Technical Communications of the 33rd International Conference on Logic Programming

ICLP 2017, August 28–September 1, 2017, Melbourne, Australia

Edited by

Ricardo Rocha

Tran Cao Son

Christopher Mears

Neda Saeedloei

Editors

Ricardo Rocha
CRACS & INESC TEC and Faculty of Sciences
University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
ricroc@dcc.fc.up.pt

Tran Cao Son
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003, USA
tson@cs.nmsu.edu

Christopher Mears
Redbubble
Melbourne, Australia
chris.mears@redbubble.com

Neda Saeedloei
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901, USA
neda@cs.siu.edu

ACM Classification 1998: D.1.6 Logic Programming, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.3 Programming Languages, D.3.2 Language Classifications: Applicative (functional) languages, Constraint and logic languages, D.3.3 Language Constructs and Features, D.3.4 Processors, F.2.2 Nonnumerical Algorithms and Problems, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.2 Semantics of Programming Languages, F.3.3 Studies of Program Constructs, H.3.3 Information Search and Retrieval, H.3.5 Online Information Services, I.2.1 Applications and Expert Systems, I.2.4 Knowledge Representation Formalisms and Methods, I.2.5 Programming Languages and Software, I.2.7 Natural Language Processing.

ISBN 978-3-95977-058-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-058-3.

Publication date
February, 2018

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/OASIcs.ICLP.2017.0
ISBN 978-3-95977-058-3 **ISSN 1868-8969** **http://www.dagstuhl.de/oasics**

OASIcs – OpenAccess Series in Informatics

OASIcs aims at a suitable publication venue to publish peer-reviewed collections of papers emerging from a scientific event. OASIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 2190-6807

http://www.dagstuhl.de/oasics

Contents

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

This volume contains the Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017), held in Melbourne, Australia, from the 28th of August to the 1st of September, 2017. Since the first conference held in Marseille in 1982, ICLP has been the premier international event for presenting research in logic programming. Contributions to ICLP are sought in all areas of logic programming, including:

- *Theory* – Semantic Foundations, Formalisms, Nonmonotonic Reasoning, Knowledge Representation.
- *Implementation* – Compilation, Virtual Machines, Parallelism, Constraint Handling Rules, Tabling.
- *Environments* – Program Analysis, Transformation, Validation, Verification, Debugging, Profiling, Testing.
- *Language Issues* – Concurrency, Objects, Coordination, Mobility, Higher Order, Types, Modes, Assertions, Programming Techniques.
- *Related Paradigms* – Inductive and Co-inductive Logic Programming, Constraint Logic Programming, Answer-Set Programming, SAT-Checking.
- *Applications* – Databases, Big Data, Data Integration and Federation, Software Engineering, Natural Language Processing, Web and Semantic Web, Agents, Artificial Intelligence, Bioinformatics, and Education.

Three kinds of submissions were accepted:

- *Technical papers*, for technically sound, innovative ideas that can advance the state of logic programming.
- *Application papers*, that impact interesting application domains;
- *System and tool papers*, which emphasize novelty, practicality. usability, and availability of the systems and tools described.

This year, ICLP adopted the hybrid publication model used in all recent editions of the conference, with journal papers and Technical Communications (TCs), following a decision made in 2010 by the Association for Logic Programming. Papers of the highest quality were selected to be published as *rapid publications* in the journal of Theory and Practice of Logic Programming (TPLP), Cambridge University Press. The TCs comprise papers which the Program Committee judged of good quality but not yet of the standard required to be accepted and published in TPLP as well as dissertation project descriptions stemming from the Doctoral Program (DP) held with ICLP.

We received 55 full submissions for the main conference, the Program Committee recommended 13 to be accepted as TCs, of which 11 were materialized in this volume (2 were withdraw). The DP, with a separate Program Committee, received 4 submissions, all of which were accepted. All papers in this volume were presented in specific sessions of ICLP 2017. The best DP paper was given the opportunity to be presented in a slot of the main conference.

We are of course deeply indebted to both Program Committee members and external reviewers, as the conference would not have been possible without their dedicated, enthusiastic and outstanding work. The Program Committee members for ICLP and the DP were:

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Mario Alviano	Gopal Gupta	Francesco Ricca
Marcello Balduccini	Amelia Harrison	Ricardo Rocha
Pedro Cabalar	Manuel V. Hermenegildo	Alessandra Russo
Mats Carlsson	Tomi Janhunen	Neda Saeedloei
Manuel Carro	Matti Järvisalo	Chiaki Sakama
Andre Augusto Cire	Serdar Kadioglu	Tom Schrijvers
Michael Codish	George Katsirelos	Takehide Soh
Alessandro Dal Palù	Andy King	Tran Cao Son
Broes De Cat	Ekaterina Komendantskaya	Theresa Swift
Marina De Vos	Lars Kotthoff	Guido Tack
Marc Denecker	Jean Marie Lagniez	Paul Tarau
Agostino Dovier	Joohyung Lee	Daniele Theseider Dupre'
Inês Dutra	Michael Leuschel	Kevin Tierney
Esra Erdem	Vladimir Lifschitz	Mirek Truszczynski
Wolfgang Faber	Michele Lombardi	Tommaso Urli
Fabio Fioravanti	Christopher Mears	Frank Valencia
Thom Fruehwirth	Alessandra Mileo	Willem-Jan Van Hoeve
Sarah Alice Gaggl	Jose F. Morales	Nadarajen Veerapen
Graeme Gange	Nina Narodytska	German Vidal
Maria Garcia De La Banda	Enrico Pontelli	Jan Wielemaker
Marco Gavanelli	Charles Prud'Homme	Stefan Woltran
Martin Gebser	Claude-Guy Quimper	Jia-Huai You
Tias Guns	C. R. Ramakrishnan	Neng-Fa Zhou

The external reviewers were:

Joaquin Arias	Georgios Karachalias	Javier Romero
Marc Bezem	Arash Karimi	Elmer Salazar
Bernhard Bliem	Michael Kifer	Vitor Santos Costa
Zhuo Chen	Ruben Lapauw	Lukas Schweizer
Jonnathan Cook	Thomas Linsbichler	Farhad Shakerin
Bernardo Cuteri	Fangfang Liu	Nada Sharaf
Fabio Aurelio D'Asaro	Yanhong A. Liu	Roni Stern
Ingmar Dasseville	Kyle Marple	Alwen Tiu
Besik Dundua	Michael Morak	Matthias van der Hallen
Andrea Formisano	Jose F. Morales	Alicia Villanueva
Michael Frank	Falco Nogatz	Yi Wang
Daniel Gall	Adrian Palacios	Philipp Wanko
Jianmin Ji	Le Thi Anh Thu Pham	Zhun Yang

We would also like to express our gratitude to the full ICLP 2017 organization committee, namely Maria Garcia de la Banda and Guido Tack, who acted as general chairs; Enrico Pontelli, who served as workshop chair; Tommaso Urli, who acted as publicity chair and designed the web pages; and, finally, Paul Fodor and Graeme Gange, who organized the programming contest.

Our gratitude must be extended to Torsten Schaub, who is serving in the role of President of the Association of Logic Programming (ALP), to all the members of the ALP Executive Committee and to Mirek Truszczyński, Editor-in-Chief of TPLP. Also, to the staff at Cambridge University Press, especially Richard Horley and Samira Ceccarelli, and to the

personnel at Schloss Dagstuhl-Leibniz Zentrum für Informatik, especially Marc Herbstritt, for their timely assistance. We would also like to thank the staff of the EasyChair conference management system for making the life of the Program Chairs easier. Thanks should go also to the authors of all submitted papers for their contribution to make ICLP alive and to the participants for making the event a meeting point for a fruitful exchange of ideas and feedback on recent developments.

Finally, we would like to thank our generous gold-tier sponsors – the Association for Logic Programming, the Association for Constraint Programming, the Monash University, the University of Melbourne, CSIRO Data61, COSYTEC and Satalia; our generous bronze-tier sponsors – Google; and our generous donors – the European Association for Artificial Intelligence, the International Journal of Artificial Intelligence, Springer, CompSustNet and Cosling.

Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei
Program Committee Chairs
August 2017

List of Authors

Marek Adrian
Universty of Calabria
Italy
m.adrian@mat.unical.it

Weronika T. Adrian
Universty of Calabria
Italy
w.adrian@mat.unical.it

Mario Alviano
University of Calabria
Italy
alviano@mat.unical.it

Giovanni Amendola
Universty of Calabria
Italy
amendola@mat.unical.it

Marcello Balduccini
Drexel University
United States of America
mb3368@drexel.edu

Roman Barták
Charles University
Czech Republic
bartak@ktiml.mff.cuni.cz

Bernhard Bliem
TU Wien
Austria
bliem@dbai.tuwien.ac.at

William E. Byrd
University of Utah
United States of America
Will.Byrd@cs.utah.edu

Melisachew Wudage Chekol
University of Mannheim
Germany
mel@informatik.uni-mannheim.de

Michael Codish
Ben-Gurion University of the Negev
Israel
mcodish@cs.bgu.ac.il

Michael Frank
Ben-Gurion University of the Negev
Israel
frankm@cs.bgu.ac.il

Daniel P. Friedman
Indiana University
United States of America
dfried@indiana.edu

Tiantian Gao
Stony Brook University
United States of America
tiagao@cs.stonybrook.edu

Michael Hanus
Institut für Informatik
Germany
mh@informatik.uni-kiel.de

Jason Hemann
Indiana University
United States of America
jhemann@indiana.edu

Manuel V. Hermenegildo
Universidad Politécnica de Madrid
Spain
manuel.hermenegildo@upm.es

Daniela Inclezan
Miami University
United States of America
inclezd@miamioh.edu

Ankush Israney
Drexel University
United States of America
avi26@drexel.edu

Nicola Leone
University of Calabria
Italy
leone@mat.unical.it

Marco Manna
University of Calabria
Italy
manna@mat.unical.it

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei

Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Amit Metodi
Cadence Design Systems
ametodi@cadence.com

Matthew Might
University of Utah
United States of America
might@cs.utah.edu

José F. Morales
IMDEA Software Institute
Spain
josef.morales@imdea.org

Morad Muslimany
Ben-Gurion University of the Negev
Israel
moradm@cs.bgu.ac.il

Rolf Schwitter
Macquarie University
Australia
Rolf.Schwitter@mq.edu.au

Heiner Stuckenschmidt
University of Mannheim
Germany
heiner@informatik.uni-mannheim.de

Nataliia Stulova
IMDEA Software Institute
Spain
nataliia.stulova@imdea.org

Paul Tarau
University of North Texas
United States of America
paul.tarau@unt.edu

Pierfrancesco Veltri
University of Calabria
Italy
veltri@mat.unical.it

Qinglin Zhang
Miami University
United States of America
zhangq7@miamioh.edu

Neng-Fa Zhou
CUNY Brooklyn College
United States of America
nzhou@sci.brooklyn.cuny.edu

Entity Set Expansion from the Web via ASP

Weronika T. Adrian[1], Marco Manna[2], Nicola Leone[3], Giovanni Amendola[4], and Marek Adrian[5]

1 **Universty of Calabria, Arcavacata di Rende (CS), Italy and AGH University of Science and Technology, Kraków, Poland**
w.adrian@mat.unical.it
2 **Universty of Calabria, Arcavacata di Rende (CS), Italy**
manna@mat.unical.it
3 **Universty of Calabria, Arcavacata di Rende (CS), Italy**
leone@mat.unical.it
4 **Universty of Calabria, Arcavacata di Rende (CS), Italy**
amendola@mat.unical.it
2 **Universty of Calabria, Arcavacata di Rende (CS), Italy**
m.adrian@mat.unical.it

Abstract

Knowledge on the Web in a large part is stored in various *semantic resources* that formalize, represent and organize it differently. Combining information from several sources can improve results of tasks such as recognizing similarities among objects. In this paper, we propose a logic-based method for the problem of entity set expansion (ESE), i.e. extending a list of named entities given a set of seeds. This problem has relevant applications in the Information Extraction domain, specifically in automatic lexicon generation for dictionary-based annotating tools. Contrary to typical approaches in natural languages processing, based on co-occurrence statistics of words, we determine the common category of the seeds by analyzing the semantic relations of the objects the words represent. To do it, we integrate information from selected Web resources. We introduce a notion of an *entity network* that uniformly represents the combined knowledge and allow to reason over it. We show how to use the network to disambiguate word senses by relying on a concept of *optimal common ancestor* and how to discover similarities between two entities. Finally, we show how to expand a set of entities, by using answer set programming with external predicates.

1998 ACM Subject Classification D.1.6 Logic Programming, H.3.3 Information Search and Retrieval, H.3.5 Online Information Services, I.2.4 Knowledge Representation Formalisms and Methods, I.2.7 Natural Language Processing

Keywords and phrases answer set programming, entity set expansion, information extraction, natural language processing, word sense disambiguation

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.1

1 Introduction

The problem we study in this paper goes under the name of *entity set expansion*. Informally, given a set of words called *seeds*, the goal is to extend the original set with new words of the same "sort". For example, starting from *Rome* and *Budapest*, one could expand these seeds with *Amsterdam*, *Athens*, *Berlin*, ..., *Warsaw*, and *Zagreb*, which are also capital cities of European Union member states. But is this the most appropriate way? In fact, an alternative expansion could be made by *Amsterdam*, *Berlin*, *Dublin*, ..., *Paris*, and *Prague*, which are

also Europe's capitals situated on rivers. Moreover, *Rome* is not only a 'capital', but also a 'drama television series', a 'female deity', and many other things, while *Budapest* is also a 'film series' and a 'rock band', apart from being a 'capital' too. Hence, which is the "best" common sort putting together the original words? Are they 'capitals' or 'films'?

The problem of entity set expansion has been widely studied in the NLP community. Several approaches have been proposed, including *bootstrapping algorithms* [10, 13] that starting from a set of seed words, discover patterns in which they appear in a given corpus, then using those patterns find more examples and repeat the process until an end condition is met. The patterns are usually lexico-syntactic, but recently more advanced ways of characterizing the words in a category to be expanded have also been proposed, e.g. *word embeddings* [3, 6]. As far as the corpus is concerned, the great potential of the Web has been recognized and used to extend the set of seeds [4, 11, 9]. Nevertheless, there are several problems with existing approaches. First, the inherent limitation of statistical methods when analyzing the words, is that they do not take into consideration possible different senses of the same word, domain-specific exceptions etc., so methods that work well for generating general lexicons may fail for domains-specific dictionaries, when the meaning of words do not always agree with statistics [5]. Moreover, the intended categories is usually as simple as a 'person' or a 'city'. We would like to go a step further and be able to discover more descriptive categories, by including the properties of the objects represented with the seeds.

To this end, we propose to use knowledge available on the Web, specifically, stored in selected *semantic resources* that represent semantics of objects, their categorization and relations with other objects. We want to use these resources to disambiguate word meanings and discover commonalities among objects represented with them. Once the common category is singled out, we want to utilize the Web-harvested knowledge, specifically stored in the hypernym database built automatically using Hearst-like patterns. This way, our approach combines structural knowledge from the semantic resources for analyzing and understanding objects, and Web-harvested knowledge to extend the set. We propose a model of an *entity network* that will allow to integrate information from several sources and reason over it. We also propose an implementation in answer set programming with external predicates to query semantic resources.

2 Semantic resources and entity networks

Currently, more and more machine-readable knowledge is available on the Web in a form of *semantic resources*, such as WordNet [7], Wikidata (`http://wikidata.org`), BabelNet [8] and WebIsADatabase[12]. These knowledge bases formalize and organize human knowledge about the world in different scope and manners, focus on various dimensions and areas.

To integrate knowledge from such resources, we propose a model that can uniformly represent information acquired from them. The basic notions we will use are *(semantic) entities* and an *(entity) network*. An entity is a pair $\varepsilon = \langle id(\varepsilon), names(\varepsilon)\rangle$, where $id(\varepsilon)$ is the identifier of ε, and $names(\varepsilon)$ is a set of (human readable) terms describing ε. From a syntactic viewpoint, $id(\varepsilon)$ is a set of strings of the form $src : code$ where src identifies the semantic resource where ε is classified, and $code$ is the local identifier within source src, while $names(\varepsilon)$ is a set of strings. For example, $\varepsilon = \langle\{$`wn:08864547`, `wd:Q40`, `bn:00007266n`$\}, \{$`Austria`, `Republic of Austria`$\}\rangle$ is an entity representing the object in real world, the Republic of Austria, referred to in WordNet (abbreviation `wn` with identifier `08864547`), Wikidata (abbreviated `wd` with item identifier `Q40`), and BabelNet (with synset identifier `bn:00007266n`).

From a semantic point of view, entities may refer to three different kinds of objects. Namely, they can either point to (i) individuals, called hereafter *instances*, such as in the previous example, where the entity denotes a particular country, or (ii) concepts that generalize a *class* of objects e.g., $\varepsilon = \langle\{$ `wn:08562388`, `wd:Q6256`, `bn:00023235n` $\}, \{$`country`$\}\ \rangle$ or to (iii) *(semantic) relations* that hold between two objects e.g., $\varepsilon = \langle\{$`wd:P31`$\}, \{$`instance of, is a, ...`$\}\ \rangle$ or $\varepsilon = \langle\{$ `wd:P131`$\}, \{$`is located in, ...`$\}\rangle$ etc. For convenience, we group the entities representing instances and classes into one group, so-called *(knowledge) units*.

An *(entity) network* is a four-tuple $\mathcal{N} = \langle Uni, Rel, Con, type\rangle$ where: (i) Uni is a set of knowledge units, both classes and instances; (ii) Rel is a set of semantic relations; (iii) $Con \subseteq Uni \times Uni$ is a set of ordered pairs denoting that two units are connected via some (one or more) semantic relations; and (iv) $type : Con \to (2^{Rel} \setminus \emptyset)$ is a function that assigns to each connection a set of semantic relations.

