessment framework [GLSW11,RRW08]. It provides the following functionality (via a web interface):

– the tutor can choose a problem type, then configure parameters of an instance generator,
– the student can view “his” problem instance (that had been generated on first access) and enter a solution candidate,
– the grading program immediately evaluates this submission and gives feedback, sometimes quite verbose,
– based on that, the student can enter modified solutions, any number of times. The exercise counts as “solved” if the student had at least one correct solution during a given time interval (say, two weeks).
A distinctive feature is that autotoool exercises are graded “semantically” — as opposed to “schematically”, by syntactic comparison with some prescribed master solution. E.g., for propositional satisfiability, the student enters an assignment, and the program checks that it satisfies all clauses of the formula, and prints the clauses that are not satisfied (see more detail in Section 2).

In the language of complexity theory, the student has to find a witness for the membership of the problem instance in a certain problem class, and the software just verifies the witness. In many cases, the software does not contain an actual solver for the class, so even looking at the source code does not provide shortcuts in solving the exercises. But see Section 5 for an example where a solver is built in for the purpose of generating random but reasonable instances.

Section 6 shows an example for an “inverse” problem, where the witness (a structure that is a model) is given, and the question (a formula of a certain shape) has to be found.

If the software just checks a witnessing property, then it might appear that it cannot check the way in which a student obtained the witness. This seems to contradict the main point of lecturing: it is about methods to solve problems, so the teacher wants to check that the methods are applied properly. In some cases, a problem instance appears just too hard for brute force attempts, so the only way of solving it (within the deadline) is by applying methods that have been taught.

Another approach for designing problems is presented in Sections 5,7,8. There, the solution is a sequence of steps of some algorithm, e.g., Decide, Propagate and Backtrack in a tree search, and the witnessing property is that each step is valid, and that the computation arrives in a final state, e.g., a solution in a leaf node, or contradiction in the root. Here, the algorithm is non-deterministic, so the student must make choices.

Another feature of autotoool is that most output and all input is textual. There is no graphical interface to construct a solution, rather the student has to provide a textual representation of the witness. This is by design, the student should learn that every object can be represented as a term. Actually, we use Haskell syntax for constructing objects of algebraic data types throughout.

For each problem type, the instructor can pose a fixed problem instance (the same for all students). For most problem types, there is also a generator for random, but reasonable instances. Quite often, the generator part of the software is more complicated than the grading part. Then, each student gets an individual problem instance, and this minimizes unwanted copying of solutions.

For fixed problem instances, autotoool can compute a “highscore” list. Here, correct solutions are ranked by some (problem-specific) measure, e.g., for resolution proofs, the number of proof steps. Some students like to compete for positions in that list and try to out-smart each other, sometimes even writing specialized software for solving the problems, and optimizing solutions. I welcome this because they certainly learn about the problem domain that way.

In the following sections, I will present exercise problems. For each problem type I’ll give
the motivation (where does the problem fit in the lecture),
– the instance type, with example,
– the solution domain type,
– the correctness property of solutions,
– examples of system answers for incorrect solution attempts,
– the parameters for the instance generator (where applicable).

The reader will note that the following sections show inconsistent concrete syntax, e.g., there are different representations for literals, clauses, and formulas. Also, some system messages are in German, some in English. These inconsistencies are the result of incremental development of the autotool framework over >10 years. The exercises mentioned in this paper can also be tried online (without any registration) at https://autotool.imn.htwk-leipzig.de/cgi-bin/Trial.cgi?lecture=199

2 Propositional Satisfiability

– instance: a propositional logic formula $F$ in conjunctive normal form,
– solution: a satisfying assignment for $F$

Motivation: At the very beginning of the course, the student should try this by any method, in order to recapitulate propositional logic, and to appreciate the NP-hardness of the SAT problem, and (later) the cleverness of the DPLL algorithm. We use the problem here to illustrate the basic approach.

Problem instance example:

```
( p || s || t) && ( q || s || t) && ( r || ! q || ! s)
&& ( p || t || ! r) && ( q || s || t) && ( r || ! s || ! t)
&& ( p || ! q || ! s) && ( q || t || ! p) && ( s || t || ! q)
&& ( p || ! q || ! t) && ( q || ! p || ! s) && ( s || ! r || ! t)
&& ( p || ! r || ! t) && ( r || s || ! q) && (! p || ! q || ! r)
&& ( q || r || ! p) && ( r || ! p || ! t) && (! p || ! r || ! s)
```

Problem solution domain: partial assignments, example

```
listToFM [( p, False ), (q, True), (s,True)]
```

Correctness property: the assignment satisfies each clause.

