WLP 2014

28th Workshop on (Constraint) Logic Programming

WFLP 2014

23rd International Workshop on Functional and (Constraint) Logic Programming

September 15-17, 2014, Wittenberg

Proceedings

Table of Contents

About WLP 2014

Preface

The first part of 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

About WFLP 2014

Preface

The second part of this volume contains the papers presented at WFLP 2014: 23rd International Workshop on Functional and (Constraint) Logic Programming held on September 15-17, 2014 in Wittenberg, at the Leucorea Conference Center of Halle-Wittenberg University.

The WFLP workshops series is running since 1992 and brings together researchers interested in functional programming, logic programming, as well as their integration.

WFLP 2014 was jointly organized and located with WLP 2014: 28th Workshop on (Constraint) Logic Programming. The WLP workshops series serves as the scientific forum of the annual meeting of the Society of Logic Programming (GLP e.V.) and brings together researchers interested in logic programming, constraint programming, and related areas like databases, artificial intelligence, and operations research.

The technical program of the workshop included an invited tutorial by Michael Hanus on Declarative Multi-paradigm Programming, an invited talk by Sven Thiele on Answer Set Programming for Systems Biology, and presentations of refereed papers.

I thank the invited speakers, authors of papers, programme committee members, external reviewers, as well as the local organizers Stefan Brass, Ramona Vahrenhold and Heike Stephan, for creating a very interesting and enjoyable workshop.

September 4, 2014 Johannes Waldmann
Leipzig

Program Committee

Elvira Albert	Complutense University of Madrid
Sergio Antoy	Portland State University
Mauricio Ayala-Rincon	University of Brasilia
William Byrd	University of Utah
Michael Hanus	Universität Kiel
Herbert Kuchen	University of Münster
Carlos Olarte	Pontificia Universidad Javeriana Cali
Peter J. Stuckey	University of Melbourne
René Thiemann	University of Innsbruck
Janis Voigtländer	University of Bonn
Johannes Waldmann	HTWK Leipzig (chair)

Additional Reviewers

Maria Christakis
Moreno Falaschi
Keller, Gabriele
Daniele Nantes-Sobrinho
Ana Cristina Rocha Oliveira,
German Vidal

Part I

28th Workshop on (Constraint) Logic Programming

Embedding Defeasible Logic Programs into Generalized Logic Programs

Martin Baláž[1], Jozef Frtús[1], Martin Homola[1], Ján Šefránek[1], and Giorgos Flouris[2]

[1] Comenius Unversity 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 → and defeasible rules by ⇒. 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:

$$\rightarrow home \qquad r_1\colon home \Rightarrow works\_hard \qquad r_2\colon works\_hard \Rightarrow wins$$

$$\rightarrow key\_player\_injured \qquad r_3\colon key\_player\_injured \Rightarrow \mathrm{not}\, wins$$

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

A *rule* over $\mathcal{L}_{At}$ is an expression r of the form $L_1, \ldots, L_n \rightarrow L_0$ where $0 \leq n$ and $L_i \in \mathcal{L}_{At}$ for each $0 \leq i \leq n$. The literal $head(r) = L_0$ is called the *head* of r and the set of literals $body(r) = \{L_1, \ldots, L_n\}$ 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 At is the set of all atoms used in a program $\mathcal{P}$, it is said that $\mathcal{P}$ is over At. 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 $\mathbf{t}, \mathbf{u}, \mathbf{f}$.

Note that a GLP $\mathcal{P}$ can be viewed as consisting of two parts, a normal logic program $\mathcal{P}^+ = \{r \in \mathcal{P} \mid head(r) \in At\}$ (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 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 $A, \text{not}\, A$ 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* $\text{not}\, A \in I$.

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

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

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*

- *$I \leq_t J$ iff $T(I) \subseteq T(J)$ and $F(I) \supseteq F(J)$,*
- *$I \leq_k J$ 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 $\mathcal{P}$ is a positive logic program $\mathcal{P}^I$ obtained from $\mathcal{P}$ by replacing in every rule of $\mathcal{P}$ all default literals which are true (resp. unknown, resp. false) in I by propositional constant $\mathbf{t}$ (resp. $\mathbf{u}$, resp. $\mathbf{f}$).*

Finally a fixed-point condition is expressed on a reduced program, which is formally captured by the operator $\Gamma_{\mathcal{P}}$.

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

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

- *I is a* partial stable model *of $\mathcal{P}$ iff $\Gamma_{\mathcal{P}^+}(I) = I$*
- *I is a* well-founded model *of $\mathcal{P}$ iff I is a k-minimal partial stable model of $\mathcal{P}$*

- *I is a* maximal stable model *of $\mathcal{P}$ iff I is a k-maximal partial stable model of $\mathcal{P}$*
- *I is a* least-undefined stable model *of $\mathcal{P}$ iff I is a partial stable model of $\mathcal{P}$ with subset-minimal $\{A \in At \mid I(A) = \frac{1}{2}\}$*
- *I is a* total stable model *of $\mathcal{P}$ iff I is a partial stable model of $\mathcal{P}$ which is total*

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

Proposition 1. *If $\mathcal{P}$ 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 $\mathcal{P}$ 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 $(\mathcal{L}, \mathcal{R})$ where $\mathcal{L}$ is a language and $\mathcal{R}$ is a set of inference rules over $\mathcal{L}$. A *language* is a set $\mathcal{L}$ of all well-formed sentences. Each *inference rule* r over $\mathcal{L}$ is of the form $\varphi_1, \dots, \varphi_n \rightarrow \varphi_0$ where $0 \leq n$ and $\varphi_i \in \mathcal{L}$ for each $0 \leq i \leq n$. The sentence $head(r) = \varphi_0$ is called the *head* of r and the set of sentences $body(r) = \{\varphi_1, \dots, \varphi_n\}$ is called the *body* of r.

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

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

A *context* is a set $\Delta \subseteq \mathcal{A}$ of assumptions. We say that Δ is *conflict-free* iff $\{\alpha, \overline{\alpha}\} \nsubseteq Cn_{\mathcal{R}}(\Delta)$ for each assumption α. A context Δ is closed iff $\Delta = Cn_{\mathcal{R}}(\Delta) \cap \mathcal{A}$, i.e., only such assumptions, which are members of Δ, are derivable from Δ. A context Δ *attacks* an assumption α iff $\overline{\alpha} \in Cn_{\mathcal{R}}(\Delta)$. A context Δ *defends* an assumption α iff each closed context attacking α contains an assumption attacked by Δ.

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 Δ is

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

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 $\mathcal{L}_{At}$, a strict (resp. defeasible) rule is an expression $L_1, \dots, L_n \rightarrow L_0$ (resp. $L_1, \dots, L_n \Rightarrow L_0$) where $0 \leq n$ and $L_i \in \mathcal{L}_{At}$ 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 $name(r)$. The name of r is a default literal from separate language $\mathcal{L}_{\mathcal{N}}$. The intuitive meaning of $name(r) = \mathrm{not}\, A$ is "by default, the defeasible rule r can be used", and consequently, the intuitive meaning of $\mathrm{not}\, name(r) = A$ is "the defeasible rule r can not be used". In the following, we will denote the defeasible rule $r = L_1, \dots, L_n \Rightarrow L_0$ with $name(r) = \mathrm{not}\, A$ as $A\colon L_1, \dots, L_n \Rightarrow L_0$.

Definition 7 (Defeasible Logic Program). *Let At be a set of atoms, $\mathcal{N}$ be a set of names, and $At \cap \mathcal{N} = \emptyset$. A* defeasible logic program *is a tuple $\mathcal{P} = (\mathcal{S}, \mathcal{D}, name)$ where $\mathcal{S}$ is a set of* strict rules *over $\mathcal{L}_{At}$, $\mathcal{D}$ is a set of* defeasible rules *over $\mathcal{L}_{At}$, and $name\colon \mathcal{D} \mapsto \mathrm{not}\, \mathcal{N}$ 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 $\mathit{vuls}(A)$ 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* $\mathcal{P} = (\mathcal{S}, \mathcal{D}, \mathit{name})$ *be a defeasible logic program. An argument* A *for a literal* L *is*

1. *a* default argument L *where* L *is a default literal.*

$$\begin{aligned} \mathit{prems}(A) &= \{L\} \\ \mathit{rules}(A) &= \emptyset \end{aligned}$$

2. *a* deductive argument $[A_1, \ldots, A_n \rightsquigarrow L]$ *where* $0 \leq n$, *each* A_i, $0 < i \leq n$, *is an argument for a literal* L_i, *and* $r = L_1, \ldots, L_n \rightsquigarrow L$ *is a rule in* $\mathcal{P}$.

$$\begin{aligned} \mathit{prems}(A) &= \mathit{prems}(A_1) \cup \cdots \cup \mathit{prems}(A_n) \\ \mathit{rules}(A) &= \mathit{rules}(A_1) \cup \cdots \cup \mathit{rules}(A_n) \cup \{r\} \end{aligned}$$

For both kinds of an argument A,

$$\mathit{vuls}(A) = \mathit{prems}(A) \cup \mathit{name}(\mathit{rules}(A) \cap \mathcal{D})$$

Example 2. Consider a defeasible logic program consisting of the only defeasible rule $r\colon \mathrm{not}\, b \Rightarrow a$. Two default arguments $A_1 = [\mathrm{not}\, a]$, $A_2 = [\mathrm{not}\, b]$ and one deductive argument $A_3 = [A_2 \Rightarrow a]$ can be constructed. We can see that $\mathit{vuls}(A_1) = \{\mathrm{not}\, a\}$, $\mathit{vuls}(A_2) = \{\mathrm{not}\, b\}$, $\mathit{vuls}(A_3) = \{\mathrm{not}\, b, \mathrm{not}\, r\}$.

The difference between a default and a deductive argument for a literal not 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* $\mathcal{P}$ *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* not 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* $\text{not}\,L$ *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 $name(r)$ 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* $\text{not}\,L$*, and V is a default literal*

- $\text{not}\,L$ *if C is an undermining conflict; or*
- *$name(r)$ where r is a defeasible rule in $rules(A) \cup rules(B)$ 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 ρ, and denoted by $res(\rho)$. The set of vulnerabilities of ρ, denoted by $vuls(\rho)$, is defined as:

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

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 $vuls(\rho)$ are undefeated (i.e. true), then in order to resolve the conflict in ρ, the contrary $res(\rho)$ of the chosen vulnerability in ρ should be concluded (i.e. true). Notions of $vuls(\rho)$ and $res(\rho)$ 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 $res(\rho) = a$ and $vuls(\rho) = \{\text{not}\,a\}$. Please note that although $\text{not}\,a$ has to be removed to resolve conflict between A and B, it remains a vulnerability of ρ since $\text{not}\,a$ is self-attacking.

Example 4. Consider the following defeasible logic program $\mathcal{P}$

$$r_1\colon\ \Rightarrow a \qquad\qquad r_2\colon\ \Rightarrow b \qquad\qquad a \rightarrow \mathrm{not}\, c \qquad\qquad b \rightarrow c$$

and arguments $A = [[\ \Rightarrow a] \rightarrow \mathrm{not}\, c]$ and $B = [[\ \Rightarrow b] \rightarrow c]$. The rebutting conflict (A, B) can be resolved in two different ways, namely $\rho_1 = (A, B, \mathrm{not}\, r_1)$ is a conflict resolution with $res(\rho) = r_1$ and $vuls(\rho) = \{\mathrm{not}\, r_2\}$, and $\rho_2 = (A, B, \mathrm{not}\, r_2)$ is another conflict resolution with $res(\rho) = r_2$ and $vuls(\rho) = \{\mathrm{not}\, r_1\}$.

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

$$\begin{aligned} r_1\colon&\quad \Rightarrow b \\ r_2\colon&\ b \Rightarrow a \\ r_3\colon&\quad \Rightarrow \mathrm{not}\, a \end{aligned}$$

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

$$A_1 = [\ \Rightarrow b] \qquad A_2 = [A_1 \Rightarrow a] \qquad A_3 = [\ \Rightarrow \mathrm{not}\, a]$$

Then $R_1 = \{(A_3, A_2, \mathrm{not}\, r_3)\}$ is the last-link strategy and $R_2 = \{(A_3, A_2, \mathrm{not}\, r_1)\}$ 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 $\overline{}$. 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 $name(r)$

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 ρ, if all assumptions in $\mathit{vuls}(\rho)$ are accepted, then, in order to resolve the conflict in ρ, the atom $\mathit{res}(\rho)$ should be concluded. To achieve this, an inference rule $\mathit{vuls}(\rho) \to \mathit{res}(\rho)$ for each conflict resolution $\rho \in R$ is added to the set of inference rules $\mathcal{R}$.

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

- $\mathcal{L} = \mathcal{L}_{At} \cup \mathcal{L}_{\mathcal{N}}$,
- $\mathcal{R} = \mathcal{S} \cup \{\mathit{body}(r) \cup \{\mathit{name}(r)\} \to \mathit{head}(r) \mid r \in \mathcal{D}\} \cup \{\mathit{vuls}(\rho) \to \mathit{res}(\rho) \mid \rho \in R\}$,
- $\mathcal{A} = \mathrm{not}\, At \cup \mathrm{not}\, \mathcal{N}$,
- $\overline{\mathrm{not}\, A} = A$ *for each atom* $A \in At \cup \mathcal{N}$.

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:

- $\mathcal{L} = \{a, \mathrm{not}\, a, b, \mathrm{not}\, b, c, \mathrm{not}\, c\} \cup \{r_1, \mathrm{not}\, r_1, r_2, \mathrm{not}\, r_2\}$
- $R = \{b \to \mathrm{not}\, c, a \to c\} \cup \{\mathrm{not}\, r_1 \to b, \mathrm{not}\, r_2 \to a\} \cup \{\mathrm{not}\, r_1 \to r_2, \mathrm{not}\, r_2 \to r_1\}$
- $\mathcal{A} = \{\mathrm{not}\, a, \mathrm{not}\, b, \mathrm{not}\, c\} \cup \{\mathrm{not}\, r_1, \mathrm{not}\, r_2\}$
- $\overline{\mathrm{not}\, A} = A$ 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 $\mathcal{F} = (\mathcal{L}, \mathcal{R}, \mathcal{A}, ^-)$, by $\mathcal{F}^+$ we denote its flattening – that is, $\mathcal{F}^+$ is the ABF $(\mathcal{L}, \{r \in \mathcal{R} \mid \mathit{head}(r) \notin \mathcal{A}\}, \mathcal{A}, ^-)$.

Definition 12 (Extension). *Let $\mathcal{P} = (\mathcal{S}, \mathcal{D}, \mathit{name})$ be a defeasible logic program, R be a set of conflict resolutions, and $\mathcal{F} = (\mathcal{L}, \mathcal{R}, \mathcal{A}, ^-)$ 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 $Cn_{\mathcal{R}}(E) \subseteq E$ and $Cn_{\mathcal{R}}(E') \subseteq E'$;*
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 $E' = \mathcal{L} \setminus \mathrm{not}\, E$.

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 $\mathit{name}(r)$ becomes true it means that the respective defeasible rule is defeated. By default $\mathit{name}(r)$ 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 $\mathit{vuls}(\rho) \rightarrow \mathit{res}(\rho)$ 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 $\mathcal{P} = (\mathcal{S}, \mathcal{D}, \mathit{name})$ 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*

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

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 $T(\mathcal{P}, R)$.*

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

A remarkable special case happens when the input program $\mathcal{P}$ 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 $\mathcal{S}$ be a generalized logic program and $\mathcal{P} = (\mathcal{S}, \emptyset, \emptyset)$ a defeasible logic program with the empty set of conflict resolutions. Then*

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

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 $\mathcal{P}$, any set of conflict resolutions R, and any extension E of $\mathcal{P}$ 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 $\mathcal{R}' = \mathcal{S} \cup \{body(r) \cup \{name(r)\} \rightarrow head(r) \mid r \in \mathcal{D}\}$. Then $Cn_{\mathcal{R}'}(E) \subseteq E$.

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 *positive defeat*) requires that for each default literal not 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.

Property 3 (Positive Defeat). For each atom $A \in \mathcal{L}_{\mathcal{N}}$, if there exists a conflict resolution $\rho \in R$ with $\mathit{res}(\rho) = A$ and $\mathit{vuls}(\rho) \subseteq E$ 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 *negative 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.

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

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).

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.*

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 $\mathit{vuls}(\rho) \rightarrow \mathit{res}(\rho)$ is added for every conflict resolution $\rho \in R$.

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.*

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 inference rules – defeasible logic programs [8] and ASPIC$^+$ [14,11]. It is known that the semantics in [8] does not satisfy rationality postulates formalized in [4], especially the closure property. Although ASPIC$^+$ 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

1. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., Przymusinski, T.C.: Dynamic logic programming. In: APPIA-GULP-PRODE 1998. pp. 393–408 (1998)
2. Baláž, M., Frtús, J., Homola, M.: Conflict resolution in structured argumentation. In: LPAR-19 (Short Papers). EPiC, vol. 26, pp. 23–34. EasyChair (2014)
3. Bondarenko, A., Dung, P.M., Kowalski, R.A., Toni, F.: An abstract, argumentation theoretic approach to default reasoning. Artif. Intell. 93(1-2), 63–101 (1997)
4. Caminada, M., Amgoud, L.: On the evaluation of argumentation formalisms. Artif. Intell. 171(5-6), 286–310 (2007)
5. Caminada, M., Carnielli, W.A., Dunne, P.E.: Semi-stable semantics. J. Log. Comput. 22(5), 1207–1254 (2011)
6. Caminada, M., Sá, S., Alcântara, J.: On the equivalence between logic programming semantics and argumentation semantics. In: ECSQARU 2014 (2013)
7. Dung, P.M.: On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. Artif. Intell. 77, 321–357 (1995)
8. García, A.J., Simari, G.R.: Defeasible logic programming: an argumentative approach. TPLP 4(2), 95–138 (2004)
9. Gelder, A.V., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 619–649 (1991)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. pp. 1070–1080. MIT Press (1988)
11. Modgil, S., Prakken, H.: Revisiting preferences and argumentation. In: IJCAI 2001. pp. 1021–1026. AAAI Press (2011)
12. Nute, D.: Defeasible reasoning and decision support systems. Decision Support Systems 4(1), 97–110 (1988)
13. Pollock, J.L.: Defeasible reasoning. Cognitive Science 11(4), 481–518 (1987)
14. Prakken, H.: An abstract framework for argumentation with structured arguments. Argument & Computation 1(2), 93–124 (2010)
15. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. Journal of Applied Nonclassical Logics 7(1), 25–75 (1997)
16. Prakken, H., Vreeswijk, G.: Logics for defeasible argumentation. In: Gabbay, D., Guenthner, F. (eds.) Handbook of Philosophical Logic, pp. 219–318. Springer (2002)
17. Przymusinski, T.C.: The well-founded semantics coincides with the three-valued stable semantics. Fundam. Inform. 13(4), 445–463 (1990)
18. Saccà, D.: The expressive powers of stable models for bound and unbound DATALOG queries. J. Comput. Syst. Sci. 54(3), 441–464 (Jun 1997)
19. Toni, F.: Assumption-based argumentation for closed and consistent defeasible reasoning. In: JSAI 2007. LNCS, vol. 4914, pp. 390–402. Springer (2008)
20. Wan, H., Grosof, B., Kifer, M., Fodor, P., Liang, S.: Logic programming with defaults and argumentation theories. In: ICLP 2009. LNCS, vol. 5649, pp. 432–448. Springer (2009)

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 CO^4 language is a Haskell-like language for specifying constraint systems over structured finite domains. A CO^4 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 CO^4 compiler. The goal is to help the programmer in understanding the resource consumption of CO^4 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

CO^4 is a constraint programming language that allows to write a constraint problem as declarative specification. The CO^4 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 CO^4, a constraint system over elements of set U is specified by a parametrized predicate `constraint` $: P \times U \to \texttt{Bool}$, 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 $p \in P$, $u \in U$ is a solution if `constraint` $(p, u) = \texttt{True}$.

For the CO^4 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 CO^4 (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 CO^4 implementation measures concrete costs of SAT-compiled function calls. The third part outlines future work in static analysis of CO^4 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 CO^4

Syntactically, CO^4 is a subset of Haskell. Domains are specified by algebraic data types (ADT), where *constructors* enumerate the values of the type.

```
data Bool       = False | True
data Color      = Red   | Green | Blue
data Monochrome = Black | White
data Pixel      = Colored Color | Background Monochrome
```

CO^4 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.

```
data Pair   a b = Pair a b
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):

```
case color of Blue      -> True
              otherwise -> False
```

Case distinctions provides conditional branching of the control-flow. Other kinds of expressions in CO^4 are constructor calls, applications, abstractions and local bindings. CO^4 provides restricted support of higher-order, polymorphic functions. Besides type definitions, constraint systems in CO^4 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 CO^4 over elements of set U is a binary predicate `constraint` : $P \times U \rightarrow$ `Bool` on U and some parameter domain P. In Listing 1.1, $P =$ `Bool` and $U =$ `Pixel`.

For a given parameter $p \in P$, $u \in U$ is a solution if `constraint` (p, u) `= True`. One advantage of specifying constraint systems in a functional language like CO^4 is that a solution can be tested against the constraint simply by evaluating `constraint` (p, u). Note that CO^4 expressions are evaluated strictly, while Haskell features a non-strict evaluation strategy.

3 Propositional Encoding of CO^4 constraints

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

To find a solution $u \in U$ for a constraint `constraint` : $P \times U \rightarrow$ `Bool` and a parameter $p \in P$, CO^4 performs the following steps:

1. The concrete program is transformed into an *abstract program*. An abstract program doesn't operate on concrete values, but on *abstract values*.
2. Evaluating the abstract program for an abstract value that represents parameter p gives a formula $f \in \mathbb{F}$ 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 σ. Optionally, testing whether `constraint` p $u =$ `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 $\mathbb{F}$ being the set of propositional formulas. Then, the set of abstract values $\mathbb{A}$ is the smallest set with $\mathbb{A} = \mathbb{F}^* \times \mathbb{A}^*$ where $\mathbb{F}^*$ denotes the set of sequences with elements from $\mathbb{F}$. An abstract value $a \in \mathbb{A}$ is a tuple $(\overrightarrow{f}, \overrightarrow{a})$ of* flags $\overrightarrow{f}$ *and* arguments $\overrightarrow{a}$.

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

Example 1. For an abstract value $a_1 \in \mathbb{A}$ to represent a value of the ADT `data Color = Red | Green | Blue | Purple` it must contain two flags $f_1, f_2 \in \mathbb{F}$ 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 $a_2 \in \mathbb{A}$ that is supposed to represent a value of type `Maybe Color`. As `Maybe` consists of two constructors, one flag $f_3 \in \mathbb{F}$ 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 $a \in \mathbb{A}$ may contain propositional variables, a can be decoded to different values according to the Boolean values that are assigned to these variables. By $\mathsf{decode}_T : \mathbb{A} \times \Sigma \to T$ we denote a mapping from abstract values and propositional assignments Σ to concrete values.

If the flags of an abstract value $a \in \mathbb{A}$ 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 $\mathsf{encode}_T : T \to \mathbb{A}$ 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:

$$\mathsf{encode}_{\texttt{Color}}(\texttt{Blue}) = ((\textsc{True}, \textsc{False}), ())$$
$$\mathsf{encode}_{\texttt{Maybe Color}}(\texttt{Just Blue}) = (\textsc{True}, ((\textsc{True}, \textsc{False}), ()))$$

As we've omitted details about abstract values we don't provide definitions for encode and 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:

$$r \texttt{ = case } x \texttt{ of \{ False -> } g \texttt{ ; True -> } h \texttt{ \}}$$

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

```
r' = let _1 = g'
         _2 = h'
     in merge_x'(_1,_2)
```

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

The function $\mathsf{merge}_x : \mathbb{A}^* \to \mathbb{A}$ 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 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 $\mathsf{merge}_{x'}(g', h')$:

$$\begin{aligned} &(x' \equiv \mathsf{encode}_{\texttt{Bool}}(\texttt{False}) \implies r' \equiv g') \\ \wedge\ &(x' \equiv \mathsf{encode}_{\texttt{Bool}}(\texttt{True}) \implies r' \equiv h') \end{aligned}$$

Informally, both clauses encode the semantics of the original case distinction in terms of abstract values: r' equals g' if x' equals $\mathsf{encode}_{\texttt{Bool}}(\texttt{False})$, 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` $: P \times U \to$ `Bool` function in a concrete program has a counterpart `constraint'` $: \mathbb{A} \times \mathbb{A} \to \mathbb{A}$ in the abstract program of the same arity. Evaluating `constraint'` p' u' on

- $p' = \mathsf{encode}_P(p)$ 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 $\mathsf{decode}_U(u', \sigma)$.

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 $f : A \to B$ in a concrete program by analyzing its counter part $f' : \mathbb{A} \to \mathbb{A}$ in the corresponding abstract program. The costs of f' depend on the size of its argument. Thus, we introduce a function $\mathsf{size} : \mathbb{A} \to \mathbb{N}$ to measure the size of an abstract value.

Example 5. There are at least two naive definitions for size: one that counts the number of nested abstract values

$$\mathsf{size}_1(\overrightarrow{f}, (a_1, \ldots, a_n)) = 1 + \sum_{i=1}^{n} \mathsf{size}_1(a_i)$$

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

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

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

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

$$s_f(n) = \max\{\mathsf{size}(f(\mathsf{encode}_A(x))) \mid x \in A \wedge \mathsf{size}(\mathsf{encode}_A(x)) \le n\}$$

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*

$$c_f(n) = \max\{\mathsf{work}(f, \mathsf{encode}_A(x)) \mid x \in A \wedge \mathsf{size}(\mathsf{encode}_A(x)) \le n\}$$

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

We can instantiate this scheme in several ways: for example, $\mathsf{work}(f, x)$ could give the number of propositional variables or clauses that are allocated while computing the abstract value $f'(\mathsf{encode}_A(x))$. 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 $b_1, \dots, b_n$ in a function f:

$$\begin{aligned} f(x) \ \texttt{=}\ \texttt{case}\ x\ \texttt{of}\ & C_1\ \texttt{->}\ b_1 \\ & \dots \\ & C_n\ \texttt{->}\ b_n \end{aligned}$$

Listing 1.2. A case distinction over n branches

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

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. $\mathsf{work}_{\mathbb{V}}(f, x)$ *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*

$$\mathsf{work}_{\mathbb{V}}(f, x) = \max\{m_i \mid 1 \le i \le n\}$$

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

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. $\mathsf{work}_{\mathbb{C}}(f, x)$ *denotes the clause-cost of function* f *in Listing 1.2 where* n *denotes the number of branches:*

$$\mathsf{work}_{\mathbb{C}}(f, x) = 2 * \mathsf{work}_{\mathbb{V}}(f, x) * n$$

$\mathsf{work}_{\mathbb{C}}$ is reasonable because two clauses are emitted for each of the $\mathsf{work}_{\mathbb{V}}(f, x)$ flags in r' and each of the n branches.

6 Profiling CO4

In the following we compare the profiling output of CO^4 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.

```haskell
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 $b \in \mathbb{A}$ 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 $\mathsf{work}_{\mathbb{V}}$ (resp. $\mathsf{work}_{\mathbb{C}}$) cost function, because

- $\mathsf{work}_{\mathbb{V}}(\mathtt{f1},\mathtt{b}) = \max\{1,1\} = 1$ as each branch in `f1’` is represented by an abstract value with one flag (because `Bool` has two constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{f1},\mathtt{b}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{f1},\mathtt{b}) * 2 = 4$ 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 $\mathsf{work}_{\mathbb{V}}$ and $\mathsf{work}_{\mathbb{C}}$ defined in Section 5, because

- $\mathsf{work}_{\mathbb{V}}(\mathtt{f2},\mathtt{b}) = \max\{2,2\} = 2$ as each branch in `f2’` is represented by an abstract value with two flags (because `T` has three constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{f2},\mathtt{b}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{f2},\mathtt{b}) * 2 = 8$ as there are $n = 2$ branches in `f2’`
- $\mathsf{work}_{\mathbb{V}}(\mathtt{g},\mathtt{t}) = \max\{1,1,1\} = 1$ as each branch in `g’` is represented by an abstract value with one flag (because `Bool` has two constructors)
- $\mathsf{work}_{\mathbb{C}}(\mathtt{g},\mathtt{t}) = 2 * \mathsf{work}_{\mathbb{V}}(\mathtt{g},\mathtt{t}) * 3 = 6$ as there are $n = 3$ branches in `g’`

Note that `f2'` allocates more variables than `f1'` because it merges branches of type `T`, which has more constructors than `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].

```
Profiling (inner-under):
("constraint'", {numCalls = 1, numVariables = 160, numClauses = 514})
("allHOInst'", {numCalls = 1, numVariables = 160, numClauses = 514})
("mapHOInst'", {numCalls = 4, numVariables = 157, numClauses = 506})
("globalLam'", {numCalls = 3, numVariables = 157, numClauses = 506})
("globalLamSat'", {numCalls = 3, numVariables = 157, numClauses = 506})
("lpo'", {numCalls = 41, numVariables = 154, numClauses = 500})
...
```

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, `constraint'` allocates the most resources according to inner-under-profiling as it is the top-level function in every abstract program.

```
Profiling (inner):
("gtNat'", {numCalls = 9, numVariables = 36, numClauses = 171})
("lpo'", {numCalls = 41, numVariables = 30, numClauses = 92})
("ordNat'", {numCalls = 9, numVariables = 27, numClauses = 63})
("eqNat'", {numCalls = 19, numVariables = 27, numClauses = 99})
("and2'", {numCalls = 26, numVariables = 20, numClauses = 44})
("eqOrder'", {numCalls = 31, numVariables = 18, numClauses = 40})
...
```

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

Profiling (inner):
("equal'", {numCalls = 717, numVariables = 1724, numClauses = 5100})
("add'", {numCalls = 359, numVariables = 352, numClauses = 704})
("and2'", {numCalls = 101, numVariables = 100, numClauses = 291})
("not'", {numCalls = 84, numVariables = 84, numClauses = 168})
("less'", {numCalls = 65, numVariables = 64, 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 `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, $\texttt{List}^{!}\ \texttt{Bool}^{?}$ denotes a list type, where each of list constructor is known, but each element of type `Bool` has an unknown constructor. Thus, such a type encodes a list of known length with unknown Boolean elements.

---

[2] available at `https://github.com/apunktbau/co4/blob/master/test/CO4/Example/QueensSelfContained.hs`

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 !, 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 CO^4 constraint system.

A possible approach to predict the resource cost is to annotate each function in a 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 $s_f, c_f : \mathbb{N} \to \mathbb{N}$.

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 $\mathsf{work}(f, \mathsf{encode}(x))$ is less or equal to $c_f(\mathsf{size}(\mathsf{encode}(x)))$ for some cost function 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 CO^4 programs in order to estimate the size of the propositional encodings.

References

1. Alexander Bau and Johannes Waldmann. Propositional Encoding of Constraints over Tree-Shaped Data. In *22nd International Workshop on Functional and (Constraint) Logic Programming*, 2013.
2. Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate Amortized Resource Analysis. *ACM Trans. Program. Lang. Syst.*, 34(3):14:1–14:62, November 2012.

3. Simon Peyton Jones, editor. *Haskell 98 Language and Libraries, The Revised Report.* Cambridge University Press, 2003.
4. David Overton, Zoltan Somogyi, and Peter J. Stuckey. Constraint-based mode analysis of mercury. In *PPDP*, pages 109–120, 2002.

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

PBR4J. Either the mapping is given indirectly by the structure of the class or directly by annotations. While PBR4J 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 variables are bound different queries are possible. PBR4J 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 PBR4J. 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 describe 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 infinite 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 PROLOG. The presented implementation of the connector is lightweight. The object term mapping is realised with only two classes. As communication interface we have presented 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 PROLOG implementation. An evaluation of the performance has shown that the `PI` in combination 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 required 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 implementation 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:

```
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:

```
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 transformes a JAVA int to a 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;
  private int pages;
  // ... constructor\ getter\ setter
}

class Author {
  private String name;
  // ... constructor\ getter\ setters
}
```

As one can see, we now have a reference from the `Book` class to the `Author` class.

```
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:

```
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:

```
@PlViews({
  @PlView(viewId="book1", orderArgs={"title"}),
  @PlView(viewId="book2", ignoreArgs={"author"}),
  @PlView(viewId="book3", functor="tome",
    modifyArgs={@PlArg(valueOf="pages", type=ATOM)}
  )}
)
class Book {
  private String title;
  private Author author;
  private int pages;

  // ... setters / getters
}
```

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 `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

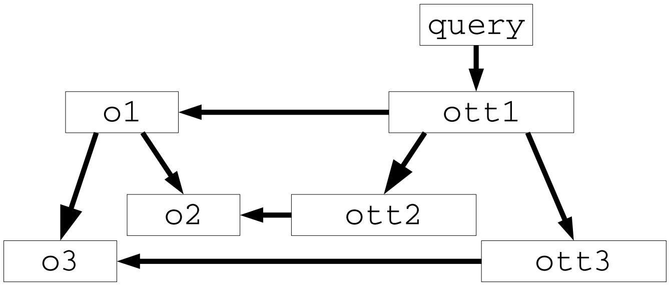


Fig. 1: A tree of `OTT` Objects

`o1` 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`

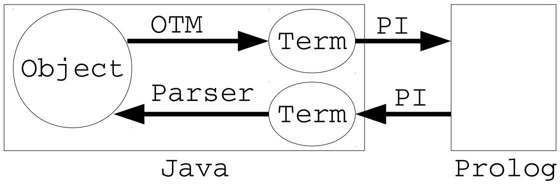


Fig. 2: Components of the Connector

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.

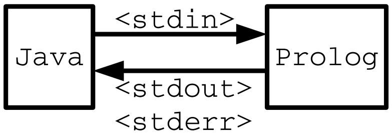


Fig. 3: Structure of the Pipe Interface

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 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.

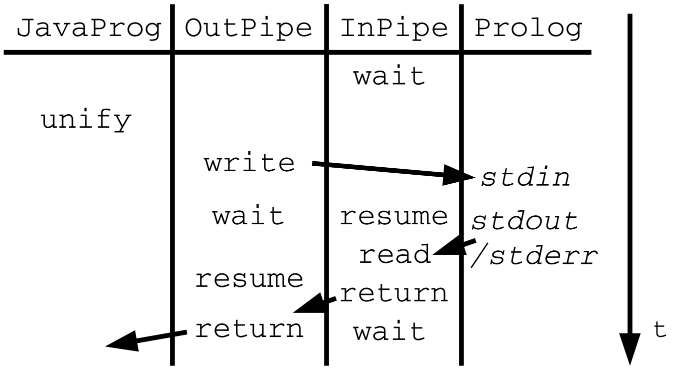


Fig. 4: Information Flow via the `PPI`

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 necessary. We have tested[1] 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.

B-Prolog	Ciao	Gnu	Swi	Xsb	Yap
3.0 sec	15.5 sec	3.0 sec	7.4 sec	3.7 sec	2.9 sec

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.

Characters	B-Prolog	Ciao	Gnu	Swi	Xsb	Yap
100	4.4 sec	15.5 sec	4.2 sec	12.5 sec	5.5 sec	8.6 sec
1000	18.4 sec	33.4 sec	18.5 sec	40.5 sec	18.6 sec	61.7 sec

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:

```
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.

	B-Prolog	Ciao	Gnu	Swi	Xsb	Yap
London	7.8 sec	25.5 sec	4.6 sec	15.3 sec	8.5 sec	5.0 sec
Sydney	6.3 sec	22.8 sec	4.4 sec	15.2 sec	7.7 sec	4.9 sec
Vienna	5.2 sec	22.0 sec	3.8 sec	13.7 sec	7.6 sec	4.7 sec

---

[1] 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

1. A. Amandi, M. Campo, A. Zunino. *JavaLog: a framework-based integration of Java and Prolog for agent-oriented programming.* Computer Languages, Systems & Structures 31.1, 2005. 17-33.
2. M. Banbara, N. Tamura, K. Inoue. *Prolog Cafe: A Prolog to Java Translator.* Proc. Intl. Conference on Applications of Knowledge Management, INAP 2005, Lecture Notes in Artificial Intelligence, Vol. 4369, Springer, 2006. 1-11.
3. M. Calejo. *InterProlog: Towards a Declarative Embedding of Logic Programming in Java.* Proc. Conference on Logics in Artificial Intelligence, 9th European Conference, JELIA, Lisbon, Portugal, 2004.
4. S. Castro, K. Mens, P. Moura. *LogicObjects: Enabling Logic Programming in Java through Linguistic Symbiosis.* Practical Aspects of Declarative Languages. Springer Berlin Heidelberg, 2013. 26-42.
5. S. Castro, K. Mens, P. Moura. *JPC: A Library for Modularising Inter-Language Conversion Concerns between Java and Prolog.* International Workshop on Advanced Software Development Tools and Techniques (WASDeTT), 2013.
6. S. Castro, K. Mens, P. Moura. *Customisable Handling of Java References in Prolog Programs.* arXiv preprint arXiv:1405.2693, 2014.
7. M. Cimadamore, M. Viroli. *A Prolog-oriented extension of Java programming based on generics and annotations.* Proc. 5th international symposium on Principles and practice of programming in Java. ACM, 2007. 197-202.

8. K. Gybels. *SOUL and Smalltalk - Just Married: Evolution of the Interaction Between a Logic and an Object-Oriented Language Towards Symbiosis.* Proc. of the Workshop on Declarative Programming in the Context of Object-Oriented Languages, 2003.
9. M. D'Hondt, K. Gybels, J. Viviane *Seamless Integration of Rule-based Knowledge and Object-oriented Functionality with Linguistic Symbiosis.* Proc. of the 2004 ACM symposium on Applied computing. ACM, 2004.
10. T. Majchrzak, H. Kuchen. *Logic java: combining object-oriented and logic programming.* Functional and Constraint Logic Programming. Springer Berlin Heidelberg, 2011. 122-137.
11. L. Ostermayer, D. Seipel. *Knowledge Engineering for Business Rules in Prolog.* Proc. Workshop on Logic Programming (WLP), 2012.
12. L. Ostermayer, D. Seipel. *Simplifying the Development of Rules Using Domain Specific Languages in Drools.* Proc. Intl. Conf. on Applications of Declarative Programming and Knowledge Management (INAP), 2013.
13. L. Ostermayer, D. Seipel. *A Prolog Framework for Integrating Business Rules into Java Applications.* Proc. 9th Workshop on Knowledge Engineering and Software Engineering (KESE), 2013.
14. L. Ostermayer, F. Flederer, D. Seipel. *A Customisable Mapping between Java Objects and Prolog Terms.* `http://www1.informatik.uni-wuerzburg.de/database/papers/otm_2014.pdf`
15. L. Ostermayer, F. Flederer, D. Seipel. *CAPJa - A Connector Architecture for Prolog and Java.* `http://www1.informatik.uni-wuerzburg.de/pub/ostermayer/paper/capja_2014.html`
16. P. Singleton, F. Dushin, J. Wielemaker. *JPL 3.0: A Bidirectional Prolog/Java Interface.* `http://www.swi-prolog.org/packages/jpl/`
17. J. Wielemaker, T. Schrijvers, T. Markus, L. Torbjörn. *SWI-Prolog.* Theory and Practice of Logic Programming. Cambridge University Press, 2012. 67-96.
18. J. Wielemaker. *SWI Prolog.* `http://www.swi-prolog.org`

Declarative Evaluation of Ontologies with Rules

Dietmar Seipel, Joachim Baumeister, and Klaus Prätor

University of Würzburg, Institute of Computer Science, Germany
`{seipel,baumeister}@informatik.uni-wuerzburg.de`
`praetor@mac.com`

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 correesponds 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 subset 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 combination 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 $\odot(A, A')$, if both atoms have the same argument tuple, and their predicate symbols are related by $\odot$, i.e., if A and A' both are

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

E.g., the relation $\odot$ can be `sub_class`, `isa`, `disjoint`, `complement`, etc. From a relationship $\odot(A, A')$ 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, ..., A', or B_i can denote both class atoms and property atoms, whereas `As`, `Bs`, ..., denote sets of class atoms and property atoms. We denote a relationship A is-a A' by `isa(A, A')`. SWRL rules $B_1 \wedge \cdots \wedge B_n \Rightarrow A$ are represented as non–ground DATALOG* facts `rule(A-Bs)` (with variable symbols), where `Bs` $= [B_1, \ldots, B_n]$ 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 ”`,`”, disjunction (or) is denoted by ”`;`”, and negation by ”`\+`”.

Incompatible Classes: Complements and Disjointness. For classes, there exists the construct *complementOf* to point to instances that do not belong to a specified class. In DATALOG*, the complement relation between two classes `C1` and `C2` is denoted by `complement(C1,C2)`. In OWL, the disjointness between two classes is defined by the *disjointWith* constructor; with `disjoint(C1,C2)` we denote the disjointness between two classes `C1` and `C2`. We call two classes `C1` and `C2` *incompatible*, if there exists a disjoint or a complement relation between them. This is detected by the following PROLOG predicate:

```
incompatible(C1, C2) :-
   ( complement(C1, C2)
   ; disjoint(C1, C2) ).
```

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*:

```
derives(C1, C2) :-
   ( isa(C1, C2)
   ; rule(A-[B]) B =.. [C1, X1], A =.. [C2, X2],
     var(X1), X1 == X2 ).

isa(C1, C2) :-
   sub_class(C1, C2).
isa(C1, C3) :-
   isa(C1, C2), sub_class(C2, C3).
```

Observe, that the call `var(X1), X1 == X2` tests if `X1` and `X2` 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 `equivalent_class` and `equivalent_property`, we can further extend the definition of `derives`: an element `E1` is derived by an element `E2`, if the elements are equivalent classes or properties. Since such an equivalence is symmetrical, the predicate `derives/2` always creates cyclic dervations of equivalent elements with length 1.

```
derives(E1, E2) :-
   ( equivalent_class(E1, E2)
   ; equivalent_property(E1, E2) ).
```

We compute the transitive closure `tc_derives` of `derives` using the following simple, standard DATALOG* scheme:

```
tc_derives(E1, E2) :-
   derives(E1, E2).
tc_derives(E1, E3) :-
   derives(E1, E2), tc_derives(E2, E3).
```

Subsequently, the reflexive transitive closure `tcr_derives` of `derives` is computed using the following PROLOG predicate:

```
tcr_derives(E1, E2) :-
   ( E1 = E2
   ; tc_derives(E1, E2) ).
```

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: the 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 $\mathcal{D}_1$ and $\mathcal{D}_2$ of DATALOG* rules:

- The upper layer $\mathcal{D}_2$ consists of the rules for the predicate `anomaly/2` and some DATALOG* rules that are stated together with them.
- The lower layer $\mathcal{D}_1$ consists of all other DATALOG* rules. E.g., the rules for predicates `derives` and `tc_derives` are in $\mathcal{D}_1$.

$\mathcal{D}_1$ 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 $\mathcal{D}_2$. The *stratification* into two layers is necessary, because $\mathcal{D}_2$ refers to $\mathcal{D}_1$ through *negation* and *aggregation*. Most PROLOG predicates in this paper support the layer $\mathcal{D}_2$.