To construct an entity network, one may start from either a set of words (i.e. raw strings) or a set of units. To this end, we use Answer Set Programming (ASP) [1] enriched with *external predicates* [2]. External predicates refer to functions (implemented separately) that encapsulate requests to semantic resources and acquire responses. It is easy to extend the current implementation with a new semantic resource: one needs only to add a new rule with an external predicate – a new (typically very simple) function, compatible with the resource's API. In fact, all the rules that query external sources establish new connections and are of the general form: *newCon*(*InputUnit*, *OutputUnit* [, *optionalArg*]*) `:-` *unitID*(*InputUnit*), &*externalPredicate*(*InputUnit*; *OutputUnit*) [, *optionalRestriction*]*.

For example, given a set of seed words, each encoded with a logical fact `seed(`*SeedWord*`)`, we use the following rules in ASP to establish connections `senseOf` from a set of seed words W to the first node of the network representing the meanings of words:

```
senseOf(SeedWord, SenseID) :- seed(SeedWord), &babelnetSense(SeedWord; SenseID).
```

Once we have the first units in the entity network, we can further expand the network with relations of the represented objects, such as hypernymy:

```
bnISA(ID, PID, PLv) :- babelnetID(ID,Lv), &babelnetISA(ID; PID),
                       babelnetDepth(BabelNetMax), Lv<BabelNetMax, PLv = Lv +1.
```

In this rule, the external predicate `&babelnetISA(`*Input; Output*`)` query BabelNet for hypernyms (superclasses) of the given input, and the optional restrictions set the limits on the number of applications of the rule.

3 Entity set expansion

Given the set W of seeds, we solve ESE by performing three major steps described next.

First, we need to understand the objects represented by the seed words. To this end, we construct a network $\mathcal{N}_1$ from W and expand the hypernymy relations via ASP as described above. From WordNet we acquire the taxonomy up to the most general concept: "entity". From other sources, in which the taxonomy is not guaranteed to be acyclic, we get the hypernyms only up to some fixed level. The output of the expansion is a directed acyclic graph, in which we determine the "correct" meanings of the seed words by identifying the "optimal common ancestors" for W. Basically, we identify via ASP program with weak constraints minimum spanning subtrees in the graph, containing one meaning for each word and one common ancestor. The output of this step is a set of units U.

Once we know the single optimal combination of word senses, we proceed to the phase of *category recognition*. In this step, we create a network $\mathcal{N}_2$ starting from the above set of U. First, we determine the common supertypes by asking the semantic resources for hypernyms up to a given limit. Then, we expand the other semantic relations that connect U to other objects. For each shared relation we obtain a set of units that are the *image* of the relation w.r.t. the seed units. If the set is a singleton, it means that the seed units are connected via the relation to the same unit. If it is not the case, then we treat the image set as the new set of seeds, for which we repeat the process of finding a common supertype and analyzing common relations (the iteration limit can be set). The output of this step is a sub-network $\mathcal{N}_3$ that describes the common properties and will be used as "verifier" in the next step.

Finally, to discover new objects of the target category, we query the WebIsADatabase for instances of the common ancestors of the seeds, setting a threshold to filter out noisly results. The obtained set of new candidate instances is then evaluated against the properties discovered earlier. We check if they are hyponyms of one of the desired common ancestors, and if they share the relations discovered for the seed set.

4 Conclusion

The problem of entity set expansion is not a new topic. With our approach, we address the old problem in a moderm semantic way. Instead of relying strictly on lexical level, we utilize the online *semantic resources*, that were not available before, to build a better representation, based on semantic relations. Our approach allows to leverage existing resources, and we believe that with the theoretical foundations and efficient ASP-based implementation of prototypes, that we already have, we can build, with further engineering effort, an integrated, configurable system.

References

1 Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

2 Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. I-DLV: the new intelligent grounder of DLV. *Intelligenza Artificiale*, 11(1):5–20, 2017.

3 José Camacho-Collados, Mohammad Taher Pilehvar, and Roberto Navigli. A unified multilingual semantic representation of concepts. In *Proc. of ACL'15*, pages 741–751, 2015.

4 Oren Etzioni, Michael Cafarella, Doug Downey, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Unsupervised named-entity extraction from the web: An experimental study. *Artificial Intelligence*, 165(1):91–134, 2005.

5 Ruihong Huang and Ellen Riloff. Inducing domain-specific semantic class taggers from (almost) nothing. In *Proc. of ACL 2010*, pages 275–285, 2010.

6 Ignacio Iacobacci, Mohammad T. Pilehvar, and Roberto Navigli. Sensembed: Learning sense embeddings for word and relational similarity. In *Proc. of ACL 2015*, pages 95–105, 2015.

7 George A Miller. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41, 1995.

8 Roberto Navigli and Simone Paolo Ponzetto. Babelnet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193:217–250, 2012.

9 Patrick Pantel, Eric Crestan, Arkady Borkovsky, Ana-Maria Popescu, and Vishnu Vyas. Web-scale distributional similarity and entity set expansion. In *Proc. of EMNLP 2009*, pages 938–947, 2009.

10 Ellen Riloff and Rosie Jones. Learning dictionaries for information extraction by multi-level bootstrapping. In *Proc. of AAAI '99 and IAAI '99*, pages 474–479, 1999.

11 Luís Sarmento, Valentin Jijkoun, Maarten de Rijke, and Eugenio Oliveira. "more like these": growing entity classes from seeds. In *Proc. of CIKM'07*, pages 959–962, 2007.

12 Julian Seitner, Christian Bizer, Kai Eckert, Stefano Faralli, Robert Meusel, Heiko Paulheim, and Simone Paolo Ponzetto. A large database of hypernymy relations extracted from the web. In *Proc. of LREC'16*, 2016.

13 Michael Thelen and Ellen Riloff. A bootstrapping method for learning semantic lexicons using extraction pattern contexts. In *Proc. of EMNLP '02*, pages 214–221, 2002.

The Pyglaf Argumentation Reasoner*

Mario Alviano

Department of Mathematics and Computer Science, University of Calabria, Italy
alviano@mat.unical.it

Abstract

The PYGLAF reasoner takes advantage of circumscription to solve computational problems of abstract argumentation frameworks. In fact, many of these problems are reduced to circumscription by means of linear encodings, and a few others are solved by means of a sequence of calls to an oracle for circumscription. Within PYGLAF, Python is used to build the encodings and to control the execution of the external circumscription solver, which extends the SAT solver GLUCOSE and implements an algorithm based on unsatisfiable core analysis.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases abstract argumentation frameworks, propositional circumscription, minimal model enumeration, incremental solving

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.2

1 Introduction

Circumscription [9] is a nonmonotonic logic formalizing common sense reasoning by means of a second order semantics, which essentially enforces to minimize the extension of some predicates. With a little abuse on the definition of circumscription, the minimization can be imposed on a set of literals, so that a set of negative literals can be used to encode a maximization objective function. Since many semantics of abstract argumentation frameworks are based on a preference relation that essentially amount to inclusion relationships, PYGLAF (`http://alviano.com/software/pyglaf/`) uses circumscription as a target language to solve computational problems of abstract argumentation frameworks.

PYGLAF is implemented in Python and uses CIRCUMSCRIPTINO [1], a circumscription solver extending the SAT solver GLUCOSE [7] with the unsatisfiable core based algorithm ONE [6] enhanced by *reiterated progression shrinking* [3], native support for cardinality constraints as in WASP [4, 5, 8], and polyspace model enumeration [2]. Linear reductions are used for all considered semantics. The communication between PYGLAF and CIRCUMSCRIPTINO is handled in the simplest possible way, that is, via stream processing. In fact, the communication is limited to a single invocation of the circumscription solver.

2 From Argumentation Frameworks to Circumscription

Let $\mathcal{A}$ be a fixed, countable set of *atoms* including $\bot$. A *literal* is an atom possibly preceded by the connective $\neg$. For a literal ℓ, let $\overline{\ell}$ denote its *complementary literal*, that is, $\overline{p} = \neg p$ and $\overline{\neg p} = p$ for all $p \in \mathcal{A}$; for a set L of literals, let $\overline{L}$ be $\{\overline{\ell} \mid \ell \in L\}$. *Formulas* are defined as

* The paper has been partially supported by the Italian Ministry for Economic Development (MISE) under project "PIUCultura – Paradigmi Innovativi per l'Utilizzo della Cultura" (n. F/020016/01-02/X27), and under project "Smarter Solutions in the Big Data World (S2BDW)" (n. F/050389/01-03/X32) funded within the call "HORIZON2020" PON I&C 2014-2020, and by Gruppo Nazionale per il Calcolo Scientifico (GNCS-INdAM).

usual by combining atoms and the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$. A *theory* is a set T of formulas including $\neg\bot$; the set of atoms occurring in T is denoted by $atoms(T)$. An *assignment* is a set A of literals such that $A \cap \overline{A} = \emptyset$. An *interpretation* for a theory T is an assignment I such that $(I \cup \overline{I}) \cap \mathcal{A} = atoms(T)$. Relation $\models$ is defined as usual. I is a *model* of a theory T if $I \models T$. Let $models(T)$ denote the set of models of T.

Circumscription applies to a theory T and a set P of literals subject to minimization. Formally, relation $\leq^P$ is defined as follows: for I, J interpretations of T, $I \leq^P J$ if $I \cap P \subseteq J \cap P$. $I \in models(T)$ is a *preferred model* of T with respect to $\leq^P$ if there is no $J \in models(T)$ such that $I \not\leq^P J$ and $J \leq^P I$. Let $CIRC(T, P)$ denote the set of preferred models of T with respect to $\leq^P$.