Typical system answers for incorrect submissions: the assignment is partial, but already falsifies a clause

```
gelesen: listToFM
[( p, False ), ( q, True ), ( s, True )]
Diese vollständig belegten Klauseln sind nicht erfüllt:
[( p || ! q || ! s )]
```

62
No clause is falsified, but not all clauses are satisfied:

gelesen: listToFM
    [( p , False ) , ( s , False ) , ( t , True ) ]

Diese Klauseln noch nicht erfüllt:
    [( p || ! q || ! t) , ( p || ! r || ! t) , ( r || s || ! q)
    , ( s || ! r || ! t) ]

*Instance generator:* will produce a satisfiable 3-SAT instance according to algorithm hgen2 [SDH02]. Parameters are the set of variables, and the number of clauses, for example,

```
Param { vars = mkSet [ p , q , r , s , t ] , clauses = 18 }
```

### 3 SAT-equivalent CNF

- instance: formula $F$ in conjunctive normal form with variables $x_1, \ldots, x_n$,
- solution: formula $G$ in conjunctive normal form with variables
  in $x_1, \ldots, x_n, y_1, \ldots, y_k$ such that $\forall x_1 \cdots \forall x_n : (F \leftrightarrow \exists y_1 \cdots \exists y_k : G)$
- measure: number of clauses of $G$.

*Motivation:* for arbitrary $F$, this is solved by the Tseitin transform [Tse70]. The student learns the underlying notion of equivalence, and that auxiliary variables may be useful to reduce formula size. The problem is also of practical relevance: for bit-blasting SMT solvers, it is important to encode basic operations by small formulas.

*Problem instance example:*

```
(x1 + x2 + x3 + x4 + x5) * (x1 + x2 + x3 + -x4 + -x5) *
(x1 + x2 + -x3 + x4 + -x5) * (x1 + x2 + -x3 + -x4 + x5) *
(x1 + -x2 + x3 + x4 + -x5) * (x1 + -x2 + x3 + -x4 + x5) *
(x1 + -x2 + -x3 + x4 + x5) * (x1 + -x2 + -x3 + -x4 + -x5) *
(-x1 + x2 + x3 + x4 + -x5) * (-x1 + x2 + x3 + -x4 + x5) *
(-x1 + x2 + -x3 + x4 + x5) * (-x1 + x2 + -x3 + -x4 + -x5) *
(-x1 + -x2 + x3 + x4 + x5) * (-x1 + -x2 + x3 + -x4 + -x5) *
(-x1 + -x2 + -x3 + x4 + -x5) * (-x1 + -x2 + -x3 + -x4 + x5)
```

*Correctness property:* existential closure of $G$ is equivalent to $F$, and $|G| < |F|$.

*Typical system answers for incorrect submissions:*

gelesen: (x1 + x2 + x3 + x6) * (-x6 + x1)

nicht äquivalent, z. B. bei Belegung(en)
    listToFM [ ( x1 , True ) , ( x2 , True ) , ( x3 , False ) ]
    , ( x4 , False ) , ( x5 , False ) ]
The equivalence check uses a BDD implementation [Wal14b].

Hint for solving the example: invent an extra variable that represents the XOR of some of the \( x_i \).

This problem type is a non-trivial highscore exercise. The given example instance (size 16) has a solution of size 12.

4 Propositional Logic Resolution

– instance: an unsatisfiable formula \( F \) in conjunctive normal form,
– solution: a derivation of the empty clause from \( F \) by resolution,
– measure: number of derivation steps.

Motivation: the student should see “both sides of the coin”: resolution proves unsatisfiability. Also, the student sees that finding resolution proofs is hard, and they are not always short (because if they were, then SAT \( \in \text{NP} \cap \text{coNP} \), which nobody believes).

Resolution of course has many applications. In the lecture I emphasize certification of UNSAT proof traces in SAT competitions.