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

```
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:

```
head_predicate(A-_, P) :-
   functor(A, P, _).

body_predicate(_-Bs, P) :-
   member(B, Bs), 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 `findall`, respectively:

```
sibling(X, Y) :-
   sub_class(X, Z), sub_class(Y, Z), X \= Y.

siblings(Xs) :-
   sibling(X, _),
   findall( Z,
      sibling(X, Z),
      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 to inefficient.

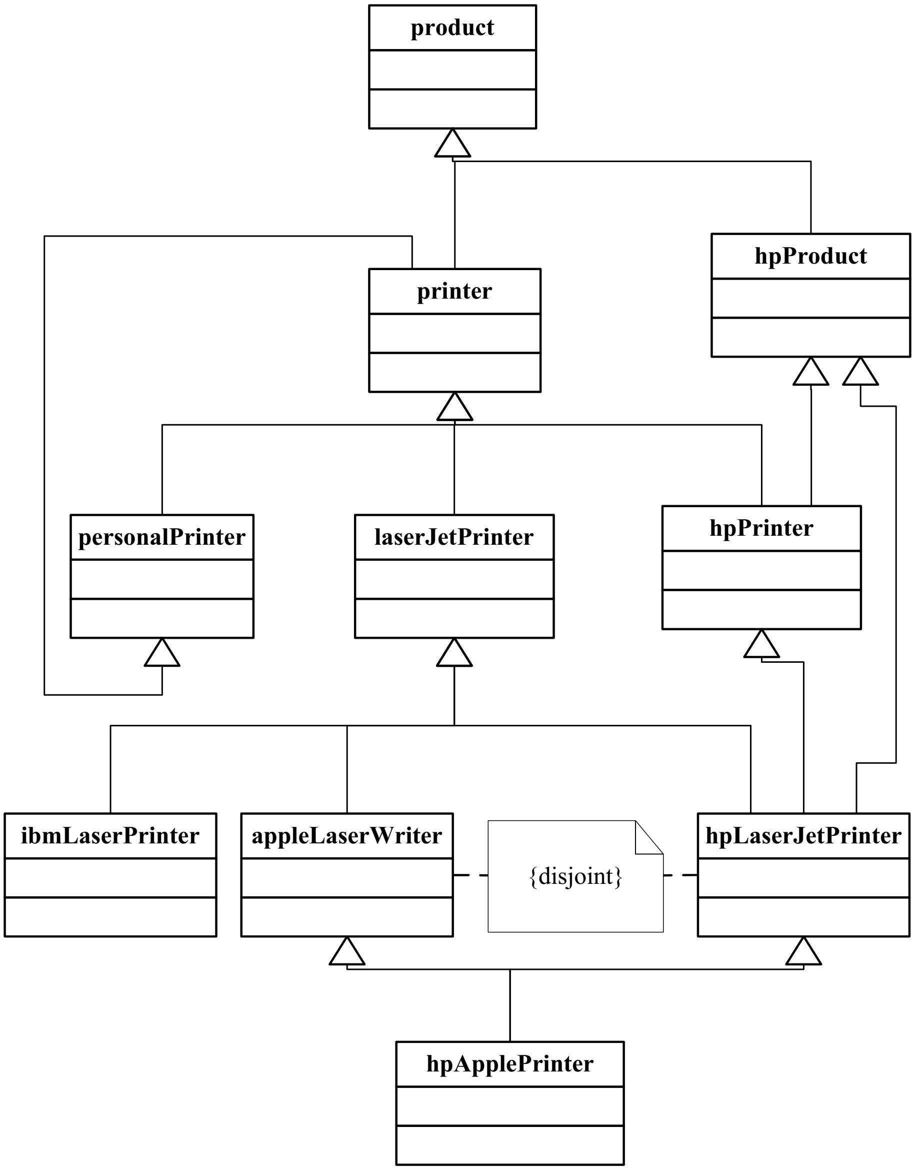
Evaluation of DATALOG*. DATALOG* rules cannot be evaluated in PROLOG or DATALOG alone for the following reasons: Current DATALOG engines cannot handle function symbols and non–ground facts, and they do not allow for the embedded computations, which we need. Standard PROLOG systems cannot easily handle 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 explicitely 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 .
p:laserJetPrinter  rdfs:subClassOf p:printer .
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 ;
    owl:disjointWith p:hpLaserJetPrinter .
```

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" ;
    rdfs:subClassOf p:hpLaserJetPrinter, p:appleLaserWriter .
```

The existence of such printers would contradict the `disjointWith` restriction between the classes `hpLaserJetPrinter` and `apperLaserWriter`. 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
    rdfs:subClassOf p:laserJetPrinter, p:hpPrinter, p:hpProduct ;
    owl:disjointWith p:appleLaserWriter .
```

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:

```
p:1001 rdf:type p:appleLaserWriter .
p:1002 rdf:type p:appleLaserWriter .
p:1003 rdf:type p:hpLaserJetPrinter .
p:1004 rdf:type p:hpLaserJetPrinter .
```

As mentioned before, there cannot exist instances of the class `hpApplePrinter`.

The ontology editor Protégé offers plugged–in reasoners, such as FaCT++, HermiT, and Racer. The ontology reasoner FaCT++ can inferr that the class `hpApplePrinter` is `EquivalentTo` the empty class `Nothing`. By clicking the question mark, an explanation can be shown. There are also databases for handling `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:

```
<rdf:Description rdf:about="printer#hpLaserJetPrinter">
  <rdfs:subClassOf rdf:resource="printer#hpPrinter"/>
  <rdfs:subClassOf rdf:resource="printer#laserJetPrinter"/>
</rdf:Description>
```

Protégé. Figure 1 shows the printer ontology in the standard ontolgy editor Protégé.

3.2 Declarative Queries in FNQuery

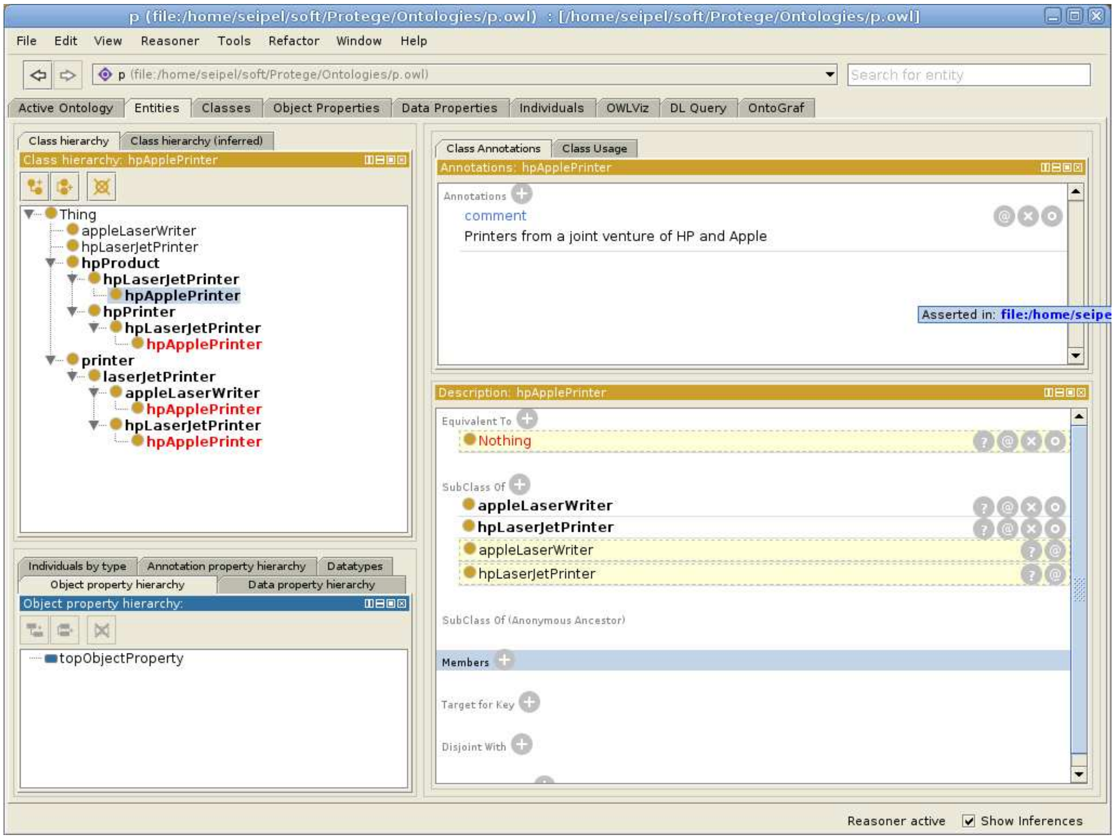
In PROLOG, an XML element can be represented as a term structure `T:As:C`, called FN–triple. `T` is the tag of the element, `As` is the list of the attribute/value pairs `A:V` of the element, and `C` is a list of FN–triples for the sub–elements.

```
'owl:Class':['rdf:ID':'appleLaserWriter']:[
   'rdfs:comment':['Apple laser ...'],
   'rdfs:subClassOf':[
      'rdf:resource':'#laserJetPrinter']:[],
   'owl:disjointWith':[
      'rdf:resource':'#hpLaserJetPrinter']:[] ]
```

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

```
% isa(+Owl, ?C1, ?C2) <-

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


p (file:/home/seipel/soft/Protege/Ontologies/p.owl) : [/home/seipel/soft/Protege/Ontologies/p.owl]
File Edit View Reasoner Tools Refactor Window Help
Search for entity
Active Ontology Entities Classes Object Properties Data Properties Individuals OWLViz DL Query OntoGraf
Class hierarchy Class hierarchy (inferred)
Class hierarchy: hpApplePrinter
Thing
appleLaserWriter
hpLaserJetPrinter
hpProduct
hpLaserJetPrinter
hpApplePrinter
hpPrinter
hpLaserJetPrinter
hpApplePrinter
printer
laserJetPrinter
appleLaserWriter
hpApplePrinter
hpLaserJetPrinter
hpApplePrinter
Individuals by type Annotation property hierarchy Datatypes
Object property hierarchy Data property hierarchy
Object property hierarchy:
topObjectProperty
Class Annotations Class Usage
Annotations: hpApplePrinter
Annotations
comment
Asserted in: file:/home/seipe
Description: hpApplePrinter
Equivalent To
Nothing
SubClass Of
appleLaserWriter
hpLaserJetPrinter
appleLaserWriter
hpLaserJetPrinter
SubClass Of (Anonymous Ancestor)
Members
Target for Key
Disjoint With
Reasoner active
Show Inferences

Printers from a joint venture of HP and Apple

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 := Owl/'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`.

```
% 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`.

```
% 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

1. J.Baumeister, D. Seipel: Anomalies in Ontologies with Rules.. Journal of Web Semantics: Science, Services and Agents on the World Wide Web 8 (2010), No. 1, pp. 55–68.
2. D. Seipel, DisLog – A System for Reasoning in Disjunctive Deductive Databases, http://www1.pub.informatik.uni-wuerzburg.de/databases/DisLog/dislog_nmr.html.
3. I. Horrocks, B. Parsia, P. Patel-Schneider, J. Hendler, Semantic Web Architecture: Stack or Two Towers?, in: F. Fages, S. Soliman (Eds.), Principles and Practice of Semantic Web Reasoning (PPSWR), No. 3703 in LNCS, Springer, 2005, pp. 37–41.
4. I. Horrocks, P. F. Patel-Schneider, S. Bechhofer, D. Tsarkov, OWL Rules: A Proposal and Prototype Implementation, Journal of Web Semantics 3 (1) (2005) pp. 23–40.
5. A. Gómez–Pérez, Evaluation of Ontologies, International Journal of Intelligent Systems 16 (3) (2001), pp. 391–409.
6. A. Preece, R. Shinghal, Foundation and Application of Knowledge Base Verification, International Journal of Intelligent Systems 9 (1994), pp. 683–702.
7. A. Preece, R. Shinghal, A. Batarekh, Principles and Practice in Verifying Rule-Based Systems, The Knowledge Engineering Review 7 (2) (1992), pp. 115–141.
8. S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer, Berlin, 1990.
9. N. Guarino, C. Welty, Evaluating Ontological Decisions with OntoClean, Communications of the ACM 45 (2).
10. OWL 2: Web Ontology Language, Document Overview (Second Edition), W3C Recommendation, http://www.w3.org/TR/owl2-overview/ (December 2012).
11. A. Pichler, A. Zöllner–Weber, Sharing and Debating Wittgenstein by Using an Ontology, Wittgenstein Archives at the University of Bergen, Norway, Literary and Linguistic Computing 2013.
12. A. L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, C. Wroe, OWL Pizzas: Practical Experience of Teaching OWL–DL: Common Errors & Common Patterns, in: Engineering Knowledge in the Age of the Semantic Web: 14th International Conference (EKAW), LNAI 3257, Springer, 2004, pp. 157–171.
13. J. Wielemaker, An Overview of the SWI-Prolog Programming Environment, in: Proc. of the 13th International Workshop on Logic Programming Environments (WLPE), 2003, pp. 1–16.
14. Y. Guo, Z. Pan, J. Heflin, LUBM: A Benchmark for OWL Knowledge Base Systems, Journal of Web Semantics 3 (2) (2005), pp. 158–182.
15. W3C, RDF 1.1 Turtle – W3C Recommendation, http://www.w3.org/TR/turtle/ (February 2014).
16. N. Zlatareva, Testing the Integrity of Non-Monotonic Knowledge Bases Containing Semi-Normal Defaults, in: Proc. of the 17th International Florida Artificial Intelligence Research Society Conference (FLAIRS), AAAI Press, 2004, pp. 349–354.
17. A. Zöllner–Weber, Ontologies and Logic Reasoning as Tools in Humanities, in: DHQ: Digital Humanities Quarterly 2009, Vol. 3, Nr. 4.

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 `autotool` 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 `autotool` 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, `autotool` 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) ]
```

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 formual 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 $\mathrm{SAT} \in \mathrm{NP} \cap \mathrm{coNP}$, 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

```
0 : ! a || b || c              6 : a || ! b || d
1 : a || b || c                7 : a || ! b || ! d
2 : a || ! c || ! d            8 : ! a || d
3 : ! a || ! c || ! d          9 : ! c || ! d
4 : a || c                     10 : ! b || c || ! d
5 : ! c || d                   11 : a || b || ! d
```

Problem solution domain: sequence of resolution steps, example:

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

Correctness property: for each step `s` it holds that `literal s` occurs in clause `left s`, and `! (literal s)` occurs in clause `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 { left = 4 , right = 8 , literal = a }
    entferne Literal a aus Klausel a || c      ergibt c
    entferne Literal ! a aus Klausel ! a || d  ergibt d
        neue Klausel        12 : c || d
nächster Befehl Resolve { left = 12 , right = 5 , literal = c }
    entferne Literal c aus Klausel c || d      ergibt d
    entferne Literal ! c aus Klausel ! c || d  ergibt d
        neue Klausel        13 : d
letzte abgeleitete Klausel ist Zielklausel? Nein.
```

Instance generator: is controlled by, for example,

```
Config { num_variables = 5 , 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

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

Correctness property: the sequence of steps determines a sequence of states (of the tree search), where

```
data State = State { decision_level :: Int
           , assignment :: M.Map Variable Info
           , conflict_clause :: Maybe Clause
           , formula :: CNF
           }
data Info = Info { value :: Bool, level :: Int, reason :: Reason }
data Reason = Decision | Alternate_Decision
            | Propagation { antecedent :: Clause }
```

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=l}` is allowed if clause `c` is a unit clause under the current assignment, with `l` 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 = l, 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 `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 Σ, two Σ-algebras A, B, both with finite universe U,
- solution: a term t over Σ 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
    Signatur
        { funktionen = listToFM [ ( p , 2 ) , ( z , 0 ) ]
        , relationen = listToFM [ ]
        , freie_variablen = mkSet [ ]
        }
, der in der Struktur
    A = Struktur
            { universum = mkSet [ 1 , 2 , 3 ]
            , predicates = listToFM [ ]
            , functions = 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
    B = Struktur
            { universum = mkSet [ 1 , 2 , 3 ]
            , predicates = listToFM [ ]
            , functions = 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 (p (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:

```
[ [  p ,  q ] , [ ! p , ! 0 <= +  x ]
, [ ! q ,  0 <= + 2 -1 * x ] , [  0 <= -3 +  x ] ]
```

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

```
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 assignment 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, var, restrict }` is valid if the following holds:

- `atoms` is a subset of the formula
- for each assignment from `var` to `current-domain var` 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:
    mkSet    [ x , y ]
but deduction is only allowed for hyper-edges of size up to 1
```