An *abstract argumentation framework* (AF) is a directed graph G whose nodes $arg(G)$ are arguments, and whose arcs $att(G)$ represent an attack relation. An *extension* E is a set of arguments. The *range* of E in G is $E_G^+ := E \cup \{x \mid \exists yx \in att(G) \text{ with } y \in E\}$. In the following, several AF semantics are characterized by means of circumscription.

For each argument x, an atom a_x is possibly introduced to represent that x is attacked by some argument that belongs to the computed extension E, and an atom r_x is possibly introduced to enforce that x belongs to the range E_G^+:

$$attacked(G) := \left\{ a_x \leftrightarrow \bigvee_{yx \in att(G)} y \;\middle|\; x \in arg(G) \right\} \tag{1}$$

$$range(G) := \left\{ r_x \rightarrow x \vee \bigvee_{yx \in att(G)} y \;\middle|\; x \in arg(G) \right\} \tag{2}$$

The following set of formulas characterize semantics not based on preferences:

$$conflict\text{-}free(G) := \{\neg\bot\} \cup \{\neg x \vee \neg y \mid xy \in att(G)\} \tag{3}$$

$$admissible(G) := conflict\text{-}free(G) \cup attacked(G) \cup \{x \rightarrow a_y \mid yx \in att(G)\} \tag{4}$$

$$complete(G) := admissible(G) \cup \left\{ \left(\bigwedge_{yx \in att(G)} a_y \right) \rightarrow x \;\middle|\; x \in arg(G) \right\} \tag{5}$$

$$stable(G) := complete(G) \cup range(G) \cup \{r_x \mid x \in arg(G)\} \tag{6}$$

Note that in (4) truth of an argument x implies that all arguments attacking x are actually attacked by some true argument. In (5), instead, whenever all attackers of an argument x are attacked by some true argument, argument x is forced to be true. Finally, in (6) all atoms of the form r_x are forced to be true, so that the range of the computed extension has to cover all arguments.

Below are several AF semantics with natural characterization in circumscription:

$$co(G) := CIRC(complete(G), \emptyset) \tag{7}$$

$$st(G) := CIRC(stable(G), \emptyset) \tag{8}$$

$$gr(G) := CIRC(complete(G), arg(G)) \tag{9}$$

$$pr(G) := CIRC(complete(G), \overline{arg(G)}) \tag{10}$$

$$sst(G) := CIRC(complete(G) \cup range(G), \{\neg r_x \mid x \in arg(G)\}) \tag{11}$$

$$stg(G) := CIRC(conflict\text{-}free(G) \cup range(G), \{\neg r_x \mid x \in arg(G)\}) \tag{12}$$

All of the above semantics are supported by PYGLAF, which provides a uniform developing platform for reasoning on argumentation frameworks.

References

1 Mario Alviano. Model enumeration in propositional circumscription via unsatisfiable core analysis. *TPLP*, 17, 2017. To appear. URL: https://arxiv.org/abs/1707.01423.

2 Mario Alviano and Carmine Dodaro. Answer set enumeration via assumption literals. In Giovanni Adorni, Stefano Cagnoni, Marco Gori, and Marco Maratea, editors, *AI*IA 2016: Advances in Artificial Intelligence - XVth International Conference of the Italian Association for Artificial Intelligence, Genova, Italy, November 29 - December 1, 2016, Proceedings*, volume 10037 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2016. doi:10.1007/978-3-319-49130-1_12.

3 Mario Alviano and Carmine Dodaro. Anytime answer set optimization via unsatisfiable core shrinking. *TPLP*, 16(5-6):533–551, 2016. doi:10.1017/S147106841600020X.

4 Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In Pedro Cabalar and Tran Cao Son, editors, *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013. doi:10.1007/978-3-642-40564-8_6.

5 Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors, *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, September 27-30, 2015. Proceedings*, volume 9345 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2015. doi:10.1007/978-3-319-23264-5_5.

6 Mario Alviano, Carmine Dodaro, and Francesco Ricca. A maxsat algorithm using cardinality constraints of bounded size. In Qiang Yang and Michael Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2677–2683. AAAI Press, 2015. URL: http://ijcai.org/Abstract/15/379.

7 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL: http://ijcai.org/Proceedings/09/Papers/074.pdf.

8 Carmine Dodaro, Mario Alviano, Wolfgang Faber, Nicola Leone, Francesco Ricca, and Marco Sirianni. The birth of a WASP: preliminary report on a new ASP solver. In Fabio Fioravanti, editor, *Proceedings of the 26th Italian Conference on Computational Logic, Pescara, Italy, August 31 - September 2, 2011*, volume 810 of *CEUR Workshop Proceedings*, pages 99–113. CEUR-WS.org, 2011. URL: http://ceur-ws.org/Vol-810/paper-106.pdf.

9 John McCarthy. Circumscription - A form of non-monotonic reasoning. *Artif. Intell.*, 13(1-2):27–39, 1980. doi:10.1016/0004-3702(80)90011-9.

Reasoning on Anonymity in Datalog+/-*

Giovanni Amendola[1], Nicola Leone[2], Marco Manna[3], and Pierfrancesco Veltri[4]

1 **DEMACS, University of Calabria, Italy**
`amendola@mat.unical.it`
2 **DEMACS, University of Calabria, Italy**
`leone@mat.unical.it`
3 **DEMACS, University of Calabria, Italy**
`manna@mat.unical.it`
4 **DEMACS, University of Calabria, Italy**
`veltri@mat.unical.it`

Abstract

In this paper we empower the ontology-based query answering framework with the ability to reason on the properties of "known" (non-anonymous) and anonymous individuals. To this end, we extend Datalog+/- with epistemic variables that range over "known" individuals only. The resulting framework, called $\mathsf{datalog}^{\exists,\mathcal{K}}$, offers good and novel knowledge representation capabilities, allowing for reasoning even on the anonymity of individuals. To guarantee effective computability, we define shyK, a decidable subclass of $\mathsf{datalog}^{\exists,\mathcal{K}}$, that fully generalizes (plain) Datalog, enhancing its knowledge modeling features without any computational overhead: OBQA for shyK keeps exactly the same (data and combined) complexity as for Datalog. To measure the expressiveness of shyK, we borrow the notion of uniform equivalence from answer set programming, and show that shyK is strictly more expressive than the DL $\mathcal{ELH}$. Interestingly, shyK keeps a lower complexity, compared to other Datalog+/- languages that can express this DL.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Datalog, query answering, Datalog+/-, ontologies, expressiveness

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.3

1 Introduction

In ontology-based query answering (OBQA), a user query q is evaluated over a logical theory consisting of an extensional database D paired with an ontology Σ. This problem is attracting the increasing attention of scientists in various fields of Computer Science, ranging from *artificial intelligence* [3, 13, 17] to *database theory* [5, 18, 6] and *logic* [22, 4, 19]. In this context, Description Logics [2] and Datalog$^{\pm}$ [10] have been recognized as the two main families of formal knowledge representation languages to specify Σ, while conjunctive queries represent the most common and studied formalism to express q.

In this paper we concentrate on the Datalog$^{\pm}$ family whose intent is to collect all expressive extensions of Datalog which are based on existential quantification, equality-generating dependencies, negative constraints, negation, and disjunction. In particular, the

* The paper has been partially supported by the Italian Ministry for Economic Development (MISE) under project "PIUCultura – Paradigmi Innovativi per l'Utilizzo della Cultura" (n. F/020016/01-02/X27), and under project "Smarter Solutions in the Big Data World (S2BDW)" (n. F/050389/01-03/X32) funded within the call "HORIZON2020" PON I&C 2014-2020.

"plus" symbol refers to any possible combination of these extensions, while the "minus" one imposes at least decidability, as already the presence of existential quantification alone makes OBQA undecidable in the general case [23, 9], also because the ontology universe may be enlarged with infinitely many "anonymous" individuals to satisfy existential rules.

Originally, this family was introduced with the aim of "closing the gap between the Semantic Web and databases" [11] to provide the *Web of Data* with scalable formalisms that can benefit from existing database technologies. And in fact it generalizes well-known subfamilies of Description Logics —such as $\mathcal{EL}$ [8] and *DL-Lite* [1]— collecting the basic tractable languages for OBQA in the context of the Semantic Web and databases. Currently, Datalog$^{\pm}$ has evolved as a major paradigm and an active field of research. As a result, a number of syntactic properties that guarantee decidability by implicitly limiting the generation and the "interaction" among anonymous individuals have been single out: *weak-acyclicity* [16], *guardedness* [9], *linearity* [11], *stickiness* [12], and *shyness* [21].