Problem instance example: clauses are numbered for later reference

\[
\begin{align*}
0 & : \neg a \lor b \lor c \\
1 & : a \lor b \lor c \\
2 & : a \lor \neg c \lor \neg d \\
3 & : \neg a \lor \neg c \lor \neg d \\
4 & : a \lor c \\
5 & : \neg c \lor \neg d \\
6 & : a \lor \neg b \lor d \\
7 & : a \lor \neg b \lor \neg d \\
8 & : \neg a \lor d \\
9 & : \neg c \lor \neg d \\
10 & : \neg b \lor c \lor \neg d \\
11 & : a \lor b \lor \neg d
\end{align*}
\]

Problem solution domain: sequence of resolution steps, example:

\[
[ \text{Resolve} \{ \text{left} = 4, \text{right} = 8, \text{literal} = a \} \\
, \text{Resolve} \{ \text{left} = 12, \text{right} = 5, \text{literal} = c \} 
]
\]

Correctness property: for each step \( s \) it holds that literal \( s \) occurs in clause \( \text{left} \ s \), and \( \neg \) (literal \( s \)) occurs in clause \( \text{right} \ s \). If so, then the system computes the resolvent clause and assigns the next number to it. The clause derived in the last step is the empty clause.

Typical system answer for an incomplete submission:

nächster Befehl Resolve \{ \text{left} = 4, \text{right} = 8, \text{literal} = a \} \\
entferne Literal a aus Klausel a \lor c \quad \text{ergibt} \ c \\
entferne Literal ! a aus Klausel ! a \lor d \quad \text{ergibt} \ d \\
neue Klausel \quad 12 : c \lor d \\
\]

nächster Befehl Resolve \{ \text{left} = 12, \text{right} = 5, \text{literal} = c \} \\
entferne Literal c aus Klausel c \lor d \quad \text{ergibt} \ d \\
entferne Literal ! c aus Klausel ! c \lor d \quad \text{ergibt} \ d \\
neue Klausel \quad 13 : d \\
letzte abgeleitete Klausel ist Zielklausel? Nein.
Instance generator: is controlled by, for example,

\[
\text{Config } \{ \text{num\_variables } = 5, \text{ literals\_per\_clause\_bounds } = (2, 3) \}\]

The implementation uses a BDD implementation [Wal14b] to make sure that the generated formula is unsatisfiable. The generator does not actually produce a proof trace, because it must exist, by refutation completeness.

5 Backtracking and Backjumping: DPLL (with CDCL)

- instance: a CNF \( F \),
- solution: a complete DPLL proof trace determining the satisfiability of \( F \).

Motivation: The DPLL algorithm [DP60,DLL62] is the workhorse of modern SAT solvers, and the basis for extensions in SMT.

This exercise type should help the student to understand the specification of the algorithm, and also the design space that an implementation still has, e.g., picking the next decision variable, or the clause to learn from a conflict, and the backjump target level.

This is an instance of the “non-determinism” design principle for exercises: force the student to make choices, instead of just following a given sequence of steps. It fits nicely with abstract descriptions of algorithms that postpone implementation choices, and instead give a basic invariant first, and prove its correctness.

Problem instance example:

\[
[ [ 3, -4 ], [ 4, 5 ], [ 3, -4, 5 ], [ 1, -2 ] ]
, [ 3, 4, 5 ], [ 1, 2, 4, 5 ], [ -1, 4, -5 ]
, [ -1, -2 ], [ 2, 3, -4 ], [ -3, -4, -5 ], [ 2, 3, -5 ]
, [ -2, -3, 4 ], [ -1, -4 ], [ -3, 4 ], [ 1, -3, -4, 5 ] ]
\]

Problem solution domain: sequence of steps, where

\[
\text{data Step} = \text{Decide Literal}
| \text{Propagate } \{ \text{use } :: \text{Clause, obtain } :: \text{Literal} \}
| \text{SAT}
| \text{Conflict Clause}
| \text{Backtrack}
| \text{Backjump } \{ \text{to\_level } :: \text{Int, learn } :: \text{Clause} \}
| \text{UNSAT}
\]

Correctness property: the sequence of steps determines a sequence of states (of the tree search), where
A state represents a node in the search tree. For each state (computed by the system), the next step (chosen by the student) must be applicable, and the last step must be SAT or UNSAT.

The reason for a variable shows whether its current value was chosen by a Decision (then we need to take the Alternate_Decision on backtracking) or by Propagation (then we need to remember the antecedent clause, for checking that a learned clause is allowed).