elements are incorrectly excluded from domain:

```
current
    Stack [ listToFM [ ( x , [ 0 ] )
             , ( y , [ 0 , 1 , 2 , 3 ] )
             , ( z , [ 0 , 1 , 2 , 3 ] ) ] ]
step
    Arc_Consistency_Deduction
        { atoms = [ P ( x , x , y ) ]
        , variable = y , 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 , 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 Fuhs, and Sibylle Schwarz for helpful comments on exercise design and implementation.

References

Apt03. Krzystztof R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

Cla11. Edmund M. Clarke. Assignment 2. `http://www.cs.cmu.edu/~emc/15414-f11/assignments/hw2.pdf`, 2011.

DLL62. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

DP60. Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

EB05. Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In Fahiem Bacchus and Toby Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

GLSW11. Hans-Gert Gräbe, Frank Loebe, Sibylle Schwarz, and Johannes Waldmann. autotool und autotool-Netzwerk. `http://www.imn.htwk-leipzig.de/~waldmann/talk/11/hds/`, 2011. HDS-Jahrestagung, TU Dresden, November 4.

Kae07. Martin Kaeser. WebCHR examples. `http://chr.informatik.uni-ulm.de/~webchr/`, 2007.

KcSFS05. Oleg Kiselyov, Chung chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In Olivier Danvy and Benjamin C. Pierce, editors, *ICFP*, pages 192–203. ACM, 2005.

KS08. Daniel Kroening and Ofer Strichman. *Decision Procedures, an Algorithmic Point of View.* Springer, 2008.

NOT06. Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *J. ACM*, 53(6):937–977, 2006.

RRW08. Mirko Rahn, Alf Richter, and Johannes Waldmann. The Leipzig autotool E-Learning/E-Testing System. `http://www.imn.htwk-leipzig.de/~waldmann/talk/08/ou08/`, 2008. Symposium on Math Tutoring, Tools and Feedback, Open Universiteit Nederland, September 19.

SDH02. Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT 2002 Competition (preliminary draft). `http://www.satcompetition.org/2002/onlinereport.pdf`, 2002.

Sim09. Helmut Simonis. Lessons learned from developing an on-line constraint programming course. `http://4c.ucc.ie/~hsimonis/lessons_abstract.pdf`, 2009. 14th Workshop on Constraint Solving and Constraint Logic Programming CSCLP 2009, Barcelona.

Sol04. Christine Solnon. An on-line course on constraint programming. `http://www.ep.liu.se/ecp/012/001/ecp012001.pdf`, 2004. First International Workshop on Teaching Logic Programming TeachLP 2004.

Tse70. G. S. Tseitin. On the complexity of derivation in propositional calculus. Leningrad Seminar on Mathematical Logic, 1970.

Wal14a. Johannes Waldmann. Skript Constraint-Programmierung. `http://www.imn.htwk-leipzig.de/~waldmann/edu/ss14/cp/folien/`, 2014. lecture slides (in German).

Wal14b. Johannes Waldmann. The Haskell OBDD Package. `https://hackage.haskell.org/package/obdd`, 2014.

Complex Certainty Factors for Rule Based Systems – Detecting Inconsistent Argumentations

Taïeb Mellouli

Department of Business Information Systems and Operations Research,
Faculty of Law and Economics, Martin Luther University Halle-Wittenberg
mellouli@wiwi.uni-halle.de

Abstract. This paper discusses the anomaly of gradually inconsistent argumentations when reasoning under uncertainty. It is argued that in several domains, uncertain knowledge modeling experts' opinions may induce inconsistencies to a certain degree in interim and final conclusions. In order to model gradual/partial inconsistency, complex certainty factors are introduced and their serial and parallel propagation within rule-based expert systems is presented. Our complex certainty factor model, representing and propagating belief and disbelief separately, sheds light on the meaning of inconsistency degrees and their persistence within argumentations under uncertainty. For the methodology capable of this separate propagation, complex certainty factors for facts are designed as two- and for rules as four-dimensional value tuples. Requiring local consistency of knowledge, we show that only two dimensions are necessary for rules, and based on this finding, deliver a simple graphical visualization suitable for expert's knowledge acquisition. Finally, we categorize gradual inconsistencies and discuss their handling.

1 Motivation: Rules, uncertainty handling, and inconsistency

Assisting experts in their decisions and actions can be performed by modeling the environment by a knowledge base and asking for entailed consequences in form of derivations for expert's expressed goals from the knowledge base. A widely used form of knowledge representation consists of facts (data) and if-then rules (production rules) for expert systems or rule-based systems. Efficient basic algorithms are known which act *either* in a forward-chaining, data-driven, bottom-up manner by applying rules from facts over derived interim results to goals (production view), *or* in a backward-chaining, goal-driven, top-down manner reducing derivations of goals to those of subgoals until reaching facts (goal/problem reduction). In case of ***certain knowledge***, a goal may admit several derivations using a collection of facts and rules and it is known that ***only one*** derivation *suffices* to show entailment from a consistent (Horn) knowledge base.

In case of ***uncertain knowledge***, the methodology of rule-based systems, logic, and logic programming cannot be transferred in a straightforward manner. In their seminal work on modeling inexact reasoning in medicine, Shortliffe and Buchanan [11] propose the use of certainty factors (CF), real numbers between -1 and 1, for facts and rules, expressing measures of increased belief (positive CF) or disbelief (negative CF) according to acquired evidence, and describe within their MYCIN diagnosis system the propagation of certainty factors for derived interim and final conclusions/goals within a forward-chaining inference framework. Besides calculating CFs for logical expressions of rule conditions and propagating CFs in rule application (serial propagation), a new issue occurs whenever several derivations exist for the same conclusion/goal, such as for the same hypothesis in medical diagnosis. Whereas such a situation is not very interesting for certain knowledge—simply taking **one** of the derivations/argumentations as a proof for a goal (with certainty)—**two** derivations for the same hypothesis, each with an uncertain belief measure out of different pieces of evidences, are regarded to constitute a situation of *incrementally acquired evidence* for the same hypothesis and would lead to a ***stronger belief*** in that hypothesis (parallel propagation).

This parallel propagation can not only be applied to two measures of increased beliefs and similarly to two measures of increased disbelief, but also to mixed belief situations where a measure of increased ***belief*** (positive CF) ***and*** a measure of increased ***disbelief*** (negative CF) are previously calculated for the ***same hypothesis*** or subgoal. This situation leads to a positive CF, if belief is of higher degree, to a negative CF, if disbelief is of higher degree, and to zero if measures of belief and disbelief are equal. The two versions of MYCIN formulas for parallel propagation do not apply to combine certain belief (+1) and certain disbelief (-1)—the case of absolute inconsistency.

This paper recognizes a deficiency in the latter kind of calculations from a modeling point of view when reasoning with experts' opinions and rules which could lead to *(degrees of) contradictions* due to *(partially) inconsistent argumentations* and derivations for goals and subgoals. We introduce *complex certainty factors* to manage these contradicting opinions leading to combined measures of increased belief and disbelief. Calculations of complex certainty factors enable to recognize conflicting subresults and propagate degrees of inconsistency until final goals and conclusions. In our opinion, the idea and visualization of the proposed complex certainty factors will throw light on the problem of gradual inconsistency within uncertainty reasoning.

The author is aware that starting with works of Heckermann and Horovitz [6] and Pierce [10], in which several anomalies in "extensional approaches" like the CF model for uncertainty reasoning are discussed and in which belief networks as an "intentional approach" based on Bayesian probabilistic inference are declared to constitute a superior model for reasoning with uncertainty, a considerable part of the AI community followed this opinion including the developers of MYCIN themselves (Heckerman and Shortliffe [7]). Extensional approaches, viewed as suffering from *modularity* together with *locality* and *detachment*, "respond only to the magnitudes of weights and not to their origins" [10] and therefore lack a proper handling of distant correlated evidences.

However, the problem of partial/gradual inconsistency addressed in this paper describes *another type of anomaly of reasoning with uncertainty* and we are not aware of a resolution of this anomaly in non- or quasi-probabilistic (extensional) or probabilistic (intentional) systems including belief networks. In Sect. 2, we emphasize the relevance of the ***inconsistency anomaly*** by considering some business applications where expert's knowledge could lead to inconsistencies. In Sect. 3, we review the MYCIN certainty factor model and discuss some general interpretation issues, such as properties of degrees of confirmation and disconfirmation. We define the notion of *local belief* *consistency* and distinguish absolute and uncertain belief/disbelief as well as absolute and partial inconsistency. The anomaly of *gradual inconsistency* is illustrated by a fictive example of experts' ratings of derivatives related to the financial crisis.

In order to model gradual inconsistency, we introduce **complex certainty factors** in Sect. 4 and present their serial and parallel propagation within a rule-based expert system in Sect. 5. In order to propagate belief and disbelief separately, more complex certainty factors for rules are necessary, still under requirements of local consistency of knowledge, they could be simplified (cp. 5.3). A simple graphical visualization for expert's knowledge acquisition follows. Detecting inconsistencies in expert's argumentations is illustrated by applying our model to the financial crisis example in subsect. 6.1.

Though our ideas to handle the anomaly of gradual inconsistency are designed using the CF model, they are applicable to other formalisms as well. Reasoning with complex certainty factors do not only sum up evaluation of a decision by a figure like certainty factor, probability or likelihood ratio, but can also evaluate distrust and skepticism (cp. 6.2) whenever different argumentations lead to partially conflicting conclusions. In 6.3, we retrospectively interpret the phenomenon of gradual inconsistency, distinguish inherent and apparent inconsistency in the course of uncertainty reasoning and show techniques to resolve recognized types of inconsistencies leading to future works (Sect. 7).

2 Expert knowledge and inconsistency in business applications

Many decision problems in business, economics, society, and politics are based on *predictive knowledge* and *expert evaluations* that are prone to hidden conflicts and inconsistencies. They represent *partial information* on cause-effect relationships with a lack of exhaustive frequency or (a-priori and conditional) probability data being a prerequisite for building belief networks. Inference systems should be able to handle experts' opinions as partial and modular knowledge about cause-effect, influence, and relevance relationships as well as selective association rules extracted by data mining techniques.

One application domain lacking complete probability data is *risk evaluation of new technologies*; only (uncertain) expert opinions about causal relationships are known concerning future consequences. Examples are relationships between greenhouse effect and global warming, between environmental contamination, damage and catastrophes, as well as effects of extensive use of mobiles and social media on children's mental growth. TV shows with debates of experts of different schools of thought often exhibit that controversial and opposite opinions may lead to *inconsistencies in argumentations*. Also, knowledge and rules of the *same* expert may sometimes induce indiscernible inconsistency within a subject. In politics, it is not rare to find "experts" who preach democracy principles and human rights but support dictatorships because of hidden economic interests. Such kind of inconsistency cannot be detected easily by TV spectators confronted with experts' opinions on complex problems such as globalization, currency devaluation, political instability, middle-east conflict and Arab spring.

Furthermore, there are some areas such as **law** and **jurisprudence** where knowledge to be applied is *normative*. Besides informative knowledge considered as descriptive, helping with conceptual understanding, normative knowledge is seen as *prescriptive* showing *how to comply* (with law). Normative orders and systems are not only relevant in jurisprudence, but also characterize scientific branches like normative economics and normative ethics. Normative knowledge includes *requirements* usually expressed using the verbal form “shall” for a necessary conclusion as a (generally formulated) judgment if a given list of conditions is fulfilled. The application of this modular knowledge in court proceedings is subject to *uncertainties* in the given *evidences* or *facts of the case*. In a criminal case, punishments may heavily differ in extent from monetary penalty to several years of prison, depending on the final refined judgment that may be a standard burglary, robbery or armed robbery. Evidences and facts of a case like “has the robber a knife, a pocket knife, etc.”, “is it considered as dangerous tool” are subject to *uncertainties*. In a course of an *analysis scheme* based on these evidences, the judge should infer *belief* about *intention* (negligent, grossly negligent, etc.). From the other hand, he examines *exculpations* (distress/emergency states) which can lead to *disbelief*. Thus, given both positive belief and disbelief in some aspects of the judgment, one is confronted with *partial inconsistency* in the concluding judgment or within an intermediate conclusion. Our method is able to *propagate these partial inconsistencies* until the concluding judgment. Only at the concluding judgment, the lawyer has to weigh pros and cons (belief and disbelief) as well as argumentations for and against, in order to finally judge the criminal case. In our opinion, *normative knowledge* is inherently *modular* and cannot deliver the necessary conditional probability tables required for belief networks. So using our complex certainty factors for modular knowledge is one method of choice.

3 Certainty factors and the inconsistency anomaly

In introducing certainty factors of facts and rules for modelling uncertain knowledge and discussing their serial and parallel propagation, we stress on several general interpretation issues: properties of belief and disbelief, difference to probability, local consistency, absolute and gradual belief and inconsistency. A motivating example concerning experts’ rating of derivatives and financial crisis illustrates the anomaly of gradual inconsistency that is shown to be improperly handled by the certainty factor model.

3.1 Certainty factors and their relationship to probabilities

A common application of uncertainty reasoning is classification and diagnosis. Some observations (symptoms, evidences) can be linked by rules to solutions (hypotheses, diagnoses, diseases). Rules are associated expert’s estimates of confirmation/disconfirmation or belief/disbelief by an (un-)certainty measure, as in a MYCIN example [11]:

IF: E1) The stain of the organism is gram positive
AND E2) The morphology of the organism is coccus
AND E3) The growth confirmation of the organism is chains
THEN: there is suggestive evidence (**CF = 0.7**)
H) that the identity of the organism is streptococcus

Generally, a **certainty factor** CF(H,E), denoted here **CF(H|E)** for convenience, is a real number in [-1…1] representing a measure of increased belief in the hypothesis H given an acquired evidence E, if it is positive, and a measure of increased disbelief in (belief against) the hypothesis H given the evidence E, if it is negative. While a certainty factor of 1 corresponds to “definitely certain” and -1 to “definitely not” or “certainly against” a hypothesis, certainty factors for linguistic utterances “weakly suggestive”, “suggestive”, and “strongly suggestive” evidence may range from 0.2 to 0.95, and for “almost certainly not”, “probably not” and “may be not” may range from -0.95 to -0.2.

A first formula for certainty factors CF(H|E) of the rule “**if** E **then** H” adopted by MYCIN in terms of a measure of (increased) belief MB(H|E) and a measure of increased disbelief MD(H|E), given an acquired evidence E, is simply the difference:

$$\mathrm{CF(H|E)} = \mathrm{MB(H|E)} - \mathrm{MD(H|E)} \tag{1}$$

Shortliffe & Buchanan [11] note that the above rule example reflects their collaborating expert’s belief that gram-positive cocci growing in chains are apt to be streptococci, where a 70% of belief in the conclusion is uttered. They noted that translated to the notation of probability, the rule with CF=0.7 seems to say P(H|E1,E2,E3) = 0.7. The expert, they say, may well agree with this, but he definitely **not agree** with the conclusion that P(¬H|E1,E2,E3) = 1 - P(H|E1,E2,E3) = 1 - 0.7 = 0.3. The expert claims, that “the three observations are evidence (to degree 0.7) in favor of the conclusion that the organism is a Streptococcus and should not be construed as evidence (to degree 0.3) against Streptococcus”. Thus, CF(¬H|E) is **not equal** 1 - CF(H|E). Accounting for this difference, Shortliffe and Buchanan [11] fix CH(H|E) = 0 for the case the hypothesis H is probabilistically independent from the evidence E, that is, for P(H|E) = P(H). In this case both MB(H|E) and MD(H|E) are equal to zero:

$$MB(H|E) = 0 \quad and \quad MD(H|E) = 0 \quad for \quad P(H|E) = P(H) \tag{2}$$

For the case the evidence E supports belief in H, P(H|E) > P(H), they define:

$$MB(H|E) = \frac{P(H|E) - P(H)}{(1) - P(H)} \quad and \quad MD(H|E) = 0 \quad for \quad P(H|E) > P(H) \tag{3}$$

By this definition, the measure of increased belief MB(H|E) can be interpreted as the ratio of increase of probability of P(H) to P(H|E) after acquiring the new evidence E relative to the possible increase distance from P(H) to 1, full certainty for H. For the case the evidence E supports disbelief in H (belief against H), P(H|E) < P(H), we get:

$$MD(H|E) = \frac{P(H) - P(H|E)}{P(H) - (0)} \quad and \quad MB(H|E) = 0 \quad for \quad P(H|E) < P(H) \tag{4}$$

Likewise MD(H|E) can be interpreted as the ratio of decrease of probability of P(H) to P(H|E) after acquiring E relative to the distance from 0, full disbelief in H, to P(H). Heckermann [5,7] multiplies denominators of (3) and (4) by the extra terms P(H|E)-0 for MB and 1-p(H|E) for MD, making the definitions symmetric in P(H) and P(H|E) and justifying parallel propagation (15). Further, when P(H) approaches 0 with P(H|E) fixed, MB(H|E) converges to P(H|E) in the original and to 1 in Heckermann’s definition. He maps the likelihood ratio $\lambda = \frac{P(E|H)}{P(E|\neg H)} \in\,]0,\infty[$ to $CF \in\,]-1,1[$ by $CF = \frac{\lambda - 1}{\lambda}$ for $\lambda \geq 1$ and $CF = \lambda - 1$ for $\lambda < 1$ and applies Bayesian inversion formulas $P(E|H) = [P(H|E) * P(E)]/P(H)$ and $P(E|\neg H) = [P(\neg H|E) * P(E)]/P(\neg H)$. We will not dwell on *probabilistic justifications of the CF model* which were already subject of many papers.

Of concern are here only *some desired properties* that remain true with these definitions of MB and MD operationalizing *degrees of confirmation and disconfirmation*:

- The measure of increased disbelief in H after acquiring evidence E is equal to the measure of belief in ¬H after acquiring evidence E and vice versa:
    - $MD(H|E) = MB(\neg H|E)$ (5)
    - $MB(H|E) = MD(\neg H|E)$ (6)
- For each rule, **not both** measures of increased belief and of increased disbelief can be positive (*local belief consistency*):
    - $MB(H|E) > 0 \rightarrow MD(H|E) = 0$ (7)
    - $MD(H|E) > 0 \rightarrow MB(H|E) = 0$ (8)

From (5) and (6) it follows according to CF definition (1) that:

$$CF(\neg H|E) = -\,CF(H|E) \tag{9}$$

Properties (7) and (8) prescribing what we call **local belief consistency** are crucial, since the same piece of evidence cannot both favor and disfavor the same hypothesis. Thus formula (1) is stated for convenience, instead of stating $CF(H|E) = MB(H|E)$, if $MB(H|E) > 0$ and $CF(H|E) = -\,MD(H|E)$, if $MD(H|E) > 0$. As Heckermann [5] states, we assume that probability and belief measures are to be understood as subjective according to the same expert with prior knowledge k about the domain. So $P(H|E)$ can be seen as $P(H|E,k)$, $P(H)$ as $P(H|k)$, $MB(H|E)$ as $MB(H|E,k)$, $MD(H|E)$ as $MD(H|E,k)$, and $CF(H|E)$ as $CF(H|E,k)$. For a fact E, $CF(E)$ can be seen as a rule's CF: $CF(E|k)$. Precisely, Heckermann denotes $CF(H|E,k)$ as $CF(H\rightarrow E, k)$ to account for the matter of fact that the expert knowledge somehow conditions the whole expert's opinion about CF of the rule and that a diagnostic rule if E then H actually models the reciprocal causality that the hypothesis/disease H causes the appearance of the evidence E.

3.2 Certainty factors of compound evidence and their serial propagation

Given an if-then-rule (R) with certainty factor CF_R

(R) **if** condition/evidence E **then** conclusion/hypothesis H (CF_R)

firstly compute the CF(E) out of CF of the members constituting the expression E and then compute $CF_R(H)$ of the conclusion by serial propagation of CFs:

1. Calculate CF(E) for E an expression using conjunction, disjunction and negation:
    - $CF(e1 \wedge e2) = $ ***and***$(CF(e1), CF(e2)) := \min(CF(e1), CF(e2))$ (10)
    - $CF(e1 \vee e2) = $ ***or***$(CF(e1), CF(e2)) := \max(CF(e1), CF(e2))$ (11)
    - $CF(\neg e) = -\,CF(e)$ (12)
2. Calculate $CF_R(H)$:
    - If $CF(E) > 0$ then $CF_R(H) = CF(E) * CF_R$ (13)
    - If $CF(E) \leq 0$ then the rule (R) is not applicable (14)

Whereas the min-function for conjunction of evidence in (10), as a possible t-norm, is adequate for e1 and e2 being completely or strongly overlapping, another t-norm $CF(e1 \wedge e2) = CF(e1)*CF(e2)$, less than $\min(CF(e1), CF(e2))$, is more adequate, if e1 and e2 are independent. We propose to attach to each rule individual variants of t-norm/t-conorm for computing CF of conjunction/disjunction of evidences according to the evidences' grade of overlapping/dependency/disjointedness (see below).

It is important to note that serial propagation do **only** apply to the case $CF(E) > 0$, or practically using a threshold, e.g. $CF(E) \geq 0.2$ as for MYCIN. Take the rule (R1) "**if** it rains **then** the grass gets wet" with certainty factor 0.9. If it rains, we can infer grass is wet with certainty factor CF_1 = 0.9. It is clear that if it doesn't rain CF(*Rain*) = -1, we **cannot** infer CF(*WetGrass*) = -1*0.9 = -0.9, since grass may be wet, for instance, because of the sprinkler being on. The asymmetry in (13) and (14) accounts for the intuition of experts working with rule-based systems, who commonly tell that the presence of evidence E increases belief in a hypothesis H, but the absence of E may have no or negligible significance on H. So for the case CF(*Rain*) = -1, we have CF(¬*Rain*) = 1 and this negated evidence is only invoked with a rule with negated evidence like "**If** it *doesn't* rain, **then** grass is not wet" that may be associated a significantly lower CF, as 0.3, depending on the expert's knowledge over other relevant causes in the domain making grass wet. This CF is nearly 0, if a sensor automatically turns the sprinkler on.

Further, knowledge engineering with certainty factors should be either causal or diagnostic in order to avoid strange feedback loops, as for the causal rule (R1) together with the diagnostic rule (R2') "**if** grass is wet, **then** sprinkler is on" with CF_2' = 0.4. Then one can infer from CF(*Rain*) = 1, that CF(*SprinklerOn*) = (1*0.9)*0.4 = 0.36. Clearly, the fact that it rains would "explain away" that the sprinkler is on, thus CF(*SprinklerOn*) should be near to zero. While inter-causal reasoning can be better handled by belief networks, the situation is better modelled by two causal rules or by one compound causal rule using disjunction: (R12) **If** *Rain* ∨ *SprinklerOn* **then** *WetGrass*. For the rule (R12), we propose to attach another t-conorm, such as CF(*R* ∨ *S*) = CF(*R*) + CF(*S*) − CF(*R*)*CF(*S*), greater than max(CF(*R*), CF(*S*)) of (11), for *R*=*Rain* being independent of *S*=*SprinklerOn* or even CF(*R* ∨ *S*) = min(1, CF(*R*) + CF(*S*)) assuming that *R* and *S* are (almost) mutually exclusive events.

3.3 Parallel CF propagation and belief substantiation of co-concluding rules

The case of parallel propagation of certainty factors applies when two rules have the same conclusion or hypothesis H (two co-concluding rules):

(R1) **if** E1 **then** H (CF_{R1})
(R2) **if** E2 **then** H (CF_{R2})

Let the certainty factors for H be: x = CF_{R1}(H) and y = CF_{R2}(H) as calculated by serial propagation of (R1) and (R2), then the resulting certainty factor for H is calculated by:

$$CF(H) = \begin{cases} x + y - x * y & for \quad x \geq 0,\ y \geq 0 & (a) \\ x + y + x * y & for \quad x \leq 0,\ y \leq 0 & (b) \\ \frac{x+y}{1-\min(|x|,|y|)} & for \quad -1 < x * y < 0 & (c) \\ undefined & for\ (x, y) \in \{(-1,1),(1,-1)\} & (d) \end{cases} \qquad (15)$$

Actually, the formulas of (15) apply to the case of more than two co-concluding rules: Simply take *x* as the result of applying (15) so far and *y* as the CF result by serial propagation of an additional rule, then combine *x* and *y* by applying (15) again. It can be shown that the application of (15) is commutative and associative. We first discuss (a) and (b), which are given in the original work of Shortliffe and Buchanan [11], then (c) and (d) in next sections. Motivated by the diagnostics domain, (15a) means that several evidences supporting the same hypothesis H substantiate suspicion **for H**. (15b) is equally motivated in case both evidences are **against** the same hypothesis **H**.

- x = 0.5, y=0.9 → CF(H) = 0.5 + 0.9 – 0.5*0.9 = 1.4 – 0.45 = 0.95
- x = -0.5, y= -0.9 → CF(H) = -0.5 – 0.9+(- 0.5)*(- 0.9) = -1.4+ 0.45 = -0.95

Formula (15b) is analogous to (15a): $x + y + x * y = -(|x| + |y| - |x| * |y|)$ for x and y being both negative ($|x|$ and $|y|$ correspond to measures of increased disbelief). In both cases, substantiation of belief (or disbelief) uses the probabilistic sum formula $a + b - a * b$, an adequate t-conorm of disjunctions for independent propositions. This is justified if distinct independent argumentation chains are available from different indications. Two ways relating parallel propagation to disjunction can be depicted. A compound rule using disjunction "if E1 ∨ E2 then H" replaces (R1) and (R2), but needs a new expert's CF estimation. A second way involving disjunction is to introduce new intermediary propositions H1 and H2 as two ways leading to H and apply rule (R1') "if E1 then H1" and (R2') "if E2 then H2", separately. Interpreting CF(H1) = CF_{R1}(H) = x and CF(H2) = CF_{R2}(H) = y as the beliefs in H regarding, in diagnostic terms, the subsets P1 and P2 of the subpopulation of patients possessing disease H who show symptoms E1 and E2, respectively, so CF(H) can be seen as the belief outcome for H related to the patient set P1∪P2. We may roughly write H = H0∨H1∨H2, where CF(H0)=0, since no belief is known for the subpopulation P0 (corresponding to H0) possessing H but not showing symptoms E1 and E2. Presuming independence of P1 and P2, the use of the probabilistic sum is justifiable. This reasoning applies to (15b) by considering ¬H instead of H and $|x|$ and $|y|$ instead of x and y as beliefs in ¬H. The independence assumptions of P1 and P2 are related to, and seem to be weaker (or equivalent) conditions for the justification of (15a) and (15b) than, the conditions of independency of E1 and E2 and their conditional independency, given H and ¬H, stated by Adams [1].

As a later appraisal for the CF model (in the new millennium) in comparison with Bayesian belief networks, Lucas ([8], Sect. 3.2-3.3, Fig. 1-2) shows that the efficiency of belief networks for large knowledge bases is due to the usage of extra structures like ***Noisy-OR*** that are shown to be *equivalently handled by formula (15a) of the CF model for co-concluding rules*. In fact similar so-called (*decomposable*) *causal independence conditions* are assumed in large practically relevant belief networks as pointed out by Lucas [8]. In order to avoid inefficiency in knowledge acquisition and processing for effect-nodes with lots of causes' parent nodes, the (very big) conditional probability tables are gathered in an implicit way out of the individual cause-effect relations (like if-then rules) and processed by formulas like (15a) of CF parallel propagation.

In case independency conditions are violated, we propose to attach other t-conorm variants to the evaluation of H, i.e. using *max*(x,y) for P1 and P2 being highly overlapping/correlated and *min*(1,x+y) for P1 and P2 being mutually exclusive (disjoint).

In this context, consider the knowledge gained from different experts: If two experts with prior knowledge k1 and k2 (evidences about the domain) assert their beliefs for the same rule, then we get something similar to CF(H|E, k1) and CF(H|E, k2). These can be seen as certainty factors for two different rules with the same conclusion H and can be handled by parallel propagation. In this case, it is convenient to assume that H1 and H2 are highly overlapping and thus CF(H) = max(CF(H|E, k1)*CF(E), CF(H|E, k2)*CF(E)). If we deal with uncertainty at a meta-level, i.e., the assertions of the experts may be themselves uncertain, then one can take the arithmetic average of the experts' beliefs concerning the same rule and assign it to one rule's CF(H|E,k1,k2).

3.4 Absolute confirmation, disconfirmation, and absolute inconsistency

Formula (15d) excludes the occurrence of "absolute inconsistency". For further discussions, let us consider the defining criteria for MB and MD postulated by Shortliffe and Buchanan [11]. Let e+ (e–) represents all confirming (all disconfirming) evidence for hypothesis H acquired to date. MB(H|e+) and MD(H|e–) increase toward 1 as confirming respectively disconfirming evidence is found and equals 1 if and only if a piece of evidence logically implies H respectively ¬H with certainty. This is achieved by (15a) and (15b), as $x + y - x * y = x + y(1 - x)$ can only be 1 if $x = 1$ or $y = 1$. For the case of ***absolute confirmation*** MB(H|e+)=1, Shortliffe and Buchanan postulate that MD(H|e–) should be set to 0 regardless of the disconfirming evidence in e–. Similarly, the case of ***absolute disconfirmation*** MD(H|e–)=1 makes all confirming evidence in e+ without value for H. They remarked that the case where (MB(H|e+)= MD(H|e–) =1 is ***absolutely inconsistent*** (contradictory) and hence the CF is undefined (15d).

The original version of (15c) was formula (1) according to previously computed MB(H|e+) and MD(H|e–) by setting CF(H|e+& e–) = MB(H|e+) – MD(H|e–). Yet, formula (15c) computes 1 when x or y is 1 and -1 when x or y is -1, as desired:

- x = 1.0, y=-0.9 → CF(H) = (1.0 - 0.9)/(1-min(|1.0|,|-0.9|) = 0.1/0.1 = 1
- x = -0.9, y=1.0 → CF(H) = (-0.9+1.0)/(1-min(|-0.9|,|1.0|) = 0.1/0.1 = 1
- x = -1.0, y=0.9 → CF(H) = (-1.0 + 0.9)/(1-min(|-1.0|,|0.9|) = -0.1/0.1 = -1

With this interpretation, we note the **discontinuum** between **{1}** for ***absolute*** and the interval **[0, 1)** for ***uncertain*** *confirmations* and *disconfirmations*. ***Certain knowledge*** is considered as knowledge of higher magnitude which ***defeats*** and ***nullifies all other uncertain knowledge***. Suppose we reason under uncertainty about the mortality likelihood of patients with some complex diseases, and for a patient we gathered evidences showing a disbelief in mortality with CF = -0.5 for the next five years. Upon knowing his death, we get a CF=1 (certainly true) that nullify our disbelief from other evidences. Another example from default reasoning: We know that all birds fly with CF=0.95 and we know that a penguin is a bird, then we can imply that a penguin may fly with high positive certainty factor. Acquiring new specific certain knowledge that a penguin can**not** fly with certainty because of heavy weight and small wings, then the resulting CF is -1 regardless of our previous uncertain belief that it is a likely flying bird.

3.5 Inconsistency in case of parallel CF propagation of mixed belief & disbelief

Considering the idea and semantics of formula (15c), we show its undesired properties as mathematical mapping and its weakness in modeling gradual inconsistency.

If evidences exist **one for** and **one against** a hypothesis, a common certainty factor CF is calculated (CF > 0, if $MB_{for} > MD_{against}$ and CF < 0 if $MB_{for} < MD_{against}$):

- x = -0.5, y=0.9 → CF(H) = (-0.5 + 0.9) / (1-min(|-0.5|,|0.9|)
  = 0.4 / (1-0.5) = 0.4 / 0.5 = 0.80
- x = 0.5, y= -0.9 → CF(H) = (0.5 + (-0.9)) / (1-min(|0.5|,|-0.9|)
  = -0.4 / (1-0.5) = -0.4 / 0.5 = -0.80

Shortliffe and Buchanan [11] firstly apply formula (1) delivering CF = MB_{for} - $MD_{against}$ = 0.9–0.5=0.4 for the first and analogously -0.4 for the second example. They enhance

(1) into formula (15c) together with van Melle [12] in the course of development of a domain-independent EMYCIN system, in order to consider that very strong belief for a hypothesis should only be slightly affected by lower ranked disbeliefs. Note that two certainty factors $CF_1 = 0.9$ and $CF_2 = 0.9$ with a combined $CF_{12} = 0.99$ would be destroyed by $CF_3 = -0.8$ to a resulting $CF_{(v1)} = 0.19$ by formula (1), whereas the formula (15c) computes $CF_{(v2)} = (0.99-0.8)/(1-0.8) = 0.19/0.2 = 0.95$. So $CF_{(v2)} = CF_{(v1)}/0.2$. The new version (15c) is a *normalization* of the difference of belief and disbelief of (1).

The formulas in (15) for CF(H) describes a function in x and y $\in$ [-1,1]. Buchanan and Duda [3] point out the following as "desired properties" of this function:

> "*When contradictory conclusions are combined (so that $x = -y$), the resulting certainty is 0. Except at the singular points (1,-1) and (-1,1), CF(H) is continuous and increases monotonically in each variable x and y.*"

The first property concerns all (x,y) in a straight line between, and excluding, the singular points (1,-1) and (-1,1). Whereas cases (1,-1) and (-1,1) are considered contradictory [11] and their CF(H) remains undefined (15d), all other situations with partially contradictory conclusions **are equally evaluated to 0**. That means, whereas for x = 1 and y = -1, an absolute contradiction is recognized, **the values x=0.999 and y= – 0.999 are evaluated to CF(H) = 0**. Even worse, we may get in ***the proximity of (1,-1) all possible values*** for CF(H). Consider x = 0.999 and y = -0,9, CF(H)=(0.999+(-0,9))/(1-min(|0.999|, |-0.9|)) = 0.099/0.1 = 0.99. This *highly contradictory situation* (0.999, -0.9) —one "strongly suggestive" opinion and one opinion "almost certainly not"—is considered *equivalent to the clear situation* x = y = 0.9 having two strongly suggestive opinions. The same problem occurs in the proximity of (-1,1). Thus, the property in the second reported sentence is to be relativized since the monotone increase in one variable is very perturbed by small changes in the other variable. Further, in the proximity of the straight line between (1,-1) and (-1,1) **small changes** may result in a **very high increase** between two extreme CF values, for instance, going from (0.999, -0.9999) to the near point (0.999, -0.99), x is fixed and y only increases by 0.0099, but the value of CF(H) increases drastically from -0.9 to +0.9 going through 0 at point (0.999, -0.999). The inability of the CF model to distinguish between **lack of evidence** $x = y = 0$ and **contradictory conclusions** $x = -y$ of different grades still remains in CF models considered to better match probability theory. For instance, the formula of parallel propagation $(x + y)/(1 + x * y)$ suggested by Heckermann [5] doesn't change the situation.

3.6 Inconsistency in argumentations – Example Financial crisis

From the above discussion, our main objection is that inconsistency may appear in conclusions and one cannot always handle the situation by a kind of summarization or calculation based on a certainty factor or other single probability figure to mirror the partial contradiction. The situation is even worse, when contradictions appear within argumentations and are not apparent in the conclusion. In order to explain the phenomenon, we introduce the fictive example for the financial crisis depicted in Figure 1: Decisions of purchasing financial products such as derivatives are based on experts' rating. Rating of a derivative D is based on ratings of A and C, the former being rated

AA+ and the latter being a mixture of a bank value papers. The situation in Figure 1 (a) shows only positive opinions about all derivatives, including C and the certainty for the composition rule for D. Based on positive certainty factors for evidences, we get a CF=0.98 for derivative C by substantiation of belief of Experts E1 and E2 by formula (15a): 0.8+0.9-0.8*0.9=0.98. Thus, a certainty factor of 0.98 results also for D, as well.

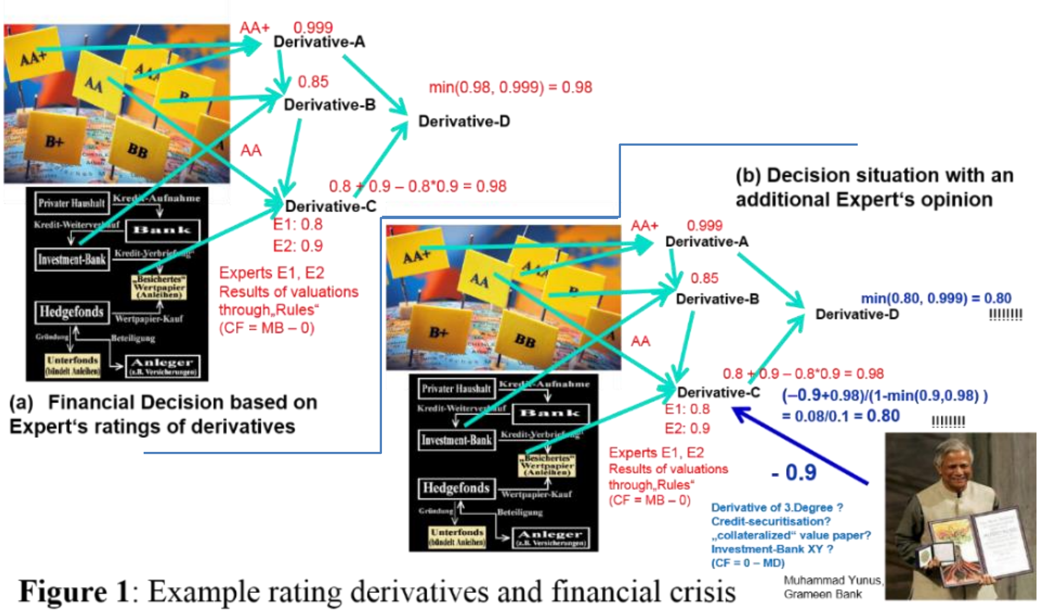


Figure 1: Example rating derivatives and financial crisis

The decision case (b) shows akin situation with a wise man giving a negative rating for derivative C, because of its high degree of composition and its connection to bank value papers of insufficiently clear origins. The wise man can be Muhammad Yunus, an economist professor awarded the nobel peace prize. He further doubts on the derivative ratings (the newspaper "*Handelszeitung*" titled on 09.12.2008 "*AAA nicht mehr das A und O*", i.e., "AAA no more the alpha and omega"). He gives a high measure of disbelief for derivative C. By using (15c) for the conclusion C merging CF12=+0.98 of Experts E1 and E2 with CF3 = -0.9 of the wise man, we get CF(C) = (0.98 + (-0.9)) / (1- min(|0.98|, |-0.9|)) = 0.08/0.1 = 0.8. This CF(C) propagates to deliver CF(D) = 0.8.

The compound certainty factor CF(C) ***obscures the reasoning situation*** at stage C within the argumentation where a *contradiction at a high grade* exits. Even worse that this ***contradiction is not apparent any more at the end*** of the argumentation, i.e. at the conclusion D, *only positive arguments are apparent without any skepticism*.

4 Complex certainty factors for reasoning with inconsistencies

The drawbacks discussed in 3.5 and 3.6 of the CF model that mixes belief and disbelief in subresults and conclusions cannot be remedied by choosing another model of uncertainty reasoning, like subjective Bayesian methods (Duda el al. (1976)) or the widely used Bayesian belief networks (Pierce (1988)). We are convinced that the evaluation by **one** real number, be it a certainty factor, an odd, a likelihood ratio, or a probability, does **not suffice** to make inconsistency visible within an argumentation. Therefore, our idea to remedy the drawbacks is to **represent** and **propagate** confirmations and disconfirmations **separately** within argumentations making possible to a disagreement, contradiction or inconsistency upon its discovery in a subresult to persist until the conclusion. To operationalize this idea, we introduce *complex certainty factors* (CCF) and by a suitable visualization stress its two-dimensionality separating belief and disbelief and making gradual inconsistency visible. The calculations for CCF are then presented for the combination of evidences/propositions by logical operators. The propagation of CCF is postponed to the next section where more complex CCF for rules are needed.

4.1 Complex certainty factors

Our requirement is that: *Distrust within an argumentation chain*
should abide incessantly till conclusion. (Req. 1)

For instance, in the financial crisis example in 3.6, although the conclusion is summarized by a positive certainty factor, disbelief in rating derivative C should be apparent in the evaluation of the conclusion (derivative D) as distrust. Also, if a conclusion would be summarized by a disbelief (negative certainty factor), distrust in form of a belief value within an argumentation should be apparent in the conclusion as well.

Our approach is based on introducing complex certainty factors (CCF) for propositions and then for rules in order to propagate belief and disbelief separately, making them apparent within argumentations till the conclusion. A CCF of a proposition (fact, subresult or conclusion) consists of two separate parts for confirmation and disconfirmation and can be written, for convenience, like a complex number:

$$\mathrm{CCF} = MB + \boldsymbol{i}\, MD \tag{16}$$

A CCF is composed of *MB* as real and *MD* as imaginary part of the complex number. The real part is called the belief/confirmation part and the imaginary part the disbelief/disconfirmation part of the CCF. Let us consider some examples of CCF:

- true (absolute confirmation): $1 + \boldsymbol{i}\,0$ $= 1$
- false (absolute disconfirmation): $0 + \boldsymbol{i}\,1$ $= \boldsymbol{i}$
- consistent belief: $0.6 + \boldsymbol{i}\,0$ $= 0.6$
- consistent disbelief: $0 + \boldsymbol{i}\,0.7$ $= \boldsymbol{i}\,0.7$
- partially inconsistent knowledge: $0.8 + \boldsymbol{i}\,0.7$ (belief & disbelief!!)
- absolutely inconsistent: $1 + \boldsymbol{i}$ (contradiction)

In the first four examples, we have either belief or disbelief and only one part (real or imaginary) is sufficient. In the last two cases, both confirmation and disconfirmation parts are positive and the resulting inconsistency is represented explicitly.

4.2 Visualization of complex certainty factors

The idea of a CCF of a proposition CCF = $MB + \boldsymbol{i}\, MD$ can be visualized in two dimensions [0,1]x[0,1] where the x-axis represents MB and y-axis MD (see Figure 2). The distinguished points are (1,0) on the MB-axis for absolute belief/confirmation (true), (0,1) for absolute disbelief (false), (0,0) for the case no information on confirmation or disconfirmation could be calculated, and (1,1) for the case of absolute contradiction.

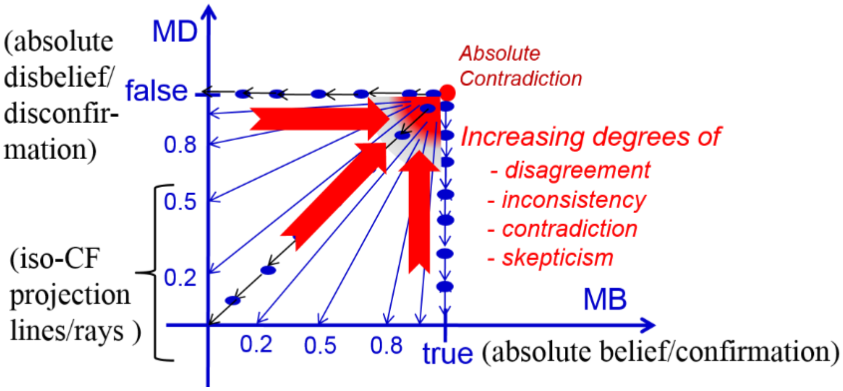


Figure 2: Visualization of complex certainty factors

The projection of the points in [0,1]x[0,1] on the MB-axis and MD-axis along the thin lines and arcs represent the certainty factor values CF, where projections points on the MB-axis are positive CFs and projection points on the MD-axis correspond to negative CFs with the same amplitude. These projection trajectories form lines like rays starting in the proximity of (1,1). Each of these projection trajectory line containing an infinity of CCF points corresponding to the same CF is called an **iso-CF line**. Whereas the certainty factor CF is the same on each iso-CF line, the grade of disagreement, inconsistency or contradiction is getting larger when going back towards the proximity of the absolute contradiction point (1,1) corresponding to CCF = $1 + \boldsymbol{i}$. For the CF model (cf. 3.4), all points (1, MD) with MD < 1 have CF=1 (certain belief nullifies partial disbelief) and all points (MB, 1) with MB < 1 have CF = -1 (certain disbelief nullifies partial belief), whereas point (1,1) of absolute contradiction has no defined CF.

The distinguished points (1,0), (0,1), (0,0) and (1,1) are the corners of the unit quadrant [0,1]x[0,1] representing the two-dimensional CCF values' range. The semantics of these four distinguished CCF corresponds to Belnap's (*useful*) four-valued logic (*for a computer how it should think*) [2]: a proposition A has truth value (1,0) iff (the computer is) just told "A is true", (0,1), iff just told "A is false", (0,0), iff neither told "A is true" nor "A is false" and (1,1), iff told both "A is true" and "A is false". Our CCF model extends Belnap's four-valued logic in case of reasoning under uncertainty, yet capturing and reasoning with both partial and absolute inconsistencies.

4.3 Calculations with complex certainty factors for compound expressions

The requirement for calculations with CCF following from (Req. 1) is as follows:

Belief and disbelief should not be admixed within CCF calculations. (Req. 2)

Beginning with operators **and**, **or** and **not** for evaluating condition parts of rules, let

$$\mathrm{CFF}(e_1) = x_1 + \boldsymbol{i}\, y_1 \qquad\qquad \mathrm{CFF}(e_2) = x_2 + \boldsymbol{i}\, y_2\,,$$

The CCF of compound propositions $e_1 \wedge e_2$, $e_1 \vee e_2$, and $\neg e_1$ are calculated as follows:

- $\mathrm{CCF}(e_1 \wedge e_2) = \boldsymbol{and}(x_1 + \boldsymbol{i}\, y_1,\ x_2 + \boldsymbol{i}\, y_2) = \mathbf{and}(x_1, x_2) + \boldsymbol{i}\ \mathbf{or}(y_1, y_2)$ (17)
- $\mathrm{CCF}(e_1 \vee e_2) = \boldsymbol{or}(x_1 + \boldsymbol{i}\, y_1,\ x_2 + \boldsymbol{i}\, y_2) = \boldsymbol{or}(x_1, x_2) + \boldsymbol{i}\ \boldsymbol{and}(y_1, y_2)$ (18)
- $\mathrm{CCF}(\neg e_1) = \boldsymbol{not}(x_1 + \boldsymbol{i}\, y_1) = y_1 + \boldsymbol{i}\, x_1$ (19)

We firstly discuss (19). For **negation NOT**, we do **neither** use the formula 1-x for probability of the complement, **nor** –x for disbelief being “negative belief” for the CF model. Rather we interchange the belief and the disbelief part in (19), since $MB(\neg e1) = MD(e1)$ and $MD(\neg e1) = MB(e1)$ following the fundamental equations (5) and (6) in 3.1. The belief and disbelief parts do not need to sum up to 1 in the CCF model.

As (17) and (18) are dual, we focus our discussion on (17). $CCF(e1 \wedge e2)$ written as **and**($CCF(e_1)$, $CCF(e_2)$) is defined through the operators on classical CF for **and**(x_1, x_2) and **or**(y_1, y_2), where general or evidence dependent t-norms and t-conorms can be used, respectively (cf. 3.2). Formula (17) incorporates de Morgan rule for “and” by using “or” for the disbelief part: As $MD(e) = MB(\neg e)$, we get $MD(e_1 \wedge e_2) = MB(\neg(e_1 \wedge e_2)) = MB(\neg e_1 \vee \neg e_2) =$ **or**($MB(\neg e_1)$, $MB(\neg e_2)$) = **or**($MD(e_1)$, $MD(e_2)$). Here, de Morgan rule, stating $\neg(e_1 \wedge e_2)$ is logically equivalent to $\neg e_1 \vee \neg e_2$, is used.

The classical CF model uses “min” as a t-norm for “and”, where the range of application is not [0,1], as for fuzzy operators, but [-1,1]. The min-function behaves as in [0,1] when combining two measures of belief. It is (incidentally) coherent for two disbelief measures, because MD is negative within the CF in this case and *min*(CF1, CF2) = *min*(-MD1, -MD2) = -*max*(MD1, MD2). This fact is usually not mentioned explicitly in presentations of the CF model. When combining a positive and a negative CF, that is, a measure of belief CF1=MB1 and a measure disbelief CF2 = -MD2, the minimum will always take -MD2 as result regardless of the intensity of belief. For instance, for MB1 = 0.9 and MD2 = 0.1, we get -0.1; likewise in the opposite case where MB1 is of lower intensity MB1 = 0.2 and MD2 = 0.4, we get -0.4. The CCF calculation for this case gives: *and*(MB1, 0+ ***i*** MD2) = *and*(MB1,0) + ***i*** *or*(0, MD2) = 0 + ***i*** MD2. Also in this case, the CCF result shows, that the MB1 disappears because of the conjunction with MB2 = 0. For these three cases having in common that they represent what we call *one-dimensional belief*, the CF result coincides with the CCF result despite of their different representation (disbelief negative or as a second dimension).

The situation changes in case of **bi-dimensional belief**. Recall that e_2 as a subresult may have a measure of belief MB2 besides MD2 and we get *and*(MB1, MB2+ ***i*** MD2) = *and*(MB1, MB2) + ***i*** MD2. That means beliefs of e_1 and of e_2 are combined into the belief part of $e_1 \wedge e_2$ and likewise the disbelief in e_2 propagates as well. Note that for the CCF conjunction, we use max-function for the evaluation of disbelief part (“or” in disbelief part in equation (17)). Thus, *max*(0, MD2) = MD2. For this special case, also other t-conorms deliver the same result *min*(1,x+y) = *min*(1, 0+MD2) = MD2 and $x + y - x*y = 0+MD2+0*MD2 = MD2$. We know that the result of CF model depends on the sign of CF2 corresponding to CCF2 = MB2 + ***i*** MD2. Following (15c), regarding x = MB2 and y = -MD2, the combined certainty factor CF2 = (x+y)/(1-*min*(|x|,|y|)=(MB2-MD2)/(1-*min*(MB2, MD2)), being positive, if MB2>MD2, negative if MB2<MD2, and 0 if MB2=MD2. For these three cases, the **CCF result** *and*(MB1, MB2) + ***i*** MD2 **differs in spirit from the CF result** *and*(MB1, CF2) using mixed belief and disbelief of CF2 and simply taking the minimum. In contrast the CCF model combines beliefs into *min*(MB1, MB2) and propagates MD2 over the disconfirmation part.

The discussion is analogous for equation (18) for calculating a CCF for disjunctions where de Morgan rule, stating $\neg(e_1 \vee e_2)$ as logically equivalent to $\neg e_1 \wedge \neg e_2$, is integrated by taking conjunction in the disbelief part of the resulting CCF.

5 Propagation of complex certainty factors

This section discusses serial and parallel propagation of CCF. We begin with discussions of some practical drawbacks of the CF models in propagating disbelief in rules and declare requirements for the CCF model in order to overcome these drawbacks. We come up with a four-dimensional CCF for rules. By exploiting local consistency of knowledge, these more complex certainty factors are shown to be reducible into two types with only two dimensions, respectively. A visualization of these two types provides an easy-to-use graphical tool for expert's knowledge acquisition.

5.1 Disbelief propagation in CF model and requirements for CCF model

Let us consider the practical difficulties in *systematically propagating disbelief* in the classical certainty factor model by considering some illustrating examples with given CF for rules and evidences:

(R1) **If** E **then** H (CF –0.8)
E (CF +0.5) → CF(H) = +0.5*(-0.8) = – 0.4

(R2) **If** E **then** H (CF 0.8) ***Rule is not applicable***
E **(CF –0.5) -/-** -/-→ CF(H) = – 0.5*0.8 = – 0.4 **(false!!)**

Rule (R2) cannot be invoked, because of negative CF of evidence. In order to propagate disbelief in E in the second example, another rule is necessary:

(R3) **If** ¬E **then** H (CF 0.6) → CF(¬E) = – (– 0.5) = 0.5
E (CF –0.5) CF(H) = 0.5*0.6 = 0.3

Thus, disbelief can only be propagated in the CF model, if an additional rule with ¬E in the premise is declared. Disbelief in H from disbelief in E can be propagated by:

(R4) **If** ¬E **then** H (CF –0.6) → CF(¬E) = – (– 0.5) = 0.5
E (CF –0.5) CF(H) = 0.5*(– 0.6) = –0.3

Because this kind of Rules (R3) or (R4) are not well-kept in expert systems besides (R1) or (R2), the MD (also if it predominates MB) is not further propagated in a „positive" argumentation chain. Let us consider the case that the expert always defines both rules; (R) with positive evidence and (dR) with negative evidence condition:

(R) **If** E **then** H CF = MB – MD
with MB = MB(H|E) and MD = MD(H|E)

(dR) **If** ¬E **then** H dCF = dMB – dMD
with dMB = MB(H|¬E) and dMD = MD(H|¬E)

Here, dMB and dMD are the measures of (increased) belief and disbelief in H under disbelief in E. While rule (R) propagates belief in evidence E, (dR) propagates disbelief in the evidence E. Although the rules of the first kind can generate disbelief as in the example (R1), this disbelief can be further propagated **only** by a rule of type (dR). Considering H as an intermediate result, CF(H) = -0.4 after applying (R1) can be further propagated only by a rule of the form (dR1) **if** ¬H **then** K. If the CF of (dR1) is positive an increased belief in K is propagated and if it is negative, disbelief in K results.

Requirement for the CCF of a rule (Req. 3)**:** Since the complex certainty factor of evidence CCF(E) = b + ***i*** d contains both MB and MD, *the CCF of a rule must contain belief and disbelief of the certainty factors of both rules (R) and (dR)*, in order that MB and MD of propositions could be further propagated simultaneously.

5.2 Complex certainty factors for rules and their serial propagation

To fulfill requirement (Req. 3), we define the complex certainty factor CCF(R) for a rule (R) **If** E **then** H
as CCF(R) **:= tt** *MB* + **tf** *MD* + **ft** *dMB* + **ff** *dMD* (20)

The four dimensions of the CCF(R) mean:

- **tt** *MB* belief in **t**ruth of evidence E results in belief of **t**ruth of hypothesis H
- + **tf** *MD* belief in **t**ruth of the evidence results in belief in **f**alsehood of H
- + **ft** *dMB* belief in **f**alsehood of the evidence results in belief in **t**ruth of H
- + **ff** *dMD* belief in **f**alsehood of evidence results in belief in **f**alsehood of H

Given this more complex certainty factors for rules, two interesting questions arise:

- How to calculate therewith? (serial/parallel propagation of CFF)
- How can CCF‘s of rules be simplified in order to be more accessible to experts?

Let us begin with **serial propagation**. Let be given

(R) **If** E **then** H with CCF(R) := **tt** *MB* + **tf** *MD* + **ft** *dMB* + **ff** *dMD*
and CCF(E) = *b* + ***i*** *d*

Then the CCF(H) is calculated as follows:

CCF(H) = CCF(E) * CCF(R) (special CCF multiplication)
:= ***or***(*b*MB, d*dMB*) + ***i*** ***or***(*b*MD, d*dMD*) (21)

Here ***or***(*b*MB, d*dMB*) represents the belief part in the hypothesis H resulting from parallel propagation of *b* (belief in E) multiplied by *MB* (**tt** part of rule's CCF) and of *d*, (disbelief in E) multiplied by *dMB* (**ft** part). Here both **tt** part and **ft** part yield belief in **t**ruth of H. Similarly, ***or***(*b*MD, d*dMD*) represents the disbelief part in H resulting from the parallel propagation of *b* (belief in E) multiplied by *MD* (**tf** part of rule's CCF) and of *d* (disbelief in E) multiplied by *dMD* (**ff** part). Here both **tf** part and **ff** part yield belief in **f**alsehood of H. As will be clear in 5.3, only one element of each ***or***-expression can be positive (***or***(x,0) = ***or***(0,x) = x). Let us consider some examples:

- CCF(R) = **tt** 0.9 + **ff** 0.4 and CCF(E) = 0.7 + ***i*** 0.3
  → CCF(H) = ***or***(0.7*0.9, 0) + ***i*** ***or***(0, 0.3*0.4) = 0.63 + ***i*** 0.12
- CCF(R) = **tf** 0.9 + **ft** 0.4 and CCF(E) = **0.7** + ***i*** 0.3
  → CCF(H) = ***or***(0, 0.3*0.4) + ***i*** ***or***(**0.7***0.9, 0) = 0.12 + ***i*** 0.63

5.3 Simplification of rules' CCF and graphical interpretation

Why have we only put two components within a rule's CCF in the above two examples? The answer is that else we would have local inconsistency of knowledge (see below). And the good news is that only two types of CCF exist for locally consistent rules:

- Type 1: MB > 0 and dMD > 0 CCF(R) = **tt** MB + **ff** dMD as the first example
- Type 2: MD > 0 and dMB > 0, CCF(R) = **tf** MD + **ft** dMB as second example

Requirement of local consistency of knowledge (Req. 4)**:** *Expert belief according to an if-then-rule should be (at least locally for this rule) consistent*:

1. From definition of local consistency of belief in (7) and (8), we have:
   $MB > 0 \rightarrow MD = 0$ and $MD > 0 \rightarrow MB = 0$ and
   analogously: $dMB > 0 \rightarrow dMD = 0$ and $dMD > 0 \rightarrow dMB = 0$ **(22)**
2. We show, that additionally:
   $MB > 0 \leftrightarrow dMD > 0$ as $P(H|E)>P(H) \leftrightarrow P(H)>P(H|\neg E)$ **(23)**
   and $MD > 0 \leftrightarrow dMB > 0$ as $P(H|E)<P(H) \leftrightarrow P(H)<P(H|\neg E)$

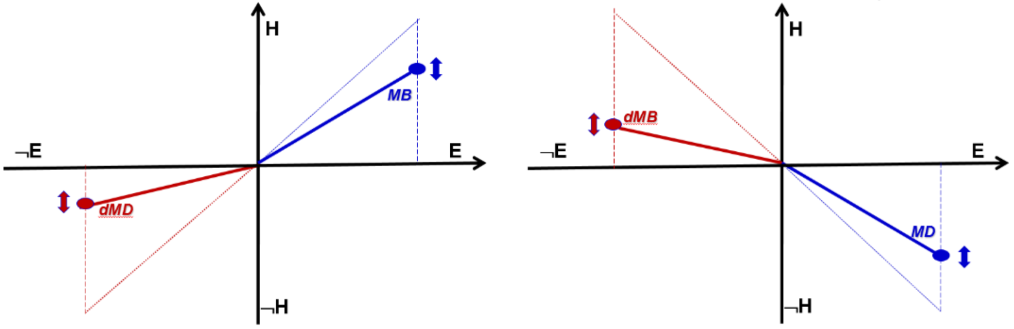
From (22) it follows that only two positive components (> 0) occurs in a CCF for each consistent rule. And from (23), it follows that ***only the above two types 1 and 2*** of the four combinations of two components are possible. Let us give a ***short proof for*** **(23):**

> Applying **Bayes rule:** $P(E|H) = [P(H|E)*P(E)]/P(H) = [P(H|E)/P(H)]*P(E) > P(E)$ for the case $P(H|E) > P(H)$, i.e. $P(H|E)/P(H) > 1$. Using **total probability principle** $P(\neg E|H) + P(E|H) = 1$, we deduce $1- P(\neg E|H) > P(E)$ and thus $P(\neg E|H) < 1-P(E)$, i.e., $P(\neg E|H) < P(\neg E)$ or $P(\neg E|H)/P(\neg E) < 1$. Applying **Bayes rule** again $P(H|\neg E) = [P(\neg E|H)*P(H)]/P(\neg E) = [P(\neg E|H)/P(\neg E)]* P(H)$, we deduce $P(H|\neg E) < P(H)$.

Is it possible to have only one positive component as in the pure logical implication under certainty? Under uncertainty, experts sometimes think that a consequence rule (R) **if** E **then** H with $MB > 0$ does **only** apply, if positive belief in E is present, and that disbelief in E doesn't mean anything for H. The proven equivalences (23) tell us that *dMD* must also be positive, that is, disbelief in H should in this case be positive. However, *dMD* doesn't have to be of same intensity as the measure of belief *MB* (cf. example in 3.2: (R1) **if** Rain **then** Wet (*MB*=0.9) but (dR1) **if** ¬Rain **then** Wet (*dMD* =0.3)).

5.4 Visualization of rules' CCF for Expert knowledge acquisition

Having reduced rule's CCF to only two types each with a special structure with only two positive components, we are able to visualize CCF for rules as in Figure 3.



(a) Type 1: CCF(R) = **tt** *MB* + **ff** *dMD* (b) **Type 2:** CCF(R) = **tf** *MD* + **ft** *dMB*

Figure 3: Visualization of complex certainty factors for rules

The visualization shows that the two positive components of a rule's CCF are always *in diagonal quadrants*, **either** Q(*E*,*H*) and Q(¬*E*,¬*H*) **or** Q(*E*,¬*H*) and Q(¬*E*,*H*). A positive measure of belief *MB* = MB(*H*|*E*) for type 1 CCF in quadrant Q(*E*,*H*) must be accompanied by a positive *dMD* = MD(H|¬*E*) = MB(¬H|¬*E*) in quadrant Q(¬*E*,¬*H*). Likewise a positive measure of disbelief *MD* = MD(*H*|*E*) = MB(¬H|*E*) for type 2 CCF in quadrant Q(*E*,¬*H*) must be accompanied by a positive *dMB* = MB(H|¬*E*) in quadrant

$Q(\neg E,H)$. The expert has only to adjust the intensity of belief and disbelief on diagonal quadrants graphically by moving the respective points (Figure 3, small double arrows). The two intensities are generally not equal, further one of them may be certain and the other uncertain, e.g., the rule **if** Pregnant **then** Women has $MB = 1$ and $dMD < 1$ because a non-pregnant human being can be a man ($\neg$Women) or a non-pregnant women.

5.5 Parallel propagation of complex certainty factors

As for the CF model, parallel CCF propagation apply when two rules have the same conclusion or hypothesis H (two co-concluding rules):

(R1) **if** E1 **then** H (CCF_{R1})
(R2) **if** E2 **then** H (CCF_{R2})

Let the certainty factors for H be: $\mathbf{x_1} + \boldsymbol{i}\,\mathbf{y_1} = CCF_{R1}(H)$ and $\mathbf{x_2} + \boldsymbol{i}\,\mathbf{y_2} = CF_{R2}(H)$ as calculated by serial propagation (21) applied to (R1) and (R2), respectively, then the resulting complex certainty factor for H is calculated by the following formulas:

$$CCF(H) = (x_1 + x_2 - x_1{*}x_2) + \boldsymbol{i}\,(y_1 + y_2 - y_1{*}y_2) \qquad (24)$$

Not only the belief part $(x_1 + x_2 - x_1{*}x_2)$ of CCF(H) substantiates both belief values of $CCF_{R1}(H)$ and $CCF_{R2}(H)$ as in (15a), but also the disbelief part $(y_1 + y_2 - y_1{*}y_2)$ substantiates both disbelief values of $CCF_{R1}(H)$ and $CCF_{R2}(H)$, too. Endorsed beliefs and endorsed disbeliefs remain separated from each other and **not combined** unlike the CF model using (15c). Having $CF_{R1}(H)= -0.5$ and $CF_{R2}(H)=0.9$ results into $CF(H) = 0.4/0.5 = 0.80$ using (15c). This corresponds to $CCF_{R1}(H) = \boldsymbol{i}\,0.5$ and $CCF_{R2}(H) = 0.9$ resulting into $CCF(H) = 0.9 + \boldsymbol{i}\,0.5$ by (24). Disbelief in H from applying (R1) remains apparent in CCF(H), unlike in the pure positive CF(H). Further, the case (15d) having an *undefined certainty factor* corresponds to $CCF_{R1}(H) = 1$ and $CCF_{R2}(H) = \boldsymbol{i}$ with the defined result $CCF(H) = 1 + \boldsymbol{i}$ signalizing an absolute inconsistency derived for H.

As discussed in 3.3, co-concluding rules can be seen as disjunctive parts for H in case of two beliefs (15a) and for $\neg$H in case of two disbeliefs in H, i.e. beliefs in $\neg$H (15b). In case of co-concluding rules, the CCF model calculate CF **simultaneously** for these **two disjunctions** and let them **separated** in the confirmation and disconfirmation part of the CCF result. Note the difference to evaluating **one disjunction** in expressions of evidences with both belief and disbelief parts in (18) where conjunction is used in the disbelief part, implicitly applying de Morgan rule. This explains the divergence of (18) and (24) in the disconfirmation/disbelief part. As discussed in 3.3 we may vary the t-conorm used for co-concluding rules according to their grade of dependency in direction to the case of overlapping ($max(x_1,x_2)$) or disjointedness ($min(1, x_1+x_2)$).

6 Application and interpretation of gradual inconsistency

The CCF model is applied to the financial crisis example of 3.6 and shown to correctly propagate disbelief and distrust in argumentations and to detect gradual inconsistency. We introduce a skepticism factor which together with the standard CF can better reflect uncertainty in derived conclusions. Finally, we interpret and classify gradual inconsistency into 2 types, namely inherent inconsistency and apparent inconsistency.

6.1 Financial crisis revisited – applying the CCF model

Let us consider the financial crisis example of section 3.6 (Figure 1) again and apply the propagation of complex certainty factors in the following Figure 4:

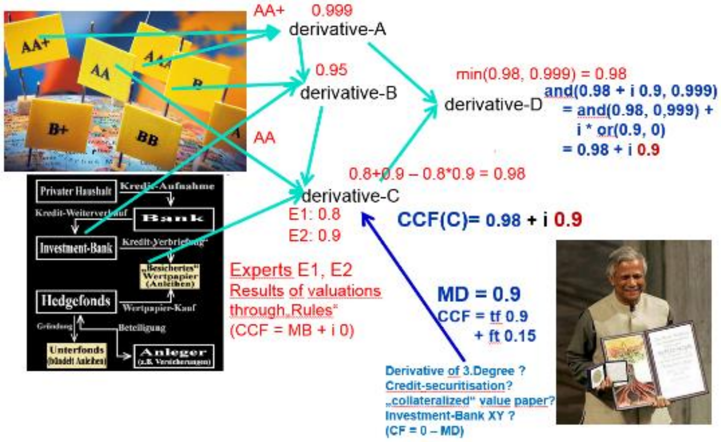


Figure 4: Applying CCF model to the financial crisis example

The disbelief of the wise man in rating of derivative C is now modeled by $CCF_{R2}(C)$ = ***i*** 0.9 resulting from serial propagation (cf. (20) and (21) in 5.2) using a rule of type 2: CCF(R2) = **<u>tf</u>** 0.9 + **<u>ft</u>** 0.15 and CF(E) = 1. Having $CCF_{R1}(C)$ = 0.98 from the ratings of experts E1 and E2, we get by applying parallel propagation (cf. (24) in 5.5) the two-dimensional complex certainty factor CCF(C) = 0.98 + ***i*** 0.9 for derivative C. The CCF of derivative D being defined as conjunctive composition of A and C is then calculated by CCF(D) = ***and***(CCF(C), CCF(A)) = ***and***(0.98 + ***i*** 0.9, 0.999) = ***and***(0.98, 0.999) + ***i*** ***or***(0.9, 0) = 0.98 + ***i*** 0.9 (using (17) incorporating de Morgan in 4.3). The rating of derivative D was 0.98 without advice of the wise man and 0.80 with his advice using the CF model. But the situation changes when applying the CCF model, since **disbelief of the wise man is now apparent** in the disconfirmation part (***i*** 0.9) of CCF(D) being MD = 0.9. This MD in CCF(D) = 0.98 + ***i*** 0.9 stands opposed to the propagated opinion of the two experts of MB = 0.98 represented by the belief part of the same CCF(D). The situation can be described as very skeptical and the grade of inconsistency is very high, as it is near to the point of absolute contradiction 1 + ***i*** (cf. Figure 2 in 4.2).

6.2 Skepticism factor

A CCF(H)=MB + ***i*** MD for a conclusion in causal reasoning or a hypothesis in diagnostic reasoning can easily be translated to a common certainty factor as in (15) by:

$$CF(H) = \frac{MB - MD}{1 - \min(MB, MD)} \qquad for\ (MB, MD) \neq (1,1) \qquad (25)$$

As this CF obscures the partial inconsistency, we associate another factor reflecting the skepticism in CF(H). This **skepticism factor** SF(H) can be defined as follows:

$$SF(H) = \frac{\min(MB, MD)}{\max(MB, MD)} \qquad for\ (MB, MD) \neq (0,0) \qquad (26)$$

As max(*MB*,*MD*) signalizes the amplitude of belief/disbelief in direction or sign of CF, the skepticism is then the ratio of the amplitude of the disbelief/belief against the CF, i.e. min(*MB*,*MD*) with respect to the former amplitude max(*MB*,*MD*). Applying (25) and (26) to our previous result CCF(D) = 0.98 + i 0.9, we get CF(D) = (0.98 – 0 .9) /(1 – min(0.98,0.9)) = 0.08/(1 – 0,9) = 0.08/0.1 = 0.8, i.e. 80% certainty in belief, but with SF(D) = min(0.98,0.9)/max(0.98,0.9) = 0.9/0.98 = 0.92, i.e. 92% of skepticism.

For points near to the diagonal (MB almost equal MD), formula (26) calculates a SF of almost 1 where CF being near to 0 (all point on the diagonal iso-CF line have CF=0, cf. figure 2). To account for the amplitude of CCF, i.e. distinguishing cases 0.2 + i 0.2 and 0.9 + i 0.9, we may integrate avg(MB,MD)=(MB+MD)/2 in (26). Also, the product MB*MD, being a t-norm for independent propositions is 1 only for absolute inconsistency, but is very small for small MB and MD values (0.04 for 0.2+$\boldsymbol{i}$ 0.2). If setting SF= MB*MD, so for a fixed SF0, the formula corresponds to an iso-SF hyperbole MD = SF0/MB crossing iso-CF lines of Figure 2. As SF is meant to be an indicator of skepticism, it can be designed as a weighted sum mixture (for $(MB, MD) \neq (0,0), \propto_i \geq 0$):

$$SF(H) = \propto_1 \frac{\min(MB, MD)}{\max(MB, MD)} + \propto_2 \frac{MB + MD}{2} + \propto_3 MB * MD \qquad with\ \propto_1 + \propto_2 + \propto_3 = 1 \qquad (27)$$

6.3 Interpretations of gradual inconsistency

After presenting the CCF model aiming at detecting absolute and partial inconsistencies, we now ask retrospectively what are the possible interpretations of a two-dimensional complex certainty factor CCF = *MB* + ***i*** *MD* with both strictly positive confirmation *MB* and disconfirmation *MD* parts? We principally discovered two types:

- Type 1: **Inherent inconsistency**: This type can either be a case of
  - absolute inconsistency of a knowledge base, or else a case of
  - inherent partial inconsistency due to opposite opinions of several experts or (partially) self-contradictory knowledge of the same expert
- Type 2: **Apparent inconsistency**: Partial inconsistencies of this type can be *resolved* in belief or disbelief *by* acquiring *more specific information* (!)

Let us discuss possible situations for the first type of **inherent inconsistency**. As shown by the financial crisis example, a typical interpretation of gradual inconsistency is that of *opposite opinions* of different experts. As discussed in Sect. 2 another kind of inherent inconsistency emerges when rules of the same expert lead to contradictory conclusions (*self-contradictory knowledge*). These types may accentuate to a case of *globally inconsistent knowledge* in a logical sense: For instance, the absolute certain facts A and B together with the locally consistent rules (R1) **if** A **then** C with MB_1 = 1 and (R2) **if** B **then** C with MD_2 = 1 lead to a (global) logical contradiction: CCF(C) = 1+$\boldsymbol{i}$. Self-inconsistent knowledge of "experts" can also arise, e.g. in politics, when consciously using *vague* or *fuzzy notions* like "fight on terrorism" without clear definition and hiding some knowledge such as "suspicious economic interest". Thus, *using unspecified / vague / fuzzy notions* and *uncovering expert's hidden knowledge* as a special case of *detecting implicit knowledge* could interpret partial/absolute inherent inconsistency.

Now, we discuss the second type of ***resolvable*** **apparent inconsistency**. Partial inconsistency can emerge in case of *missing specific information*, e.g., due to predictive knowledge about future events. Knowing no more specific information about an animal than being a bird leads to a high MB for “Flying”, e.g., CF = CCF = 0.95. Upon knowing that the bird is a penguin with heavy weight and small wings MD(Flying) = 1, the CCF becomes by parallel propagation CCF = 0.95 + ***i***. At a first glance, this example from default reasoning represents a situation of high degree of inconsistency. Following the interpretation of [11] (cf. 3.4), distrust is nullified upon knowing absolutely certain belief or disbelief (+ ***i***). For this, the knowledge should be consistent (not only locally).

Generally, a two-dimensional CCF with high skepticism should trigger further analysis of the situation, in order to interpret this skepticism into a type of opposite opinions or any type of contradictions, or else a case of *ambiguity, incertitude or ignorance* because of *lack of specific or complete information*. Let us only know that an animal is a bird with heavy weight. Against a high measure of belief applying the rule that “almost all birds fly” MB(Flying) = 0.95, we may deduce from another rule, that “animals with heavy weights are very likely flightless”, a high measure of disbelief in “Flying”, e.g, MD(Flying) = 0.99. Thus, the resulting CCF = 0.95 + ***i*** 0.99 indicates that the animal at hand is a flightless bird, but with high skepticism. The interpretation of this inconsistency of high degree can only be resolved, upon knowing more specific information about the examined animals (in general, objects or object subclasses), e.g., whether the bird has **small or large wings**. In the first case, like the example of a **penguin,** it is definitely **flightless** and in the second case, it may **fly** like the example of a **pelican** in spite of its heavy weight which may attain 15kg (length of more than 1.80m, cf. English Wikipedia entry “list of largest birds”). Therefore, the skepticism gives rise to goal-driven acquisition of more specific knowledge about the object instance and object subclasses as well as about probabilistic and causal relationships between them.

Acquiring *more specific knowledge* helps in resolving partial inconsistency in a process of *disambiguation or mitigation of uncertainties*. These aspects are related to the phenomenon of **missing explanatory attributes**, variables, or propositions, known in decision theory. Acquiring more specific knowledge may be performed by observations of the examined objects (e.g., birds, patients) or learning more about other attributes of examined objects (wings, symptoms). Data mining techniques may help finding relationships to (missing) attributes or properties of examined objects and object classes.

If no specific knowledge is available, a disambiguation can be represented by means of a **case analysis** on some not sufficiently specified attributes (this is a crucial point further discussed in the concluding remarks). For the latter example, the answer could be “**if** the heavy weight bird has large wings, **then** it is likely to fly **else** it is flightless”. This **case analysis** recognizes **cases with stronger / certain beliefs** or **disbeliefs**.

The remaining question is now: How to distinguish between inherent inconsistency and apparent inconsistency (Type 1 and Type 2 of gradual inconsistency)? The former type leads to genuine contradictions and the latter is resolvable upon knowing sufficient specific information. For the distinction, one may reason under what is known in decision theory as **perfect information** (that is usually absent in uncertainty reasoning). If the inconsistency persists, then it is of type 1, i.e. an inherent inconsistency. If the inconsistency could be resolved under perfect information, it is likely of type 2. If all cases of possible perfect information are sketched (all possible worlds), then we have exactly a resolvable apparent inconsistency of type 2 and the knowledge base is likely to be consistent, at least in the subset of knowledge concerning the derivations.

7 Conclusion and future work

In this paper, we disclosed the phenomenon of gradual inconsistencies encountered in knowledge processing with uncertainty. We provided new formalism and inference tools based on complex certainty factors for rule-based systems capable of detecting inconsistencies in argumentations and of propagating them until final derivations. Our two-dimensional CCF for facts help visualizing grades of inconsistency and our four-dimensional CCF for rules, reduced into two types each with only two dimensions, suggest that derivation of goals should simultaneously consider belief and disbelief.

Our interpretation and classification of gradual inconsistencies in inherent and apparent inconsistencies stress the issue of reasoning under *incomplete knowledge* and the usefulness of *case analysis* for the resolution of inconsistencies. We are now extending the CCF rule-based approach to a *non-Horn environment* of reasoning where *case analysis inference* is embedded *in consequence chains*. This type of inference, presented in our earlier work [9] for disjunctive logic programming and for two-, three- and four-valued logic, is capable of *nested case analysis inference* for goals and subgoals and of message-passing of assumptions in argumentation chains for *cases* (enabling conditioning and summation as in belief networks). Integrating the CCF methodology, we then could offer a competitive inference system under uncertainty—without anomalies.

References

1. Adams, J.B.: A probability model of medical reasoning and the MYCIN model. Mathematical Biosciences, 32, pp. 177-186 (1976)
2. Belnap, N.D.: A useful four-valued logic. In: Dunn, J.M. and Epstein, G. (eds.) Modern use of multiple-valued logic, pp. 8-37. Reidel Dordrecht, Holland (1977)
3. Buchanan, B.G. and Duda, R.O.: Principles of Rule-Based Expert Systems. Technical report STAN-CS-82-926, Dept. computer science, Stanford University (1982)
4. Duda, R. O., Hart, P. E., and Nilsson, N. J.: Subjective Bayesian Methods for Rule-Based Inference Systems. AI Center, Tech. Note 134, Stanford Res. Institute. California (1976)
5. Heckermann, D.E.: Probabilistic interpretations for MYCIN's certainty factors. In: Kanal, L. and Lemmer, J. (eds.) Uncertainty in AI, pp. 67-196. North-Holland, New York (1986)
6. Heckermann, D.E. and Horovitz: The myth of modularity in rule-based systems. In Kanal, L. and Lemmer, J. (eds.) Uncertainty in AI 2, pp. 23-34. North-Holland, New York (1988)
7. Heckermann, D.E. and Shortliffe, E. H.: From Certainty Factors to Belief Networks, Artificial Intelligence in Medicine, Volume 4, Issue 1, pp. 35-52 (1986)
8. Lucas, P.J.F.: Certainty-factor-like structures in Bayesian networks. Knowledge-Based Systems 14, pp. 327-335 (2001)
9. Mellouli, T.: TMPR: A Tree-Structured Modified Problem Reduction Proof Procedure and Its Extension to Three-Valued Logic. Journal of Automated Reasoning 12, pp. 47-87 (1994)
10. Pierce, J.: Probabilistic Reasoning in Intelligent systems: Networks of Plausible Inference. Morgan Kaufmann Publishers, Inc., San Francisco, California (1988)
11. Shortliffe, E.H., and Buchanan, B.G.: A Model of Inexact Reasoning in Medicine. Mathematical Biosciences 23, pp. 351-379 (1975)
12. van Melle, W.: A domain independent system that aids in constructing knowledge base consultation programs". PhD thesis. Stanford university, STAN-CS-80-820 (1980)

Part II

23rd International Workshop on Functional and (Constraint) Logic Programming

Declarative Multi-paradigm Programming

Michael Hanus

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

Abstract. This tutorial provides an overview and introduction to declarative programming exploiting multiple paradigms, in particular, functional, logic, and constraint programming. To demonstrate the possibility to support these paradigms within a single programming model, we survey the features of the declarative multi-paradigm language Curry.

1 Overview

Compared to traditional imperative languages, declarative programming languages provide a higher and more abstract level of programming that leads to reliable and maintainable programs. However, there is no distinct "declarative programming" paradigm. Instead, there are various programming paradigms and related languages based on different methods to structure declarative knowledge. *Functional programming* is based on the lambda calculus and provide functions as computational entities. *Logic programming* is based on first-order predicate logic and uses predicates as basic programming entities. *Constraint programming* offers constraint solvers to reason about models described with the help of various constraint structures. Although the motivation to exploit high-level programming is similar in all paradigms, the concrete languages associated to them are quite different. Thus, it is a natural idea to combine these worlds of programming into a single paradigm, and attempts for doing so have a long history. However, the interactions between functional and logic programming features are complex in detail so that the concrete design of such declarative multi-paradigm languages is a non-trivial task. This is demonstrated by a lot of research work on the semantics, operational principles, and implementation of functional logic languages since more than two decades. Fortunately, recent advances in the foundation and implementation of functional logic languages have shown reasonable principles that lead to the design of practically applicable programming languages.

This tutorial provides an overview on the principles of integrated functional logic languages. As a concrete programming language, we survey the declarative multi-paradigm language Curry[1] [13, 20]. It is developed by an international initiative of researchers in this area and intended to provide a common platform for research, teaching, and application of integrated functional logic languages.

---

[1] `http://www.curry-language.org`

Details about functional logic programming and Curry can be found in recent surveys [5, 18] and in the language report [20].

The integration of functional and logic programming has various advantages. Beyond the fact that one can use the best features of declarative languages in a single language, like strong typing, higher-order functions, optimal (lazy) evaluation from functional programming, or non-determinism, computing with partial information, and constraint solving from logic programming, there are also clear advantages compared to the individual paradigms. For instance, the combination of lazy evaluation and non-determinism leads to a demand-driven exploration of the search space which is sometimes more efficient and optimal for particular classes of programs [2]. Moreover, non-declarative features, which are regularly used in practical logic programs, can be avoided in functional logic languages, e.g., by functional notation or declarative I/O [22].

The combined features offered by functional logic languages led to new design patterns [3, 6], better abstractions for application programming (e.g., programming with databases [7, 11], GUI programming [14], web programming [15, 16, 19], string parsing [10]), and new techniques to implement programming tools, like partial evaluators [1] or test case generators [12, 21]. In particular, functional patterns, as proposed in [4], exploit non-determinism from logic programming and demand-driven pattern matching from functional programming in order to achieve a powerful executable specification method. For instance, functional patterns have been used for XML processing [17] where it has been shown that specialized logic programming approaches [8, 9] can be implemented with a few lines of code in Curry. Some of these techniques are reviewed in this tutorial.

References

1. M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
2. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.
3. S. Antoy and M. Hanus. Functional logic design patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pages 67–87. Springer LNCS 2441, 2002.
4. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
5. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
6. S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.
7. B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pages 316–332. Springer LNCS 4902, 2008.

8. F. Bry and S. Schaffert. A gentle introduction to Xcerpt, a rule-based query and transformation language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, 2002.
9. F. Bry and S. Schaffert. Towards a declarative query and transformation language for XML and semistructured data: Simulation unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, pages 255–270. Springer LNCS 2401, 2002.
10. R. Caballero and F.J. López-Fraguas. A functional-logic perspective of parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pages 85–99. Springer LNCS 1722, 1999.
11. S. Fischer. A functional logic database library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pages 54–59. ACM Press, 2005.
12. S. Fischer and H. Kuchen. Systematic generation of glass-box test cases for functional logic programs. In *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'07)*, pages 63–74. ACM Press, 2007.
13. M. Hanus. A unified computation model for functional and logic programming. In *Proc. of the 24th ACM Symposium on Principles of Programming Languages (Paris)*, pages 80–93, 1997.
14. M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.
15. M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.
16. M. Hanus. Type-oriented construction of web user interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pages 27–38. ACM Press, 2006.
17. M. Hanus. Declarative processing of semistructured web data. In *Technical Communications of the 27th International Conference on Logic Programming*, volume 11, pages 198–208. Leibniz International Proceedings in Informatics (LIPIcs), 2011.
18. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
19. M. Hanus and S. Koschnicke. An ER-based framework for declarative web programming. *Theory and Practice of Logic Programming*, 14(3):269–291, 2014.
20. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
21. C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proc. of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM Press, 2008.
22. P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

Interpreting XPath by Iterative Pattern Matching with Paisley

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

[1] Ilmenau University of Technology
[2] `<semantics/>` GmbH

Abstract. The Paisley architecture is a light-weight EDSL for non-deterministic pattern matching. It automates the querying of arbitrary object-oriented data models in a general-purpose programming language, using API, libraries and simple programming patterns in a portable and non-invasive way. The core of Paisley has been applied to real-world applications. Here we discuss the extension of Paisley by *pattern iteration*, which adds a Kleene algebra of pattern function composition to the unrestricted use of the imperative host language, thus forming a hybrid object-oriented–functional–logic framework. We subject it to a classical practical problem and established benchmarks: the node-set fragment of the XPath language for querying W3C XML document object models.

1 Introduction

The Paisley embedded domain-specific language (EDSL) and library adds the more declarative style of *pattern matching* to the object-oriented context of Java programming [9,10]. Paisley offers a combination of features that is, by our knowledge, not offered by any other existing pattern matching solution for a main-stream language: it is strictly typed, fully compositional, non-invasive and supports non-determinism, full reification and persistence.

The non-deterministic aspects of Paisley have been demonstrated to provide a fairly powerful basis for embedded logic programming in the object-oriented host environment. In [11] we have discussed how to solve cryptarithmetic puzzles with the backtracking facilities of Paisley, and how to obtain non-trivial efficient search plans by object-oriented meta-programming with Paisley pattern objects.

Here we describe and evaluate recent extensions to Paisley that greatly increase its capabilities as an embedded functional-logic programming system, supporting more complex control and data flow than the combinatorial generate&test tactics of [11]. In particular, the novel contributions are:

1. in the core layer, a calculus of functions on patterns and its Kleene algebra, thus providing regular expressions of nested patterns;
2. in the application layer, a library extension that demonstrates their use by matching W3C XML Document Object Models with XPath [3] expressions;
3. a practical case study that evaluates the approach in comparison with standard on-board facilities of the host language Java.

Variants: 1 `.//a[@href]` 2 `.//p//a[@href]` 3 `.//p[.//a[@href]]`