2 Datalog$^{\pm}$ with epistemic variables

Following the Datalog$^{\pm}$ philosophy, on the one hand we extend the family with a novel knowledge representation feature that allows for consciously reasoning on the properties of "known" (non-anonymous) and anonymous individuals in different ways; on the other hand, we single out sufficient syntactic conditions to ensure decidability. More specifically, we start from a classical well-established setting introduced by [11], where an ontology Σ is a set of $\mathsf{datalog}^{\exists}$ (a.k.a. "existential") rules of the form $\forall\mathbf{X}\forall\mathbf{Y}\,(\phi(\mathbf{X},\mathbf{Y}) \rightarrow \exists\mathbf{Z}\; p(\mathbf{X},\mathbf{Z}))$, and a query $q(\mathbf{X})$ is an expression of the form $\exists\mathbf{Y}\,(\phi(\mathbf{X},\mathbf{Y}), \neg p_1(\mathbf{Z}_1), \ldots, \neg p_k(\mathbf{Z}_k))$, where symbol '$\neg$' stands for default negation (a.k.a. negation as failure). Both in rules and queries, $\phi(\mathbf{X},\mathbf{Y})$ is a conjunction of atoms; also, $p(\mathbf{X},\mathbf{Z})$ and each $p_i(\mathbf{Z}_i)$ are atoms (with $\mathbf{Z}_i \subseteq \mathbf{X}\cup\mathbf{Y}$ required as standard "safety" condition). Then, we enhance the framework with *epistemic variables* (denoted by $\widehat{X}$, $\widehat{Y}$, $\widehat{Z}$, ...) that complement standard variables (denoted by X, Y, Z, ...) adding some interesting modeling capabilities. We call $\mathsf{datalog}^{\exists,\mathcal{K}}$ the resulting language. Roughly, epistemic variables range over "known" (non-anonymous) individuals only; while standard variables range over all individuals.

Consider for example the database $D = \{person(john), person(tim), hasFather(john, tim)\}$, and the $\mathsf{datalog}^{\exists,\mathcal{K}}$ ontology Σ consisting of the following rules:

$$person(X) \rightarrow \exists Y\; hasFather(X,Y) \qquad (\rho_1)$$

$$hasFather(X,\widehat{Y}) \rightarrow hasKnownFather(X) \qquad (\rho_2)$$

The first rule states that every person has a father (note that the father is guaranteed to exists, even if he could be an unknown individual); while the second, using the epistemic variable $\widehat{Y}$, specifies the persons who have a known father. From ρ_1 we derive that also *tim* has a father, but his identity is not known. (Technically, this is represented by some fact $hasFather(tim, \eta)$ where η is a term not occurring in the ontology domain, namely an "anonymous" individual or a "null" in the database terminology.) From ρ_2, $hasKnownFather(john)$ is derived, while $hasKnownFather(tim)$ is not derived as $\widehat{Y}$ ranges over the ontology domain $\{john, tim\}$. Let us now consider the query:

$$q(X) = \exists Y\; hasFather(X,Y),\; \neg hasKnownFather(X),$$

which asks for those people whose father is not known. By evaluating q over $D\cup\Sigma$, we get the answer $X = tim$, as expected since the identity of tim's father is not known; while the father of john is known.

Roughly, epistemic variables behave as the operator K already in use in Description Logics [14]. In this context, expression $\mathsf{K}C$ is interpreted as the set of individuals on the ontology domain that are instances of the concept C in all models, or equivalently the "known" objects which are instances of C. For example, the inclusion axiom $\mathsf{K}C \sqsubseteq D$ of Description Logics can be expressed via the $\mathsf{datalog}^{\exists,\mathcal{K}}$ rule $C(\widehat{X}) \rightarrow D(\widehat{X})$.

3 A decidable and expressive language: shyK

Besides enhancing the KR features of the framework, we want to ensure decidable query answering. To this end, we single out a $\mathsf{datalog}^{\exists,\mathcal{K}}$ language called shyK. Intuitively, consider a database D, a shyK ontology Σ, and a chase step $\langle \rho, h \rangle(I) = I'$ employed in the construction of $chase(D, \Sigma)$ (for more details on the chase procedure, see [20]). The syntactic properties underlying shyK guarantee that: (1) if a standard variable X occurs in two different atoms of the body of ρ, then $h(X)$ is a constant; and (2) if two different standard variables X and Y occur both in the head of ρ and in two different atoms of the body of ρ, then $h(X) = h(Y)$ implies $h(X)$ is a constant.

We reduce the evaluation of conjuntive queries over shyK ontologies to the evaluation of conjunctive queries over shy ontologies.

▶ **Theorem 1.** QEVAL *for conjunctive queries over* shyK *ontologies is:* (*i*) ExpTime*-complete in combined complexity, and* (*ii*) PTime*-complete in data complexity.*

To measure the expressiveness of shyK we compare it with the DL $\mathcal{ELH}$. More precisely, consider an ontology Σ. We say that an ontology Σ' is *equivalent to* Σ if, for each database D over $\mathcal{R}(\Sigma)$, it holds that $chase(D, \Sigma')|_{\mathcal{R}(\Sigma)} = chase(D, \Sigma)$. Hence, a class $\mathcal{C}_1$ is *strictly more expressive* than $\mathcal{C}_2$ if (*i*) for each $\Sigma \in \mathcal{C}_2$ there is $\Sigma' \in \mathcal{C}_1$ being equivalent to Σ, and (*ii*) for some $\Sigma \in \mathcal{C}_1$ there is no $\Sigma' \in \mathcal{C}_2$ being equivalent to Σ.

We show that shyK is strictly more expressive than $\mathcal{ELH}$. In particular, we provide a polynomial-time transformation that maps each $\mathcal{ELH}$ ontology to an equivalent shyK one. Since the reduction is polynomial, this also shows that $\mathcal{ELH}$ is no more succinct than shyK.

▶ **Theorem 2.** shyK *is strictly more expressive than* $\mathcal{ELH}$.

4 Conclusion

In conclusion, we extend datalog rules to deal with both epistemic variables and existential quantification. The resulting framework offers good and novel knowledge representation capabilities, allowing for reasoning even on the anonymity of individuals. We define shyK, a $\mathsf{datalog}^{\exists,\mathcal{K}}$ language that supports epistemic variables, fully generalizes datalog, and that guarantees the decidability of OBQA for conjunctive queries with epistemic variables. Finally, to measure the expressive power of shyK, we borrow the notion of uniform equivalence from answer set programming [15]. Then, we compare shyK with the well-known Description Logic $\mathcal{ELH}$ [7, 8], showing that shyK is strictly more expressive than $\mathcal{ELH}$. Interestingly, shyK keeps a lower computational complexity, compared to other Datalog$^{\pm}$ languages that can express this Description Logic (namely guarded and its extensions).

References

1 Alessandro Artale, Diego Calvanese, Roman Kontchakov, and Michael Zakharyaschev. The dl-lite family and relations. *J. Artif. Intell. Res.*, 36:1–69, 2009.

2 Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

3 Jean-François Baget, Michel Leclère, Marie-Laure Mugnier, and Eric Salvat. On rules with existential variables: Walking the decidability line. *Artif. Intell.*, 175(9-10):1620–1654, 2011.

4 Vince Bárány, Georg Gottlob, and Martin Otto. Querying the guarded fragment. *Logical Methods in Computer Science*, 10(2), 2014.

5 Meghyn Bienvenu, Balder ten Cate, Carsten Lutz, and Frank Wolter. Ontology-based data access: A study through disjunctive datalog, csp, and MMSNP. *ACM Trans. Database Syst.*, 39(4):33:1–33:44, 2014.

6 Pierre Bourhis, Marco Manna, Michael Morak, and Andreas Pieris. Guarded-based disjunctive tuple-generating dependencies. *ACM TODS*, 41(4), November 2016.

7 Sebastian Brandt. On subsumption and instance problem in ELH w.r.t. general tboxes. In *Proceedings of DL 2004*, 2004.

8 Sebastian Brandt. Polynomial time reasoning in a description logic with existential restrictions, GCI axioms, and - what else? In *Proceedings of ECAI 2004*, pages 298–302, 2004.

9 Andrea Calì, Georg Gottlob, and Michael Kifer. Taming the infinite chase: Query answering under expressive relational constraints. *J. Artif. Intell. Res.*, 48:115–174, 2013.

10 Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. Datalog$^{\pm}$: a unified approach to ontologies and integrity constraints. In *Proceedings of ICDT 2009*, pages 14–30, 2009.

11 Andrea Calì, Georg Gottlob, and Thomas Lukasiewicz. A general datalog-based framework for tractable query answering over ontologies. *J. Web Sem.*, 14:57–83, 2012.

12 Andrea Calì, Georg Gottlob, and Andreas Pieris. Towards more expressive ontology languages: The query answering problem. *Artif. Intell.*, 193:87–128, 2012.

13 Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Data complexity of query answering in description logics. *Artif. Intell.*, 195:335–360, 2013.

14 Francesco M. Donini, Maurizio Lenzerini, Daniele Nardi, Andrea Schaerf, and Werner Nutt. Adding epistemic operators to concept languages. In *Proceedings of KR 1992*, pages 342–353, 1992.

15 Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of ICLP 2003*, pages 224–238, 2003.

16 Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

17 Georg Gottlob, Stanislav Kikot, Roman Kontchakov, Vladimir Podolskii, Thomas Schwentick, and Michael Zakharyaschev. The price of query rewriting in ontology-based data access. *Artif. Intell.*, 213:42–59, 2014.

18 Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Query rewriting and optimization for ontological databases. *ACM Trans. Database Syst.*, 39(3):25:1–25:46, 2014.

19 Georg Gottlob, Andreas Pieris, and Lidia Tendera. Querying the guarded fragment with transitivity. In *Proceedings of ICALP 2013*, pages 287–298, 2013.