A SAT step asserts that the current assignment satisfies the formula. A Conflict c step asserts that Clause c is falsified by the current assignment. The next step must be Backtrack (if the decision level is below the root) or UNSAT (if the decision level is at the root, showing that the tree was visited completely).

A Step Propagate {use=c, obtain=1} is allowed if clause c is a unit clause under the current assignment, with 1 as the only un-assigned literal, which is then asserted. A Decide step just asserts the literal.

Then following problem appears: if the student can guess a satisfying assignment σ of the input formula, then she can just Decide the variables, in sequence, according to σ, and finally claim SAT. This defeats the purpose of the exercise.

The following obstacle prevents this: each Decide must be negative (assert that the literal is False). This forces a certain traversal order. It would then still be possible to “blindly” walk the tree, using only Decide, Conflict (in the leaves), and Backtrack. This would still miss a main point of DPLL: taking shortcuts by propagation. In the exercise, this is enforced by rejecting all solutions that are longer than a given bound.

Instance generator: will produce a random CNF F. By completeness of DPLL, a solution for F (that is, a DPLL-derivation) does exist, so the generator would be done here. This would create problem instances of vastly different complexities (with solutions of vastly different lengths), and this would be unjust to students. Therefore, the generator enumerates a subset S of all solutions of F, and then checks that the minimal length of solutions in S is near to a given target.

The solver is written in a “PROLOG in Haskell” style, using Control.Monad.Logic [KcSFS05] with the fair disjunction operator to model choices, and allow a breadth-first enumeration (where we get shorter solutions earlier).

There is some danger that clever students extract this DPLL implementation (autotool is open-sourced) to solve their problem instance. I think this approach
requires an amount of work that is comparable to solving the instance manually, so I tolerate it.

**DPLL with CDCL (conflict driven clause learning):** in this version of the exercise, there is no **Backtrack**, only **Backjump** `{ to_level = 1, learn = c }`. This step is valid right after a conflict was detected, and if clause $c$ is a consequence of the current antecedents that can be checked by **reverse unit propagation**: from $\text{not } c$ and the antecedents, it must be possible to derive the empty clause by unit propagation alone. This is a “non-deterministic version” of Algorithm 2.2.2 of [KS08].

I introduced another point of non-determinism in clause learning: the student can choose any decision level to backjump to. Textbooks prove that one should go to the second most recent decision level in the conflict clause but that is a matter of efficiency, not correctness, so we leave that choice to the student.

If the **Backjump** does not go high enough, then learning the clause was not useful (it is just a **Backtrack**). If the **Backjump** does go too high (in the extreme, to the root), then this will lead to duplication of work (re-visiting parts of the tree). Note that the target node of the backjump is re-visited: we return to a state with a partial assignment that was seen before. But this state contains the learned clause, so the student should use it in the very next step for unit propagation, and only that avoids to re-visit subtrees.

A challenge problem: the following pigeonhole formula is unsatisfiable for $n > m$, but this is hard to see for the DPLL algorithm: “there are $n$ pigeons and $m$ holes, each pigeon sits in a hole, and each hole has at most one pigeon” [Cla11]. I posed this problem for $n = 5, m = 4$. The resulting CNF on 20 variables ($v_{p,h}$: pigeon $p$ sits in hole $h$) has 5 clauses with 4 literals, and 40 clauses with 2 literals. My students obtained a DPLL solution with 327 steps, and DPLL-with-CDCL solution with 266 steps. (Using software, I presume.)

6 Evaluation in Finite Algebras

- instance: a signature $\Sigma$, two $\Sigma$-algebras $A, B$, both with finite universe $U$,
- solution: a term $t$ over $\Sigma$ with $t_A \neq t_B$.

Motivation: the introduction, or recapitulation, of predicate logic basics. The exercise emphasizes the difference and interplay between syntax (the signature, the term) and semantics (the algebras).