```
void collect  (int variant,  Node node, Collection⟨Element⟩ results,
               boolean sideways) {
  if  (node instanceof Element) {
    Element elem = (Element)node;
    if  (elem.getTagName().equals(variant == 1 ? "a" : "p")) {
      if  (variant  > 1) {
        Collection⟨Element⟩ sub = variant == 2 ? results :  new ArrayList⟨⟩();
        collect(1,  elem, sub, false);
      }
      if  (variant == 1 && !elem.getAttribute("href").equals("")  ||
          variant  == 3 && !sub.isEmpty())
        results.add(elem);                                        // solution
    }
  }
  if  (node.getFirstChild() != null)
    collect(variant,  node.getFirstChild(),  results,  true);     // depth
  if  (sideways && node.getNextSibling() != null)
    collect(variant,  node.getNextSibling(), results,  true);     // breadth
}
```

Fig. 1. Three related XML queries. *Top:* XPath expressions; *bottom:* Java code.

2 Motivation

The basic motivation for a pattern matching EDSL is that patterns as language constructs make data-gathering code more declarative, and thus more readable, more writable and more maintainable. If done properly, this additional expressive power can be used to factor code compositionally in ways that are not straightforwardly possible in a conventional object-oriented approach, where the logic and data flow of a complex query often appear as cross-cutting concerns.

As an example that highlights the issues of logical compositionality, consider a family of three related XPath expressions, depicted in Fig. 1, that select nodes from XHTML documents. The first selects, from the context node and its descendants, all `<a>` elements (hyperlink anchors) that have a `href` (target) attribute specified. The second selects only those `<a>` elements with `href` attributes that are nested within `<p>` elements (paragraphs). The third selects all `<p>` elements that have some `<a>` element with `href` attribute nested within.

The task is to implement the specified search procedures, as object-oriented queries of the standard W3C X(HT)ML Document Object Model. Evidently, one wishes to implement most of the operational code generically, with reuse and minimal adaptation for each of the three variants. Additionally, the second and third variant should be able to use the first recursively. To emphasize the role of reuse in this example, Fig. 1 gives a unified implementation, where the identifier variant is grayed out to emphasize that all its occurrences serve only the static choice between the variants. The common algorithm is to traverse nodes by depth-first search, and add all valid matches, possibly repeatedly, to a `results` collection supplied by the caller. Two further improvements of the solution are suggested as exercises to the reader:

1. Of course, code that works for these three, but no other XPath expressions is hardly generic. Abstract to a large, preferrably unbounded, set of useful XPath expressions. Refactor the code to separate commonalities and individual degrees of freedom, as cleanly as possible.
2. The use of a `Collection` for matches implies eager evaluation. This is not efficient for the recursive calls from variant 3 to variant 1, where searching can be aborted after the first successful match (which falsifies `sub.isEmpty()`). Refactor the code to allow for lazy evaluation, as transparently as possible.[3]

See section 5 and Fig. 4 below for our proposed solution. We do not claim that these refactoring steps are infeasible in any particular previous framework. But we shall demonstrate that the Paisley approach naturally gives both pragmatic design guidelines, and a concrete notation for the adequate, abstract, modular and efficient expression of the underlying algorithm and its instantiations.

3 Basic Matching with Paisley

The design principles, semantics and APIs of Paisley patterns have been discussed in detail in [9,10,11]. The EDSL is extremely lightweight: It requires no language or compiler extension; API calls and programming idioms are sufficient for its full expressive power. The core API is summarized in Fig. 2.

The root of the pattern hierarchy is the abstract base class `Pattern⟨A⟩` of patterns that can process objects of some arbitrary type `A`. A pattern `Pattern⟨A⟩ p` is applied to some target data `x` of type `A` by calling `p.match(x)`, returning a **boolean** value indicating whether the match was successful.

Complex patterns are composed from application-layer building blocks that implement classifications and projections, the reification of instance tests and getter methods, respectively. These can be defined independently for arbitrary (public interfaces of) data models, requiring no intrusion into legacy code. Paisley comes with predefined bindings for common Java data models, such as the `Collection` framework; more can be added by the user. Each projection x from type `A` *to* `B` induces by contravariant lifting a construction of type `Pattern⟨A⟩` *from* `Pattern⟨B⟩`, conveniently implemented as a method `Pattern⟨A⟩ getX(Pattern⟨B⟩ p)`.

Information is extracted from a successfully matched pattern via embedded pattern variables. A pattern `Variable⟨A⟩ v` matches any value `A x`, and stores a reference to `x` that can be retrieved by invoking `v.getValue()`. Variables behave like imperative rather than logical variables; subsequent matches merely overwrite the stored value, no unification is implied.[4]

---

[3] This task is inherently much more difficult in conventional programming languages than in logic and lazy functional approaches; cf. [5].

[4] This design choice enables the use of Paisley for general, object-oriented data models, where stable notions of equality, let alone an induction principle, cannot be taken for granted. See section 4 for the handy implications of the imperative perspective.

```
abstract class Pattern⟨A⟩ {
  public abstract boolean match(A target);
  public boolean matchAgain();

  public static ⟨A⟩ Pattern⟨A⟩
    both(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);
  public static ⟨A⟩ Pattern⟨A⟩
    either(Pattern⟨? super A⟩ first, Pattern⟨? super A⟩ second);

  public Pattern⟨A⟩ someMatch();
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
  public A getValue();

  public ⟨B⟩ List⟨A⟩ eagerBindings(Pattern⟨? super B⟩ root, B target);
  public ⟨B⟩ Iterable⟨A⟩ lazyBindings(Pattern⟨? super B⟩ root, B target);

  public ⟨B⟩ Pattern⟨B⟩ bind(Pattern⟨? super B⟩ root, Pattern⟨? super A⟩ sub);
  public Pattern⟨A⟩ star(Pattern⟨? super A⟩ root);
  public Pattern⟨A⟩ plus(Pattern⟨? super A⟩ root);
}
```

Fig. 2. Interface synopsis (core).

Patterns can be composed conjunctively and disjunctively by the binary combinators both and either, or their n-ary variants all and some (not shown), respectively. As in traditional logic programming, disjunction is realized as backtracking: After each successful match for a pattern p, p.matchAgain() can be invoked to backtrack and find another solution. Solutions can be exhausted, in an imperative style of encapsulated search, by the following programming idiom:

```
if (p.match(x)) do
  doSomething();
while (p.matchAgain());
```

When *not* using an exhaustive loop, the computation of alternative solutions can be deferred indefinitely; the required state for choice points is stored residually in the instances of the logical combinators themselves.

The search construct can be abbreviated further to a functional style if the desired result of each match is the value stored in a single pattern variable v. Invoking v.eagerBindings(p, x) or v.lazyBindings(p, x) returns objects that give the value of v for all successive matches, collected beforehands in an implicit loop, or computed on iterator-driven demand, respectively. The latter can also deal with infinite sequences of solutions.

For cases where only the satisfiability of a pattern p, but not the actual sequence of solutions, is of concern, one can invoke p.someMatch(), yielding a wrapper that cuts all alternatives after the first solution. Thus, laziness is exploited for efficiency and, when used in conjunction with other patterns with multiple relevant solutions, spurious multiplication of solution sets is avoided.

4 Advanced Pattern Calculus

4.1 Pattern Substitution and Iteration

As discussed in [10], Paisley patterns and their variables obey an algebraic calculus where most of the expected mathematical laws hold, despite the low-level imperative nature of their implementation.

A variable `v` known to occur[5] in a pattern `p` can be substituted by a subpattern `q`, by invoking `v.bind(p, q)`, mnemonically read as $(\lambda v.\, p)q$. The implementation delegates to `p` and `q` sequentially, and hence does not require access to the internals of either operand.

Substitution can also be given a recursive twist; a pattern `p` with a "hole" `v` can be nested. The patterns `v.star(p)` and `v.plus(p)` correspond to the * and $^+$-closures, respectively, of the search path relation between `p` and `v`, in the sense that the usual recursive relations hold,

$$\begin{aligned} \mathsf{v.star(p)} &\equiv \mathsf{either(v,\ v.plus(p))} \\ \mathsf{v.plus(p)} &\equiv \mathsf{v.bind(p,\ v.star(p))} \end{aligned}$$

which is already almost the complete, effective implementation, up to lazy duplication of `v.plus(p)` for each iteration level that is actually reached. Note that iteration depth is conceptually unbounded, and solutions are explored in depth-first pre-order, due to the way `either` is used. Thus patterns form a Kleene algebra up to equivalence of solution *sets* but not *sequences*.

Using these iteration operators, complex nondeterministic pattern constructors can be defined concisely. For instance, the contravariant lifting of the multi-valued, transitive *descendant* projection in XML document trees, applied to a pattern `p`, becomes simply

```
v.bind(v.plus(child(v)), p)
```

where `Variable⟨Node⟩ v` is fresh, and the multi-valued projection `Pattern⟨Node⟩ child(Pattern⟨Node⟩)` is implemented in terms of the `org.w3c.dom.Node` getter methods `getFirstChild()` and `getNextSibling()`. Note how the two sources of nondeterminism, regarding horizontal (`child`) and vertical (`plus`) position in the document tree, respectively (cf. the distinct recursive calls in Fig. 1), combine completely transparently.

4.2 Pattern Function Abstraction

Functions taking patterns to patterns have so far featured in two important roles. In operational form, implemented as lifting methods, they are the basis for contravariant representation of projection patterns. In algebraic form, as `bind`-based abstractions from a variable, they enable pattern composition and iteration. Being so ubiquitous in the Paisley approach, they deserve their own

---

[5] The actual condition is definite assignment rather than occurrence, for technical reasons.

```
abstract class Motif⟨A, B⟩ {
  public abstract Pattern⟨B⟩ apply(Pattern⟨? super A⟩);

  public static ⟨A⟩ Motif⟨A, A⟩ id();
  public        ⟨C⟩ Motif⟨C, B⟩ on(Motif⟨C, ? super A⟩ other);

  public static ⟨A⟩ Motif⟨A, A⟩ star(Motif⟨A, ? super A⟩ f);
  public static ⟨A⟩ Motif⟨A, A⟩ plus(Motif⟨A, ? super A⟩ f);

  public Iterable⟨A⟩ lazyBindings (B target);
  public List⟨A⟩ eagerBindings(B target);
}

class Variable⟨A⟩ extends Pattern⟨A⟩ {
  // ...
  public ⟨B⟩ Motif⟨A, B⟩ lambda(Pattern⟨B⟩ root);
}
```

Fig. 3. First-class pattern function (motif) interface.

reification. Fig. 3 shows the relevant API. An instance of class Motif⟨A, B⟩ is the reified form of a function taking patterns over A to patterns over B, or by contravariance, the pattern representation of an access operation that takes data of type B to data of type A, by its apply method.

The Motif class provides the algebra of the category of patterns, namely the identical motif id() and the type-safe composition of motifs by the on(Motif) instance method. The variable-related operations star/plus and lazy-/eagerBindings each have a point-free counterpart in Motif, which create anonymous variables to hold intermediate results on the fly. For instance, the XML *descendant* pattern defined above can be expressed less redundantly in point-free form as

`plus(child).apply(p)`

given a child access reification Motif⟨Node,Node⟩ child.

Conversely, motif abstraction is defined for pattern variables, which obeys the beta reduction rule:

$$\mathsf{v.lambda(p).apply(q)} \equiv \mathsf{v.bind(p, q)}$$

There is often a trade-off in convenience between functions in operational and reified form, that is as either pattern lifting methods or motifs. Since there is no automatic conversion between methods and function objects in Java prior to version 8, our strategy in the Paisley library is to provide both redundantly.

5 Motivation Revisited

Figure 4 repeats the XPath examples expressions, now contrasting the domain-specific XPath code with the general-purpose Java/Paisley code. The latter has been underlined for easy reference. The solution makes full use of the shared

Variants: **1** `.//a[@href]` **2** `.//p//a[@href]` **3** `.//p[.//a[@href]]`

```
Pattern⟨Node⟩ dslash(String name, Pattern⟨Element⟩... constraints) {
  return descendantOrSelf(element(both(tagName(name), all(constraints))));
}

Variable⟨Element⟩ outerVar = new Variable⟨⟩();
Pattern⟨Node⟩ outerForm = dslash(”p”, outerVar);
Variable⟨Element⟩ innerVar = new Variable⟨⟩();
Pattern⟨Node⟩ innerForm = dslash(”a”, innerVar, attr(”href”, neq(””)));

Iterable⟨Element⟩ collect1(Document root) {
  return innerVar.lazyBindings(innerForm, root);
}
Iterable⟨Element⟩ collect2(Document root) {
  return innerVar.lazyBindings(outerVar.bind(outerForm, innerForm), root);
}
Iterable⟨Element⟩ collect3(Document root) {
  return outerVar.lazyBindings(outerVar.bind(outerForm, innerForm.someMatch()), root);
}
```

Fig. 4. Three related XML queries revisited (from Figure 1). *Top:* XPath expressions; *middle:* Java code template; *bottom:* template instantiations.

common structure of the three variants, and achieves complete separation of concerns:

- The search procedure implied by the operator `//` is not spread out as recursive control flow, but reified and encapsulated as the descendantOrSelf pattern lifting, predefined in the XPathPatterns static factory class of Paisley, discussed in detail in the following section.
- XPath subexpressions are denoted concisely and orthogonally.
  - The generic form `.//`*tag*... that reoccurs in all variants is abstracted as the extensible pattern construction dslash.
  - The particular inner and outer forms, `.//a[@href]` and `.//p`, respectively, are defined once and for all, independently of each other.
- The semantics of XPath queries are effected concisely and orthogonally.
  - The general principle of encapsulated search is expressed naturally by a call to lazyBindings. The third variant concisely prunes the recursively encapsulated search *after the first solution*, via the pattern modifier someMatch().
  - The particular configuration of subexpressions for all variants boils down to a simple choice of the bound variable and pattern-algebraic composition. Note that both are reified and hence first-class citizens of Java. Thus, each variant boils down to a one-line expression, where the points of variability are ordinary subexpressions that can be chosen statically (as shown) or dynamically, with arbitrarily complex meta-programming.

```
enum Axis {
  public Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Test {
  public abstract Motif⟨? extends Node, Node⟩ getMotif();
}

abstract class Predicate {
  public abstract boolean accepts(NodeSet context, int position, Node candidate);
  public Motif⟨? extends Node, Node⟩ apply(Motif⟨? extends Node, Node⟩ base);
}

class NodeSet {
  private Collection⟨? extends Node⟩ elems;
  public NodeSet(Iterable⟨? extends Node⟩ elems);
  public int size();
  public Iterable⟨Node⟩ filter(Predicate pred);
}

class Path {
  Path(Motif⟨? extends Node, Node⟩ motif);
  public Motif⟨? extends Node, Node⟩ getMotif();
  public Path step(Axis axis, Test test, Predicate... predicates);
}
```

Fig. 5. XPath base classes in Paisley.

6 The **Paisley** XPath Interpreter

As a case study of real-world data models and queries, we have implemented the *navigational* (proper path) fragment of the XPath 1.0 language as a Paisley pattern combinator library. The complete implementation consists of the factory classes XMLPatterns for generic DOM access and XPathPatterns for XPath specifics, with currently 433 and 282 lines of code, respectively. Given an XPath parser, it can be extended to a full-fledged interpreter, and hence a non-embedded DSL, by a straightforward mapping of abstract syntax nodes to pattern operations.

6.1 Language Fragment

The XPath 1.0 language comes with many datatypes and functions that do not contribute directly to its main goal, namely the addressing of nodes in an XML document. For simplicity, we restrict our treatment of the language to a fragment streamlined for that purpose. Typical uses of XPath within XSLT or XQuery [2] conform to this fragment. We conjecture that the missing features can be added without worsening essential operational complexity and runtime performance.

We omit external variables and functions, and all operations on the datatypes of strings, numbers and booleans, as well as intangible document nodes such as comments and processing instructions. The supported sublanguage is reflected one-to-one by API operations based in the class hierarchy depicted in Fig. 5. (Operation signatures are given in Fig. 11 in the appendix.) Its focus is on so-called path expressions of the general syntactic form

$$\mathit{path} ::= \mathit{abs\_path} \mid \mathit{rel\_path} \qquad\qquad \mathit{abs\_path} ::= \texttt{/}\,\mathit{rel\_path}^{?}$$

$$\mathit{step} ::= \mathit{axis}\ \texttt{::}\ \mathit{test}\ (\,\texttt{[}\ \mathit{predicate}\ \texttt{]}\,)^{*} \qquad\qquad \mathit{rel\_path} ::= (\mathit{rel\_path}\ \texttt{/}\ )^{?}\,\mathit{step}$$

Here *axis* is one of the twelve XPath document axes, omitting the deprecated namespace declaration axis. The nonterminal *test* is the tautological test `node()`, the text node test `text()` or an explicit node name test. For *predicate*, used to filter selected nodes, we accept not the full XPath expression language, but only

$$\mathit{predicate} ::= \mathit{path} \mid \mathit{integer} \mid \texttt{not}\ \mathit{predicate} \mid \texttt{(}\ \mathit{predicate}\,(\texttt{and} \mid \texttt{or})\ \mathit{predicate}\ \texttt{)}$$

where a *path* predicate holds if and only if it selects at least one node (existential quantification). Positive and nonpositive *integer* predicates $\texttt{[}\pm i\texttt{]}$ select the i-th node from the candidate sequence, and the $(n - i)$-th node, respectively, from the sequence of n candidates in total. The former is a valid abbreviation for `[position()==i]` in standard XPath; we add the latter as an analogous abbreviation for `[position()==last()-i]`. Logical connectives on predicates are defined as usual.

Various abbreviation rules apply; for instance, the shorthand `.//a[@href]` expands to the verbose form `self::node()/descendantOrSelf::node()/child ::a[attribute::href]`. This translates to the following semantic object:

```
relative().step(Axis.self, node())
           .step(Axis.descendantOrSelf, node())
           .step(Axis.child, name("a"),
                 exists(relative().step(Axis.attribute, "href")))
```

The relation between XPath predicates and candidate sequences, misleadingly called “node-sets” in the standard, is rather idiosyncratic (they can not be modeled adequately as plain sets) and the major challenge in this case study. A node-set is implicitly endowed with either of two predefined orders, namely *forward* or *reverse document order*. These orders are loosely specified by a pre-order traversal of nodes, up to attribute permutation. A predicate filters the nodes in a node-set not purely by point-wise application, but may depend on some context, namely the position of the node in the node-set, starting from one, and the total number of members. This information is available in XPath via the “functions” `position()` and `last()`, respectively. It is realized in the API by the parameter position of method Predicate.accepts and the method size() of class NodeSet, respectively, to be supplied from the method filter of class NodeSet.

6.2 Pattern-Based Interpreter Design

The node-extracting semantics of the XPath language can be rendered naturally in the Paisley framework by contravariant lifting. An XPath expression can be

$$\begin{aligned} \mathsf{ancestor} &\equiv \mathsf{plus}(\mathsf{parent}) & \mathsf{descendant} &\equiv \mathsf{plus}(\mathsf{child}) \\ \mathsf{ancestorOrSelf} &\equiv \mathsf{star}(\mathsf{parent}) & \mathsf{descendantOrSelf} &\equiv \mathsf{star}(\mathsf{child}) \\ \mathsf{followingSibling} &\equiv \mathsf{plus}(\mathsf{nextSibling}) & \mathsf{precedingSibling} &\equiv \mathsf{plus}(\mathsf{previousSibling}) \end{aligned}$$

$$\mathsf{following} \equiv \mathsf{ancestorOrSelf.on}(\mathsf{followingSibling}).\mathsf{on}(\mathsf{descendantOrSelf})$$

$$\mathsf{preceding} \equiv \mathsf{ancestorOrSelf.on}(\mathsf{precedingSibling}).\mathsf{on}(\mathsf{descendantOrSelf})$$

$$\mathsf{self} \equiv \mathsf{id}$$

Fig. 6. Non-primitive XPath axes.

```
class Path {
  // ...
  public Path step(Axis axis, Test test, Predicate... predicates) {
    Motif⟨? extends Node, Node⟩ r = axis.getMotif().on(test.getMotif());
    for (Predicate p : predicates)
      r = p.apply(r);
    return new Path(getMotif().on(r));
  }
}
```

Fig. 7. Implementation of composite path expressions.

applied to any node of an XML document, here implemented as DOM, and extracts some other nodes, possibly of a more special type, such as elements or attributes. This gives rise to a semantic type Motif⟨? **extends** Node, Node⟩.

Except for the context-sensitivity of predicates, all language constructs could simply been given motif semantics and lumped together by composition.

XPath axes are all defined concisely in terms of motifs. Primitive DOM access operations from factory class XMLPatterns directly define the attribute, child and parent axes. Given the additional primitives next-/ previousSibling, which are only implicit in the XPath standard, all other axes are definable in terms of elementary motif algebra; see Figure 6. Node tests are straightforward applications of test pattern lifting.

Atomic path expressions are realized by the document root access motif and the identity motif, for absolute and relative paths, respectively. Composite path expressions conceptually compose their constituents left to right. The exception are predicates, which are implemented as motif transforms in order to deal with context sensitivity. These are applied in order to the step basis, that is the composition of axis and test, for local filtering, and the result is then composed with the front of the path expression; see Fig. 7.

The real challenge is the implementation of node-set predicates: On the one hand, context information about relative element position and total node-set size must be provided, which transcends the context-free realm of pattern and motif composition. On the other hand, elements should be enumerated lazily, in