20 David S. Johnson and Anthony C. Klug. Testing containment of conjunctive queries under functional and inclusion dependencies. *J. Comput. Syst. Sci.*, 28(1):167–189, 1984. URL: `https://doi.org/10.1016/0022-0000(84)90081-3`, `doi:10.1016/0022-0000(84)90081-3`.

21 Nicola Leone, Marco Manna, Giorgio Terracina, and Pierfrancesco Veltri. Efficiently computable datalog∃ programs. In *Proceedings of KR 2012*, 2012.

22 Héctor Pérez-Urbina, Boris Motik, and Ian Horrocks. Tractable query answering and rewriting under description logic constraints. *J. Applied Logic*, 8(2):186–209, 2010.

23 Riccardo Rosati. The limits of querying ontologies. In *Proceedings of ICDT 2007*, pages 164–178, 2007.

Rule Based Temporal Inference

Melisachew Wudage Chekol[1] and Heiner Stuckenschmidt[2]

1 **Data and Web Science Group, University of Mannheim, Mannheim, Germany**
`mel@informatik.uni-mannheim.de`
2 **Data and Web Science Group, University of Mannheim, Mannheim, Germany**
`heiner@informatik.uni-mannheim.de`

Abstract

Time-wise knowledge is relevant in knowledge graphs as the majority facts are true in some time period, for instance, (Barack Obama, president of, USA, 2009, 2017). Consequently, temporal information extraction and temporal scoping of facts in knowledge graphs have been a focus of recent research. Due to this, a number of temporal knowledge graphs have become available such as YAGO and Wikidata. In addition, since the temporal facts are obtained from open text, they can be weighted, i.e., the extraction tools assign each fact with a confidence score indicating how likely that fact is to be true. Temporal facts coupled with confidence scores result in a probabilistic temporal knowledge graph. In such a graph, probabilistic query evaluation (marginal inference) and computing most probable explanations (MPE inference) are fundamental problems. In addition, in these problems temporal coalescing, an important research in temporal databases, is very challenging. In this work, we study these problems by using probabilistic programming. We report experimental results comparing the efficiency of several state-of-the-art systems.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases temporal inference, temporal knowledge graphs, probabilistic reasoning

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.4

1 Introduction

The advance of open information extraction and data mining have guided the automatic construction and completion of big knowledge graphs (KGs). This is often done by crawling the web and extracting facts and relations using machine learning techniques, for instance NELL [4]. Some of the KGs contain high quality, human curated facts for instance YAGO [20], Wikidata [30], DBpedia [1] and some contain probabilistic facts for instance Google's Knowledge Vault [10], NELL, DeepDive [27], ReVerb [15], and ProbKB [6]. Additionally, present-day knowledge graphs contain partially temporally annotated facts.

Time-wise knowledge can be found from patient and employee histories to event and streaming data. For instance, the fact that Barack Obama was the president of USA is valid only from 2009 to 2017. When such facts are derived via machine learning techniques, they are produced with some degree confidence indicating how likely they are to be true. We refer to knowledge graphs (KGs) that contain temporally annotated probabilistic facts as *probabilistic temporal knowledge graphs*. The emergence of such KGs poses new challenges in probabilistic reasoning. In this respect, recently, Markov logic networks (MLNs) is used for conflict resolution in uncertain temporal KGs [5]. In particular, the authors investigate maximum a-posteriori inference (MAP–computing the most probable temporal KG) for debugging noisy temporal data. This problem is known to be intractable. On the other hand, the main focus of this work is to investigate marginal inference (computing the probabilities

of queries) through the use of ProbLog (a probabilistic programming language). Moreover, we leverage a tractable rule language called probabilistic soft logic for computing most probable explanations (MPE) for probabilistic temporal KGs.

In parallel with fact and relation extraction, schema induction and rule learning[1] have been widely investigated [7, 25, 26]. Most rule learning systems, such as AIME+ [16], SHERLOCK [26], and ProbFOIL [25], produce weighted Horn rules that can be encoded into ProbLog and probabilistic soft logic (PSL). Besides such rules can be conveniently *temporalized* to take into account the temporal scope of facts. Thereby, alleviating the notoriously difficult problem of temporal rule learning. As an example consider a probabilistic Horn rule which represents "a person lives in the same place where the company she works for is located":

$0.5 :: \mathsf{livesin}(x, z) :- \mathsf{worksfor}(x, y), \mathsf{locatedin}(y, z).$

This rule can be temporalized as:

$0.5 :: \mathsf{livesin}(x, z, t, t') :- \mathsf{worksfor}(x, y, t_b, t_e), \mathsf{locatedin}(y, z, t'_b, t'_e), \mathsf{overlaps}(t_b, t_e, t'_b, t'_e)$,

where the overlaps tests if there is an overlap between the intervals $[t_b, t_e]$ and $[t_b, t'_e]$ and $[t, t'] = [t_b, t_e] \cap [t'_b, t'_e]$. ProbLog is equipped with built-in predicates that allow to represent the predicate overlaps, this permits us to perform inference in temporal knowledge graphs where inference rules often contain arithmetic predicates to determine temporal overlap. Similarly, PSL provides a programming interface for creating user defined functions.

A relevant problem in probabilistic temporal KGs is *temporal coalescing*. Temporal coalescing is the process of merging facts with identical non-temporal arguments and adjacent or overlapping time-intervals. This problem has been thoroughly investigated in the database community in a non-probabilistic setting (look for instance [3]). In this paper we investigate two approaches for coalescing probabilistic temporal facts. Overall, the contributions of this paper are the following: (i) we study temporal coalescing in a probabilistic setting and propose efficient algorithms, (ii) we provide coalescing-based query rewriting for marginal and MPE inference tasks, and (iii) we perform extensive experimental analysis over the Wikidata KG.

Outline. The paper is organized as follows. Next, we briefly introduce ProbLog, PSL and knowledge graphs. In Section 3, we present temporal coalescing of KGs. Section 4 describes representation of probabilistic temporal KGs in ProbLog. We briefly outline temporal KGs in probabilistic soft logic (Section 5). In Section 6, we evaluate our approach using Wikidata and four state-of-the-art systems. We review related work in Section 7 and provide conclusion in Section 8.

2 Background

2.1 ProbLog

ProbLog is a probabilistic extension of Prolog [22]. A ProbLog program consists of a set of definite clauses with their corresponding probabilities. The probability of a clause indicates the likelihood of that clause, i.e., it is a measure of how likely the clause is to hold or be true. Given a ProbLog program $T = \{p_1 :: c_1, \ldots, p_n :: c_n\}$, each ground c_i (a clause with no variables) is called a *fact*. Facts allow us to represent triples of a KG and definite clauses enable to encode background knowledge or schema of a KG. A ProbLog program $T = \{p_1 :: c_1, \ldots, p_n :: c_n\}$ defines a probability distribution over ground logic programs

[1] Often first order Horn rules are produced by inductive logic programming and machine learning techniques.

$L \subseteq L_T = \{c_1, \ldots, c_n\}$ of T:

$$P(L|T) = \prod_{c_i \in L} p_i \prod_{c_i \in L_T \setminus L} (1 - p_i)$$

2.1.1 Marginal query

An important problem in ProbLog is computing the probability of a query (known as the success probability). The probability of a query q over a ProbLog program T is obtained as:

$$P(q|T) = \sum_{L \subseteq L_T} P(q|L)P(L|T), \quad P(q|L) = \begin{cases} 1 \text{ if } \exists\theta : L \models q\theta \\ 0 \text{ otherwise} \end{cases}$$

where θ denotes a possible substitution for q. $P(q|T)$ is the probability that q is provable over the distribution of logic programs of T. In this paper, we make use of ProbLog to compute the marginal probabilities of temporal queries over probabilistic temporal KGs. Another important problem in ProbLog is computing the most probable explanation of a set of facts.

2.1.2 MPE inference

MPE inference is the task of finding the most likely interpretation (joint state) of all non-evidence facts NE given some evidence facts E, i.e., $\text{argmax}_{ne}\, P(NE = ne \mid E = e)$. MPE inference in a probabilistic temporal KG corresponds to computing the most probable temporal KG with the highest probability. Another Horn-based probabilistic logic programming language is PSL. MPE inference in PSL is known to be tractable.