This exercise type shows the design principle of inversion: since we usually define syntax first (terms, formulas), and semantics later (algebras, relational structures), it looks natural to ask “find an algebra with given property”. Indeed I have such an exercise type ("find a model for a formula"), but here I want the other direction.
Problem instance example:

Finden Sie einen Term zur Signatur

\[
\text{Signatur} = \{ \text{funktionen} = \text{listToFM} \[(p, 2), (z, 0)\], \text{relationen} = \text{listToFM} [], \text{freie_variablen} = \text{mkSet} []\}
\]

, der in der Struktur

\[
\text{A = Struktur} = \{ \text{universum} = \text{mkSet} [1, 2, 3], \text{predicates} = \text{listToFM} [], \text{functions} = \text{listToFM} [p, \{ (1, 1, 3), (1, 2, 3), (1, 3, 3), (2, 1, 2), (2, 2, 1), (2, 3, 1), (3, 1, 3), (3, 2, 1), (3, 3, 2) \}], (z, \{(3)\}) \}
\]

eine anderen Wert hat als in der Struktur

\[
\text{B = Struktur} = \{ \text{universum} = \text{mkSet} [1, 2, 3], \text{predicates} = \text{listToFM} [], \text{functions} = \text{listToFM} [p, \{ (1, 1, 1), (1, 2, 3), (1, 3, 3), (2, 1, 2), (2, 2, 1), (2, 3, 1), (3, 1, 3), (3, 2, 1), (3, 3, 2) \}], (z, \{(3)\}) \}
\]

here, \(k\)-ary functions are given as sets of \((k+1)\)-tuples, e.g., \((2, 2, 1) \in p\) means that \(p(2, 2) = 1\).

Problem solution domain: terms \(t\) over the signature, e.g.,

\[
p (p (p (z (), z ()) , z ()) , z ()) , z () , z ()
\]

Correctness property: value of term \(t\) in \(A\) is different from value of \(t\) in \(B\).

Example solution: the student first notes that the only difference is at \(p_A(1, 1) = 3 \neq 1 = p_B(1, 1)\), so the solution can be \(p(s, s)\) where \(s_A = 1 = s_B\). Since \(z_A() = 3, p_A(3, 3) = 2, p_A(3, 2) = 1\), a solution is

\[
p (p (p (z()), z()), z()) , p (p (z()), z()), z())
\]
Instance generator: is configured by the signature, and the size of the universe. It will build a random structure $A$, and apply a random mutation, to obtain $B$. It also checks that the point of mutation is reachable by ground terms, and none of them are too small.

7 Satisfiability modulo Theories: DPLL(T)

- instance: a conjunction $F$ of clauses, where a clause is a disjunction of literals, and a literal is a Boolean literal or a theory literal
- solution: a DPLL(T) proof trace determining the satisfiability of $F$.

Motivation: Satisfiability modulo Theories (SMT) considers arbitrary Boolean combinations of atoms taken from a theory $T$, e.g., the theory of linear inequalities over the reals. DPLL(T) is a decision procedure for SMT that combines the DPLL algorithm with a “theory solver” that handles satisfiability of conjunctions of theory literals [NOT06].

E.g., the Fourier-Motzkin algorithm (FM) for variable elimination is a theory solver for linear inequalities. It is not efficient, but I like it for teaching: it has a nice relation to propositional resolution, and it is practically relevant as a pre-processing step in SAT solvers [EB05]. Also, some students took the linear optimization course, some did not, so I do not attempt to teach the simplex method.

We treated DPLL in Section 5, and give only the differences here.

Problem instance example:

$$\begin{align*}
\{ \{ p, q \}, \{ ! p, ! 0 <= + x \}, \{ ! q, 0 <= + 2 -1 * x \}, \{ 0 <= -3 + x \} \}
\end{align*}$$

this represents a set of clauses, where $! p$ is a (negative) Boolean literal, and $0 <= -3 + x$ is a (positive) theory literal.

Problem solution domain: sequence of steps, where

```haskell
data Step = Decide Literal
  | Propagate { use :: Conflict, obtain :: Literal }
  | SAT
  | Conflict Conflict
  | Backtrack
  | Backjump { to_level :: Int, learn :: Clause }
  | UNSAT

data Conflict = Boolean Clause | Theory
```

Note that this Step type results from that of Section 5 by replacing Clause with Conflict in two places (arguments to Propagate and Conflict).
Correctness property: The sequence of steps determines a sequence of states (of the tree search). As long as we use only the Boolean Clause :: Conflict constructor, we have a DPLL computation — that may use theory atoms, but only combines them in a Boolean way. The underlying theory solver is only used in the following extra cases:

A Conflict Theory step is valid if the conjunction of the theory literals in the current assignment is unsatisfiable in the theory. E.g., ! 0 <= x and 0 <= -3 + x is not a Boolean conflict, but a theory conflict.