```
class Predicate {
  // ...
  public Motif⟨Node, Node⟩ apply(final Motif⟨? extends Node, Node⟩ base) {
    return new Motif⟨? extends Node, Node⟩ () {
      public Pattern⟨Node⟩ apply(Pattern⟨? super Node⟩ p) {
        return new MultiTransform⟨Node, Node⟩ (p) {
          protected Iterable⟨Node⟩ apply(Node n) {
            return new NodeSet(base.lazyBindings(n)).filter(Predicate.this); // [∗]
  }}; }}; }
}

class NodeSet {
  // ...
  public NodeSet(Iterable⟨? extends Node⟩ elems) {
    this.elems = cache(elems) ;
  }
  public Iterable⟨Node⟩ filter(final Predicate pred) {
    return new Iterable⟨Node⟩ () {
      public Iterator⟨Node⟩ iterator() {
        return new FilterIterator⟨Node⟩ (elems.iterator()) {
          int i = 0;                                                  // [∗]
          protected boolean accepts(Node candidate) {
            return pred.accepts(NodeSet.this, ++i, candidate);        // [∗]
  }}; }}; }
}
public static Predicate exists(final Path cond) {
  return new Predicate() {
    public boolean accepts(NodeSet context, int position, Node node) {
      return cond.getMotif().apply(any()).match(node);               // [∗]
    }
  };
}
public static Predicate index(final int i) {
  return new Predicate() {
    public boolean accepts(NodeSet context, int position, Node node) {
      return position == (i > 0 ? i : context.size() − i);          // [∗]
    }
  };
}
```

Fig. 8. Implementation of context-sensitive predicate filtering. See text for underlining and [∗].

order to make tests for non-emptiness efficient and avoid scanning for unneeded solutions. Obviously, evaluation must switch transparently from lazy to eager strategy if the size of the node-set is observed. And lastly, for elegance reasons, we strive for an implementation that is as declarative as possible, with very limited amounts of specific imperative coding.

The solution is depicted in Figure 8. Generic Paisley API method calls are underlined for easy reference. The action of a predicate on a base motif requires control over the solution node-set. Hence it needs to intercept both the pattern parameter at the motif level and the target node parameter at the pattern level, by means of two nested anonymous classes. Then a lazy disjunction is spliced in, which enumerates the solutions of the base motif, wraps them in a local node-set and filters them context-sensitively.

The counterpart on the node-set side of the implementation works as follows: It wraps the lazy enumeration of candidate nodes is a collection, via the auxiliary method `cache` (definition not shown). This collection caches enumerated items for repeated access, and forces eager evaluation if its `size()` method is called.

The actual filtering operation yields a lazy enumeration that intercepts iterator creation, again by means of two nested anonymous classes. The iterator of the cached candidate collection is overwritten by an instance of the auxiliary abstract class `FilterIterator⟨A⟩` (definition not shown) that in-/excludes elements ad-hoc, determined by the result of its method **boolean** `accepts(A)`. This acceptance test is then routed back to the context-sensitive acceptance test of the given predicate, by addition of a single minuscule piece of explicit imperative (stateful) programming, namely a counter `i` for the relative position. Note that, Java formal noise apart, the actual problem-specific code consists of five single-line statements, marked with [$*$].

Existential predicates forget their results, hence the invocation of the catch-all pattern `any()`. They prune the search after the first hit, as witnessed by the absence of a call to `matchAgain()` on the freshly created pattern. Eager evaluation is only ever triggered by index predicates via a call to `context.size()`. Logical predicate connectives are defined point-wise (not shown).

7 Experimental Evaluation

We have tested the performance and scalability of our implementation using material from XMark [8], a benchmark suite for XML-based large databases. The homepage of XMark [7] offers a downloadable tool to generate pseudo-random well-typed XML files according to a published DTD, with a linear size parameter. We have used the tools to produce test data files for size parameter values from 0.04 to 0.40 in increments of 0.04, where 0.01 corresponds roughly to 1 MiB of canonical XML. From the 20 published XQuery benchmark queries we have chosen as test cases the four where the entire logic is expressed in XPath rather than XQuery operations; see Fig. 9 (and Table 1 in the appendix).

For each pair of data file and query expression, we processed the document with `lazyBindings` of the XPath motif, and computed running times for extracting

Query	**XPath Expression**
Q01	`/site/open_auctions/open_auction/bidder[1]/increase/text()`
Q06	`//site/regions//item`
Q15	`/site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()`
Q16	`/site/closed_auctions/closed_auction[annotation/description/parlist/listitem/parlist/listitem/text/emph/keyword/text()]`

Fig. 9. XPath expression test cases from the XMark benchmark.

the *first* solution and *all* solutions (in a loop), as well as the *effective* time (total time divided by number of solutions).

Timing values were obtained as real time with System.nanoTime(), median value of ten repetitions, with interspersed calls to System.gc(). In order to compensate deferred computation costs in the DOM implementation, we reused the same document instances for all successive queries. All experiments were performed on an Intel Core i5-3317U quad-core running at 1.70 GHz, with 8 GiB of physical memory, under Ubuntu 14.04 LTS (64bit), and Oracle Java SE 1.8.0_05-b13 with HotSpot 64bit Server VM 25.5-b02 and 800 MiB heap limit.

Our first quantitative goal, beyond highlighting the elegance and effectiveness of Paisley-style pattern *specification*, is to demonstrate the efficiency payoff of lazy pattern *execution*. Our second quantitative goal comes back to methodological arguments from the motivation section of this paper. Pattern matching has been hailed as a declarative tool that brings expressiveness for data querying extremely close to the hosting programming language environment. Tools for non-embedded domain-specific languages may be more powerful in many respects, but the burden of proof is on them that this power is worth the trouble incurred by the impedance mismatch with ordinary host code. Nevertheless, we should verify that the benefits of tight embedding produced by our methodological approach are not squandered by the implementation.

To that end, we repeated our experiments with the next-closest tool at hand, namely the Java on-board XPath implementation accessible as javax.xml.xpath.*.[6] We compared both preparation and running times of Paisley XPath patterns with XPathExpression objects obtained via XPath.compile(String). The first solution and all solutions were obtained by calling XPathExpression.evaluate with the type parameter values NODE and NODESET, respectively.

The Paisley approach fares well, even as a non-embedded DSL. The Paisley preparation process (a simple recursive descent parser for full XPath generated with ANTLR[7], followed by direct translation of abstract syntax nodes to pattern operations) is consistently faster than on-board compilation to XPathExpression objects, although the task may have been sped up marginally by considering

---

[6] In our Java environment, com.sun.org.apache.xpath.internal.jaxp.XPathImpl.

[7] http://www.antlr.org/

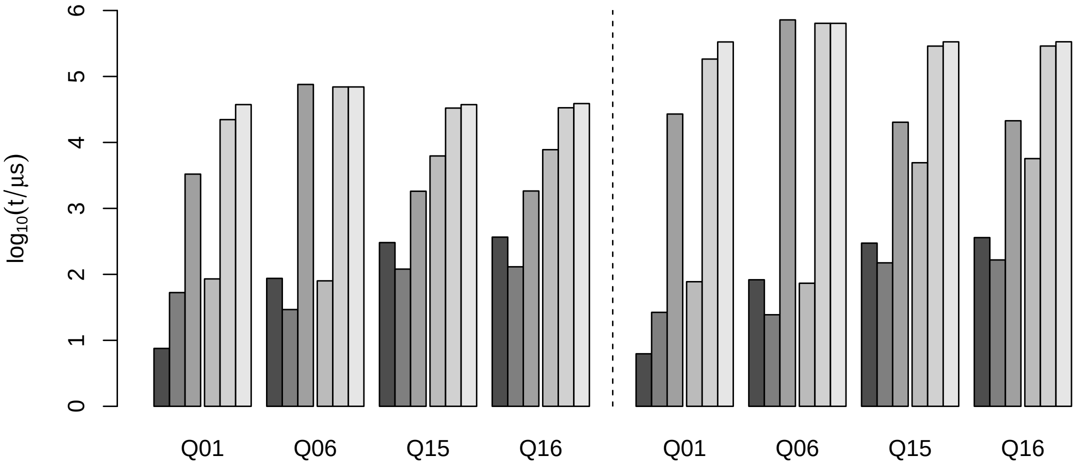


Fig. 10. Running times for Paisley and Java on-board XPath implementations. *Left* – size parameter 0.04; *right* – size parameter 0.40. Colors: *dark to light* – Paisley eff/first/all; on-board eff/first/all. Logarithmic scale; lower end of scale arbitrary.

only a subset of the language after parsing. (See Table 2 in the appendix for details.) When patterns are constructed in embedded DSL style, using the API rather than textual input, static safety is improved, and even the small overhead eliminated, at the same time.

The differences in running times are so drastic that they can only be visualized meaningfully in logarithmic scale. Proportions range from on-board facilities being 13 % faster for all solutions of Q06 at size 0.04, to being over 26 000 times slower for the first solution of Q06 at size 0.40. With the exception of the exhaustion of "brute-force" case Q06, Paisley is 1–2 (all solutions) or 2–4 (first solution) orders of magnitude faster, respectively; see Fig. 10. Accessing only the first solution with the on-board tools is particularly disappointing, being only marginally faster than exhausting all solutions. It appears that our motivation is confirmed, and conventionally developed tools are ill-suited to scalable lazy evaluation, where external demand governs internal control. (More detailed results are given in Fig. 12 the appendix.)

8 Conclusion

The experimental figures for the on-board tools have been obtained without any tweaking of features; hence there is large uncertainty in the amount of possible improvements. Therefore the comparison should not be taken too literally. But we feel that it is fair in a certain sense nevertheless: Our Paisley implementation has been obtained in the straightest possible manner, also without any tweaking. Its advantage lies thus chiefly in the fact that we have chosen the application domain, namely the specified XPath fragment, and tailored the tool design to avoid any complexity inessential to the task at hand. It is this light-weight flexibility by effective manual programming we wish to leverage with the Paisley approach – the capabilities of existing, more heavy-weight tools with respect to automated, adaptive specialization are generally no match.

8.1 Related Work

Related work with regard to language design and implementation, and to object-oriented-logic programming, has been discussed in [9,10] and [11], respectively.

Purely declarative accounts of XPath, although theoretically interesting, have little technical impact on our main concern, the concrete embedding in a mainstream programming language. A few random examples: In [6], XPathLog is presented, adding variable binding capabilities and sound Herbrandt semantics to XPath, for a full-fledged logic programming language. In [4], an algorithmic analysis of XPath in terms of modal logic is given. The language fragment and experimental approach used there is a model predecessor for our own work, including the particular benchmark. In the recent paper [1], the implementation of XPath in the Haskell-like functional-logic programming language TOY is discussed. A combinatorial approach to XPath constructs very similar to Paisley is taken, which appears to corroborate our claims of a natural design.

References

1. Almendros-Jiménez, J., Caballero, R., García-Ruiz, Y., Sáenz-Pérez, F.: XPath query processing in a functional-logic language. ENTCS 282, 19–34 (2012)
2. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language (Second Edition). W3C, `http://www.w3.org/TR/2010/REC-xquery-20101214/` (2010)
3. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C, `http://www.w3.org/TR/1999/REC-xpath-19991116/` (1999)
4. Franceschet, M., Zimuel, E.: Modal logic and navigational XPath: an experimental comparison. In: Proc. Workshop Methods for Modalities. pp. 156–172 (2005)
5. Hughes, J.: Why Functional Programming Matters. Computer Journal 32(2), 98–107 (1989)
6. May, W.: XPath-Logic and XPathLog: A logic-based approach for declarative XML data manipulation. Tech. Rep. 149, Institut für Informatik, Albert-Ludwigs-Universität Freiburg (2001)
7. Schmidt, A.: XMark – an XML benchmark project (2009), `http://www.xml-benchmark.org/`
8. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: Proc. 28th VLDB. pp. 974–985. Morgan Kaufmann (2002)
9. Trancón y Widemann, B., Lepper, M.: Paisley: pattern matching à la carte. In: Proc. 5th ICMT. LNCS, vol. 7307, pp. 240–247. Springer (2012)
10. Trancón y Widemann, B., Lepper, M.: Paisley: A pattern matching library for arbitrary object models. In: Proc. 6th ATPS. LNI, vol. 215, pp. 171–186. Gesellschaft für Informatik (2013)
11. Trancón y Widemann, B., Lepper, M.: Some experiments on light-weight object-functional-logic programming in Java with Paisley. In: Declarative Programming and Knowledge Management. LNCS, vol. 8439. Springer (2014), in press

A Appendix: Supplementary Material

Table 1. Size of input files and solution spaces for XPath expression test cases from the XMark benchmark.

	Size		**# Solutions**			
Parameter	**File** (kiB)	**# Nodes**	**Q01**	**Q06**	**Q15**	**Q16**
0.04	4 648	191 083	439	870	6	5
0.08	9 212	379 719	884	1 740	17	15
0.12	13 696	569 434	1 301	2 604	29	28
0.16	18 100	748 766	1 732	3 480	32	29
0.20	22 964	946 554	2 142	4 350	37	34
0.24	27 396	1 129 553	2 610	5 214	44	40
0.28	31 976	1 315 902	3 045	6 090	62	58
0.32	36 616	1 502 611	3 453	6 960	60	53
0.36	41 076	1 690 597	3 857	7 824	61	54
0.40	45 600	1 877 979	4 310	8 700	68	59

Table 2. Preparation times for Paisley and Java on-board XPath implementations.

		Q01	**Q06**	**Q15**	**Q16**
Paisley	(μs)	56.40	101.73	89.26	154.32
On-Board		233.44	204.73	162.61	215.92
Ratio		4.14	2.01	1.82	1.40

```
enum Axis {
  ancestor, ancestorOrSelf, attribute,  child,  descendant, descendantOrSelf,
  following, followingSibling,  parent, preceding, precedingSibling, self
  // ...
}
public static Test node  ();
public static Test text   ();
public static Test name  (String tagName);
public static Predicate exists  (Path cond);
public static Predicate index   (int i);
public static Predicate not      (Predicate p);
public static Predicate and      (Predicate p, Predicate q);
public static Predicate or       (Predicate p, Predicate q);
public static Path absolute ();
public static Path relative  ();
```

Fig. 11. XPath operations as combinators in the Paisley application-layer library.

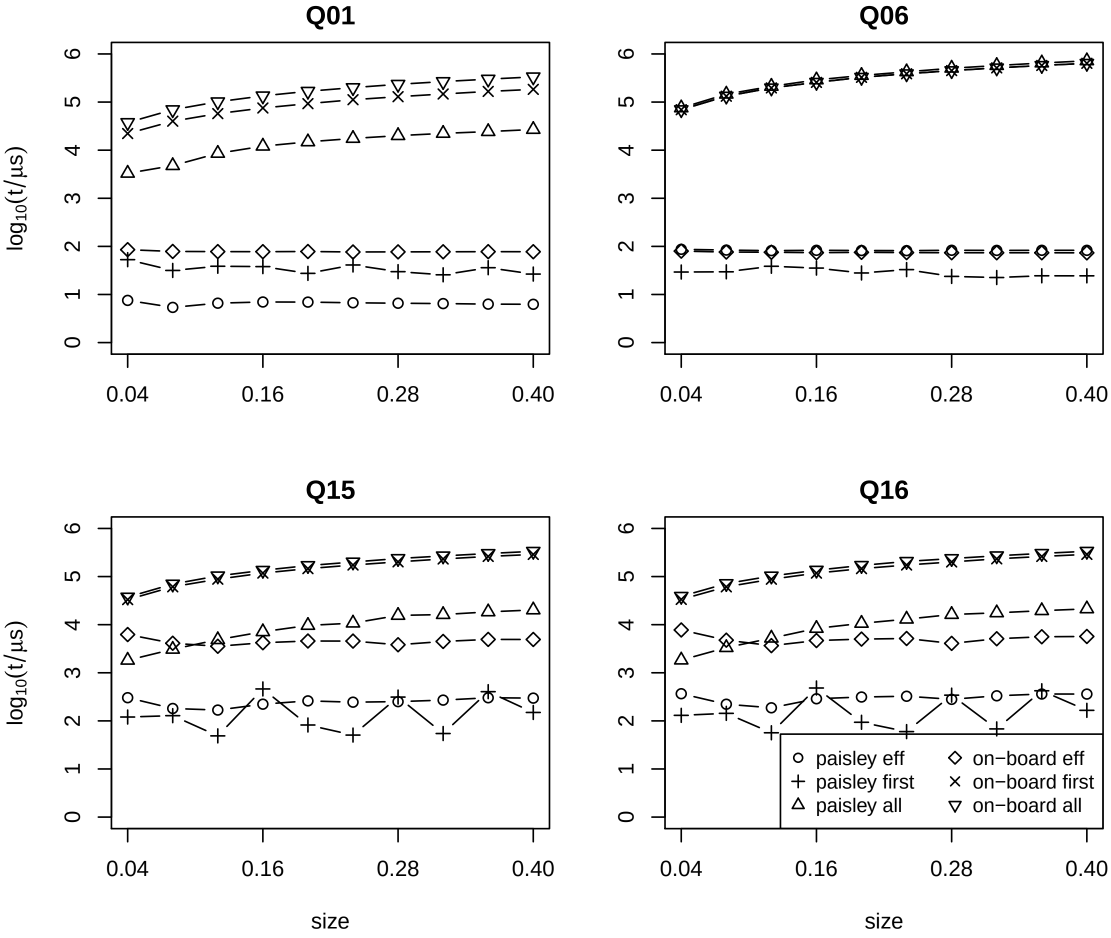


Fig. 12. Running times for Paisley and Java on-board XPath implementations. Logarithmic scale; lower end of scale arbitrary.

Exploring Non-Determinism in Graph Algorithms

Nikita Danilenko

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

Abstract. Graph algorithms that are based on the computation of one or more paths are often written in an implicitly non-deterministic way, which suggests that the result of the algorithm does not depend on a particular path, but any path that satisfies a given property. Such algorithms provide an additional challenge in typical implementations, because one needs to replace the non-determinism with an actual implementation. In this paper we explore the effects of using non-determinism explicitly in the functional logic programming language Curry. To that end we consider three algorithms and implement them in a prototypically non-deterministic fashion.

1 Introduction

Consider a graph $G = (V, E)$, two vertices $s, t \in V$ and the question, whether there is a cycle in G that contains both vertices (i.e. both are contained in the same strong connected component). It is easy to come up with a solution for this task – simply check whether there are paths from s to t and from t to s in G, if so, the answer is "yes", otherwise it is "no". Now all one needs to do is to come up with how to specify the existence of paths, which is just as straight-forward. Next, one could ask for an actual cycle in case such a cycle exists. For every path p from s to t and every path q from t to s, the composition of p and q is such a cycle, i.e. the existence of a cycle does not depend on a particular choice of a path and there may be several different cycles.

When the above problems are considered in light of logic programming, one can use non-determinism to express the independence of choice and to compute one or more results from the specification alone. Aside from the usual benefit of the declarative approach of what to compute and not how, one can also observe that the different intermediate results (cycles or paths) yield the same overall answer (yes or no). While this is somewhat trivial in the above example, there are more sophisticated graph algorithms that rely on the computation of *some* value with a special property and it may not be obvious at all that different choices of such values yield the same results in the end. This invariance can be considered as a certain kind of confluence and the possibility to observe this invariance is a useful tool, particularly in teaching.

In this paper we consider graph algorithms that are based on the computation of paths. We begin with a simple observation relating the path search strategy to the solution search strategies and proceed to provide solutions to two non-trivial graph problems, namely the *maximum matching problem* and the *maximal flow problem* and provide examples of matchings and flows. Our main focus is on experimentation and applications in teaching, where theoretical results are likely to be followed by examples. Observing implicit non-determinism explicitly is of great assistance in this context since one can witness the invariance of the solution with respect to different choices that lead to it.

All code in this paper is written in Curry [12] and compiled with KiCS2 [5], which can be obtained at `http://www-ps.informatik.uni-kiel.de/kics2/`. Throughout the paper we refer to several functions and modules, all of which can be found using the search engine Curr(y)gle `https://www-ps.informatik.uni-kiel.de/kics2/currygle/`. A polished version of the presented code is available at GitHub under `https://github.com/nikitaDanilenko/ndga`.

2 Preliminaries

A graph is a pair $G = (V, E)$, where V is a non-empty finite set and $E \subseteq V \times V$. This is to say that we consider all graphs to be directed and realise graphs in which the direction does not matter as symmetric directed graphs. For any $v \in V$ we set $N_{\rightarrow}(v) := \{\, w \in V \mid (v, w) \in E \,\}$ and $N_{\leftarrow}(v) := \{\, w \in V \mid (w, v) \in E \,\}$; elements of $N_{\rightarrow}(v)$ are called *successors* and those of $N_{\leftarrow}(v)$ *predecessors* of v. A path in a graph is an injective sequence of vertices, i.e. a sequence that does not contain multiple occurrences of the same vertex. Since V is finite, every path traverses at most $|V|$ vertices. We represent vertices by integers, edges by pairs of vertices and paths by lists of distinct vertices, where the distinctness is required implicitly. We discuss this design decision in Section 3.

```
type Vertex = Int
type Edge   = (Vertex, Vertex)
type Path   = [Vertex]
```

Graphs are represented by an adjacency list model, where the adjacency lists are sorted in ascending natural order of the integers. These adjacency lists are stored in a finite map, one implementation of which is provided in the standard KiCS2 library *FiniteMap* as the data structure *FM key value*.

```
data Graph = Graph (FM Vertex [Vertex])
```

Additionally, we use sets of vertices explicitly for the maintenance of visited vertices. We use finite maps provided by KiCS2 for a representation of sets of vertices, where the functions *emptyFM*, *addToFM*, *elemFM*, *delFromFM* are provided, too. The function *emptyFM* is parametrised over an irreflexive order predicate which in our case is $(<) :: Int \rightarrow Int \rightarrow Bool$.

```
type VertexSet = FM Vertex ()
```

```
empty :: VertexSet
empty = emptyFM (<)

insert :: Vertex → VertexSet → VertexSet
insert i m = addToFM m i ()

inSet :: Vertex → VertexSet → Bool
inSet = elemFM

remove :: Vertex → VertexSet → VertexSet
remove = flip delFromFM

vertexListToSet :: [ Vertex] → VertexSet
vertexListToSet = foldr insert empty
```

For the purpose of demonstration we use the two example graphs from the Figures 1, 2 in the following two sections and assume that they are implemented in the values *graph1* and *graph2* respectively.

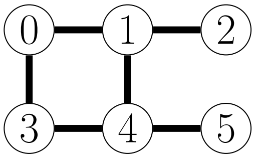


Fig. 1. Example graph G_1

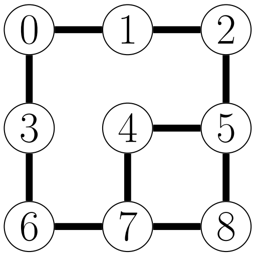


Fig. 2. Example graph G_2

3 Reachability and Paths

One standard example of logic programming in graph theory is the check whether a vertex set ts is reachable from a given vertex s. This is the case iff $s \in ts$ or if there is a successor $i \in V$ of s, such that ts is reachable from i. Since successors can lead back to already visited vertices, one additionally has to check, if the successor has been visited or not[1]. The implementation of the above may be as follows.

```
reachable :: Graph → Vertex → VertexSet → Success
reachable g from ts = find empty from where

  find vis s | s ‘inSet‘ ts                      = success
             | isEdge g s i ∧ ¬ (i ‘inSet‘ vis) = find (insert s vis) i where i free
```

This implementation contains a high level of abstraction and is very close to the original specification. Also, instead of defining how to access the successors of

---

[1] Avoiding this check can easily result in non-termination: consider the graph $(\{0,1,2\}, \{(0,1),(1,0),(0,2)\})$ with DFS. Then checking whether 2 is reachable from 0 diverges, because the loop $(0,1,0)$ is entered before checking the other successors of 0. A similar example shows the non-termination in case of using BFS.

s, this implementation relies only on a test whether a certain edge is contained in the graph and the use of a free variable to find a fitting edge. The technique of translating existential quantifications into free variables is common in logic programming.

Instead of just checking for the existence of a path, one can ask for an actual path between two vertices. It is easy to implement such a search by modifying the above function.

```
path :: Graph → Vertex → VertexSet → Path
path g from ts = find empty from where
  find vis s | s `inSet` ts                         = [s]
             | isEdge g s i ∧ ¬ (i `inSet` vis) = s : find (insert s vis) i where i free
pathToSingle :: Graph → Vertex → Vertex → Path
pathToSingle g from t = path g from (vertexListToSet [t])
```

Again, the function is non-deterministic in its choice of the successor. There are two interesting consequences of this implementation. First of all, using the interactive mode of KiCS2 one can find all paths between two given vertices without any additional work. Alternatively, when all paths are required in a program, one can use set functions [2] to encapsulate the search and collect all results. Set functions are provided in the KiCS2 library *SetFunctions*, which provides functions *setK* (for $K \in \{0, \ldots, 7\}$)

$$setK :: (a_0 \to .. \to a_{K-1} \to b) \to a_0 \to .. \to a_{K-1} \to Values\ b$$

which turn a function of a given arity into a set function, where *Values b* is the list of all results. Assuming that the graph from Figure 2 is implemented as *graph2* we can test the following results.

```
kics2> set3 path graph2 0 (vertexListToSet [5,8])
(Values [[0,1,2,5],[0,3,6,7,4,5],[0,3,6,7,8]])
kics2> set3 pathToSingle graph2 0 8
(Values [[0,1,2,5,4,7,8],[0,1,2,5,8],[0,3,6,7,4,5,8],[0,3,6,7,8]])
```

The second consequence is that the path search is based upon the order in which the free variable *i* is instantiated, which in turn depends on the search strategy. The search strategy can be chosen in KiCS2 either interactively (by using :*set STRATEGY*) or directly in the code. The latter can be accomplished by explicitly representing the search space as a *SearchTree* from the homonymous Curry module and using strategy dependent operations on the tree to obtain actual results. For example, we get:

```
kics2> someValueWith bfsStrategy (pathToSingle graph1 2 5)
[2,1,4,5]
kics2> someValueWith dfsStrategy (pathToSingle graph1 2 5)
[2,1,0,3,4,5]
```

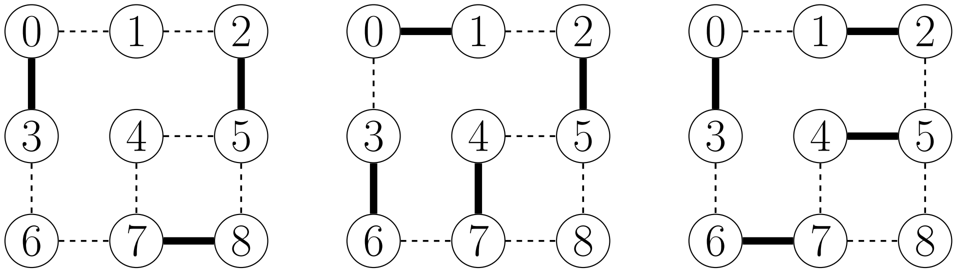
Note that these two results correspond precisely to what the respective graph-theoretic strategies yield. In fact, in the above implementation one can decide between choosing a successor and descending the recursive call (DFS) or checking other successors first and descending afterwards (BFS). However, both strategies will find all paths eventually, so that the above observation holds for the first result, but not necessarily for subsequent results (this can be observed using set functions, but not with *someValueWith*, because the latter is deterministic).

```
kics2> set3With bfsStrategy pathToSingle graph1 2 5
(Values [[2, 1, 4, 5], [2, 1, 0, 3, 4, 5]])
kics2> set3With dfsStrategy pathToSingle graph1 2 5
(Values [[2, 1, 0, 3, 4, 5], [2, 1, 4, 5]])
```

Since we check whether a vertex has been visited before by hand, the condition that the vertices of every resulting *Path* are distinct is maintained. Using an implementation based upon the constrained constructor pattern one can disallow the creation of invalid paths. However, the *search* itself requires access to previously visited vertices to avoid (infinite) repetition and thus its implementation would remain essentially the same as above. We omit the smart constructor approach for the sake of simplicity.

4 Maximum Matchings

A more sophisticated application of non-deterministic path search is the algorithm for finding maximum matchings. A matching in a symmetric graph is a set $M \subseteq E$ that is symmetric and functional[2]. In other words a matching consists of edges that do not share common vertices. A matching is called *maximum matching* iff there is no matching with a strictly greater cardinality. Clearly, maximum matchings are not necessarily unique, since they are maximal elements with respect to cardinality. Figure 3 shows some examples of matchings in the example graph from Figure 2, where the bold lines show matching edges and the dashed lines are non-matching edges.



A matching. A max. matching. Another max. matching.

Fig. 3. Examples of matchings.

---

[2] I.e. $\forall x, y, z \in V : (x, y) \in M \wedge (x, z) \in M \Rightarrow y = z$.

There are several natural applications for maximum matchings, like assigning jobs to people, processes to machines or, more classically, spouses to one another. In all cases one may wish to make as many one-to-one assignments as possible, which translates directly into the search for a maximum matching. All of these examples can be modelled with so-called *bipartite* graphs, which are graphs that allow a partition of vertices into two sets, such that all edges of the graph connect only vertices from two different sets. For bipartite graphs there is a simple imperative algorithm that computes maximum matchings, while the algorithms for the non-bipartite case is significantly more sophisticated [9].

The Berge theorem [3] characterises maximum matchings and provides an algorithm for finding such matchings. We state it exactly as in [6].

Theorem 1 (Characterisation of maximum matchings, Berge). *Let $M \subseteq E$ be a matching. Let $\oplus$ denote the symmetric difference. For a path p we denote the set of the edges along p by $E(p)$. A path is called M-augmenting iff it starts and ends in a vertex that is not contained in some edge of M and alternates between edges of $E \setminus M$ and those of M. Then we have*

1. *If there are no M-augmenting paths in G, then M is a maximum matching.*
2. *If there is an M-augmenting path p in G, then $M \oplus E(p)$ is a larger matching.*

This theorem can be used for an actual computation of a maximum matching. Starting with the empty matching one searches for an augmenting path and if such a path exists the current matching is expanded. If on the other hand no such path exists, the current matching is already a maximum matching.

For the remainder of this section we assume all graphs to be symmetric, in particular this concerns all graph arguments in functions. This condition can (and should!) be checked in a proper application, but to avoid additional code that is not necessary for the presentation, we assume this check to have been performed beforehand implicitly.

Let us first deal with the matching augmentation. Given a matching M and an augmenting path p Berge's theorem states that $M \oplus E(p)$ is a larger matching. Once we have computed this matching, we will check whether there is an augmenting path, using $M \oplus E(p)$ instead of M, which requires the computation of $E \setminus (M \oplus E(p))$. As we have stated in [6] it is simple to see that

$$E \setminus (M \oplus E(p)) = (E \setminus M) \oplus E(p) \ ,$$

so that both M and $E \setminus M$ are updated in the same fashion. Suppose that

```
xorBiEdges :: Graph → [Edge] → Graph
```

traverses a list of edges and for every edge it adds its undirected version (i.e. both directions) to the graph if the edge is not already present in the graph and removes both directions otherwise. Then the augmentation update can be implemented as follows, where we assume that m is the current matching and *notM* is its complement in E (i.e. *notM* $= E \setminus m$).

```
augmentBy :: Path → Graph → Graph → (Graph, Graph)
augmentBy path m notM = (m ‘xorBiEdges‘ es, notM ‘xorBiEdges‘ es) where
  es = zip path (tail path)
```

Without logic means checking the existence of an augmenting path and finding such a path in the positive case is quite technical[3]. Using logic means however, we can specify an augmenting path by first dealing with alternating paths and then adding the conditions for the first and last vertex. Suppose we have a target set of vertices $ts :: VertexSet$, a starting vertex s and a list of graphs $(g : gs) :: [Graph]$ that we want to traverse in sequence. Then an alternating path from s to ts through $g : gs$ exists iff $s \in ts$ holds or there is a successor i of s in g and there exists an alternating path from i to ts through $gs +\!\!+ [g]$. The intention is that we will use the list $[notM, m]$ in our application, where m is the current matching and $notM$ is its complement in E. Except for the traversal of multiple graphs in a cyclic order[4], the above specification is very similar to the one that specified the existence of a path. Similarly, the corresponding function is easy to modify to return an alternating path as well.

```
alternatingPath :: [Graph] → Vertex → VertexSet → Path
alternatingPath grs from ts = find empty from grs where
  find vis s (g : gs)
     | s ‘inSet‘ ts                      = [s]
     | isEdge g s i ∧ ¬ (i ‘inSet‘ vis) = s : find (insert s vis) i (gs ++ [g]) where i free
```

With this function we can find augmenting paths, too. Let us assume that we have a function at hand that computes those vertices of a symmetric graph which do not have any neighbours (i.e. successors, due to symmetry).

```
noSuccessors :: Graph → VertexSet
```

By definition an augmenting path starts and ends in a vertex that is not covered by the matching and since it is a path the start vertex should also be different from the end vertex, which is not guaranteed by the *alternatingPath* function from above. Fortunately, this is easily corrected by simply excluding the start vertex from the target vertex set. Again, we assume that m is a matching and $notM$ is its complement in E.

```
augmentingPath :: Graph → Graph → Path
augmentingPath m notM | s ‘inSet‘ unc = alternatingPath [notM, m] s (s ‘remove‘ unc)
  where unc = noSuccessors m
        s free
```

This function is prototypical by design – guessing a possible start vertex and trying to find a path is obviously not the most efficient way of finding an augmenting path. Instead, one could either modify the search function in a fashion

---

[3] The usual procedure is to implement a modified breadth-first search, which explicitly alternates between two graphs.

[4] We have implemented the cyclic list traversal inefficiently by adding the first element to the end of the list for demonstration purposes only. It is simple to replace this implementation by either functional lists or queues, both of which allow adding an element to the end in constant time.

that searches from a set of vertices instead of a single one or to use an explicitly parallel search strategy. These improvements lead to a less declarative look-and-feel, which is why we use the above version.

If there is no augmenting path in the graph then this function does not return any value. We use negation as failure with set functions to obtain a function that repeats the search for an augmenting path until there is no such path left.

```
maximumMatching :: Graph → Graph
maximumMatching g = go (emptyGraph (graphSize g), g) where
  go (m, notM) | isEmpty ps = m
               | otherwise  = go (augmentBy (chooseValue ps) m notM)
              where ps = set2 augmentingPath m notM
```

The function *graphSize* returns the number of vertices in the graph, *emptyGraph* creates an empty graph of a given size, *isEmpty* :: *Values a* → *Bool* checks whether the list of values is empty or not and *chooseValue* :: *Values a* → *a* nondeterministically chooses a value from the list of all values. Additionally, we can parametrise the *maximumMatching* function by a search strategy that is passed to *set2With* which is a version of *set2* that is parametrised by a search strategy.

We test the above function in the interactive mode of KiCS2 on the example graph G_2 from Figure 2. The function *graphEdges* :: *Graph* → [*Edge*] computes the edges of a graph and we use it for an uncluttered output.

```
kics2> graphEdges (maximumMatching graph2)
[(0,1),(1,0),(2,5),(3,6),(4,7),(5,2),(6,3),(7,4)]
More values? [Y(es)/n(o)/a(ll)] Y
<< four more times the same result >>
More values? [Y(es)/n(o)/a(ll)] Y
[(0,1),(1,0),(2,5),(3,6),(5,2),(6,3),(7,8),(8,7)]
More values? [Y(es)/n(o)/a(ll)] n
```

Observe that the first results are the first maximum matching from Figure 3.

As we have mentioned before, one typically distinguishes the cases of bipartite and non-bipartite graphs, because the search for an augmenting path is simpler in the former case. The above implementation does not rely on the graph being bipartite and will in fact work for non-bipartite graphs too, even those where usual algorithms will fail. The essence of this failure is known to be that every search is guided by some vertex ordering and in the non-bipartite case one can always create examples where a vertex is marked as visited prematurely, thus excluding this vertex from possible further searches. This problem cannot occur in the bipartite case – if there is an augmenting path, it can be found with every vertex ordering. The above implementation however, uses all possible vertex orderings and thus always finds a path if it exists. Clearly, this comes at the price of not being efficient (polynomial), but it is still interesting to observe that the function itself still yields the correct results, but only its complexity changes.

Another interesting observation is that such an implementation is very well suited for presentation, particularly in teaching. We have stated before that maximum matchings are not necessarily unique and the above function can be used interactively in KiCS2 to find different maximum matchings. Similarly, for any given matching there might be several augmenting paths and one can again check that the choice of a particular path does not matter for the maximality of the result. In graphs with unique maximum matchings the confluence of the algorithm is observable by considering all solutions and removing duplicates. In the graph from Figure 1 there is exactly one maximum matching, but it can be found with the above function in 184 ways[5]. This demonstration of confluence in again interesting in teaching to indicate that even a possibly non-deterministic algorithm can yield a deterministic result.

Finally, we point out that there are more efficient algorithms that compute maximum matchings, namely the Hopcroft-Karp algorithm [13] and the Mucha-Sankowski [15] algorithm. The latter is based upon Gaussian elimination and not directly related to a search for paths. The former algorithm, however, computes in every augmentation step not just a single augmenting path, but a set of shortest, pairwise disjoint augmenting paths that is maximal with respect to inclusion. A function that computes such a set can be implemented as a combination of two searches (first a breadth-first search, followed by a modified depth-first search, cf. [7]). Interestingly, the algorithm still contains a very similar non-deterministic component as above, since there can be several sets that are all maximal and again, the correctness of the output (and sometimes even the output itself) does not depend on a particular choice of such a set. The actual implementation is not particularly difficult, but it *explicitly* relies on the fact that the BFS returns a graph that is the union of all shortest paths from the source set to the target set, rather than the property that some path has been found using BFS *implicitly.*

5 Maximal Flows in Networks

A problem that is conceptually related to maximum matchings is that of a maximal flow through a network. A network is a quadruple $N = ((V, E), s, t, c)$ that consists of an asymmetric graph[6] (V, E), two designated vertices $s, t \in V$ (where s is called *source* and t is called *sink*) such that $s \neq t$ and a capacity function[7] $c : E \to \mathbb{N}$. To avoid unnecessary brackets whenever X is a set and $h : E \to X$, we write $h(x, y)$ instead of $h((x, y))$. For any function $g : E \to \mathbb{N}$ let

$$\partial g : V \to \mathbb{Z}, \quad v \mapsto \left(\sum_{w \in N_{\to}(v)} g(v, w) \right) - \left(\sum_{w \in N_{\leftarrow}(v)} g(w, v) \right) .$$

The function ∂g measures for every vertex the difference between the amount of all outgoing values and the incoming values. A *flow* is a function $f : E \to \mathbb{N}$ that

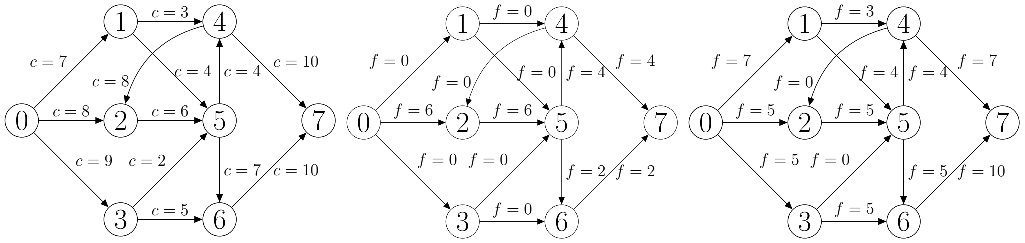
---

[5] Using *liftIO length* $\circ$ *values2list* $\circ$ *set1 maximumMatching* instead of just the *maximumMatching* function yields the number of successfully found matchings.

[6] That is that for all $v, w \in V$ such that $(v, w) \in E$, we have $(w, v) \notin E$.

[7] Typically, one chooses $c : E \to \mathbb{Q}_{\geq 0}$, but since only finite graphs are considered, it is possible to multiply c by the least common multiple of its image values and, if necessary, to divide them later.

satisfies $f \leq c$ (pointwise) and for all $v \in V \setminus \{ s, t \}$ we have $\partial f(v) = 0$. This is known as the Kirchhoff's law "what goes is, must come out". The *value of a flow* f is defined as $|f| := \partial f(s)$ and a maximal flow is a flow that has a maximal flow value. In typical applications there are no edges leading into the source, which then allows the intuition that the flow value is the amount of goods that is sent through the network from the source.



A network. A flow. A maximal flow.

Fig. 4. Examples of a network and flows.

Flow problems are related (among many others) to routing problems, where one wishes to send a certain amount of goods through different distribution lines that have a limited capacity only (e.g. traffic or electrical current). The Kirchhoff law then simply states that there is no loss of goods along the way. There has been extensive research on finding maximal flows (cf. [11, 10, 7] for overviews and results) and efficient algorithms are known. We consider the original algorithm and variations thereof. The original algorithm for finding maximal flows is due to a theorem by Ford and Fulkerson, which is based upon [14].

Theorem 2 (Characterisation of maximal flows, Ford & Fulkerson). *Let $N = ((V, E), s, t, c)$ be a network and $f : E \to \mathbb{N}$ a flow. Let*

$$c_f : E \cup E^{-1} \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} c(v, w) - f(v, w) & : (v, w) \in E \\ f(w, v) & : \textit{otherwise} \end{cases}$$

and $E_f := \{ e \in E \cup E^{-1} \mid c_f(e) > 0 \}$. We call c_f the residual capacity w.r.t. f. *Then the following hold:*

(1) If there is no path from s to t in (V, E_f), then f is a maximal flow.
(2) If p is a path from s to t in (V, E_f), let $\varepsilon := \min \{ c_f(e) \mid e \in E(p) \}$ and

$$f_p : E \to \mathbb{N}, \quad (v, w) \mapsto \begin{cases} f(v, w) + \varepsilon & : (v, w) \in E(p) \cap E \\ f(v, w) - \varepsilon & : (w, v) \in E(p) \cap E \\ f(v, w) & : \textit{otherwise.} \end{cases}$$

Then f_p is a flow and $|f_p| = |f| + \varepsilon$ (p is called a flow-augmenting path*).*

This theorem is very similar to the Berge theorem from the previous section. In fact, it is known that the problem of finding maximum matchings in bipartite graphs can be solved using a particular instance of the flow problem. However, this technique works only for bipartite graphs in which an explicit bipartition is known, while the presented strategy does not require an explicit bipartition.

The theorem of Ford and Fulkerson provides an algorithm for finding maximum flows, which checks for the existence of a path and improves the flow in the positive case. When searching for any path, the algorithm is known as the Ford-Fulkerson algorithm, which is not necessarily polynomial in the graph size. When searching for shortest paths, this algorithm is known as the Edmonds-Karp algorithm [8] and has a complexity that is polynomial in graph size. Assuming a deterministic choice of the first element of a list of non-deterministically found augmenting paths, this fact can be reflected in Curry by specifying an explicit strategy for the path search[8]. An additional variation of the algorithm is finding a set of paths from s to t that are disjoint up to s and t, such that the set is maximal with respect to inclusion and use all paths from this set for an improvement. This is a version of the Dinitz [7] algorithm and is more efficient than the Edmonds-Karp algorithm, since it is possible to implement the search for such a set in a fashion that is just as complex as finding a single path.

For an implementation we consider capacities and flows to be functions from $V \times V$ to $\mathbb{N}$, which yield zeroes on $(V \times V) \setminus E$. For any $g, h : V \times V \to \mathbb{Z}$ set

$$\mathrm{swap}(g) : V \times V \to \mathbb{Z}, \quad (x, y) \mapsto g(y, x)$$

$$g \sqcap h : V \times V \to \mathbb{Z}, \quad e \mapsto \begin{cases} h(e) & : g(e) \neq 0 \\ 0 & : \text{otherwise.} \end{cases}$$

Then "swap" is an uncurried version of the function *flip* and $\sqcap$ is the left-forgetful intersection of functions. Let $\oplus$, $\ominus$ be the pointwise addition and subtraction of functions respectively and $\bullet$ the multiplication of a function with a constant.

From now on we assume that c is a capacity and f is a flow. The residual capacity c_f as in the Ford-Fulkerson theorem can be computed as

$$c_f := c \ominus f \oplus \mathrm{swap}(f) \ .$$

Indeed, for every $v, w \in V$ we find that due to the asymmetry at most one of the two values $f(v, w)$ and $f(w, v)$ can be non-zero, which yields

$$\begin{aligned} (c \ominus f \oplus \mathrm{swap}(f))\,(v, w) &= c(v, w) - f(v, w) + f(w, v) \\ &= \begin{cases} c(v, w) - f(v, w) & : (v, w) \in E \\ f(w, v) & : (v, w) \in E^{-1} \\ 0 & : \text{otherwise} \end{cases} = c_f(v, w) \ , \end{aligned}$$

---

[8] However, we use a non-deterministic choice of an augmenting path to be able to observe the different choices.

where the last step is only true up to the extension of c_f to $V \times V$. Now let p be a path from s to t in (V, E_f) and let $\varepsilon := \min \{ c_f(e) \mid e \in E(p) \}$. Then set

$$\sigma_p : V \times V \to \mathbb{Z}, \quad e \mapsto \begin{cases} 1 & : e \in E(p) \\ 0 & : \text{otherwise,} \end{cases}$$

i.e. σ_p is the characteristic function of $E(p)$ in $V \times V$, and

$$u_p := \varepsilon \bullet (c \sqcap (\sigma_p \ominus \mathrm{swap}(\sigma_p)) \ .$$

The value u_p is a “point-free” version of the flow update indicated by the Ford-Fulkerson theorem: the term $\sigma_p \ominus \mathrm{swap}(\sigma_p)$ produces a function that yields 1 along the edges on p and -1 along all the reversed edges along p. The intersection produces a function that is 1 along the edges along p which are contained in E and -1 on those that are contained in E^{-1}. One easily verifies that $f_p = f \oplus u_p$. With this we can compute as follows, where all of the arithmetic rules below follow immediately from their pointwise counterparts.

$$\begin{aligned} c_{f_p} &= c \ominus f_p \oplus \mathrm{swap}(f_p) = c \ominus (f \oplus u_p) \oplus \mathrm{swap}(f \oplus u_p) \\ &= c \ominus f \ominus u_p \oplus \mathrm{swap}(f) \oplus \mathrm{swap}(u_p) = (c \ominus f \oplus \mathrm{swap}(f)) \ominus u_p \oplus \mathrm{swap}(u_p) \\ &= c_f \ominus u_p \oplus \mathrm{swap}(u_p) \ . \end{aligned}$$

Thus we can update the flow and the residual capacity using only the changes provided by the path, which reduces the number of necessary computations. To implement paths with values along traversed edges, we use the data type

```
data Path a = Final Vertex | From Vertex a (Path a)
```

and assume a function *toEdges* :: *Path a* → [(*Edge*, *a*)] to be at hand that collects all edges along the path with their corresponding values. We then model capacities and flows using finite maps *FM* with (*Vertex*, *Vertex*) keys and *Int* values[9].

```
type EdgeMap = FM (Vertex, Vertex) Int
```

The functions (⊕), (⊖), (⊓) :: *EdgeMap* → *EdgeMap* → *EdgeMap*, (•) :: *Int* → *EdgeMap* → *EdgeMap* and *swap* :: *EdgeMap* → *EdgeMap* are rather simple to define using standard operations on *FiniteMap*s (e.g. (⊕) = *plusFM_C* (+)). We can then implement the flow augmentation as follows, where the first argument is an augmenting path, the second one denotes the original capacity, the third one is the current residual capacity and the fourth one is the current flow.

```
augmentBy :: Path Int → EdgeMap → EdgeMap → EdgeMap → (EdgeMap, EdgeMap)
augmentBy p c c_f f = ((c_f ⊖ u_p) ⊕ swap u_p, f ⊕ u_p) where
```

---

[9] We could define a data type for natural numbers as well, but then manual conversion between naturals and integers requires some additional overhead; since this implementation is for demonstration, only, we assume the correct usage.

```
  u_p   = eps • (c ⊓ (σ_p ⊖ swap σ_p))             -- the update
  eps   = minlist (map snd edges)                  -- the minimum along the path
  σ_p   = fromList (map (λ(e, _) → (e, 1)) edges)  -- the characteristic function
  edges = toEdges p
```

Note that the actual result is an exact copy of the computations from above. The maximisation function can then be realised in a fashion very similar to the one we used for matchings, but explicitly parametrised over a search strategy. This implementation adds an additional non-deterministic component, namely the choice of the actual augmenting path. The strategy for this choice is given by the top-level search strategy.

```
data Network = Network Graph Vertex Vertex EdgeMap

maximalFlowWith :: Strategy (Path Int) → Network → EdgeMap
maximalFlowWith str (Network _ s t c) = go (c, empty) where

  go (c_f, f) | isEmpty ps = f
              | otherwise = go (augmentBy (chooseValue ps) c c_f f)
            where ps = set1With str findAugmenting c_f

  findAugmenting = augmenting s t
```

All that remains is the *augmenting* function. In essence, it is another path search, but this time we use the capacity map to check for existing edges, because edges in the residual graph exist iff their capacity is positive.