2.2 Probabilistic Soft Logic

Probabilistic Soft Logic (PSL) uses first-order logic to specify templates for Hinge-Loss Markov Random Fields (HR-MRFs) [2]. A PSL knowledge base KB contains a set of formulae $\{(F_i, w_i)\}$, where F_i is a disjunction of literals (an atom or its negation) and $w_i \in \mathbb{R}_{\geq 0}$ is a real-valued weight. A formula F_i is called *hard* (resp. *soft*) if its weight $w_i = \infty$ (resp. $w_i \in \mathbb{R}_{\geq 0}$). A hard formula must be true in all the possible worlds of the KB. To avoid confusion (and perform experimental comparison of ProbLog and PSL over the same dataset) of probabilities in ProbLog and weights in PSL, we assume that the weights in PSL are between 0 and 1. A PSL formula is written as: $H_1 \vee \cdots \vee H_m \leftarrow B_1 \wedge \cdots \wedge B_n$, where $H_1, \ldots, H_m$ are predicates in the head and $B_1, \ldots, B_n$ are predicates in the body of the rule. PSL defines a probability distribution over all possible interpretations I of all ground atoms. The probability density function for I is defined as:

$$f(I) = Z^{-1} exp[-\sum_{r \in R} w_r (d_r(I))^p]; \quad z = \int_I exp[-\sum_{r \in R} w_r (d_r(I))^p],$$

where R denotes the set of ground formulas; w_r denotes the weight of rule r; Z is the normalization constant; p provides two different loss functions: linear ($p = 1$) and quadratic ($p = 2$), the choice of a loss function depends on the application[2]; $d_r(I)$ is r's *distance to*

[2] The "linear loss function chooses interpretations that completely satisfy one rule at the expense of higher distance from satisfaction for conflicting rules, whereas the quadratic loss function favors interpretations that satisfy all rules to some degree" [2].

satisfaction under the interpretation I, $d_r(I) = max\{0, I(r_{body}) - I(r_{head})\}$, which is defined using Lukasiewicz's relaxation of Boolean operators $\wedge$, $\vee$, and $\neg$:

$$I : b_i \to [0,1] \qquad I(b_i \wedge b_j) = max\{0, I(b_i) + I(b_j) - 1\}$$
$$I(\neg b_i) = 1 - I(b_i) \qquad I(b_i \vee b_j) = min\{I(b_i) + I(b_j), 1\}$$

The most important inference problem in PSL is MPE inference.

2.2.1 MPE inference

A most probable explanation query corresponds to the problem of finding a most probable state of a probabilistic KB. Formally, MPE inference is the task of computing $P(y|x)$ the most probable assignment for a set of variables y given observations x: $\text{argmax}_y P(y|x)$. This problem is known to be tractable. Hence, it allows to perform MPE inference efficiently in probabilistic temporal KGs.

2.3 Knowledge Graphs

A knowledge graph is a set of triples that can be encoded in the W3C standard RDF data model [19]. Let I and L be two disjoint sets denoting the set of IRIs (identifying resources) and literals (character strings or some other type of data), respectively. We abbreviate the union of these sets ($\mathsf{I} \cup \mathsf{L}$) as IL . A triple of the form $(s, r, o) \in \mathsf{I} \times \mathsf{I} \times \mathsf{IL}$ is called an *RDF triple*[3]; s is the *subject*, r is the *predicate*, and o is the *object* of the triple. Each triple can be thought of as an edge between the subject and the object labeled by the predicate; hence a set of RDF triples is referred to as an *RDF graph*. We use the term *knowledge graph* loosely to refer to an RDF graph. Automatic extraction of facts produces highly calibrated probabilistic annotations for the facts. Additionally, some of these facts can be timestamped with time intervals. Knowledge graphs that contain such facts are called probabilistic temporal KGs.

2.3.1 Probabilistic Temporal Knowledge Graphs

A temporal knowledge graph is obtained by labeling triples in the graph with a temporal element [18]. The temporal element represents the time period in which a triple is valid, i.e., the *valid time* of the triple. We consider a discrete time domain T as a linearly ordered finite sequence of *time points*; for instance, days, minutes, or milliseconds. The finite domain assumption ensures that there are finitely many possible worlds in ProbLog and Probabilistic Soft Logic (see discussion in subsequent sections). A *time interval* is an ordered pair $[t_b, t_e]$ of time points, with $t_b \leq t_e$ and $t_b, t_e \in \mathsf{T}$, which denotes the closed interval from t_b to t_e[4]. We will work with the interval-based temporal domain to define our data model.

▶ **Definition 1** (Temporal KG)**.** A temporal KG is a KG where some facts $\mathsf{g}_i = (s, r, o)$ in the graph have a valid time $[t_b, t_e]$, i.e., $\mathsf{g}_i = (s, r, o, t_b, t_e)$. We refer to g_i as a *temporal fact*.

For a temporal KG G, its *snapshot* at time t is the graph $G(t)$ (the non-temporal KG): $G(t) = \{(s, r, o) \mid (s, r, o, t, t) \in G\}$. The KG associated with a temporal KG, denoted $u(G)$, is $\bigcup_t G(t)$, the union of the graphs $G(t)$. We define *temporal entailment* as follows. For temporal

[3] We do not consider blank nodes.
[4] It is possible to extend to other interval-based representations such as $[t_b, t_e)$, left-closed, right-open interval.

KGs G_1 and G_2, $G_1 \models_t G_2$ if $G_1(t) \models G_2(t)$ for each t; $\models_t$ denotes temporal entailment [18] and $\models$ is the standard RDF entailment [19]. An extension of temporal KGs with uncertainty is studied in [5]. The authors leverage Markov logic networks to provide semantics. In this paper, we employ ProbLog and Probabilistic soft logic (PSL) for representation and reasoning tasks.

▶ **Definition 2** (Probabilistic temporal KG)**.** A probabilistic temporal knowledge graph is a tuple $K = (G, F)$ with $G = \{(g_1, p_1), \ldots, (g_n, p_n)\}$ a temporal KG where each temporal fact $g_i \in G$ is labeled with a probability p_i; and $F = \{(f_1, p_1), \ldots, (f_m, p_m)\}$ is a finite set of first order logic formulas representing background knowledge or schema and p_i denotes the probability of clause f_i.

In this paper, we restrict F to be Horn clauses that express temporal inference rules and use the Problog syntax to represent them (discussed in the next section).

▶ **Example 3.** Consider the following probabilistic temporal KG representing Michael Jordan's playing career:

(g_1)	$(\mathsf{MichaelJordan}, \mathsf{playsfor}, \mathsf{ChicagoBulls}, 1984, 1993)$	0.99
(g_2)	$(\mathsf{MichaelJordan}, \mathsf{playsfor}, \mathsf{WashingtonWizards}, 2001, 2003)$	0.7
(g_3)	$(\mathsf{ChicagoBulls}, \mathsf{locatedin}, \mathsf{Chicago}, 1966, \mathtt{now})$	1.0

The time point `now` denotes the current time instant or *until changed* from temporal databases. The temporal fact g_3 represents the fact that "the basketball team Chicago Bulls is located in the Chicago city from 1966 until `now`".

Temporalizing inference rules. Knowledge graphs often contain background knowledge to control and manage the quality of data and query answers. These background knowledge can be captured by first order logic. However, most rule extraction systems produce Horn rules (that we call inference rules), for instance, the SHERLOCK system [26] and ontological pathfinding (OP) [7] efficiently learn several thousands of first order Horn rules. Horn clauses are expressive enough to represent complex schema axioms (background knowledge). The extraction of Horn rules with temporal constraints is notoriously difficult and has been afforded limited attention from the research community. However, the majority of the rules produced by rule learning systems can be converted into temporal rules by using the following: (i) add two variables t_b and t_e that represent time points of intervals to each predicate (i.e., $r(x, y)$ becomes $r(x, y, t_b, t_e)$) and (ii) if the number of predicates in the body is more than one, introduce an arithmetic predicate which is used to test temporal overlap (i.e., $r_3(x, z) :- r_1(x, y) \wedge r_2(y, z)$ becomes $r_3(x, z, t, t') :- r_1(x, y, t_b, t_e) \wedge r_2(y, z, t'_b, t'_e) \wedge \mathsf{overlaps}(t_b, t_e, t'_b, t'_e)$ where $[t, t'] = [t_b, t_e] \cap [t'_b, t'_e]$). We refer to this process as *temporalizing inference rules*.

3 Coalescing Probabilistic Temporal Knowledge Graphs

Coalescing is a technique used in temporal databases for duplicate elimination [9, 3]. Coalescing has a number of advantages: reduces the size of the probabilistic temporal KG, avoids incorrect answers in query evaluation. For instance, consider the query '*did Michael Jordan play for Chicago bulls from 1984 to 1996?*' on the temporal facts of Fig. 1 before coalescing. The result is no, however, the same query on the coalesced fact (brown part of the figure), returns yes. Uncoalesced facts can arise in various cases: during query evaluation via projection or union operations, by not enforcing coalescing in update or insertion operations,

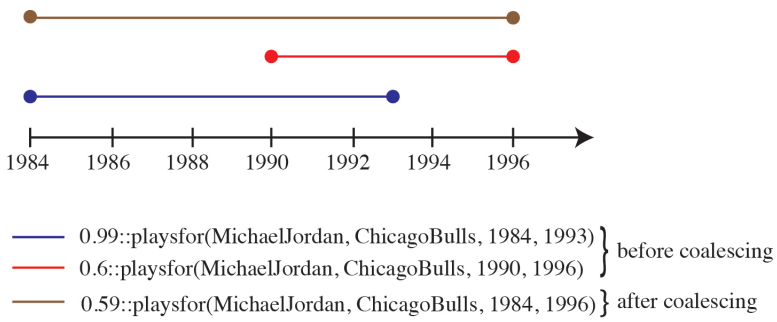


Figure 1 Coalescing probabilistic temporal facts.

and through information extraction from diverse sources or accuracy of the extractor. In this section, we discuss temporal coalescing of KGs, in the next section, we address coalescing for query evaluation.