A Propagate {use = Theory, obtain = l} step is valid if l is a theory literal that is implied by the theory literals in the current assignment, in other words, if ! l together with these literals is unsatisfiable in the theory.

Example solution:

[ Propagate {use = Boolean [ 0 <= -3 + x ], obtain = 0 <= -3 + x } , Propagate {use = Theory, obtain = 0 <= + x } , Propagate {use = Boolean [ ! p , ! 0 <= + x ], obtain = ! p } , Propagate {use = Boolean [ p , q ], obtain = q} , Propagate {use = Boolean [ ! q , 0 <= + 2 -1 * x ], obtain = 0 <= + 2 -1 * x } , Conflict Theory , UNSAT ]

E.g., to validate the second step (theory propagation), the T-solver checks that the conjunction of (0 <= -3+x) (from the current assignment) and !(0 <= x) (negated consequence) is unsatisfiable. We arrive at a T-conflict at the root decision level, so the input formula is unsatisfiable.

8 Solving Finite Domain Constraints

- instance: a relational Σ-structure R over finite universe U, and a conjunction F of Σ-atoms
- solution: a complete FD tree search trace determining the satisfiability of F.

Motivation: Finite Domain (FD) constraints can be seen as a mild generalization of propositional SAT. Methods for solution are similar (tree search), but have differences. In particular, I use FD constraints to discuss (arc) consistency notions, as in [Apt03], and this automated exercise type also makes that point.

The design principle is again non-determinism: the student has to make a choice among several possible steps. In particular, propagation and conflict detection are done via arc consistency deduction.

Problem instance example:

Give a complete computation of an FD solver that determines satisfiability of:

[ P ( x , y , z ) , P ( x , x , y ) , G ( y , x ) ]

in the structure:
Algebra

{ universe = [ 0 , 1 , 2 , 3 ]
, relations = listToFM
[ ( G , mkSet
[ [ 1 , 0 ] , [ 2 , 0 ] , [ 2 , 1 ]
, [ 3 , 0 ] , [ 3 , 1 ] , [ 3 , 2 ] ]
)
, ( P , mkSet
[ [ 0 , 0 , 0 ] , [ 0 , 1 , 1 ] , [ 0 , 2 , 2 ]
, [ 0 , 3 , 3 ] , [ 1 , 0 , 1 ] , [ 1 , 1 , 2 ]
, [ 1 , 2 , 3 ] , [ 2 , 0 , 2 ] , [ 2 , 1 , 3 ]
, [ 3 , 0 , 3 ] ]
)]
}

Problem solution domain: sequence of steps, where (u is the universe)
data Step u = Decide Var u
| Arc_Consistency_Deduction
{ atoms :: [ Atom ], variable :: Var, restrict_to :: [ u ]
} | Solved
| Backtrack
| Inconsistent

Correctness property: the sequence of steps determines a sequence of states (of
the tree search) where a state is a Stack containing a list of domain assignments
(for each variable, a list of possible values)
data State u = Stack [ M.Map Var [u] ]

A state is Solved if each instantiation of the current assignment (at the top
of the stack) satisfies the formula. A state is conflicting if the current assign-
ment contains a variable with empty domain. In a conflicting state, we can do
Backtrack (pop the stack) or claim Inconsistent (if the stack has one element
only).

A step Decide v e pops an assignment a off the stack, and pushes two
assignments back: one where the domain of v is the domain of v in a, without
e (that is where we have to continue when backtracking), and the other where
the domain of v is the singleton [e]

A step Arc_Consistency_Deduction { atoms, variable, restrict } is valid
if the following holds:

– atoms is a subset of the formula
– for each assignment from variable to current-domain variable without restrict: it
cannot be extended to an assignment that satisfies atoms.

This constitutes a proof that the domain of v can be restricted to restrict.
We have non-determinism here, as we are not enforcing that the restricted set
is minimal. If the restricted set is empty, we have detected a conflict. Since we
want a minimal design, there is no other Step constructor for stating conflicts.

There are several arc consistency concepts in the literature. Ours has these
properties:
we allow to consider a set of atoms (its conjunction), but we can restrict its size (to one, then we are considering each atom in isolation)
we can restrict the number of variables that occur in the set of atoms. this number is the size of the hyper-edges that are considered for hyperarc-consistency. For 1, we get node consistency; for 2, standard arc consistency.
from this number, we omit those variables that are uniquely assigned in the current state. This allows to handle atoms of any arity: we just have to Decide enough of their arguments (so their domain is unit), and can apply arc consistency deduction on those remaining.