```
augmenting :: Vertex → Vertex → EdgeMap → Path Int
augmenting s t capacity = go (emptyFM (<)) s where

  go vis from | from ≡ t                        = Final from
              | cfi > 0 ∧ ¬ (from `inSet` vis) = From from cfi (go (insert from vis) i)
             where i free
                   cfi = capacity ! (from, i)

(!) :: FiniteMap a Int → a → Int
m ! key = lookupFMWithDefault m 0 key
```

The non-determinism is again enclosed in the path search. Just as was the case with matchings, maximal flows are usually not unique and the presented implementation can be used to find all possibilities. Still, every maximal flow has the same flow value and this fact can be observed by defining the ∂ function.

Again, we test our implementation with the example network from Figure 4 and wrap the call in the function *showEdgeMap* :: *EdgeMap* → *String* that pretty-prints key-value pairs as *key* → *value*.

```
kics2> showEdgeMap (maximalFlowWith bfsStrategy) network1
(2,5) → 5, (0,2) → 5, (0,1) → 7, (1,4) → 3, (0,3) → 5, (1,5) → 4, (5,6) → 5,
(4,7) → 7, (3,6) → 5, (5,4) → 4, (6,7) → 10
More values? [Y(es)/n(o)/a(ll)] y
(2,5) → 3, (0,2) → 3, (0,1) → 7, (1,4) → 3, (0,3) → 7, (1,5) → 4, (5,6) → 5,
(4,7) → 7, (3,6) → 5, (3,5) → 2, (5,4) → 4, (6,7) → 10
More values? [Y(es)/n(o)/a(ll)] n
```

The first flow is exactly the maximal flow from Figure 4 and the second one demonstrates that maximal flows can variate in their edges as well as the flow values along the edges. Clearly, both flows have a flow value of 17.

6 Discussion

We have demonstrated how to apply non-deterministic path computations to compute the solution to some selected graph problems. The presented functions are not the most efficient ones by design, but intended as prototypes for demonstration. This prototypical approach has the additional advantage of being simple and declarative. Clearly, several parts of our implementations can be improved or described in a more declarative or more efficient fashion, but our focus is on the non-deterministic path computations, which are the essence of all the described algorithms.

The overall strategy of all computations in this paper can be considered as the computation of the preimages of a given function which needs to be maximised. For instance in case of flows this function is $|\cdot| : F \to \mathbb{N},\ f \mapsto |f|$, where F is the set of all flows in a given network. Similarly, every improvement step is a preimage computation for the function $\mathit{improve} : P \to F,\ p \mapsto f_p$ where P is the set of all flow-augmenting paths with respect to a given flow f. It is interesting to note that every preimage choice is a branching point in the overall computation and that every new choice can allow different branches that will still lead to the same final result. In the case of maximum matchings two different maximum matchings are distinct only in one or more edges, for flows we can observe significantly more variation, since not only the edges that have non-zero flow can be different, but even different non-zero flow values for the same edge are possible.

Being able to observe such differences is interesting in its own right, but can be of particular interest in teaching. In case of the maximum matching function there is no difference between the search strategies. The maximal flow function on the other hand behaves differently, as we have stated above, and the choice of a depth-first strategy combined with an "unlucky" vertex ordering yields a non-polynomial complexity, while the very same concrete program with a breadth-first search strategy is in a completely different complexity class.

In our applications, results may be computed repeatedly through different branches. Since set functions are implemented using lists, removing duplicates is possible but costly, since a straight-forward graph comparison takes $\mathcal{O}(|E|)$ operations. For matchings this is slightly better, since every matching has only $\mathcal{O}(|V|)$ edges, which makes the naïve duplicate removal less inefficient. Still, with focus on experimentation and teaching, an inefficient duplicate removal is still rather simple, because the Curry function $\mathit{nub} :: [\alpha] \to [\alpha]$ removes duplicates from a given list (of ground terms).

The presented implementation and the ideas behind it are not exclusive to Curry, but passing search strategies to a set function is already a built-in feature of Curry and particularly KiCS2. However, KiCS2 translates Curry programs to Haskell and using non-deteminism monads (see [4]) and replacing logic variables by overlapping rules (as in [1]) one can obtain a purely functional implementation. Such an implementation should be portable to every other language that supports higher-order functions. It should not be too difficult to translate the above functions into a relational setting, e.g. in Prolog, after removing the encapsulated non-determinism. Additionally, negations need to be handled with care in general, but in our case we used negations only to check for "being not contained in the visited vertices", which can be inlined and implemented by hand without explicit negation. However, Prolog uses a built-in DFS, which disallows the parametrisation over the search strategy. It is difficult to estimate how declarative and structurally complex the resulting program will be in another language. While we omitted some auxiliary functions, we still consider our code to be rather simple and straight-forward.

Acknowledgements: I thank Rudolf Berghammer for encouraging this work, Frank Huch for sparking its idea and the highly appreciated feedback of the reviewers.

References

1. S. Antoy and M. Hanus. Overlapping Rules and Logic Variables in Functional Logic Programs. In *ICLP*, pages 87–101, 2006.
2. S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proc. of the 11th International ACM SIGPLAN Conference on Principle and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
3. Claude Berge. Two Theorems in Graph Theory. In *PNAS*, volume 43 (9), pages 842–844. National Academy of Sciences, 1957.
4. B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming Functional Logic Programs into Monadic Functional Programs. In Julio Mariño, editor, *WFLP*, volume 6559 of *LNCS*, pages 30–47. Springer, 2010.
5. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A New Compiler from Curry to Haskell. In *Proc. of the 20th Int. Workshop on Functional and (Constraint) Logic Prog. (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
6. Nikita Danilenko. Using Relations to Develop a Haskell Program for Computing Maximum Bipartite Matchings. In Wolfram Kahl and Timothy G. Griffin, editors, *RAMICS*, volume 7560 of *LNCS*, pages 130–145. Springer, 2012.
7. Yefim Dinitz. Dinitz' Algorithm: The Original Version and Even's Version. In Oded Goldreich, Arnold L. Rosenberg, and Alan L. Selman, editors, *Essays in Memory of Shimon Even*, volume 3895 of *LNCS*, pages 218–240. Springer, 2006.
8. J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, 19(2):248–264, 1972.
9. Jack Edmonds. Paths, trees and flowers. *Canadian J. Math.*, 17:449–467, 1965.
10. A. V. Goldberg and S. Rao. Beyond the Flow Decomposition Barrier. *J. ACM*, 45(5):783–797, 1998.
11. A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum-Flow Problem. *J. ACM*, 35(4):921–940, 1988.
12. Michael Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
13. J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
14. L. R. Ford Jr. and D. R. Fulkerson. Maximal Flow through a Network. *Canadian J. Math.*, 8:399–404, 1956.
15. M. Mucha and P. Sankowski. Maximum Matchings via Gaussian Elimination. In *FOCS*, pages 248–255. IEEE Computer Society, 2004.

Curry without Success

Sergio Antoy[1] Michael Hanus[2]

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

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

Abstract. Curry is a successful, general-purpose, functional logic programming language that predefines a singleton type *Success* explicitly to support its logic component. We take the likely-controversial position that without *Success* Curry would be as much logic or more. We draw a short history and motivation for the existence of this type and justify why its elimination could be advantageous. Furthermore, we propose a new interpretation of rule application which is convenient for programming and increases the similarity between the functional component of Curry and functional programming as in Haskell. We outline some related theoretical (semantics) and practical (implementation) consequences of our proposal.

1 Motivation

Recently, we coded a small Curry [16] module to encode and pretty-print JSON formatted documents [14]. The JSON format encodes floating point numbers with a syntax that makes the decimal point optional. Our Curry System prints floating numbers with a decimal point. Thus, integers, which were converted to floats for encoding, were printed as floats, e.g., the integer value 2 was printed as “2.0”. We found all those point-zeros annoying and distracting and decided to get rid of them. To avoid messing with the internal representation of numbers, and risking losing information, our algorithm would look for “.0” at the end of the string representation of a number in the JSON document and remove it. In the *List* library, we found a function, *isSuffixOf*, that tells us whether to drop the last two characters, but we did not find a function to drop the last 2 characters. How could we do that?

In the library we found the usual *drop* and *take* functions that work at the beginning of a string s. Hence, we could reverse s, drop 2 characters, and reverse again. We were not thrilled. Or we could take from s the first $n-2$ characters, where n is the length of s. We were not thrilled either. In both cases, conceptually the string is traversed 3 times (probably in practice too) and extraneous functions are invoked. Not a big deal, but there must be a better way. Although the computation is totally *functional*, we started to think *logic*.

Curry has this fantastic feature called *functional patterns* [4]. With it, we could code the following:

```
fix_int (x ++ ".0") = x                                    (1)
```

Now we were thrilled! This is compact, simple and obviously correct. Of course, we would need a rule for cases in which the string representation of a number does not end in ".0", i.e.:

```
fix_int (x ++ ".0") = x
fix_int x = x
```
(2)

Without the last rule *fix_int* would fail on a string such as "2.1". With the last rule the program would be incorrect because *both* rules would be applied for a number that ends in ".0". The latter is a consequence of the design decision that established that the order of the rules in a program is irrelevant—a major departure of Curry from popular functional languages. One of the reasons of this design decision is *Success*.

2 History

Putting it crudely, a functional logic language is a functional language extended with logic variables. The only complication of this extension is what to do when some function f is applied to some unbound logic variable u. There are two options, either to residuate on u or to narrow u. Residuation suspends the application of *f*, and computes elsewhere in the program in hopes that this computation will narrow u so that the suspended application of f can continue. Narrowing instantiates u to values that sustain the computation. For example, given the usual concatenation of lists:

```
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```
(3)

Narrowing $u \mathbin{++} t$, where u is unbound and t is any expression, instantiates u to `[]` and $u' \mathbin{:} us$ and continues these computations either one at the time or concurrently depending on the control strategy.

In early functional logic languages [1, 17], in the tradition of logic programming, only predicates (as opposed to any function) are allowed to instantiate a logic variable. In the early days of Curry, we were not brave enough. Indiscriminate narrowing, such as that required for (1), which is based on (3), was uncharted territory and we decided that all functions would residuate except a small selected group called *constraints*. These functions are characterized by returning a singleton type called *Success*.

Narrowing has the remarkable property of solving equations [23]. Indeed, the rule in (1) works by solving an equation by narrowing. An application *fix_int*(s), where s is a string, attempts to solve $s = x$++`".0"`. A solution, "the" if any exists, gives the desired result x. Returning *Success* rather than *Boolean*, as the constrained equality does, had the desirable consequence that we would not "solve" an equation by deriving it to *False*, but had the drawback of introducing a new variant of equality, implemented in Curry by the operation "=:=", and the undesirable consequence that "some expressions were more equal than others" [19].

As a consequence of our hesitation, narrowing was limited to the arguments of constraints—a successful model well-established by Prolog. However, this model is at odds with a language with a functional component with normal (lazy) order of evaluation. Without functional nesting, there is no easy way to tell whether or not some argument of some constraint should be evaluated. Consider a program to solve the 8-queens puzzle:

```
permute x y = ...  succeed if y is a permutation of x
safe y = ...  succeed if y is a safe placement of queens
```
(4)

A solution of the puzzle is obtained by *permute* [1..8] y `&&` *safe* y, where y is likely a free variable. The constraint *permute* fully evaluates y upon returning even if *safe* may only look at the first two elements and determine that y is not safe.

This prompted the invention of “non-deterministic functions”, i.e., a function-like mechanism, that may return more than one value for the same combination of arguments, but is used as an ordinary function. With this idea, Example (4) is coded as:

```
permute x = ...  return any permutation of x
safe y = ...  as before
```
(5)

In this case, a solution of the puzzle is obtained by *safe* y *where* y $=$ *permute* [1..8]. Since y is nested inside *safe*, *permute* [1..8] can be evaluated only to the extent needed by its context. A plausible encoding of *permute* is:

```
permute [] = []
permute (x : xs) = nd_insert x (permute xs)

nd_insert x ys = x : ys
nd_insert x (y : ys) = y : nd_insert x ys
```
(6)

The evaluation of *permute* [1..8] produces any permutation of [1..8] *only if* both rules defining *nd_insert* are applied when the second argument is a non-empty list. Thus, the well-established convention of functional languages that the first rule that matches an expression is the only one being fired had to be changed in the design of Curry.

3 Proposed Adjustments

Our modest proposal is to strip the *Success* type of any special meaning. Since *Success* is isomorphic to the *Unit* type, which is already defined in the *Prelude*, probably it becomes redundant. Future versions of the language could keep it for backward compatibility, but deprecate it.

The first consequence of this change puts in question the usefulness of “=:=”, the constrained equality. Equations can be solved using the Boolean equality “==” bringing Curry more in line with functional languages. To solve an equation by narrowing, we simply evaluate it using the standard rules defining Boolean equality. For example, below we show these rules for a polymorphic type *List*:

```
[] == [] = True
(x:xs) == (y:ys) = x==y && xs==ys
[] == (_:_) = False
(_:_) == [] = False
```
(7)

However, we certainly want to avoid binding variables with instantiations that derive an equation to *False* since these bindings are not solutions. Avoiding these bindings is achieved with the following operation:

```
solve True = True
```
(8)

and wrapping an equation with *solve*, i.e., to solve $x = y$, we code *solve* $(x == y)$. Nostalgic programmers could redefine “=:=” as:

```
x =:= y = solve (x == y)                                   (9)
```

When an equation occurs in the condition of a rule, the intended behavior is implied, i.e., the rule is fired only when the condition is (evaluates to) *True*.

In a short paragraph above, we find the symbols "=", "==" and "=:=". The first one is the (mathematical) equality. The other two are (computational) approximations of it with subtle differences. Our proposal simplifies this situation by having only "==" as the implementation of "=", as in functional languages, without sacrificing any of Curry's logic aspects.

The second consequence of our proposal is to review the rule selection strategy, i.e., the order, or more precisely its lack thereof, in which rules are fired. We have already hinted at this issue discussing example (2). Every rule that matches the arguments and satisfies the condition of a call is non-deterministically fired. A motivation for the independence of rule order was discussed in example (6). The ability of making non-deterministic choices is essential to functional logic programming, and it must be preserved, but it can be achieved in a different way.

The predefined operation "?" non-deterministically returns either of its arguments. This operation allows us to express non-determinism in a way different from rules with overlapping left-hand sides [3]. For instance, the non-determinism of the operation *nd_insert* in (6) can be moved from the left-hand sides of its defining rules to the right-hand sides as in the following definition:

```
nd_insert x ys = (x : ys) ? nd_insert2 x ys
nd_insert2 x (y : ys) = y : nd_insert x ys                 (10)
```

Indeed, some Curry compilers, like KiCS2 [8], implement this transformation.

The definition of Curry at the time of this writing [16] establishes that the order of the rules defining an operation is irrelevant. The same holds true for the conditions of a rule, except in the case in which the condition type is Boolean, and for flexible case expressions. Our next proposal is to change this design decision of Curry. Although this is somehow independent of our first proposal to remove the *Success* type, it is reasonable to consider both proposals at once since both simplify the use of Curry.

We propose to change the current definition of rule application in Curry as follows. To determine which rule(s) to fire for an application $t = f(t_1, \dots, t_n)$, where f is an operation and $t_1, \dots, t_n$ are expressions, use the following strategy:

1. Scan the rules of f in textual order. An unconditional rule is considered as a conditional rule with condition *True*.
2. Fire the first rule whose left-hand side matches the application t and whose condition is satisfied. Ignore any remaining rule.
3. If no rule can be applied, the computation fails.
4. If a combination of arguments is non-deterministic, the previous points are executed independently for each non-deterministic choice of the combination of arguments. In particular, if an argument is a free variable, it is non-deterministically instantiated to all its possible values.

As usual in a non-strict language like Curry, arguments of an operation application are evaluated as they are demanded by the operation's pattern matching and condition.

However, any non-determinism or failure during argument evaluation is not passed inside the condition evaluation. A precise definition of “inside” is in [6, Def. 3]. This is quite similar to the behavior of set functions to encapsulate internal non-determinism [6]. Apropos, we discuss in Section 5 how to exploit set functions to implement this concept.

Before discussing the advantages and implementation of this concept, we explain and motivate the various design decisions taken in our proposal. First, it should be noted that this concept distinguishes non-determinism outside and inside a rule application. If the condition of a rule has several solutions, this rule is applied if it is the first one with a true condition. Second, the computation proceeds non-deterministically with all the solutions of the condition. For instance, consider an operation to look up values for keys in an association list:

```
lookup key assoc
  | assoc == (_ ++ [(key,val)] ++ _)
  = Just val
  where val free
lookup _ _ = Nothing
```

(11)

If we evaluate *lookup* 2 $[(2,14),(3,17),(2,18)]$, the condition of the first rule is solvable. Thus, we ignore the remaining rules and apply only the first rule to evaluate this expression. Since the condition has the two solutions $\{\mathit{val} \mapsto 14\}$ and $\{\mathit{val} \mapsto 18\}$, we yield the values *Just* 14 and *Just* 18 for this expression. Note that this is in contrast to Prolog’s if-then-else construct which checks the condition only once and proceeds just with the first solution of the condition. If we evaluate *lookup* 2 $[(3,17)]$, the condition of the first rule is not solvable but the second rule is applicable so that we obtain the result *Nothing*.

On the other hand, non-deterministic arguments might trigger different rules to be applied. Consider the expression *lookup* $(2 \mathbin{?} 3)$ $[(3,17)]$. Since the non-determinism in the arguments leads to independent rule applications (see item 4), this expression leads to independent evaluations of *lookup* $2\,[(3,17)]$ and *lookup* $3\,[(3,17)]$. The first one yields *Nothing*, whereas the second one yields *Just* 17.

Similarly, free variables as arguments might lead to independent results since free variables are equivalent to non-deterministic values [5]. For instance, the expression *lookup* 2 xs yields the value *Just* v with the binding $\{xs \mapsto (2,v):\_\}$, but also the value *Nothing* with the binding $\{xs \mapsto \texttt{[]}\}$ (as well as many other solutions). Again, this behavior is different from Prolog’s if-then-else construct which performs bindings for free variables inside the condition independently of its source. In contrast to Prolog, our design supports completeness in logic-oriented computations even in the presence of if-then-else.

The latter desirable property has also implications for the handling of failures occurring when arguments are evaluated. For instance, consider the expression “*lookup 2 failed*” (where *failed* is a predefined operation which always fails whenever it is evaluated). Because the evaluation of the condition of the first rule fails, the entire expression evaluation fails instead of returning the value *Nothing*. This is motivated by the fact that we need the value of the association list in order to check the satisfiability of the condition, but this value is not available.

To see the consequences of an alternative design decision, consider the following contrived definition of an operation that checks whether its argument is the unit value `()` (which is the only value of the unit type):

```
isUnit x | x == () = True
isUnit _ = False
```
(12)

In our proposal, the evaluation of *isUnit failed* fails. In an alternative design (like Prolog's if-then-else construct), one might skip any failure during condition checking and proceed with the next rule. In this case, we would return the value *False* for the expression *isUnit failed*. This is quite disturbing since the (deterministic!) operation *isUnit*, which has only one possible input value, could return two values: *True* for the call *isUnit* `()` and *False* for the call *isUnit failed*. Moreover, if we call this operation with a free variable, like *isUnit x*, we obtain the single binding $\{x \mapsto \texttt{()}\}$ and value *True* (since free variables are never bound to failures). Thus, either our semantics would be incomplete for logic computations or we compute too many values. In order to get a consistent behavior, we require that failures of arguments demanded for condition checking lead to failures of evaluations.

Changing the meaning of rule selection from an order-independent semantics to a sequential interpretation is an important change in the design of Curry. However, this change is relevant only for a relatively small amount of existing programs. First, most of the operations in a functional logic program are inductively sequential [2], i.e., they are defined by rules where the left-hand sides do not overlap. Hence, the order of the rules does not affect the definition of such operations. Second, rules defined with traditional Boolean guards residuate if they are applied to unknown arguments, i.e., it is usually not intended to apply alternative conditions to a given call. This fits to a sequential interpretation of conditions. Moreover, our proposal supports the use of conditional rules in a logic programming manner with unknown arguments, since this "outside" non-determinism does not influence the sequential condition checking.

Nevertheless, there are also cases where a sequential interpretation of rules is not intended, e.g., in a rule-oriented programming style, which is often used in knowledge-based or constraint programming. Although we argued that one can always translate overlapping patterns into rules with non-overlapping patterns by using the choice operator "`?`", the resulting code might be less readable. Finally, we have to admit that in a declarative language ignoring the order of the rules is more elegant though not always as convenient. Hence, a good compromise would be a compiler pragma that allows to choose between a sequential or an unordered interpretation of overlapping rules.

4 Advantages

In this section we justify through exemplary problems the advantages of the proposed changes.

Example 1. With the proposed semantics, (2) is a simple and obviously correct solution of the problem, discussed in the introduction, of "fixing" the representation of integers in a JSON document.

Example 2. As in the previous example, our proposed semantics is compatible with functional patterns. Hence, (11) can be more conveniently coded as:

```
lookup key (_ ++ [(key,val)] ++ _) = Just val
lookup _ _ = Nothing                                    (13)
```

Example 3. Consider a *read-eval-print* loop of a functional logic language such as Curry. A top-level expression may contain free variables that are declared by a *free clause* such as in the following example:

```
x ++ y == [1,2,3,4] where x, y free                     (14)
```

Of course, the *free clause* is absent if there are no free variables in the top-level expression. The free variables, when present, are easily extracted with a “deep” pattern as follows:

```
breakFree (exp++" where "++wf++" free"))
  = (exp,wf)
breakFree exp                                           (15)
  = (exp,"")
```

For this code to work, the rules of `breakFree` must be tried in order and the second one must be fired only if the first one fails.

Example 4. Suppose that World Cup soccer scores are represented in either of the following forms:

```
GER _:_ USA
GER 1:0 USA                                             (16)
```

where the first line represents a game not yet played and the second one a game in which the digits are the goals scored by the adjacent team (a single digit suffices in practice). The following operation parses scores:

```
parse (team1++" _:_ "++team2) = (team1,team2,Nothing)
parse (team1++[' ',x,':',y,' ']++team2)
  | isDigit x && isDigit y                              (17)
  = (team1,team2, Just (toInt x,toInt y))
parse _ = error "Wrong format!"
```

Example 5. The *Dutch National Flag* problem [13] has been proposed in a simple form to discuss the termination of rewriting [12]. A formulation in Curry of this simple form is equally simple:

```
dnf (x++[White,Red]++y) = dnf (x++[Red,White]++y)
dnf (x++[Blue,Red]++y) = dnf (x++[Red,Blue]++y)    (18)
dnf (x++[Blue,White]++y) = dnf (x++[White,Blue]++y)
```

However, (18) needs a termination condition to avoid failure. With our proposed semantics, this condition is simply:

```
dnf x = x                                               (19)
```

With the standard semantics, a much more complicated condition is needed.

5 Implementation

A good implementation of the proposed changes in the semantics of rule selection requires new compilation schemes for Curry. However, an implementation can also be obtained by a transformation over source programs when existing advanced features of Curry are exploited. This approach provides a reference semantics that avoids explicitly specifying all the details of our proposal, in particular, the subtle interplay between condition solving and non-determinism and failures in arguments. Hence, we define in this section a program transformation that implements our proposed changes within existing Curry systems.

Initially, we discuss the implementation of a single rule with a sequence of conditions, i.e., a program rule of the form

$$\begin{array}{lllll} l & \mid & c_1 & = & e_1 \\ & \vdots & & & \\ & \mid & c_k & = & e_k \end{array} \tag{20}$$

According to our proposal, if the left-hand side l matches a call, the conditions $c_1, \ldots, c_k$ are sequentially evaluated. If c_i is the first condition that evaluates to `True`, all other conditions are ignored so that (20) becomes equivalent to

$$l \;\mid\; c_i = e_i$$

Note that the subsequent conditions are ignored even if the condition c_i also evaluates to *False*. Thus, the standard translation of rules with multiple guards, as defined in the current report of Curry [16], i.e., replacing multiple guards by nested if-then-else constructs, would yield a non-intended semantics. Moreover, non-determinism and failures in the evaluation of actual arguments must be distinguished from similar outcomes caused by the evaluation of the condition, as discussed in Section 3.

All these requirements call for the encapsulation of condition checking where "inside" and "outside" non-determinism are distinguished and handled differently. Fortunately, recent developments for encapsulated search in functional logic programming [6, 10] provide an appropriate solution of this problem. For instance, [10] proposes an encapsulation primitive *allValues* so that the expression $(\mathit{allValues}\ e)$ evaluates to the set of values of e where only internal non-determinism inside e is considered. Thus, we can use the following expression to check a condition c with our intended meaning:[3]

$$\texttt{if notEmpty (allValues (solve } c\texttt{)) then } e_1 \texttt{ else } e_2 \tag{21}$$

According to [10], the meaning of this expression is as follows:

1. Test whether there is some evaluation of c to *True*.
2. If the test is positive, evaluate e_1.
3. If there is no evaluation of c to *True*, evaluate e_2.

The semantics of *allValues* ensures that non-determinism and failures caused by expressions not defined inside c, in particular, parameters of the left-hand side l of the operation, are not encapsulated. The Curry implementations PAKCS [15] and KiCS2 [8] provide set functions [6] instead of *allValues* which allows the implementation of this conditional in a similar way.

---

[3] [10] defines only an operation *isEmpty*. Hence we assume that *notEmpty* is defined by the rule $\mathit{notEmpty}\ x = \mathit{not}\ (\mathit{isEmpty}\ x)$.

Our expected semantics demands that a rule with a solvable condition be applied for *each* true condition, in particular, with a possible different binding computed by evaluating the condition. To implement this behavior, we assume an auxiliary operation *ifTrue* that combines a condition and an expression. This operation is simply defined by

$$\texttt{ifTrue True x = x} \tag{22}$$

Then we define the meaning of (20) by the following transformation:

$$\begin{aligned}
l\ \texttt{=}\ & \texttt{if notEmpty (allValues (solve } c_1\texttt{))} \\
& \quad \texttt{then (ifTrue } c_1\ e_1\texttt{) else} \\
& \vdots \\
& \texttt{if notEmpty (allValues (solve } c_k\texttt{))} \\
& \quad \texttt{then (ifTrue } c_k\ e_k\texttt{) else failed}
\end{aligned} \tag{23}$$

There are obvious simplifications of this general scheme. For instance, if $c_k = \mathit{True}$, as frequently is the case, the last line of (23) becomes e_k.

This transformation scheme is mainly intended as the semantics of sequential condition checking rather than as the final implementation (similarly to the specification of the meaning of guards in Haskell [20]). A sophisticated implementation could improve the actual code. For instance, each condition c_i is duplicated in our scheme. Moreover, it seems that conditions are always evaluated twice. However, this is not the case if a lazy implementation of encapsulated search via *allValues* or set functions is used, as in the Curry implementation KiCS2 [10]. If c_i is the first solvable condition, the emptiness test for $(\mathit{allValues}\ c_i)$ can be decided after computing a *first* solution. In this case, this solution is computed again (and now also all other solutions) in the then-part in order to pass its computed bindings to e_i. Of course, a more primitive implementation might avoid this duplicated evaluation.

Next we consider the transformation of a sequence of rules

$$\begin{aligned}
& l_1\ \ r_1 \\
& \vdots \\
& l_k\ \ r_k
\end{aligned} \tag{24}$$

where each left-hand side l_i is a pattern $f\ p_{i1} \ldots p_{in_i}$ for the same function f and each r_i is a sequence of condition/expression pairs of the form "$|\ c\ =\ e$" as shown in (20).[4] We assume that the pattern arguments p_{ij} contain only constructors and variables. In particular, functional patterns have been eliminated by moving them into the condition using the function pattern unification operator "=:<=" (as shown in [4]). For instance, rule (1) is transformed into

$$\texttt{fix\_int xs | (x ++ ".0") =:<= xs = x} \tag{25}$$

Finally, we assume that subsequent rules with the same pattern (up to variable renaming) are joined into a single rule with multiple guards. For instance, the rules (2) can be joined (after eliminating the functional pattern) into the single rule

---

[4] In order to handle all rules in a unique manner, we consider an unconditional rule "l_i = e_i" as an abbreviation for the conditional rule "l_i `|  True` = e_i".

```
fix_int xs
  | (x ++ ".0") =:<= xs = x
  | True                = xs
```
(26)

Now we distinguish the following cases:

- The patterns in the left-hand sides $l_1, \ldots, l_k$ are inductively sequential [2], i.e., the patterns can be organized in a tree structure such that there is always a discriminating (inductive) argument: since there are no overlapping left-hand sides in this case, the order of the rules is not important for the computed results. Therefore, no further transformation is necessary in this case. Note that most functions in typical functional logic programs are defined by inductively sequential rules.
- Otherwise, there might be overlapping left-hand sides so that it is necessary to check all rules in a sequential manner. For this purpose, we put the pattern matching into the condition so that the patterns and conditions are checked together. Thus, a rule like

  $$f\, p_1 \ldots p_n \;\texttt{|}\; c = e$$

  is transformed into

  $$\begin{array}{l} f\, x_1 \ldots x_n \;\texttt{| (\textbackslash}p_1 \ldots p_n\; \texttt{->}\; c\;\texttt{)}\; x_1 \ldots x_n \\ \qquad\qquad\texttt{= (\textbackslash}p_1 \ldots p_n\; \texttt{-> ifTrue}\; c\; e\texttt{)}\; x_1 \ldots x_n \end{array}$$

  where $x_1, \ldots, x_n$ are fresh variables (the extension to rules with multiple conditions is straightforward). Using this transformation, we obtain a list of rules with identical left-hand sides which can be joined into a single rule with multiple guards, as described above.

For instance, the definition of `fix_int` (26) is transformed into

```
fix_int xs =
  if notEmpty (allValues (solve (x++".0"=:<=xs)))
  then (ifTrue (x++".0" =:<= xs) x)
  else xs
```
(27)

For an example of transforming rules with overlapping patterns, consider an operation that reverses a two-element list and leaves all other lists unchanged:

```
rev2 [x,y] = [y,x]
rev2 xs    = xs
```
(28)

According to our transformation, this definition is mapped into (after some straightforward simplifications):

```
rev2 xs =
  if notEmpty (allValues (\[x,y] -> True) xs)
  then (\[x,y] -> [y,x]) xs
  else xs
```
(29)

Thanks to the logic features of Curry, one can also use this definition to generate appropriate argument values for *rev2*. For instance, if we evaluate the expression *rev2 xs* (where *xs* is a free variable), the Curry implementation KiCS2 [8] has a finite search space and computes the following bindings and values:

```
{xs = []} []
{xs = [x1]} [x1]
{xs = [x1,x2]} [x2,x1]
{xs = (x1:x2:x3:x4)} (x1:x2:x3:x4)
```

As mentioned above, the transformation presented in this section is intended to serve as a reference semantics for our proposed changes and to provide a prototypical implementation. There are various possibilities to improve this implementation. For instance, if the right-hand side expressions following each condition are always evaluable to a value, i.e., to a finite expression without defined operations, the duplication of the code of the condition as well as the potential double evaluation of the first solvable condition can be easily avoided. As an example, consider the following operation that checks whether a string contains a non-negative float number (without an exponent):

```
isNNFloat (f1 ++ "." ++ f2)
          | all isDigit f1 && all isDigit f2 = True     (30)
isNNFloat _= False
```

If *c* denotes the condition

```
(f1 ++ "." ++ f2) =:<= s &&
all isDigit f1 && all isDigit f2                        (31)
```

by functional pattern elimination [4], program (30) is equivalent to

```
isNNFloat s | c = True
isNNFloat _ = False                                     (32)
```

Applying our transformation, we obtain the following code with the duplicated condition *c*:

```
isNNFloat s =
  if notEmpty (allValues (solve c))
  then (ifTrue c True)                                  (33)
  else False
```

Since the expressions on the right-hand side are always values (*True* or *False*), we can put these expressions into the sets computed by *allValues*. Then the check for a solvable condition becomes equivalent to check the non-emptiness of these value sets so that we return non-deterministically some value of this set.[5] This idea can be implemented by the following scheme which does not duplicate the condition and evaluates it only once (the actual code can be simplified but we want to show the general scheme):

```
isNNFloat s =
  if notEmpty s1 then chooseValue s1 else False
 where                                                  (34)
  s1 = allValues (ifTrue c True)
```

Note that this optimization is not applicable if it is not ensured that the right-hand side expressions are always evaluable to values. For instance, consider definition (28) of

---

[5] The predefined operation *chooseValue* non-deterministically returns some value of a set.

rev2 and the expression *head* (*rev2* $[nv, 0]$), where nv is an expression without a value (e.g., failure or non-termination). With our current transformation (29), we compute the value 0 for this expression. However, the computation of the set of all values of (*rev2* $[nv, 0]$) w.r.t. the first rule defining *rev2* does not yield any set since the right-hand side $[0, nv]$ has no value. This explains our transformation scheme (23) which might look complicated at a first glance.

However, there is another transformation to implement overlapping rules like (28) with our intended semantics. If the rules are unconditional, one can "complete" the missing constructor patterns in order to obtain an inductively sequential definition. For the operation *rev2*, we obtain the following definition:

```
rev2 [x,y] = [y,x]
rev2 [] = []
rev2 [x] = [x]
rev2 (x:y:z:xs) = x:y:z:xs
```

(35)

Since a case can be more efficiently executed than an encapsulated computation, this alternative transformation might lead to larger but more efficient target code.

6 Related Work

Declarative programming languages support the construction of readable and reliable programs by partitioning complex procedures into smaller units—mainly using case distinction by pattern matching and conditional rules. Since we propose a new interpretation of case distinctions for functional logic programs, we compare our proposal with existing ones with similar objectives.

The functional programming language Haskell [20] provides, similarly to Curry, also pattern matching and guarded rules for case distinctions. Our proposal for a new sequential interpretation of patterns increases the similarities between Curry and Haskell. Although Curry provides more features due to the built-in support to deal with non-deterministic and failing computations, our proposal is a conservative extension of Haskell's guarded rules, i.e., it has the same behavior as Haskell when non-determinism and failures do not occur. To see this, consider a program rule with multiple conditions:

$$\begin{array}{lll} l & \mid & c_1 \;=\; e_1 \\ & \vdots & \\ & \mid & c_k \;=\; e_k \end{array} \qquad (36)$$

Since non-deterministic computations do not exist in Haskell and failures lead to exceptions in Haskell, we assume that, if this rule is applied in Haskell to an expression e, there is one condition c_i which evaluates to *True* and all previous conditions $c_1, \ldots, c_{i-1}$ evaluate to *False*. If we consider the same rule translated with the transformation scheme (23), obviously each condition *notEmpty* (*allValues* (*solve* c_j)) reduces to *False* for $j = 1, \ldots, i-1$ and to *True* for $j = i$. Thus, the application of this rule reduces e to (*ifTrue* c_i e_i) and, subsequently, to e_i, as in Haskell.

The logic programming language Prolog [11] also supports pattern matching and, for sequential conditions, an if-then-else construct of the form "c `->` e_1 `;` e_2". Although Prolog can deal, similarly to Curry, with non-deterministic and failing computations, the if-then-else construct usually restricts the completeness of the search space due to cutting the choice points created by c before executing e_1. Hence, only the first solution of c is used to evaluate e_1. Furthermore, inside and outside non-determinism is not distinguished so that variables outside the condition c might be bound during its evaluation. This has the effect that predicates where if-then-else is used are often restricted to a particular mode. For instance, consider the re-definition of *rev2* (28) as a predicate in Prolog using if-then-else:

```
rev2(Xs,Ys) :- Xs=[X,Y] -> Ys=[Y,X] ; Ys=Xs.          (37)
```

If we try to solve the goal $\mathit{rev2}(\mathit{Xs}, \mathit{Ys})$, Prolog yields the single answer $\mathit{Xs} = [A, B]$, $\mathit{Ys} = [B, A]$. Thus, in contrast to our approach, all other answers are lost.

Various encapsulation operators have been proposed for functional logic programs [7] to encapsulate non-deterministic computations in some data structure. Set functions [6] have been proposed as a strategy-independent notion of encapsulating non-determinism to deal with the interactions of laziness and encapsulation (see [7] for details). We can also use set functions to distinguish successful and non-successful computations, similarly to negation-as-failure in logic programming, exploiting the possibility to check result sets for emptiness. When encapsulated computations are nested and performed lazily, it turns out that one has to track the encapsulation level in order to obtain intended results, as discussed in [10]. Thus, it is not surprising that set functions and related operators fit quite well to our proposal.

Computations with failures for the implementation of an if-then-else construct and default rules in functional logic programs have been also explored in [18, 22]. In these works, an operator, *fails*, is introduced to check whether every reduction of an expression to a head-normal form is not successful. The authors show that this operator can be used to define a single default rule, but not the more general sequential rule checking of our approach. Moreover, nested computations with failures are not considered by these works. As a consequence, the operator *fails* might yield unintended results if it is used in nested expressions. For instance, if we use *fails* instead of *allValues* to implement the operation *isUnit* defined in (12), the evaluation of *isUnit failed* yields the value *False* in contrast to our intended semantics.

7 Conclusions

We proposed two changes to the current design of Curry. The first one concerns the removal of the type *Success* and the related constraint equality "=:=". This simplifies the language since it relieves the programmer from choosing the appropriate equality operator. The second one concerns a strict order in which rules and conditions are tried to reduce an expression. This makes the language design more similar to functional languages like Haskell so that functional programmers will be more comfortable with Curry. Nevertheless, the logic programming features, like non-determinism and evaluating functions with unknown arguments, are still applicable with our new semantics. This distinguishes our approach from similar concepts in logic programming which simply cuts alternatives.

However, our proposal comes also with some drawbacks. We already mentioned that in knowledge-based or constraint programming applications, a sequential ordering of rules is not intended. Hence, a compiler pragma could allow the programmer to choose between a sequential or an unordered interpretation of overlapping rules.

A further drawback of our approach concerns the run-time efficiency. We argued that solving "==" equations by narrowing with standard equational rules can replace the constraint equality "=:=". Although this is true from a semantic point of view, the constraint equality operator "=:=" is more efficient from an operational point of view. If x and y are free variables, the equational constraint "x=:=y" is deterministically solved by binding x to y (or vice versa), whereas the Boolean equality "x==y" is solved by non-deterministically instantiating x and y to identical values. The efficiency improvement of performing bindings is well known, e.g., it is benchmarked in [9] for the Curry implementation KiCS2. On the other hand, the Boolean equality "x==y" is more powerful since it can also solve negated conditions, i.e., evaluate "x==y" to *False* by binding x and y to different values.

Hence, for future work it is interesting to find a compromise, e.g., performing variable bindings when "x==y" should be reduced to *True* without any surrounding negations. A program analysis could be useful to detect such situations at compile time.

Finally, the concurrency features of Curry must be revised. Currently, concurrency is introduced by the concurrent conjunction operator "`&`" on constraints. If the constraint type *Success* is removed, other forms of concurrent evaluations might be introduced, e.g., in operators with more than one demanded argument ("==", "+",...), explicit concurrent Boolean conjunctions, or only in the I/O monad similarly to Concurrent Haskell [21].

Despite all the drawbacks, our proposal is a reasonable approach to simplify the design of Curry and make it more convenient for the programmer.

8 Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249.

References

1. H. Aït-Kaci and A. Podelski. Towards a meaning of LIFE. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 255–274. Springer LNCS 528, 1991.
2. S. Antoy. Definitional trees. In *Proc. of the 3rd International Conference on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.
3. S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at `http://cs.pdx.edu/~antoy/homepage/publications/alp97/full.pdf`.
4. S. Antoy and M. Hanus. Declarative programming with function patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pages 6–22. Springer LNCS 3901, 2005.
5. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proceedings of the 22nd International Conference on Logic Programming (ICLP 2006)*, pages 87–101. Springer LNCS 4079, 2006.

6. S. Antoy and M. Hanus. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pages 73–82. ACM Press, 2009.
7. B. Braßel, M. Hanus, and F. Huch. Encapsulating non-determinism in functional logic computations. *Journal of Functional and Logic Programming*, 2004(6), 2004.
8. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
9. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. Implementing equational constraints in a functional language. In *Proc. of the 15th International Symposium on Practical Aspects of Declarative Languages (PADL 2013)*, pages 125–140. Springer LNCS 7752, 2013.
10. J. Christiansen, M. Hanus, F. Reck, and D. Seidel. A semantics for weakly encapsulated search in functional logic programs. In *Proc. of the 15th International Symposium on Principle and Practice of Declarative Programming (PPDP'13)*, pages 49–60. ACM Press, 2013.
11. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog - the standard: reference manual*. Springer, 1996.
12. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1/2):69–116, 1987.
13. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
14. The JSON Data Interchange Standard.
15. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2013.
16. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
17. J. Lloyd. Programming in an integrated functional and logic language. *Journal of Functional and Logic Programming*, 1999(3):1–49, 1999.
18. F.J. López-Fraguas and J. Sánchez-Hernández. A proof theoretic approach to failure in functional logic programming. *Theory and Practice of Logic Programming*, 4(1):41–74, 2004.
19. G. Orwell. *Animal Farm: A Fairy Story*. Secker and Warburg, London, UK, 1945.
20. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
21. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 295–308. ACM Press, 1996.
22. J. Sánchez-Hernández. Constructive failure in functional-logic programming: From theory to implementation. *Journal of Universal Computer Science*, 12(11):1574–1593, 2006.
23. J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity, and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

A Partial Evaluator for Curry

Michael Hanus and Björn Peemöller

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany
{mh|bjp}@informatik.uni-kiel.de

Abstract. We present a partial evaluator for functional logic programs written in Curry. In contrast to previous approaches to the partial evaluation of functional logic programs, we take into account the features used in contemporary Curry programs, in particular, non-deterministic operations and recursive `let` expressions. For this purpose, we base our partial evaluator on FlatCurry, an intermediate language for the representation of Curry programs. We sketch our approach and present initial benchmarks of our implementation.

1 Introduction

Partial evaluation of programs is a technique to anticipate the evaluation of computations once at compile time instead of performing them (possibly several times) at run time. This is possible if some part of the input data, also called *static* data, is known at compile time. In this case, some parts of the program are evaluated so that a *residual* program, i.e., a specialized version of the original one, is returned. Since some computations have been performed at compile time, the run time of the specialized program could be considerably decreased. The static data does not need to be some user input, but can also be subexpressions in the original program. *Offline* partial evaluators obtain information about static data from a separate static analysis phase (binding-time analysis), whereas *online* partial evaluators obtain this information on the fly and propagate it during the partial evaluation process. In this work we follow the online partial evaluation approach.

Partial evaluation has already been studied for different kinds of programming languages, like functional languages, logic languages, as well as for combined functional logic languages. An interesting aspect of the partial evaluation of functional logic programs is the fact that the effects of supercompilation [23] can be obtained by applying the operational semantics of the source language (narrowing) at partial evaluation time [5]: if a function is called with unknown arguments, narrowing instantiates these arguments such that the rules defining this function can be applied. Hence, one mainly needs to control the partial evaluator, i.e., avoiding infinite unfoldings and instantiations of logic variables, in order to obtain residual programs.

Thanks to this insight, partial evaluators for functional logic languages can be constructed with techniques similarly to the implementation of these languages. For instance, Albert et al. [3] proposed a partial evaluator for Curry [17] based on the intermediate language FlatCurry. Since FlatCurry makes the evaluation strategy of Curry programs explicit [1], the use of FlatCurry led to a partial evaluator able to optimize practical Curry programs. Unfortunately, when this partial evaluator was constructed, the use of non-deterministic operations, although proposed some years ago [14], was not well established. Therefore, the partial evaluation scheme was based on term rewriting and restricted to confluent programs, i.e., all operations were required to be deterministic, and recursive `let` expressions were also not taken into account. Thus, if this partial evaluator is applied to programs containing non-deterministic operations, which is a useful programming pattern in contemporary functional logic programs [6,8], the resulting programs are not semantically equivalent to the source programs.

In order to deal with realistic Curry programs, it is crucial for a partial evaluator to cover the full source language, including both logic features such as non-determinism and functional features such as recursive `let` expressions. Therefore, we extend in this work the partial evaluator of Albert et al. [3] to cover the full language of FlatCurry. In contrast to [3], we base our partial evaluator on an operational semantics [1] which is adequate for contemporary Curry programs.

We start with an introduction to the functional logic language Curry in Sect. 2 before we sketch the structure of the partial evaluator in Sect. 3. The partial evaluation scheme is presented in Sect. 4, whereas control issues are discussed in Sect. 5. We evaluate our implementation with some benchmarks in Sect. 6 before we conclude in Sect. 7.

2 Curry

We briefly review the basic concepts of the functional logic language Curry. More details can be found in recent surveys on functional logic programming [7,15] and in the language report [17].

The syntax of Curry [17] is close to Haskell [21], i.e., type variables and names of defined operations usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of an operation f to an expression e is denoted by juxtaposition ("$f\ e$"). In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. If the data type of Booleans and a negation operation are defined by

```
data Bool = False | True
not True  = False
not False = True
```

the expression "`not x where x free`" non-deterministically reduces to `False` with the binding `x = True`, and to `True` with the binding `x = False`. A further kind of non-determinism is supported in Curry by the *choice* operator "`?`", which can be considered as predefined by the overlapping rules

```
x ? _ = x
_ ? y = y
```

Thus, we can define a *non-deterministic operation* `coin` yielding the values `0` and `1` by

```
coin = 0 ? 1
```

If non-deterministic operations are used as arguments in other operations, a semantical ambiguity might occur. Consider the operation

```
double x = x + x
```

and the expression "`double coin`". If we evaluated this expression by term rewriting, we could have the reduction

```
double coin  →  coin + coin  →  0 + coin  →  0 + 1  →  1
```

leading to the unintended result `1`. Note that this result cannot be obtained with a strict reduction strategy where arguments are evaluated prior to the function calls. In order to avoid dependencies on the evaluation strategies and exclude such unintended results, Curry is based on the rewriting logic CRWL, proposed by González-Moreno et al. [14] as a logical (execution- and strategy-independent) foundation for declarative programming with non-strict and non-deterministic operations. This logic specifies the *call-time choice* semantics [18] where values of the arguments of an operation are determined before the operation is evaluated. In a lazy strategy, this can be enforced by sharing actual arguments. For instance, the expression above can be lazily evaluated provided that all occurrences of `coin` are shared so that all of them consistently reduce to either `0` or `1`.

3 Overview of the Partial Evaluator

Before describing the details of the partial evaluation process, we provide an overview of the partial evaluator and its usage. Since our partial evaluator is an extension of the first partial evaluator for Curry described in [3], our representation is oriented towards the original description.

Our partial evaluator is intended to specialize some parts of a given input program in order to create an optimized, residual program. In order to support the specification of expressions to be optimized, we assume that these expressions are annotated with `PEVAL`. For example, we assume a program which contains the function definition

```
main xs = map (twice square) xs
```

We can then annotate the main expression (or parts of it) as follows:

```
main xs = PEVAL (map (twice square) xs)
```

Actually, `PEVAL` is the identity function, i.e., it has the type `a → a`. As a consequence, annotations with `PEVAL` do not change the semantics of the original program. After annotating the program, the process of partial evaluation is fully automatic. The process itself consists of the following phases (depicted in Fig. 1):

1. The partial evaluator is called for a given program, containing annotated expressions as described above. This source program is converted into the standard intermediate representation for Curry programs, called FlatCurry (see Sect. 4.1).
2. The process continues by extracting the set of annotated expressions and creating a copy of the original program without annotations.
3. Both form the input for the partial evaluation phase, which is later described.
4. The output of the partial evaluation is a set of semantically equivalent, potentially more efficient expressions. These expressions are converted to new function definitions to allow reuse, a process called *renaming*.
5. The evaluation process tends to produce some "intermediate" functions which only pass their parameters to another function. Therefore, the process is finished by a *compression* phase which removes such intermediate functions by inlining and simplifies expressions to produce a more efficient and legible result.
6. Finally, the annotated expressions of the form `(PEVAL` e`)` are replaced with their (hopefully more efficient) equivalents e', where e' is the renaming of e. This optimized program is then stored as a FlatCurry program.

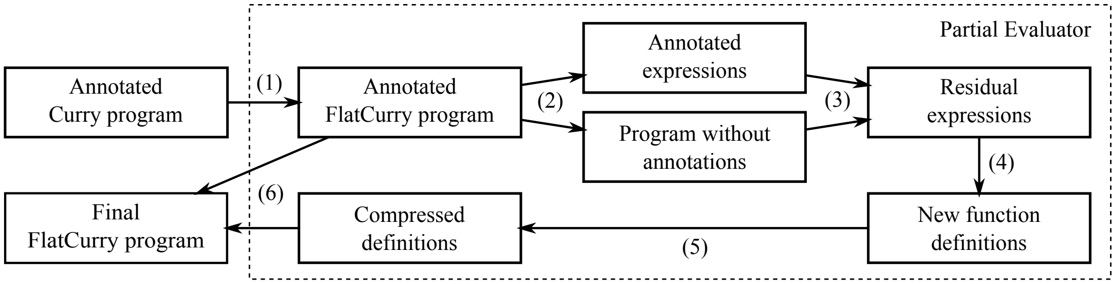


Fig. 1. Overview of the partial evaluation process with phases (1) to (6)

For instance, with the usual definitions of `map`, `twice`, and `square`, the example above is transformed into

```
main xs = map0 xs
map0 xs = case xs of []    → []
                     y:ys  → let z = (y*y) in (z*z) : map0 ys
```

so that the overhead of the higher-order operations `map` and `twice` is eliminated.

The fact that the partial evaluator internally operates on the FlatCurry format is no restriction, since this format is used by current Curry compilers anyway, e.g., PAKCS [16] or KiCS2 [11]. Hence, the partial evaluator can easily be incorporated into a compilation chain.

4 The Partial Evaluation Scheme

As already mentioned, the partial evaluator described in [3] lacks support for two language features, namely non-deterministic operations and `let` expressions. For instance, consider the definition

```
main = PEVAL (double coin)
```

w.r.t. the definitions of `coin` and `double` shown in Sect. 2. The partial evaluator [3] unfolds the call to `double` in the body of `main` to `(0 ? 1) + (0 ? 1)`, so that the residual program yields the values `0`, `1`, `1`, and `2` for `main`. However, according to the call-time choice semantics of Curry [14,18], the correct result would be `0` or `2` but not `1`. This problem arises from the residual semantics of the original partial evaluator, which is based on term-rewriting so that non-determinism in shared subexpressions is duplicated in the residual programs.

The second missing feature are (mutually recursive) `let` expressions, i.e., bindings where the variables to be bound might occur in the right-hand side of the bindings. For example, it is not possible to partially evaluate the program

```
ones = let ones = 1 : ones in ones
main = PEVAL (take 2 ones)
```

One might encounter that this does not impose a real restriction because recursive let-bindings could be interpreted by recursive function definitions (at the cost of some overhead). While this is possible for the example above, it is not whenever a non-deterministic value should be shared. For instance, consider the following program:

```
digits = let digits = (0 ? 1) : digits in digits
main   = PEVAL (take 2 digits)
```

$$
\begin{array}{rcll}
P & ::= & \overline{D_m} & \text{(program)} \\
D & ::= & f(\overline{x_n})\ \texttt{=}\ e & \text{(defined function)} \\
e & ::= & x & \text{(variable)} \\
  & \mid & c(\overline{e_k}) & \text{(constructor call)} \\
  & \mid & f(\overline{e_k}) & \text{(function call)} \\
  & \mid & \texttt{let}\ \{\ \overline{x_n\ \texttt{=}\ e_n}\ \}\ \texttt{in}\ e & \text{(recursive let binding)} \\
  & \mid & \texttt{let}\ \overline{x_n}\ \texttt{free in}\ e & \text{(free variables)} \\
  & \mid & e_1\ \texttt{?}\ e_2 & \text{(disjunction)} \\
  & \mid & \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} & \text{(case expression, } p_i \text{ pairwise different)} \\
p & ::= & c(\overline{x_n}) & \text{(constructor pattern)}
\end{array}
$$

Fig. 2. The FlatCurry representation of programs

Because of the `let` binding, the decision to bind `digit` to either `0` or `1` is shared, and, in consequence, the expression `main` evaluates to either `[0,0]` or `[1,1]`. If we replaced the definition of `digits` by a top-level operation, as in

```
digits = (0 ? 1) : digits
main   = PEVAL (take 2 digits)
```

the expression `main` would produce the additional results `[0,1]` and `[1,0]`. Thus, recursive `let` expressions cannot be transformed into operations but must be explicitly considered by a partial evaluator.

The usage of both features in contemporary Curry programs is the motivation for us to develop a new partial evaluator. In contrast to [3], we do not use a semantics based on term rewriting. Instead, we base our work on the natural semantics for FlatCurry proposed in [1] which is intended to specify the call-time choice semantics of non-deterministic operations by modeling a heap structure to express sharing. A similar semantics has been used in [13] in a partial evaluator for first-order functional programs. In contrast to our approach, non-determinism, which is essential for Curry, has not been considered there.

4.1 FlatCurry

FlatCurry is a simple intermediate language used by Curry compilers [11,16]. Moreover, it is also the basis of precise descriptions of the semantics of Curry [3] and semantics-based tools for Curry (e.g., [2,3,4]). The syntax of this representation is depicted in Fig. 2, where we denote a sequence of objects $o_1, \ldots, o_n$ by $\overline{o_n}$. A FlatCurry program P consists of a sequence of function definitions D such that each function must be defined by a single rule with a linear left-hand side, i.e., the variables $\overline{x_n}$ must be pairwise different. The right-hand side of a function definition is an expression e composed of variables (x, y, z, …), constructors (A, B, C, …), and function calls (f, g, h, …). In the following, we denote by ϕ a constructor c or a function f. For the sake of simplicity, we assume that literals occurring in the source program, like numbers or characters, are represented as nullary constructors. Additionally, we allow local (mutually recursive) bindings of variables, the introduction of free (logic) variables, disjunctions (to represent overlapping left-hand sides in the source language), and pattern matching. The patterns p_i in `case` expressions are required to be pairwise different and only consist of constructors applied to variables. In consequence, nested patterns in the source language are represented by nested `case` expressions. For example, the list concatenation `conc` is represented in FlatCurry as

```
conc(xs,ys) = case xs of { []    →  ys
                         ; z:zs  →  z : conc(zs,ys) }
```

Note that, in contrast to [1], we do not distinguish between *flexible* and *rigid* `case` expressions. Although they behave differently on free (logic) variables [17], this difference is not relevant for partial evaluation [3]. Furthermore, we omit the representation of external functions like arithmetics, which are implemented in the partial evaluator but do not play a significant role in the evaluation scheme. Finally, we do not consider higher-order applications in the syntax of FlatCurry since they can be represented by an operation $apply$ where partial applications are interpreted as constructor calls [3].

4.2 Natural Semantics

We base our partial evaluator on a variant of the operational semantics of FlatCurry [1], also referred to as the *natural semantics* of FlatCurry. The semantics uses a heap structure to specify sharing of expressions and computes the *(flat) value* of an expression which is either a logic variable (w.r.t. the associated heap) or a constructor applied to variables.

$$Heap = \mathcal{V} \to \{\text{free}, \blacksquare\} \uplus Exp \qquad\qquad Value ::= x \mid c(\overline{x_n})$$

A *heap* is a partial mapping from a set of variables $\mathcal{V}$ to either an expression (Exp is the set of expressions according to the syntax of FlatCurry), a special symbol “free” to represent a free variable,[1] or a symbol “$\blacksquare$” representing a black hole.[2] We denote the empty heap by $[]$, and the value associated to a variable x in a heap Γ by $\Gamma[x]$. $\Gamma[x \mapsto e]$ denotes a heap Γ' with $\Gamma'[x] = e$ and $\Gamma'[y] = \Gamma[y]$ for all $y \neq x$.

We use judgements of the form $\Gamma : e \Downarrow \Delta : v$ which express the fact that “the expression e under the heap Γ evaluates to the value v and the (possibly modified) heap Δ”. The basic inference rules of the natural semantics are depicted in Fig. 3. We briefly describe these rules and explain the differences to the original version of [1].

(Value) Evaluation of a value directly returns the value without modifying the heap.

(VarExp) This rule implements sharing of subexpressions. If a variable to be evaluated is bound to an expression, the expression is evaluated and its value is returned. In addition, the heap is updated with the value. During evaluation of the expression, the binding is replaced by $\blacksquare$, in contrast to [1]. This allows the detection of black holes and is necessary for the correctness of the semantics [10] (see also Appendix A for a detailed explanation).

(Flatten) To correctly implement sharing, arguments of function or constructor calls must be represented in the heap. This is usually achieved by a preprocessing step called *flattening* or *normalization* [1,19], but it can also be performed on demand.

---

[1] [1] represents free variables by circular `let` bindings of the form `let {`x `=` x`} in` e, but this prohibits the correct representation of such bindings occurring in the source code.

[2] We use a special symbol for black holes instead of simply removing the binding for a variable (as in [19]) in order to distinguish black holes from unbound variables.

$$\text{(Value)}\qquad \Gamma : v \Downarrow \Gamma : v \qquad \text{where } v = c(\overline{x_n}) \text{ or } v \in \mathcal{V} \text{ with } \Gamma[v] = \text{free}$$

$$\text{(VarExp)}\qquad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \qquad \text{where } e \notin \{\text{free}, \blacksquare\}$$

$$\text{(Flatten)}\qquad \frac{\Gamma[y \mapsto e_i] : \phi(x_1, \ldots, x_{i-1}, y, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v}{\Gamma : \phi(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v} \qquad \text{where } e_i \notin \mathcal{V}, y \text{ fresh}$$

$$\text{(Fun)}\qquad \frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \qquad \text{where } f(\overline{x_n}) = e \in P,\ \sigma = \{\overline{x_n \mapsto y_n}\}$$

$$\text{(Let)}\qquad \frac{\Gamma[\overline{y_k \mapsto \sigma(e_k)}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let \{ } \overline{x_k = e_k} \texttt{ \} in } e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_k \mapsto y_k}\}, \overline{y_k} \text{ fresh}$$

$$\text{(Or)}\qquad \frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1 \ ?\ e_2 \Downarrow \Delta : v} \qquad \text{where } i \in \{1, 2\}$$

$$\text{(Free)}\qquad \frac{\Gamma[\overline{y_n \mapsto \text{free}}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let } \overline{x_n} \texttt{ free in } e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

$$\text{(Select)}\qquad \frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}} \Downarrow \Theta : v} \qquad \text{where } \begin{array}{l} p_i = c(\overline{x_n}), \\ \sigma = \{\overline{x_n \mapsto y_n}\} \end{array}$$

$$\text{(Guess)}\qquad \frac{\Gamma : e \Downarrow \Delta[x \mapsto \text{free}] : x \qquad \Delta[x \mapsto \sigma(p_i), \overline{y_n \mapsto \text{free}}] : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \texttt{case } e \texttt{ of \{ } \overline{p_k \to e_k} \texttt{ \}} \Downarrow \Theta : v}$$
$$\text{where } i \in \{1, \ldots, k\}, p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

Fig. 3. Natural semantics

(Fun) This rule unfolds a function call, where the result is obtained by evaluation of the function's right-hand side. We assume that the program P is a global parameter of the calculus. Generally, whenever new variables are introduced by the program, we apply a renaming substitution σ to prohibit name clashes.

(Let) The bindings of a `let` construct are added to the heap after all variables have been renamed to fresh variable names.

(Or) This rule non-deterministically chooses one of the arguments to be futher evaluated. In consequence, this rule introduces non-determinism into the calculus itself.

(Free) Like variables bound to expressions, logic variables are renamed and afterwards bound in the heap.

(Select) For `case` expressions whose argument evaluates to a constructor-rooted term the right-hand side of the corresponding alternative is selected and evaluated.

(Guess) For `case` expressions whose argument evaluates to a logic variable, one of the alternatives is non-deterministically chosen to be evaluated. The variable is then bound to the corresponding pattern where the variables inside the pattern are also bound as logic variables.

4.3 Ensuring Termination

Following the general idea of partial evaluation of functional logic programs [5] as well as logic programs [20], we evaluate an annotated expression e with a (possibly incomplete) standard derivation $[\,] : e \Downarrow \Delta : e'$. In order to ensure the termination of the partial evaluation process, we defer the evaluation of some expressions. For example, consider the program

```
loop xs = loop xs
main xs = PEVAL (loop xs)
```

The evaluation of the expression `main` does not terminate due to the recursive function call to `loop`. To achieve termination of the partial evaluation process, we modify the natural semantics as follows:

1. The evaluation of an expression can be deferred to avoid non-termination.
2. An operation *proceed* is used to decide whether a function call should be unfolded or deferred.

This *residualizing natural semantics* is similar to [3,4] but more complex due to the use of a heap for sharing instead of term rewriting. Regarding the first modification, we extend the representation of values with a new symbol $\langle\!\langle \cdot \rangle\!\rangle$ which encloses expressions whose evaluation should be deferred.

$$Value \ ::= \ \ldots \mid \langle\!\langle e \rangle\!\rangle \quad \text{(annotated expression)}$$

This annotation directly corresponds to the `PEVAL` annotation in source programs. Second, we extend the inference system with the operation *proceed*, deciding whether a function call should be unfolded, and replace the rule Fun with:[3]

$$\text{(FunEval)} \quad \frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \quad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ proceed(\Gamma, f(\overline{y_n})) = \text{true} \end{array}$$

$$\text{(FunDefer)} \quad \Gamma : f(\overline{y_n}) \Downarrow \Gamma : \langle\!\langle f(\overline{y_n}) \rangle\!\rangle \quad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ proceed(\Gamma, f(\overline{y_n})) = \text{false} \end{array}$$

Approaches for the concrete definition of *proceed* will be discussed in Sect. 5.1. Furthermore, we extend the rule Value to also return deferred expressions unchanged and constrain the rule VarExp in that the value must not be a deferred expression. Finally, we add two more rules for deferred expressions where the annotation is lifted upwards:

$$\text{(VarDefer)} \quad \frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto e'] : \langle\!\langle x \rangle\!\rangle} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

$$\text{(CaseDefer)} \quad \frac{\Gamma : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \langle\!\langle \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

4.4 Dealing with Partial Information

In contrast to the evaluation performed in a standard interpreter, the partial evaluation process has to deal with partial knowledge in the form of unbound variables. For instance, if the right-hand side of a function declaration like `f x = PEVAL (g x)` should be evaluated, there is no binding information for the parameter variable `x`.

---

[3] Actually, the operation *proceed* also takes into account the context of reductions already performed, but we omit them here for the sake of simplicity.

A possible solution is to handle such unbound variables as logic variables, as done in [5], so that they are bound to appropriate values by the partial evaluator. Since it has been shown in [2] that the back-propagation of these bindings can lead to incorrect residual programs, [3] uses a residualizing semantics which represents such bindings by `case` expressions in the residual program. However, this is only necessary for unbound variables. Explicitly introduced logic variables are known to be free during the actual evaluation so that they can be bound during partial evaluation time. For instance, consider the expression

```
let x free in case x of { True  →  1 }
```

Here we can bind `x` to `True`, since this binding is not visible outside the scope of this expression, select the (single) branch as the value of the `case` expression, and continue by evaluating its right-hand side. In consequence, our implementation evaluates this expression to `1`, while the partial evaluator described in [3] cannot evaluate the expression any further.

Hence, we distinguish unbound variables from logic variables by *not binding* them in the heap. Furthermore, we assume that rule Value is also applicable to variables not bound in the heap so that unknown variables reduce to themselves. Thus, only the rules for `case` expressions have to be changed, where it is now also possible that the scrutinized value is an unknown variable. Following the idea of [3], we generate *residual case expressions* to defer the inspection of the variable to the run time of the specialized program. Therefore, we extend the definition of values to

$$Value \ ::= \ \ldots \mid \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to v_k}\ \} \qquad \text{(residual case expression)}$$

where the variable x inspected in the `case` expression is not bound in the corresponding heap. Because `case` expressions are now contained in the set of values, we also have to consider them as the value of a variable or an expression examined by another `case` expression. Hence, we add the following rules:

$$\text{(CaseUnbound)} \quad \frac{\varGamma : e \Downarrow \varDelta : x}{\varGamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \varDelta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}} \quad \text{where } x \notin dom(\varDelta)$$

$$\text{(CaseCase)} \quad \frac{\varGamma : e \Downarrow \varDelta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p'_j \to \langle\!\langle e'_j \rangle\!\rangle}\ \}}{\varGamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \varDelta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p'_j \to \left\langle\!\!\left\langle \begin{matrix} \texttt{case}\ e'_j\ \texttt{of} \\ \{\ \overline{p_k \to e_k}\ \} \end{matrix} \right\rangle\!\!\right\rangle}\ \}}$$

$$\text{(VarCase)} \quad \frac{\varGamma[x \mapsto \blacksquare] : e \Downarrow \varDelta : \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}{\varGamma[x \mapsto e] : x \Downarrow \varDelta[x \mapsto \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}] : x} \quad \text{where } e \notin \{\text{free}, \blacksquare\}$$

$$\text{(CaseVarCase)} \quad \frac{\varGamma : e \Downarrow \varDelta : x}{\varGamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \varDelta : \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p'_j \to \left\langle\!\!\left\langle \begin{matrix} \texttt{case}\ x\ \texttt{of} \\ \{\ \overline{p_k \to e_k}\ \} \end{matrix} \right\rangle\!\!\right\rangle}\ \}} \quad \text{where } \varDelta[x] = \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p'_j \to e'_j}\ \}$$

The general idea is to lift `case` expressions inspecting an unbound variable upwards and to defer the evaluation of the alternatives. Such deferred expressions are not further evaluated in the residual semantics but later extracted by the global iterative process as the initial expressions of a new specialization run. Because the alternatives are then evaluated independently, it will be possible to take the binding information of the `case` expression into account. For instance, if we consider the expression

```
case x of { True  →  not x }
```

a subsequent evaluation of the right-hand side `not x` may respect the binding of `x` to `True` and, thus, directly evaluate to `False`.

4.5 Dereferencing the Heap

After evaluating an expression to a residual value, this value might contain variables which are either free or bound to expressions in the corresponding heap. To be able to replace parts of the input program with residual values, these bindings have to be added to the values to form valid expressions, a process we call *dereferencing the heap*. Conceptually, for a given configuration $\Gamma : e$, we retrieve the set of variables transitively reachable from e and bound in Γ and add the corresponding bindings to the expression. For residual `case` expressions, we also respect the bindings represented by the `case` expression. The bindings are divided into logic variables (fv) and variables bound to expressions (bv) and added to the original expression:

$$drf(\Gamma, e) = \begin{cases} \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to drf(\Gamma[x \mapsto p_k], e_k)}\ \} & \text{if } e = \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \\ \langle\!\langle \texttt{let}\ fv(\Gamma, e)\ \texttt{free in let}\ \ bv(\Gamma, e)\ \ \texttt{in}\ e \rangle\!\rangle & \text{otherwise} \end{cases}$$

For instance, if we consider the configuration

$$[\texttt{y} \mapsto \text{free}] : \texttt{case x of \{True -> x; False -> y\}}$$

then dereferencing will produce the expression

$$\texttt{case x of \{ True} \to \langle\!\langle \texttt{let \{ x = True \} in x} \rangle\!\rangle \texttt{;False} \to \langle\!\langle \texttt{let y free in y} \rangle\!\rangle\ \texttt{\}}$$

5 Control

Our partial evaluation algorithm follows the general procedure of Alpuente et. al. [5], which is parametric w.r.t. an *unfolding rule* used to construct a finite derivation for an expression and an *abstraction operator* used to guarantee that only finitely many expressions are evaluated. The basic algorithm is depicted in Fig. 4 and works as follows. Given an input program P and a set of annotated expressions E, the algorithm starts by applying an unfolding rule which evaluates each expression according to the residual semantics presented in the previous section and extracts the results by drf. If there is more than one derivation in the residual semantics due to the non-deterministic inference rules Or and Guess, the different extracted results of the derivation are combined by the choice operator "`?`". If there is no derivation at all, the result is represented by the predefined operation `failed`. In the next step, an abstraction operator is applied to this set, adding the new expressions to the set of already evaluated expressions. This phase yields a new set which may need further evaluation, hence, this process is iteratively repeated until no more expressions are added to the set. This iteration is necessary for the correctness of partial deduction [20] in order to achieve a "closed" set of expressions

```
Input: A program $P$ and a set of expressions $E$
Output: A set of expressions $S$
$i := 0;\ E_0 := E;$
repeat
    $E' := unfold(E_i, P);$
    $E_{i+1} := abstract(E_i, E');$
    $i := i + 1;$
until $E_i = E_{i+1}$ (modulo renaming);
return $S := E_i$
```

Fig. 4. Basic algorithm for partial evaluation

that covers all expressions possibly occurring in the residual program. To generate the resulting program, the same unfolding rule has to be applied to the resulting set of expressions to generate the corresponding resultants, i.e., the rules of the residual program (in our implementation, this step is integrated into the algorithm). Finally, the set of generated resultants are compressed to eliminate intermediate and redundant functions (see [5] for details).

This procedure distinguishes two levels of control, namely the *local level*, managed by the unfolding rule to avoid infinite evaluations, and the *global level*, managed by the abstraction operator to avoid infinitely repetitions of the partial evaluation algorithm. To ensure termination of the whole process, both *local* and *global* termination is required.

5.1 Local Control

Termination of the unfolding rule directly corresponds to termination of the residual semantics presented in Sect. 4. For this purpose, the semantics has already been extended by an oracle $proceed(\Gamma, e)$ responsible for the decision whether a function call should be unfolded or not. There exist several well-known techniques in the literature to come to this decision, e. g., depth-bounds, loop-checks [9], well-founded orderings [12], or well-quasi orderings [22]. Our implementation currently supports the following simple strategies:

None No unfolding is performed for user-defined functions.
One Only one function call is unfolded for each evaluation.
Each At most one call is unfolded for each user-defined function, subsequent calls are deferred.
All All function calls are unfolded, which corresponds to the original inference system. This does not guarantee termination but may be useful if the user is sure that the process terminates.

Note that, regardless of the chosen strategy, built-in functions (such as arithmetics) are evaluated in any case, since they are known to terminate.

Expressions that have been deferred during evaluation will be extracted and eventually added to the set of expressions to be evaluated, depending on the operation $abstract$ (see Sect. 5.2 for details). Generally, a strategy that allows more evaluation steps in one derivation than another strategy might seem superior. If an evaluation is split into multiple derivations with deferred subexpressions, each of these subexpressions has to be evaluated anew and leads to a new residual function to be generated. In contrast, longer derivations will produce less deferred subexpressions and, hence, less residual functions. Nevertheless, although a simpler strategy may produce more intermediate expressions, there are better chances that some of these expressions have already been encountered before, reducing the overall number of expressions to be evaluated. Furthermore, the final compression phase will eliminate intermediate functions so that even the simple strategies perform very well in practice.

5.2 Global Control

The local control is parametric w.r.t. the decision whether to stop or to proceed with the evaluation, since it is safe to terminate the evaluation at any point. This flexibility does not apply to the global control because we cannot stop the iterative extension of the set of expressions until all function calls in this set are "closed" w.r.t. the set of expressions. An expression e is closed w.r.t. a set of expressions if it is an instance of an expression in the set and all expressions in the matching substitution are recursively closed (see [5] for details). This condition is necessary to ensure the correctness of the partial evaluator so that the specialized program computes the same solutions as the original program. In order to avoid the construction of infinite sets of expressions, expressions in this set are generalized to ensure termination of this process.

Hence, the operation $abstract$ returns a safe approximation of $E_i \cup E'$ so that each expression in the set of $E_i \cup E'$ is closed w.r.t. the result of $abstract(E_i, E')$. More precisely, an expression $e' \in E'$ is added to the set E_i according to the following rules (note that the result of unfolding is either a variable, a deferred expression, a constructor appplication, a `case` expression, or a choice of these results):

1. If e' is a variable, it is discarded.
2. If e' has the form $\langle\!\langle e \rangle\!\rangle$, one of the following options is considered:
   (a) add e to the set E_i,
   (b) discard the expression e, or
   (c) compute the most specific generalization of e and some expression $e' \in E'$, say $\hat{e}$, and try to add both $\hat{e}$ and the expressions in the corresponding substitutions σ and θ, where $e = \sigma(\hat{e})$ and $e' = \theta(\hat{e})$.
3. For all other cases (constructor calls, `case` expressions, choices), the corresponding subexpressions are considered.

Like for the unfolding rule, the abstraction can be parameterized by a criterion to decide the option taken in (2). Our implementation currently supports abstractions using a well-founded ordering or an embedding ordering to distinguish between (2a) and (2c), i.e., smaller expressions are added but larger expressions are generalized.

To achieve a good level of specialization, it is crucial to recognize different variants of one expression as equivalent in order to discard them in (2b). This is more complex in our framework compared to [3], since we take `let` expressions into account. For example, consider the equivalent expressions "`map(square,xs)`" and "`let {f = square} in map(f,xs)`". If we do not recognize them as variants, they might be generalized to `map(f,xs)` which could not further be specialized. Therefore, we *normalize* expressions by applying α-conversion and flattening [19] before computing their abstractions.

Benchmark	Time for PE	Original	Specialized	Speedup
`allOnes`	280	180	140	1.29
`doubleApp`	330	190	160	1.19
`doubleFlip`	330	230	210	1.10
`lengthApp`	320	120	90	1.33
`kmp`	6100	1000	50	20.00
`foldr (+) 0 xs` (sum)	300	600	400	1.50
`foldr (+) 0 (map square xs)`	340	1280	800	1.60
`foldr (++) [] xs` (concat)	400	440	220	2.00
`map (twice square) xs`	420	1600	1410	1.13
`foldr (?) failed ys` (choose)	330	50	40	1.25
`head (perm ys)`	410	2960	40	74.00

Table 1. Benchmarks of selected partial evaluation examples (in msec)

6 Experimental Results

In this section we evaluate the implementation of our partial evaluator by some benchmarks. We compile both the partial evaluator and the benchmarks with the PAKCS Curry compiler (version 1.11.3, based on SICStus Prolog 4.2.3). All benchmarks were executed on a Linux machine (Debian Wheezy) with an Intel Core i5-750 (2.66GHz) processor and 4GiB of memory. The timings were performed using the profiling operation `profileTimeNF` of PAKCS and denote the time required for computing the normal form of the respective result in milliseconds (the arguments passed to the various functions were evaluated before to bring out the speedup obtained by partial evaluation). The benchmark examples have been specialized with one unfolding per evaluation and without any abstraction, since all examples terminated. Experiments with both a well-founded ordering or a well-quasi ordering resulted in the same or worse performance. Table 1 presents the time required for the partial evaluation process itself, for executing the original and the specialized program, and the gained speedup.

In the first group of benchmarks, we consider some typical examples of partial deduction and functional program transformations. These are simple functions working on lists or trees as (intermediate) data structures: `allOnes` computes the length of its input list, represented as Peano numbers, and constructs a new list of the same length with `1` as all elements, `doubleApp` is the concatenation of three lists, `doubleFlip` flips a tree structure twice, returning the same tree, `lengthApp` computes the length of the concatenation of two lists, and `kmp` implements a generic string pattern matcher. The first four functions were specialized without static input data, while the `kmp` example was specialized w.r.t. a fixed pattern of length 4, explaining both the time needed for partial evaluation and the gained speedup.

In the second group, we benchmark some examples with higher-order functions: the computation of the sum of list elements using `foldr`, the sum of squared numbers, the concatenation of a list of lists, and repeatedly applying a function to a list. All functions are applied to an input list `xs` containing 200,000 elements. The speedup is generally achieved because of the removal of intermediate data structures. For instance, the Curry expression “`foldr (+) 0 (map square xs)`” is specialized to the following residual FlatCurry definition:

```
sumSquare(xs) = case xs of { []    →  0
                           ; y:ys  →  (y*y) + sumSquare(ys) }
```

Finally, we evaluate two (complicated) variants of the function `choose`, which non-deterministically chooses one element of a given list `ys` containing 10,000 elements:

```
choose (x:xs) = x ? choose xs
```

Our partial evaluator computes this simple implementation of `choose` for the first example. The result for the second example only differs from `choose` in the order in which the two non-deterministic alternatives are taken, which stems from the implementation of `perm`. The huge speedup is achieved because of the omission of the non-deterministic intermediate list structure.

To summarize, our partial evaluator shows promising results and is capable of performing optimizations such as deforestation [24] and transformation of higher-order functions to first-order ones. In addition, non-deterministic operations are correctly specialized in contrast to [3], and the results for deterministic operations are almost identical.

7 Conclusions and Future Work

We have presented a new partial evaluation scheme for the functional logic language Curry based on its intermediate representation FlatCurry. The partial evaluator is based on an adaptation of the natural semantics of FlatCurry, extending the semantics to deal with the requirements of partial evaluation such as ensuring termination. In contrast to the original partial evaluator [3], which is based on term rewriting without sharing, the new implementation correctly handles both recursive `let` expressions and non-deterministic operations and, thus, supports full (Flat)Curry. As our benchmarks demonstrate, the implementation is capable of powerful optimizations both to deterministic and non-deterministic programs.

For future work, we intend to formally prove the correctness of the partial evaluation scheme, which should be manageable due to the similarity of the original and residual semantics. Another aspect for further investigations is the improvement of the abstraction operator. While the abstraction is necessary to ensure termination, a too general abstraction reduces the quality of the specialization. Thus, more sophisticated abstraction operators might be beneficial.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. E. Albert, M. Hanus, and G. Vidal. Using an abstract representation to specialize functional logic programs. In *Proc. of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR 2000)*, pages 381–398. Springer LNCS 1955, 2000.
3. E. Albert, M. Hanus, and G. Vidal. A practical partial evaluator for a multi-paradigm declarative language. *Journal of Functional and Logic Programming*, 2002(1), 2002.

4. E. Albert, M. Hanus, and G. Vidal. A residualizing semantics for the partial evaluation of functional logic programs. *Information Processing Letters*, 85(1):19–25, 2003.
5. M. Alpuente, M. Falaschi, and G. Vidal. Partial evaluation of functional logic programs. *ACM Transactions on Programming Languages and Systems*, 20(4):768–844, 1998.
6. S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30. Springer LNCS 1298, 1997.
7. S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.
8. S. Antoy and M. Hanus. New functional logic design patterns. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 19–34. Springer LNCS 6816, 2011.
9. R.N. Bol. Loop checking in partial deduction. *Journal of Logic Programming*, 16(1&2):25–46, 1993.
10. B. Braßel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.
11. B. Braßel, M. Hanus, B. Peemöller, and F. Reck. KiCS2: A new compiler from Curry to Haskell. In *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, pages 1–18. Springer LNCS 6816, 2011.
12. M. Bruynooghe, D. De Schreye, and B. Martens. A general criterion for avoiding infinite unfolding. *New Generation Computing*, 11(1):47–79, 1992.
13. S. Fischer, J. Silva, S. Tamarit, and G. Vidal. Preserving sharing in the partial evaluation of lazy functional programs. In A. King, editor, *Logic-based Program Synthesis and Transformation (revised and selected papers from LOPSTR'07)*, pages 74–89. Springer LNCS 4915, 2008.
14. J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.
15. M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168. Springer LNCS 7797, 2013.
16. M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2013.
17. M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org`, 2012.
18. H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.
19. J. Launchbury. A natural semantics for lazy evaluation. In *Proc. 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 144–154. ACM Press, 1993.
20. J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
21. S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.
22. M.H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In *Proc. of the 1995 International Logic Programming Symposium*, pages 465–479. MIT Press, 1995.
23. V.F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), 1986.
24. P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Black Hole Detection

As mentioned in Sect. 4.2, rule VarExp of the natural semantics shown in Fig. 3 replaces the variable binding $x \mapsto e$ by $x \mapsto \blacksquare$ in the heap when evaluating the associated expression e. This allows the detection of black holes (a self-dependent infinite loop) [19], as done in some implementations of functional (logic) languages. For instance, an attempt to evaluate the expression "`let {x = x} in x`" would result in a finite but incomplete derivation tree, whereas it would trigger the construction of an infinite derivation tree if the binding $x \mapsto e$ was kept.

The detection of black holes included in the semantics seems to be an optimization that could be omitted for deterministic programs [19]. However, it is crucial in combination with non-determinism in order to prevent the binding of a variable to *different* values in the *same* derivation, as shown in [10]. For example, consider the expression "`let { x = T ? case x of { T → F }} in x`". If we do not replace the variable binding in rule VarExp, the following derivation would be possible:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma : \texttt{T} \Downarrow \Gamma : \texttt{T}}{\Gamma : \texttt{T ? case x of \{ T} \to \texttt{F \}} \Downarrow \Gamma : \texttt{T}}}{\Gamma : \texttt{x} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{T}} \qquad [\texttt{x} \mapsto \texttt{T}] : \texttt{F} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{F}}{\Gamma : \texttt{case x of \{ T} \to \texttt{F \}} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{F}}}{\Gamma : \texttt{T ? case x of \{ T} \to \texttt{F \}} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{F}}}{\Gamma : \texttt{x} \Downarrow [\texttt{x} \mapsto \texttt{F}] : \texttt{F}}}{[\,] : \texttt{let \{ x = T ? case x of \{ T} \to \texttt{F \} \} in x} \Downarrow [\texttt{x} \mapsto \texttt{F}] : \texttt{F}}$$

$$\text{where } \Gamma = [\texttt{x} \mapsto \texttt{T ? case x of \{ T} \to \texttt{F \}}]$$

In this derivation, the variable `x` is looked up in the heap twice, where at first the right (non-deterministic) branch is chosen and afterwards the left branch. Hence, `x` is bound to `T` as well as `F`, which violates the single assignment property of call-time choice.

With our semantics, there is one successful and one failing derivation but no derivation where `x` is bound to `T` as well as `F`:

$$\dfrac{\dfrac{\dfrac{[\texttt{x} \mapsto \blacksquare] : \texttt{T} \Downarrow [\texttt{x} \mapsto \blacksquare] : \texttt{T}}{[\texttt{x} \mapsto \blacksquare] : \texttt{T ? case x of \{ T} \to \texttt{F \}} \Downarrow [\texttt{x} \mapsto \blacksquare] : \texttt{T}}}{[\texttt{x} \mapsto \texttt{T ? case x of \{ T} \to \texttt{F \}}] : \texttt{x} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{T}}}{[\,] : \texttt{let \{ x = T ? case x of \{ T} \to \texttt{F \} \} in x} \Downarrow [\texttt{x} \mapsto \texttt{T}] : \texttt{T}}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{[\texttt{x} \mapsto \blacksquare] : \texttt{x} \Downarrow \text{failure}}{[\texttt{x} \mapsto \blacksquare] : \texttt{case x of \{ T} \to \texttt{F \}} \Downarrow}}{[\texttt{x} \mapsto \blacksquare] : \texttt{T ? case x of \{ T} \to \texttt{F \}} \Downarrow}}{[\texttt{x} \mapsto \texttt{T ? case x of \{ T} \to \texttt{F \}}] : \texttt{x} \Downarrow}}{[\,] : \texttt{let \{ x = T ? case x of \{ T} \to \texttt{F \} \} in x} \Downarrow}$$

B Residualizing Semantics

Since the various rules of the residualizing semantics used in our partial evaluator are distributed over the paper and some of them were only informally sketched, we summarize in the following the complete set of rules of our residualizing semantics.