Example solution: starts like this:

```
[ Decide x 0
, Arc_Consistency_Deduction
  { atoms = [ P ( x , x , y ) , G ( y , x ) ]
  , variable = y , restrict_to = [ ]
  }
, Backtrack
, Decide x 1
, Arc_Consistency_Deduction
  { atoms = [ P ( x , x , y ) , G ( y , x ) ]
  , variable = y , restrict_to = [ 2 ]
  }
]
```

After the first step (Decide x 0), the state is

```
Stack [ listToFM [ ( x , [ 0 ] )
 , ( y , [ 0 , 1 , 2 , 3 ] )
 , ( z , [ 0 , 1 , 2 , 3 ] ) ]
 , listToFM [ ( x , [ 1 , 2 , 3 ] )
 , ( y , [ 0 , 1 , 2 , 3 ] )
 , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
```

Typical system answers for incorrect submissions: hyperarc size restriction is violated:

current

```
Stack
  [ listToFM
    [ ( x , [ 0 , 1 , 2 , 3 ] )
    , ( y , [ 0 , 1 , 2 , 3 ] )
    , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
```

step

```
Arc_Consistency_Deduction
  { atoms = [ P ( x , x , y ) , G ( y , x ) ]
  , variable = y , restrict_to = [ 2 ]
  }
```
these atoms contain 2 variables with non-unit domain:

\[ \text{mkSet} \ [ \ x , \ y \ ] \]

but deduction is only allowed for hyper-edges of size up to 1

elements are incorrectly excluded from domain:

current

\[
\text{Stack} \ [ \ \text{listToFM} \ [ \ ( \ x , \ [ \ 0 \ ] ) \ ] , \ ( \ y , \ [ \ 0 , 1 , 2 , 3 \ ] ) \ ] , \ ( \ z , \ [ \ 0 , 1 , 2 , 3 \ ] ) \ ] \]

step

\[
\text{Arc\_Consistency\_Deduction} \ \\
\{ \ \text{atoms} = [ \ P ( \ x , \ x , \ y ) \ ] \ \\
, \ \text{variable} = \ y , \ \text{restrict\_to} = [ \ 1 \ ] \ \}
\]

these elements cannot be excluded from the domain of the variable, because the given assignment is a model for the atoms:

\[
[ \ ( \ 0 , \ \text{listToFM} \ [ \ ( \ x , \ 0 ) , \ ( \ y , \ 0 ) ) \ ] \ ]
\]

*Instance generator:* uses the same idea as for DPLL: generate a random instance, solve it breadth-first, and check for reasonable solution length.

9 Related Work and Conclusion

We have shown automated exercises for constraint programming, and also presented the intentions behind their design. In particular, we described how to test the student's understanding of constraint solving algorithms by making use of non-determinism, similar in spirit to the inference systems (proof rules) in [Apt03]. These exercise types are part of the autotool framework for generating exercise problem instances, and grading solutions semantically.

There are several online courses for constraint programming. Few of them seem to contain online exercises. In all cases, computerized exercises (offline or online) focus on teaching a specific constraint language, as a means of modelling, e.g., Gnu-Prolog [Sol04], ECLIPSe [Sim09], CHR [Kae07].

The exercises from the present paper do not focus much on modelling, and learning a specific language. The aim is to teach the semantics of logical formulas, and fundamental algorithms employed by constraint solvers. One could say that each exercise uses a different problem-specific language. Each exercise is graded automatically, and immediately, while giving feedback that helps the student.

So, the approaches are not competing, but complementary.

*Acknowledgments:* Many thanks to the students of my Constraint Programming course (Sommersemester 2014) for working on these exercises, fighting for high-scores, and reporting bugs in the software and in a draft of this report; and to Alexander Bau, Carsten Fuls, and Sibylle Schwarz for helpful comments on exercise design and implementation.
References


Complex Certainty Factors for Rule Based Systems
— Detecting Inconsistent Argumentations

(Invited Talk)

Taieb Mellouli

Martin-Luther-Universität Halle-Wittenberg, Lehrstuhl für Wirtschaftsinformatik
und Operations Research
Universitätsring 3, 06108 Halle (Saale), Germany
mellouli@wiwi.uni-halle.de