(Value)
$$\Gamma : v \Downarrow \Gamma : v \qquad \text{where } \begin{array}{c} v = c(\overline{x_n}) \text{ or } v = \langle\!\langle e' \rangle\!\rangle \text{ or } v \in \mathcal{V} \\ \text{with } (v \notin \mathit{dom}(\Gamma) \text{ or } \Gamma[v] = \text{free}) \end{array}$$

(VarExp)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto v] : v} \qquad \text{where } \begin{array}{l} e \notin \{\text{free}, \blacksquare\} \\ \text{and } (v \in \mathcal{V} \text{ or } v = c(\overline{x_n})) \end{array}$$

(VarDefer)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto e'] : \langle\!\langle x \rangle\!\rangle} \qquad \text{where } e \notin \{\text{free}, \blacksquare\}$$

(VarCase)
$$\frac{\Gamma[x \mapsto \blacksquare] : e \Downarrow \Delta : \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \}}{\Gamma[x \mapsto e] : x \Downarrow \Delta[x \mapsto \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \}] : x} \qquad \text{where } e \notin \{\text{free}, \blacksquare\}$$

(Flatten)
$$\frac{\Gamma[y \mapsto e_i] : \phi(x_1, \ldots, x_{i-1}, y, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v}{\Gamma : \phi(x_1, \ldots, x_{i-1}, e_i, e_{i+1}, \ldots, e_k) \Downarrow \Delta : v} \qquad \text{where } e_i \notin \mathcal{V}, y \text{ fresh}$$

(FunEval)
$$\frac{\Gamma : \sigma(e) \Downarrow \Delta : v}{\Gamma : f(\overline{y_n}) \Downarrow \Delta : v} \qquad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ \mathit{proceed}(\Gamma, f(\overline{y_n})) = \text{true} \end{array}$$

(FunDefer)
$$\Gamma : f(\overline{y_n}) \Downarrow \Gamma : \langle\!\langle f(\overline{y_n}) \rangle\!\rangle \qquad \text{where } \begin{array}{l} f(\overline{x_n}) = e \in P, \sigma = \{\overline{x_n \mapsto y_n}\}, \\ \mathit{proceed}(\Gamma, f(\overline{y_n})) = \text{false} \end{array}$$

(Let)
$$\frac{\Gamma[\overline{y_k \mapsto \sigma(e_k)}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let}\ \{\ \overline{x_k = e_k}\ \}\ \texttt{in}\ e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_k \mapsto y_k}\}, \overline{y_k} \text{ fresh}$$

(Or)
$$\frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1\ ?\ e_2 \Downarrow \Delta : v} \qquad \text{where } i \in \{1, 2\}$$

(Free)
$$\frac{\Gamma[\overline{y_n \mapsto \text{free}}] : \sigma(e) \Downarrow \Delta : v}{\Gamma : \texttt{let}\ \overline{x_n}\ \texttt{free in}\ e \Downarrow \Delta : v} \qquad \text{where } \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh}$$

(Select)
$$\frac{\Gamma : e \Downarrow \Delta : c(\overline{y_n}) \qquad \Delta : \sigma(e_i) \Downarrow \Theta : v}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v} \qquad \text{where } \begin{array}{l} p_i = c(\overline{x_n}), \\ \sigma = \{\overline{x_n \mapsto y_n}\} \end{array}$$

(Guess)
$$\frac{\Gamma : e \Downarrow \Delta[x \mapsto \text{free}] : x \qquad \Delta[x \mapsto \sigma(p_i), \overline{y_n \mapsto \text{free}}] : \sigma(e_i) \Downarrow \Theta : v}{\begin{array}{c} \Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Theta : v \\ \text{where } i \in \{1, \ldots, k\}, p_i = c(\overline{x_n}), \sigma = \{\overline{x_n \mapsto y_n}\}, \overline{y_n} \text{ fresh} \end{array}}$$

(CaseDefer)
$$\frac{\Gamma : e \Downarrow \Delta : \langle\!\langle e' \rangle\!\rangle}{\Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \langle\!\langle \texttt{case}\ e'\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}$$

(CaseUnbound)
$$\frac{\Gamma : e \Downarrow \Delta : x}{\begin{array}{c} \Gamma : \texttt{case}\ e\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \Downarrow \Delta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to \langle\!\langle e_k \rangle\!\rangle}\ \} \\ \text{where } x \notin \mathit{dom}(\Delta) \end{array}}$$

(CaseCase)
$$\frac{\Gamma : e \Downarrow \Delta : \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p'_j \to \langle\!\langle e'_j \rangle\!\rangle}\ \}}{\Gamma : \begin{array}{c} \texttt{case}\ e\ \texttt{of} \\ \{\ \overline{p_k \to e_k}\ \} \end{array} \Downarrow \Delta : \begin{array}{l} \texttt{case}\ x\ \texttt{of} \\ \quad \{\ \overline{p'_j \to \langle\!\langle \texttt{case}\ e'_j\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \} \end{array}}$$

(CaseVarCase)
$$\frac{\Gamma : e \Downarrow \Delta : x}{\begin{array}{c} \Gamma : \begin{array}{c} \texttt{case}\ e\ \texttt{of} \\ \{\ \overline{p_k \to e_k}\ \} \end{array} \Downarrow \Delta : \begin{array}{l} \texttt{case}\ y\ \texttt{of} \\ \quad \{\ \overline{p'_j \to \langle\!\langle \texttt{case}\ x\ \texttt{of}\ \{\ \overline{p_k \to e_k}\ \} \rangle\!\rangle}\ \} \end{array} \\ \text{where } \Delta[x] = \texttt{case}\ y\ \texttt{of}\ \{\ \overline{p'_j \to e'_j}\ \} \end{array}}$$

Automatic Testing of Operation Invariance

Tobias Gödderz and Janis Voigtländer

University of Bonn, Germany, {goedderz,jv}@cs.uni-bonn.de

Abstract. We present an approach to automatically generating operation invariance tests for use with Haskell's random testing framework QuickCheck. The motivation stems from a paper by Holdermans [8] which showed how to address certain shortcomings of straightforward testing of implementations of an abstract datatype. While effective, his solution requires extra generation work from the test engineer. Also, it may not even be doable if the person responsible for testing has no knowledge about, and program-level access to, the internals of the concrete datatype implementation under test. We propose and realize a refinement to Holdermans' solution that improves on both aspects: Required operation invariance tests can be formulated even in ignorance of implementation internals, and can be automatically generated using Template Haskell.

1 Introduction

It is good software engineering practice to test one's, and possibly other's, code. In declarative languages, QuickCheck [4] and related tools [1, 3, 10] are an attractive option, combining convenient ways of generating test input data and a DSL for expressing properties to be tested.

One recurring situation where property-based testing is desirable is in connection with implementations of an abstract datatype (ADT) specification. The scenario is that some interface (API) is given as a collection of type signatures of operations along with a collection of equational axioms that are expected to hold of those operations. Then there is an, or possibly several, implementations of that specification. The user of such an implementation should not need to be concerned with its internals, but would still like to be convinced of its correctness. It seems very natural to simply turn the given axioms into QuickCheck-like properties, and to accept an implementation as correct if it passes all those axioms-become-tests. However, it has been known for a while that such an approach is not enough to uncover all possible errors [6, 9]. Subtle bugs can remain hidden, due to an unfortunate interplay of buggy implementation and programmed equality (while actual semantic equality is impossible to test in general). Recently, Holdermans [8] presented a solution that works by adding operation invariance tests, to ensure that the assumed notion of equality is not just an equivalence relation, but actually a congruence relation. His solution has its own problems though. Specifically, it requires hand-writing a certain kind of additional test input data generator. But not only is that something which may depend on implementation internals (so is not necessarily something that a user having access only to the datatype's external API could do); it is also extra work and an additional source of potential errors: get that generator wrong by not covering enough ground, and even the additional tests will not be enough to really establish overall correctness. We set out to improve on these aspects.

2 The Need for Operation Invariance Tests

This section is largely a recap of material from Holdermans' paper. Specifically, we also use his very compelling example of how the approach of simply (and only) testing the axioms from the specification of an ADT may lead to a false sense of security.

Assume a set of people is interested in integer FIFO queues, and in particular in separately specifying, implementing, and using them. The specification is just a mathematical entity, consisting of a listing of signatures of desirable operations:

$$\begin{aligned}
&\mathit{empty} &&:: \mathsf{Queue}\\
&\mathit{enqueue} &&:: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{Queue}\\
&\mathit{isEmpty} &&:: \mathsf{Queue} \to \mathsf{Bool}\\
&\mathit{dequeue} &&:: \mathsf{Queue} \to \mathsf{Queue}\\
&\mathit{front} &&:: \mathsf{Queue} \to \mathsf{Int}
\end{aligned}$$

and of axioms expected to hold:

Q_1: $\mathit{isEmpty}\ \mathit{empty} = \mathsf{True}$

Q_2: $\mathit{isEmpty}\ (\mathit{enqueue}\ x\ q) = \mathsf{False}$

Q_3: $\mathit{front}\ (\mathit{enqueue}\ x\ \mathit{empty}) = x$

Q_4: $\mathit{front}\ (\mathit{enqueue}\ x\ q) = \mathit{front}\ q \qquad \textbf{if}\ \mathit{isEmpty}\ q = \mathsf{False}$

Q_5: $\mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty}) = \mathit{empty}$

Q_6: $\mathit{dequeue}\ (\mathit{enqueue}\ x\ q) = \mathit{enqueue}\ x\ (\mathit{dequeue}\ q) \qquad \textbf{if}\ \mathit{isEmpty}\ q = \mathsf{False}$

An implementation in Haskell would be a module

module Queue (Queue, *empty*, *enqueue*, *isEmpty*, *front*, *dequeue*) **where**
...

that contains some definition for Queue as well as definitions for the operations adhering to the type signatures from the specification. Importantly, the type Queue is exported without any data constructors, so from the outside of the module, Queue values can only be created, manipulated, and inspected using the provided operations. A prospective user of the implementation can import the above module and call those operations in application code. Now, the user would like to have some certainty that the implementation is correct. The appropriate notion of correctness is adherence to the axioms from the mathematical specification, on which user and implementer should have agreed beforehand. One way for the implementer to convince the user of the correctness is to do extensive property-based testing to establish validity of the specification's axioms for the provided implementation. Using QuickCheck, the test suite would consist of the following properties:

$q_1 = \mathit{property}\ (\mathit{isEmpty}\ \mathit{empty} == \mathsf{True})$

$q_2 = \mathit{property}\ (\lambda x\ q \to \mathit{isEmpty}\ (\mathit{enqueue}\ x\ q) == \mathsf{False})$

$q_3 = \mathit{property}\ (\lambda x \to \mathit{front}\ (\mathit{enqueue}\ x\ \mathit{empty}) == x)$

```
q4 = property (λx q → isEmpty q == False
                        ⟹
                        front (enqueue x q) == front q)
q5 = property (λx → dequeue (enqueue x empty) == empty)
q6 = property (λx q → isEmpty q == False
                        ⟹
                        dequeue (enqueue x q) == enqueue x (dequeue q))
```

These could even be written down by the user person in ignorance of any internals of the implementation. However, the use of == on Queue values requires availability of an appropriate instance of the Eq type class (which is how Haskell organizes overloading of ==). Let us assume the implementer provides such an instance. Moreover, we have to assume that the implementer also provides an appropriate random generator for Queue values (the q values quantified via the lambda-abstractions above – whereas for the Int values quantified as x we can take for granted that a random generator already exists), because otherwise the properties cannot be tested. In general, the implementer may even have to provide several random Queue generators for different purposes, for example since otherwise certain preconditions in axioms might be too seldom fulfilled, thus preventing effective testing.[1] But in the example here this is not an issue, since both empty and nonempty queues will appear with reasonable likelihood among the generated test data.

So if a Queue implementation passes all the above tests, it should be correct, right? Unfortunately not. As Holdermans [8] demonstrates, it can happen, even under all the assumptions above, that an implementation passes all the tests but is still harmfully incorrect. Let us repeat that faulty implementation in full as well (also since we will later want to refer to it in checking the adequacy of our own testing solution to the overall problem). Here it is:

```
data Queue = BQ [Int] [Int]
bq :: [Int] → [Int] → Queue
bq [ ] r = BQ (reverse r) [ ]
bq f  r = BQ f r
empty :: Queue
empty = bq [ ] [ ]
enqueue :: Int → Queue → Queue
enqueue x (BQ f r) = bq f (x : r)
isEmpty :: Queue → Bool
isEmpty (BQ f _) = null f
```

---

[1] QuickCheck's restricting conditional operator $\Longrightarrow$, written as ==>, does not count "*A*" being false as evidence for "*A* implies *B*" being true. Thus, in order to reach a certain number of positive test cases for "*A* implies *B*", that many cases with both "*A*" and "*B*" being true must be encountered (and, of course, not a single case with "*A*" true but "*B*" false). Consequently, QuickCheck gives up testing if not enough cases with "*A*" being true are encountered.

```
front :: Queue → Int
front (BQ f _) = last f

dequeue :: Queue → Queue
dequeue (BQ f r) = bq (tail f) r
```

This implementation uses the "smart constructor" *bq* to preserve the invariant that, for every BQ *f r*, it holds that *null f* implies *null r*, which makes *front* simpler to implement. Ironically, in the implementation above the error nevertheless lies in the definition of *front*, which should have been *head f* rather than *last f*.

As already mentioned, in order to test properties q_1–q_6, we need a definition of $==$ for Queue and a random generator for Queue values. The implementer is kind enough to provide both (and both are perfectly sane) within the Queue module:

```
instance Eq Queue where
  q == q′ = toList q == toList q′

toList :: Queue → [Int]
toList (BQ f r) = f ++ reverse r

instance Arbitrary Queue where
  arbitrary = do f ← arbitrary
                 r ← arbitrary
                 return (bq f r)
```

Now we can run tests *quickCheck* q_1 through *quickCheck* q_6, and find that all are satisfied. And this is not just happenstance, by incidentally not hitting a counterexample during the random generation of test input data. No, it is a mathematical fact that the implementation above satisfies the $==$-equations in properties q_1–q_6. And yet the implementation is incorrect. To see this, consider:

```
> front (dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty))))
3
```

Clearly, a correct implementation of a FIFO queue should have output 2 here. In fact, Q_2–Q_6 can be used to prove, by equational reasoning, that

$$\mathit{front}\ (\mathit{dequeue}\ (\mathit{enqueue}\ 3\ (\mathit{enqueue}\ 2\ (\mathit{enqueue}\ 1\ \mathit{empty})))) = 2$$

The key to what went wrong here (the implementation being incorrect and yet passing all the tests derived from the specification axioms) is that properties q_1–q_6 do not, and in fact cannot, test for semantic equivalence. They can only be formulated using the programmed equivalence $==$. That would be fine if $==$ were not just *any* equivalence relation that satisfies q_1–q_6 (which we know it does), but actually a *congruence relation* that does so. For otherwise, $==$ cannot correspond to the semantic equivalence $=$ intuitively used in Q_1–Q_6. Being a congruence relation means to, in addition to being an equivalence relation, be compatible with all operations from the ADT specification. And that is not the case for the implementation given above. For example,

```
> let q = dequeue (enqueue 3 (enqueue 2 (enqueue 1 empty)))
> let q′ = enqueue 3 (dequeue (enqueue 2 (enqueue 1 empty)))
> q == q′
True
```

but

```
> front q == front q′
False
```

That does not necessarily mean that, morally, the implementation of == given for Queue (and used in the tests) is wrong. The real bug here resides in the implementation of *front*, but it cannot be detected by just checking q_1–q_6; one additionally needs appropriate compatibility axioms.[2]

So Holdermans [8] observes that one should check additional axioms like

```
q₇ = property (λq q′ → q == q′ ⟹ front q == front q′)
```

Actually, he wisely adds a non-emptiness check to the precondition to prevent both *front q* and *front q′* from leading to a runtime error. But even then, the new axiom (or any of the other added compatibility axioms) does not – except in very lucky attempts – actually uncover the bug in the implementation. The problem is that it is very unlikely to hit a case with $q == q'$ for random q and q'. And even if QuickCheck hits one such case every now and then, it is not very likely that it is *also* a counterexample to *front q == front q′*. So in the vast majority of test runs QuickCheck simply gives up, and we are none the wiser about whether the compatibility axioms do hold for the given implementation or do not.

Holdermans' solution to *this* problem is to invest manual effort into randomly generating pairs of q and q' that ought to be considered equivalent queues. Concretely,

```
data Equiv a = a :≡: a

instance Arbitrary (Equiv Queue) where
  arbitrary = do z ← arbitrary
                 x ← from z
                 y ← from z
                 return (x :≡: y)
    where
      from xs = do i ← choose (0, length xs − 1)
                   let (xs₁, xs₂) = splitAt i xs
                   return (bq xs₁ (reverse xs₂))
```

Given some Show instances for Queue and Equiv Queue, testing the newly formulated property

```
q₇ = property (λ(q :≡: q′) → not (isEmpty q) ⟹ front q == front q′)
```

would yield, for example:

---

[2] In general, one should *also* convince oneself in the first place that == is indeed an equivalence relation as well (reflexive, symmetric, transitive), but often this will be a triviality. For example, any definition of the form $x == x' = f\ x == f\ x'$ for some function f, like in the Eq instance for Queue, is already guaranteed to give an equivalence relation if == on the target type of f is known to be an equivalence relation.

```
> quickCheck q7
*** Failed! Falsifiable (after 5 tests):
BQ [2,0] [] :≡: BQ [2] [0]
```

That, finally, is useful information which can be used to figure out the bug in the implementation of *front*.

But this success depends on the implementer having provided a good definition for the Arbitrary (Equiv Queue) instance above, specifically, a good *from*-function for "perturbing" a randomly generated queue in different ways. Note, this is not something the user could do themselves, or exercise any control over, since implementing that instance crucially depends on having access to the internals of the queue implementation module. If the implementer makes a mistake in writing that function/instance, then the bug in the original implementation of *front* will possibly remain unnoticed. For example, in an extreme case, the implementer may accidentally write the Arbitrary (Equiv Queue) instance in such a way that *from* is semantically equivalent to *return*, in which case q_7 and all other compatibility axioms will be tested positively, despite the implementation of *front* still being incorrect. Holdermans notes that one might test the chosen perturbation functionality itself to ensure not having introduced an error there. But this, which would correspond to a test $\mathit{quickCheck}\ (\lambda (q :\equiv: q') \rightarrow q == q')$ does not help with judging the suitability of the perturbation in terms of having "enough randomness"; clearly $\mathit{from}\ xs = \mathit{return}\ xs$ would make this test succeed as well. And the inverse property, that any pair of equivalent queues $q == q'$ has indeed a chance of also being generated as $q :\equiv: q'$, is unfortunately *not* expressible as a QuickCheck test.

Our purpose now is to avoid the need of defining an Arbitrary (Equiv Queue) instance. Thus, we will empower the user to detect operation invariance violations in an ADT implementation without having access to internals or relying on the implementer to generate the required pairs of equivalent values for tests.

3 Thoughts on Random Terms

A simple idea would be to work purely symbolically as long as possible. For example, the values

$$q = \mathit{dequeue}\ (\mathit{enqueue}\ 3\ (\mathit{enqueue}\ 2\ (\mathit{enqueue}\ 1\ \mathit{empty})))$$

and

$$q' = \mathit{enqueue}\ 3\ (\mathit{dequeue}\ (\mathit{enqueue}\ 2\ (\mathit{enqueue}\ 1\ \mathit{empty})))$$

that we first mentioned as evidence for a case with $q == q'$ but $\mathit{front}\ q \mathrel{/=} \mathit{front}\ q'$[3], are semantically equivalent according to Q_1–Q_6. Naturally, *any* pair of conceptually equivalent queues can be shown (mathematically) to be equivalent by using those original axioms. After all, that is the whole point of having the ADT specification comprising of those axioms in the first place.

---

[3] We have $\mathrel{/=}$ as the negation of programmed equivalence $==$.

So a suitable strategy for generating terms q and q' that would be eligible as $q :\equiv: q'$ pairs might be to start from the given signatures of operations, randomly generate a well-typed term q from those operations, then randomly apply a random selection of the axioms Q_1–Q_6 at random places in the term q to obtain another term q'. Using pairs of such q and q' for tests like $\mathit{front}\ q == \mathit{front}\ q'$ (now actually computing the result, no longer working symbolically) would indeed, in principle, have the power to detect the bugs that violate operation invariance. In fact, this is exactly the strategy we initially pursued.

However, it does not work out in practice. Already generating random terms is known to be non-trivial, though solutions exist [5]. Moreover applying random axioms at random places is difficult in general to get right in the sense of ending up with a probability distribution that is effective in terms of hitting cases that uncover bugs. For example, given an integer queue, only the root node can be of type Bool, which will lead to a low probability of axiom Q_2 to be applicable at a randomly selected node and an even lower probability of it to be selected for application, especially for large terms. Conversely first deciding on an axiom to apply and then randomly generating a (sub)term at which it is indeed applicable leads to similar uncertainties about coverage.

Additionally, there is a problem with "infeasible" terms. For integer queues, that is every term where *dequeue* or *front* is applied to a subterm t with $\mathit{isEmpty}\ t$ being True. Assume for now that x in each term $\mathit{enqueue}\ x\ q$ might only be an integer (and not a more complex subterm $\mathit{front}\ q'$).[4] Then the tree of each term is a path. Its leaf will be *empty*, its root node one of *enqueue*, *isEmpty*, *front*, or *dequeue*. All inner nodes can only be one of *enqueue* or *dequeue*. If at any node in the path more of its descendants are of type *dequeue* than *enqueue*, then the term is not valid. The probability that a term of length n is feasible is $\frac{1}{2}$ for $n = 1$, $\frac{252}{1024}$ for $n = 10$, and about 0.125 for $n = 40$. This might be much worse for ADTs with more complex terms.

Without prior knowledge about which operations are problematic and which axioms contribute to situations of equivalent values (queues, in the example) whose equivalence is not preserved by applying an operation, there is not much hope with the naive approach described above. But of course we precisely want to avoid having to invest any such knowledge about a specific ADT or implementation, since our goal is a generic solution to the problem. Also, working purely symbolically would not be practical for another reason: some axioms, like Q_4 and Q_6, have preconditions that need to be established before application. So in the hypothetical rewriting of q into q' by random applications of axioms described above, we would either have to symbolically prove such preconditions along the way (arbitrarily hard in general), or resort to actually computing values of subterms of q to establish whether some axiom is applicable there or not.

4 Our Solution

So what can we do instead? We want to avoid having to apply arbitrarily many rewrites at random places deep in some terms. Toward a solution, let us discuss the case of a binary operation f with two argument positions to be filled by a value of the abstract

---

[4] This could be seen as having an operation $\mathit{enqueue}_x :: \mathsf{Queue} \to \mathsf{Queue}$ for every $x :: \mathsf{Int}$.

datatype (unlike in the case of `Queue`, where no operation takes more than one queue as input).[5] The required operation invariance test would be that $t_1 :\equiv: t'_1$ and $t_2 :\equiv: t'_2$ imply $f\ t_1\ t_2 == f\ t'_1\ t'_2$, where $t_i :\equiv: t'_i$ means that t_i and t'_i are convertible, i.e., there is a sequence of axiom applications that leads from term t_i to term t'_i. We claim that it would be enough to focus on the two argument positions of f separately, and to consider only single axiom applications. Indeed, assume the above test leads to a counterexample, i.e., $f\ t_1\ t_2\ /= f\ t'_1\ t'_2$. Then there are already t, t', and t'' with $f\ t\ t'\ /= f\ t\ t''$ or $f\ t'\ t\ /= f\ t''\ t$ and where t' and t'' are exactly one axiom application apart. Why? By the given assumptions, we can form a sequence $f\ t_1\ t_2 = \ldots = f\ t'_1\ t_2 = \ldots = f\ t'_1\ t'_2$ of single axiom applications, where in the first part of that sequence axioms are only applied inside the first argument position of f, later only inside the second one. Now if t, t', and t'' with the claimed properties would not exist, then it would have to be the case that $f\ t_1\ t_2 == \ldots == f\ t'_1\ t_2 == \ldots == f\ t'_1\ t'_2$ and thus $f\ t_1\ t_2 == f\ t'_1\ t'_2$ by transitivity of $==$[6]. But this is a contradiction to $f\ t_1\ t_2\ /= f\ t'_1\ t'_2$.

So, generalizing from the case of a binary operation, we have that in order to establish operation invariance overall, it suffices to test that for every operation f it holds $f \ldots t' \ldots == f \ldots t'' \ldots$ for every argument position and all pairs t' and t'' of terms related by exactly one axiom application and where the other argument positions are appropriately filled with random input (but with the same on left and right; this is what the "…" indicate in $f \ldots t' \ldots == f \ldots t'' \ldots$). Still, the axiom application relating t' and t'' could be somewhere deeply nested, i.e., t' could be $f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots lhs \ldots) \ldots) \ldots) \ldots$ and t'' be $f_1 \ldots (f_2 \ldots (\ldots (f_n \ldots rhs \ldots) \ldots) \ldots) \ldots$, where *lhs* and *rhs* are the two sides of an instantiated axiom, while all the other parts ("…") in the two nestings agree. We could generate tests arising from this, an important observation being that for the "…" parts, since they are equal on both sides (not only equivalent), we can simply use random values – no random symbolic *terms* are necessary, which is an important gain. Note that, as Haskell is purely functional, it is not necessary to model dependencies between arguments to obtain completeness. Actually, we restrict to a subset of all tests, namely ones where the axiom application in fact is at the root of t' and t''. That is, we only employ tests $f \ldots lhs \ldots == f \ldots rhs \ldots$, not e.g. $f \ldots (f_1 \ldots lhs \ldots) \ldots == f \ldots (f_1 \ldots rhs \ldots) \ldots$ – a pragmatic decision, but also one that has turned out well so far. We have not encountered a situation where this narrowing of test cases has missed some bug, except in very artificial examples manufactured just for that purpose. The intuitive reason seems to be that by doing all tests $f \ldots lhs \ldots == f \ldots rhs \ldots$, which of course also includes the tests $f_1 \ldots lhs \ldots == f_1 \ldots rhs \ldots$, for varying *lhs*/*rhs* as obtained from all axioms, one casts a fine enough net – situations where these all go through, along with all standard instantiations $lhs == rhs$ obtained from the axioms, but where $f \ldots (f_1 \ldots lhs \ldots) \ldots == f \ldots (f_1 \ldots rhs \ldots) \ldots$ would fail, appear to be extremely rare.

To summarize, we propose to test operation invariance via properties of the following form, for each combination of one operation, one argument position, and one axiom:

$$f\ x_1 \ldots (axiom_{lhs}\ y_1 \ldots y_m) \ldots x_n == f\ x_1 \ldots (axiom_{rhs}\ y_1 \ldots y_m) \ldots x_n$$

---

[5] The consideration of binary operations here is without loss of generality. Dealing with unary operations is simpler, and dealing with operations of higher arity than two is analogous to dealing with binary operations, just requires more index fiddling in the discussion.

[6] Note that while the operation invariance property of $==$ is still under test, we were willing to simply take for granted that $==$ is at least an equivalence relation. See also Footnote 2

where like the x_i (alluded to as the "other" values above), the y_j – which are meant to fill any variable positions in a symbolic axiom $lhs = rhs$ – can be simply values. No need for symbolic terms, since neither in the x_i nor in the y_j positions we need to assume any further possible axiom rewrites. Also, no need to generate terms/values in an implementation-dependent way, since the signature and axiom information from the ADT specification suffices. The discussion in the paragraphs preceding this one implies that the proposed approach is sound (if a counterexample is found, then there is a genuine problem with the ADT implementation under test), but not in general complete (even if the axioms and our subset of operation invariance properties are tested exhaustively, some bug in the ADT implementation might be missed – due to our pragmatic decision not to consider deeper nested rewrites like $f\ x_1 \ldots (f_1\ y_1 \ldots (axiom_{lhs}\ z_1 \ldots z_l) \ldots y_m) \ldots x_n == f\ x_1 \ldots (f_1\ y_1 \ldots (axiom_{rhs}\ z_1 \ldots z_l) \ldots y_m) \ldots x_n$).

The full set of tests of the proposed form for integer queues follows. An index $enqueue_1$ or $enqueue_2$ indicates that the first or second argument position of *enqueue* is filled by the particular axiom, respectively. Parameters with a prime (i.e. q' and x') fill the other arguments of the tested operation as opposed to being variables of the relevant axiom.

```
enqueue1_q3 = property (λx q′ → let lhs = front (enqueue x empty)
                                    rhs = x
                                in enqueue lhs q′ == enqueue rhs q′)
enqueue1_q4 = property (λx q q′ → isEmpty q == False
                                  ⟹
                                  let lhs = front (enqueue x q)
                                      rhs = front q
                                  in enqueue lhs q′ == enqueue rhs q′)
enqueue2_q5 = property (λx x′ → let lhs = dequeue (enqueue x empty)
                                    rhs = empty
                                in enqueue x′ lhs == enqueue x′ rhs)
isEmpty_q5 = property (λx → let lhs = dequeue (enqueue x empty)
                                rhs = empty
                            in isEmpty lhs == isEmpty rhs)
enqueue2_q6 = property (λx q x′ → isEmpty q == False
                                  ⟹
                                  let lhs = dequeue (enqueue x q)
                                      rhs = enqueue x (dequeue q)
                                  in enqueue x′ lhs == enqueue x′ rhs)
isEmpty_q6 = property (λx q → isEmpty q == False
                              ⟹
                              let lhs = dequeue (enqueue x q)
                                  rhs = enqueue x (dequeue q)
                              in isEmpty lhs == isEmpty rhs)
dequeue_q6 = property (λx q → isEmpty q == False
                              ⟹
```

```
                               let lhs = dequeue (enqueue x q)
                                   rhs = enqueue x (dequeue q)
                               in not (isEmpty lhs) ⟹
                                  dequeue lhs == dequeue rhs)
front_q6 = property (λx q → isEmpty q == False
                             ⟹
                             let lhs = dequeue (enqueue x q)
                                 rhs = enqueue x (dequeue q)
                             in not (isEmpty lhs) ⟹
                                front lhs == front rhs)
```

There are two additional tests that syntactically belong to the others, but do not really add anything:

```
dequeue_q5 = property (λx → let lhs = dequeue (enqueue x empty)
                                rhs = empty
                            in not (isEmpty lhs) ⟹
                               dequeue lhs == dequeue rhs)
front_q5 = property (λx → let lhs = dequeue (enqueue x empty)
                              rhs = empty
                          in not (isEmpty lhs) ⟹
                             front lhs == front rhs)
```

This is because both sides of Q_5 are empty queues, but neither *dequeue* nor *front* work on those.

5 A Practical Implementation

The tests shown in the previous section can (almost) strictly syntactically be derived from the specification. For every operation, every argument the operation has, and every axiom, one test can be obtained – by applying the operation at the selected argument to the axiom if the types allow it. In addition, constraints may have to be added per operation to prevent their application to invalid values, like the constraints *not* (*isEmpty lhs*) in *dequeue_q*$_6$ and *front_q*$_6$ (and in *dequeue_q*$_5$ and *front_q*$_5$).

A tool or library that generates these tests automatically needs type information about both operations and axioms. Details about the implementation of the datatype and operations, or the specific terms in the axioms, are not necessary. As relatively arbitrarily typed code must be generated, it seems to be at least very tricky to do this with plain Haskell and without a lot of manual help. Thus, to automate our solution as much as possible, we used Template Haskell [11]. As a result, none of the shown tests need to be hand-written by the user.

As a case study, we demonstrate here different ways how our tool/library (in a new module Test.OITestGenerator) can be used to generate appropriate QuickCheck tests. First, the axioms have to be specified. For this, OITestGenerator exports a datatype AxiomResult *a*, its constructor =!=, and a restricting conditional operator $\Rrightarrow$[7] that works

---

[7] which is written as ===>

akin to $\Longrightarrow$ as known from QuickCheck. Axioms with variables are written as functions with corresponding arguments, returning an AxiomResult a where a is the codomain of the axiom's left- and right-hand side.

$$q_1 :: \mathsf{AxiomResult\ Bool}$$
$$q_1 = \mathit{isEmpty}\ \mathit{empty} =!= \mathsf{True}$$
$$q_2 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Bool}$$
$$q_2 = \lambda x\ q \to \mathit{isEmpty}\ (\mathit{enqueue}\ x\ q) =!= \mathsf{False}$$
$$q_3 :: \mathsf{Int} \to \mathsf{AxiomResult\ Int}$$
$$q_3 = \lambda x \to \mathit{front}\ (\mathit{enqueue}\ x\ \mathit{empty}) =!= x$$
$$q_4 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Int}$$
$$q_4 = \lambda x\ q \to \mathit{not}\ (\mathit{isEmpty}\ q) \Rrightarrow \mathit{front}\ (\mathit{enqueue}\ x\ q) =!= \mathit{front}\ q$$
$$q_5 :: \mathsf{Int} \to \mathsf{AxiomResult\ Queue}$$
$$q_5 = \lambda x \to \mathit{dequeue}\ (\mathit{enqueue}\ x\ \mathit{empty}) =!= \mathit{empty}$$
$$q_6 :: \mathsf{Int} \to \mathsf{Queue} \to \mathsf{AxiomResult\ Queue}$$
$$q_6 = \lambda x\ q \to \mathit{not}\ (\mathit{isEmpty}\ q) \Rrightarrow \mathit{dequeue}\ (\mathit{enqueue}\ x\ q) =!= \mathit{enqueue}\ x\ (\mathit{dequeue}\ q)$$

This is already enough preparation to generate the basic tests, i.e., direct translations of the axioms, with the provided function *generate_basic_tests*. It gets one argument, a list of Axioms. Axiom is a container holding a) the name of an axiom, which (the axiom) must be a function returning an AxiomResult a, and b) possibly custom generators for the function's arguments. It has one constructor function *axiom*, which takes a Name – custom generators can be assigned to an Axiom via *withGens*, which is explained later. Then *generate_basic_tests* returns an expression of type [Property]. Property is the type of QuickCheck tests, as returned by the function *property* in the earlier given versions of q_1–q_6.

In using *generate_basic_tests* to generate the basic tests from the above functions returning AxiomResults, two of Template Haskell's syntactic constructs will be needed: The first are *splices*. A splice is written $(...), where ... is an expression. A splice may occur instead of an expression. The splice will be evaluated at compile time and the syntax tree returned by it will be inserted in its place. The second construct used is '..., where ... is a name of a function variable or data constructor. Then '... is of type Name and its value represents the name of the ... that was quoted. Using these constructs, we can write:

$$\mathit{adt\_basic\_tests} :: [\mathsf{Property}]$$
$$\mathit{adt\_basic\_tests} = \$(\mathbf{let}\ \mathit{axs} = \mathit{map}\ \mathit{axiom}\ [\text{'}q_1, \text{'}q_2, \text{'}q_3, \text{'}q_4, \text{'}q_5, \text{'}q_6]$$
$$\mathbf{in}\ \mathit{generate\_basic\_tests}\ \mathit{axs})$$

Now, *adt_basic_tests* can be executed via *mapM_ quickCheck adt_basic_tests*.

Before the, for our purposes more interesting, operation invariance tests can be generated, constraints for *dequeue* and *front* must be specified. Such a constraint function has the purpose of deciding whether a set of arguments is valid for the respective operation. Thus it takes the same arguments, but returns a Bool independently of the operation's return type.

```
may_dequeue :: Queue → Bool
may_dequeue = not ∘ isEmpty

may_front :: Queue → Bool
may_front = not ∘ isEmpty
```

Given these, the operation invariance tests can be generated by the provided function *generate_oi_tests*. For operations there exists a datatype Op similar to Axiom, with one constructor function *op*. The function *withConstraint* may be used to add a constraint to an operation.[8] It may also be used multiple times on the same operation, in which case the constraints are connected with a logical "and".

```
adt_oi_tests :: [Property]
adt_oi_tests = $(let ops = [op 'empty
                           ,op 'enqueue
                           ,op 'isEmpty
                           ,withConstraint (op 'dequeue) 'may_dequeue
                           ,withConstraint (op 'front) 'may_front]
                     axs = map axiom ['q1,'q2,'q3,'q4,'q5,'q6]
                 in generate_oi_tests axs ops)
```

Note that the repeated local definition of *axs* (in both *adt_basic_tests* and *adt_oi_tests*) is necessary due to Template Haskell's stage restrictions. It is not possible to refer to a top-level declaration in the same file, because it is still being compiled when the splices are executed. Note also that *empty* could be omitted here from the list of operations as it takes no arguments.

Running the tests automatically generated above is enough to detect the buggy implementation of *front*!

```
+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q3).
+++ OK, passed 100 tests (100% Queue.enqueue@1/Main.q4).
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q5).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q5).
*** Gave up! Passed only 0 tests.
*** Gave up! Passed only 0 tests.
+++ OK, passed 100 tests (100% Queue.enqueue@2/Main.q6).
+++ OK, passed 100 tests (100% Queue.isEmpty@1/Main.q6).
+++ OK, passed 100 tests (100% Queue.dequeue@1/Main.q6).
*** Failed! Falsifiable (after 5 tests):
3
BQ [4] [4,−3]
```

The test that fails here is *front_q*$_6$ (which would have appeared in the output as `Queue.front@1/Main.q6`). Note that two tests were generated that are correctly typed but have no valid input (as already observed further above when writing down properties by hand); namely *dequeue_q*$_5$ and *front_q*$_5$ alias `Queue.dequeue@1/Main.q5` and

---

[8] As opposed to constraints for axioms, which are specified using $\Rrightarrow$ in the function whose name is passed to *axiom*.

`Queue.front@1/Main.q5`. They could be avoided by using other functions exported by OITestGenerator and explained below.

Generators can be passed to an operation via *withGens* in a list, with one generator name for each argument, as in *withGens* (*op* '*enqueue*) ['*arbitrary*, '*arbitrary*]. It is not allowed to omit generators when using *withGens*; instead *arbitrary* must be passed if no special generator should be used for some position. The function *withGens* can be intermingled with *withConstraint* and may also be used on Axioms.

Also, there is a convenience function *generate_axiom's_tests* which takes only one Axiom and a list of Ops. It is useful when certain combinations of axioms and operations should be excluded. It can also be used when only specific argument positions of an operation should be excluded for an axiom. The function *but* :: Op → Arg → Op, when called as *o* ‘*but*‘ *i*, excludes the *i*th argument from *o* when generating tests. Arg has a single constructor function *arg* :: Int → Arg. The function *but* may be called multiple times on the same operation to exclude multiple arguments. To supplement it, there also is *only* :: Op → Arg → Op to include only one argument. For illustration:

```
all_q5 :: [Property]
all_q5 = $(let ops = [op 'empty
                     ,op 'enqueue
                     ,op 'isEmpty
                     ,op 'dequeue `but` arg 1
                     ,op 'front `but` arg 1]
           in generate_axiom's_tests (axiom 'q5) ops)
```

Of course, in this case, *dequeue* and *front* could simply be omitted completely as they do only have one argument.

Another convenience function is *generate_single_test*, which again works similarly to *generate_oi_tests*, but takes only one Axiom and one Op instead of lists, and generates only a single test. It may be used when more control is needed.

```
enqueue1_q3 = $(generate_single_test (axiom 'q3) (op 'enqueue `only` 1))
enqueue1_q4 = $(generate_single_test (axiom 'q4) (op 'enqueue `only` 1))
enqueue2_q5 = $(generate_single_test (axiom 'q5) (op 'enqueue `only` 2))
isEmpty1_q5 = $(generate_single_test (axiom 'q5) (op 'isEmpty `only` 1))
enqueue2_q6 = $(generate_single_test (axiom 'q6) (op 'enqueue `only` 2))
isEmpty1_q6 = $(generate_single_test (axiom 'q6) (op 'isEmpty `only` 1))
dequeue1_q6 = $(generate_single_test (axiom 'q6) (op 'dequeue `only` 1))
front1_q6   = $(generate_single_test (axiom 'q6) (op 'front `only` 1))
```

No constraints are passed here because the superfluous tests $\mathit{dequeue}_1\_q_5$ and $\mathit{front}_1\_q_5$ were purposefully omitted, and because no axiom but Q_5 can result in an empty queue.

As writing such a list of tests is cumbersome, there is a function *show_all_tests* :: Maybe (String → Int → String → String) → [Name] → [Name] → ExpQ which takes an optional formatting function, a list of axiom names, and a list of operation names, and produces a String-typed expression whose content is exactly the code above (plus $\mathit{dequeue}_1\_q_5$ and $\mathit{front}_1\_q_5$, which were removed manually from the output).

```
single_test_str = $(let ops = ['empty,'enqueue,'isEmpty,'dequeue,'front]
                        axs = ['q1,'q2,'q3,'q4,'q5,'q6]
                    in show_all_tests Nothing axs ops)
```

If constraints have to be added to the generated code, they must be added manually. On the other hand, all *‘only‘ n* could be omitted in the case above, since there is no operation with two arguments of the same type.[9] Instead of Nothing above, a custom formatting function can be passed that generates the names of the properties.

The implementation is available as a Cabal package at `http://hackage.haskell.org/package/qc-oi-testgenerator`. Insights into the implementation, as well as more discussion of formal aspects of the overall approach, can be found in the first author's diploma thesis [7].

6 Conclusion and Discussion

We have presented and implemented an approach for automatically checking operation invariance for Haskell implementations of abstract datatypes. The user writes down axioms by closely following standard QuickCheck syntax, and both the ordinary QuickCheck tests as well as additional operation invariance tests are derived from that.

It might have been more desirable to obtain the necessary information about axioms directly from existing QuickCheck tests, freeing the user from possibly having to rewrite them. For this it would be necessary, in the implementation, to obtain the syntax tree of a given Haskell declaration and find the axiom by looking for a call to $==$. Then, type information for the left- and right-hand side would be needed. As of today, neither is possible with Template Haskell: The Info data structure returned by *reify* has a field of type Maybe Dec to hold the syntax tree of the right-hand side of a declaration. However, according to the documentation of Template Haskell's most current version 2.9.0.0, there is no implementation for this and the field always contains Nothing. Another way to obtain the syntax tree would be to write the axioms in expression quotations. In both cases, though, Template Haskell offers no way of obtaining the necessary type information, as those can only be retrieved for Names using *reify*, but not for arbitrary syntax trees. Also due to Template Haskell not offering a way of calling the type checker or other convenient ways to help generate correctly typed code yet, the implementation does not currently support polymorphic types. A workaround making it possible to test polymorphic abstract datatypes is to construct a (suitably chosen, cf. [2]) monomorphic instance and rename all participating functions. That way, *reify* returns the monomorphic types.

On the conceptual side, it would be attractive to gain more insight into how effective the kind of tests we generate are in finding bugs in general. This might be achieved by a formalization of our approach and/or collecting experimental evidence from more case studies. Here we discuss only a small variation on the running example, which illustrates an interesting aspect concerning the implementation of equality:

---

[9] The function *generate_single_test* throws a compile time error unless there is exactly one way to combine the given Axiom and Op.

The error in the shown version of *front* is hidden from the basic tests only because the implemented equality compares two values by how the implementer thinks they *should* behave. An observational equality like the following one, which even does not touch the internals, would not so be fooled.

$$\begin{aligned} q == q' &\mid isEmpty\ q \mathrel{/=} isEmpty\ q' &&= \mathsf{False} \\ &\mid isEmpty\ q &&= \mathsf{True} \\ &\mid otherwise &&= front\ q == front\ q' \wedge dequeue\ q == dequeue\ q' \end{aligned}$$

Still, the basic tests will not suffice in general. As a very artificial example, consider the queue implementation used so far, but now without the error in *front*, and replace the following two functions:

$$bq\,f\ r = \mathsf{BQ}\ (f \mathbin{+\!\!+} reverse\ r)\ [\,]$$

$$\begin{aligned} enqueue\ x\ q@(\mathsf{BQ}\,f\ r) &\mid isEmpty\ q = bq\,f\ (r \mathbin{+\!\!+} [x]) \\ &\mid otherwise = \mathsf{BQ}\,f\ (r \mathbin{+\!\!+} [x]) \end{aligned}$$

This error will never be found with the basic tests and using the above observational equality. So using operation invariance tests is still a good idea.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions.

References

[1] C. Amaral, M. Florido, and V.S. Costa. PrologCheck – Property-based testing in Prolog. In *FLOPS*, pages 1–17. Springer, 2014.

[2] J.-P. Bernardy, P. Jansson, and K. Claessen. Testing polymorphic properties. In *ESOP*, pages 125–144. Springer, 2010.

[3] J. Christiansen and S. Fischer. EasyCheck – Test data for free. In *FLOPS*, pages 322–336. Springer, 2008.

[4] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279. ACM, 2000.

[5] J. Duregård, P. Jansson, and M. Wang. Feat: Functional enumeration of algebraic types. In *Haskell Symposium*, pages 61–72. ACM, 2012.

[6] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[7] T. Gödderz. Verification of abstract data types: Automatic testing of operation invariance. Diploma thesis, University of Bonn, Germany, 2014. `http://tobias.goedderz.info/dt-inf.pdf`.

[8] S. Holdermans. Random testing of purely functional abstract datatypes: Guidelines for dealing with operation invariance. In *PPDP*, pages 275–284. ACM, 2013.

[9] P. Koopman, P. Achten, and R. Plasmeijer. Model based testing with logical properties versus state machines. In *IFL*, pages 116–133. Springer, 2012.

[10] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck: Automatic exhaustive testing for small values. In *Haskell Symposium*, pages 37–48. ACM, 2008.

[11] T. Sheard and S. Peyton Jones. Template meta-programming for Haskell. In *Haskell Workshop*, pages 1–16. ACM, 2002.