A temporal knowledge graph G is called duplicate-free, if for all pairs of facts $p(s, o, t_b, t_e)$, $p(s, o, t'_b, t'_e) \in G$, it holds that: $[t_b, t_e] \cap [t'_b, t'_e] = \emptyset$. In other words, if the non-temporal terms of two temporal facts are the same, then their temporal terms must be disjoint (non-overlapping). Temporal coalescing is the process of merging facts with identical non-temporal arguments and adjacent or overlapping time-intervals. Often, a temporal database is assumed to be duplicate-free and coalescing is done by merging the time intervals of facts with the same non-temporal arguments. However, in a probabilistic setting, to perform coalescing is not straightforward because it is not clear what the probability of the new (coalesced) fact should be and it could be dependent of the application. Performing coalescing on a probabilistic temporal knowledge graph removes duplicates.

▶ **Definition 4** (Coalescing). Formally, two probabilistic temporal facts $p_1 :: r(s, o, t_b, t_e)$, $p_2 :: r'(s', o', t'_b, t'_e)$ can be coalesced if $s = s'$, $r = r'$, $o = o'$ and the overlap of $[t_b, t_e]$ and $[t'_b, t'_e]$ is non-empty. The probability of the coalesced fact $p_3 :: r(s, o, [t, t'])$ with $[t, t'] = [t_b, t_e] \cup [t'_b, t'_e]$ is computed using Table 1.

Rule-based coalescing. One approach to coalescing is to use the following rule-based technique. In order to coalesce all the facts of a probabilistic temporal KG, we can construct Horn rules for each relation in the KG. Thus, in ProbLog, rule-based coalescing can be done as follows: for each relation r_i in a probabilistic temporal KG K, use the following rule:

$$r_i(x, y, t, t') :- r_i(x, y, t_b, t_e), r_i(x, y, t'_b, t'_e), t \textbf{ is } min(t_b, t'_b), t' \textbf{ is } max(t_e, t'_e), t_b \leq t'_e,\ t'_b \leq t_e.$$

The expression $t_b \leq t'_e,\ t'_b \leq t_e$ tests temporal overlap of the intervals $[t_b, t_e]$ and t'_b, t'_e. Besides, **is**, *min*, and *max* are built-in predicates representing assignment, minimum, and maximum functions respectively. The probability of the coalesced facts is the product when done in ProbLog, however, this can be replaced by using Table 1 (to compute probabilities). This approach uses one rule for each relation. If a KG has 80000 relations, we need the same number of coalescing rules to coalesce the KG. Hence, this operation can be very expensive, however, it is done only once. Furthermore, this operation can be done more efficiently outside the ProbLog setting.

▶ **Example 5.** Consider coalescing the probabilistic temporal facts shown in Figure 1 using the rule-based approach. This operation merges the two facts into one with the probability of the new fact being the product of the two.

Table 1 Computing probabilities of coalesced facts based on Allen's interval relations. The computation holds for the inverse relations. g_{new} is obtained by merging (taking the union) of the intervals of the two facts and $p_{new} = p_1 + p_2 - p_1 * p_2$, based on the product rule of probability.

Interval relation	Temporal facts	Coalesced fact
Equal	(g_1, p_1) (g_2, p_2)	$(g_1, max(p_1, p_2))$
During	(g_1, p_1) (g_2, p_2)	(g_2, p_2)
Starts	(g_1, p_1) (g_2, p_2)	(g_2, p_2)
Finishes	(g_1, p_1) (g_2, p_2)	(g_2, p_2)
overlaps	(g_1, p_1) (g_2, p_2)	(g_{new}, p_{new})
Meets	(g_1, p_1) (g_2, p_2)	(g_{new}, p_{new})
Before	(g_1, p_1) (g_2, p_2)	{}

Algorithmic coalescing. Another approach for coalescing probabilistic temporal facts is based on the following. In short, the algorithm continues as follows: for each relation r search all temporal facts with the same non-temporal (s, r, o) elements, order these facts according to their time period, for overlapping time periods build the maximum time period (union of time periods), delete the temporal facts whose time periods are contained in the maximum time period, and finally assign probability to the coalesced temporal facts using Table 1. There are two important tasks in probabilistic temporal KGs: marginal and MPE inference. In order to perform these reasoning, we use ProbLog.

4 ProbLog-based Representation of Probabilistic Temporal KGs

In order to compute the probabilities of temporal queries, we represent probabilistic temporal KGs in ProbLog. This can be done by introducing a predicate for each temporal fact. To elaborate, we use a simple correspondence between temporal facts (s, r, o, t_b, t_e) and ProbLog predicates of the form $r(s, o, t_b, t_e)$ such that s, r, o, t_b, and t_e are temporal KG symbols. Thus, whenever a temporal fact (s, r, o, t_b, t_e) is satisfied, the corresponding predicate $r(s, o, t_b, t_e)$ is satisfied and the converse also holds. More formally, given a probabilistic temporal KG $K = (G, F)$ with $G = \{(g_1, p_1), \ldots, (g_n, p_n)\}$ and $F = \{(f_1, p_1), \ldots, (f_m, p_m)\}$, a ProbLog representation K_p of K is obtained as follows: (i) replace each probabilistic temporal fact $(g_i = (s, r, o, t_b, t_e), p_i)$ with $p_i :: r(s, o, t_b, t_e) \in G_p$, and (ii) replace each Horn formula (f_j, p_j) with $p_j :: f_j \in F_p$ where $p_j \in (0, 1]$. Thus, $K_p = (G_p, F_p)$ is a ProbLog program with G_p being facts and F_p is a set of definite clauses.

▶ **Example 6.** Consider the following temporal ProbLog KG $K_p = (G_p, F_p)$ based on Wikidata[5], G_p contains the facts g_4–g_6 obtained by translating the temporal facts g_1–g_3 of Example 3 into ProbLog. F_p contains the formulas f_1–f_3 shown below. The formula f_1 expresses the fact that if someone *plays for* (resp. *works for*, resp. *coaches*) a club located in a city over an overlapping period of time, then that person likely lives in the same city as where the club is located. f_2 represents if a person plays for a team and that the team participates in a league over an overlapping period of time, then that person plays in that league. Finally, f_3 is used to express disjointness of temporal relations, to be precise, a person cannot play (resp. work, resp. coach) for two different teams at the same time.

$$(f_1)\quad 0.5 :: \mathsf{livesin}(x,z,t,t') :- \mathsf{playsfor/worksfor/coaches}(x,y,t_b,t_e), \mathsf{locatedin}(y,z,t'_b,t'_e), t \textbf{ is } \mathsf{max}(t_b,t'_b), t' \textbf{ is } \mathsf{min}(t'_b,t'_e), t \leq t'.$$

$$(f_2)\quad 0.7 :: \mathsf{playsInLeague}(x,z,t,t') :- \mathsf{playsfor}(x,y,t_b,t_e), \mathsf{teamPlaysInLeague}(y,z,t'_b,t'_e), t \textbf{ is } \mathsf{max}(t_b,t'_b), t' \textbf{ is } \mathsf{min}(t'_b,t'_e), t \leq t'.$$

$$(f_3)\quad 0.9 :: false :- \mathsf{playsfor/worksfor/coach}(x,y,t_b,t_e), \mathsf{playsfor/worksfor/coach}(x,z,t_b,t_e), y \neq z, t \textbf{ is } \mathsf{max}(t_b,t'_b), t' \textbf{ is } \mathsf{min}(t'_b,t'_e), t \leq t'.$$

$$(g_4)\quad 0.99 :: \mathsf{playsfor}(\mathsf{MichaelJordan}, \mathsf{ChicagoBulls}, 1984, 1993)$$

$$(g_5)\quad 0.7 :: \mathsf{playsfor}(\mathsf{MichaelJordan}, \mathsf{WashingtonWizards}, 2001, 2003)$$

$$(g_6)\quad 1.0 :: \mathsf{locatedin}(\mathsf{ChicagoBulls}, Chicago, 1966, \mathsf{now})$$

In f_1–f_3, max and min are ProbLog built-in predicates, **is** is an assignment operator. The arthimetic expression t **is** $\mathsf{max}(t_b, t'_b), t'$ **is** $\mathsf{min}(t'_b, t'_e), t \leq t'$ tests if there is an overlap between the intervals $[t_b, t_e]$ and $[t'_b, t'_e]$.

The semantics of probabilistic temporal KGs in ProbLog can be given in terms of Herbrand interpretations. Let C be the set of IRIs and Literals that appear in some probabilistic temporal KG $K = (G, F)$ and let $K_p = (G_p, F_p)$ be its ProbLog representation, the Herbrand base of F_p can be constructed by replacing all the variables in F_p with the constants in C. For a finite set C and a set of time points T, each temporal fact in K_p can be mapped, using a substitution θ, into a subset of the Herbrand base of F with respect to C and T. A Herbrand interpretation is a subset of the Herbrand base. A Herbrand interpretation H is a Herbrand model of F_p iff it satisfies all groundings of the formulas in F_p. As in ProbLog, since the schema $F_p = \{p_i :: f_i\}$ of K_p is fixed and there is a one-to-one mapping between ground f_i clauses and Herbrand interpretations, a probabilistic temporal KG also defines a probability distribution over its Herbrand interpretations [22]. The probabilities of the facts and Horn rules determine a probability distribution in ProbLog. Formally, given a probabilistic temporal KG $K = (G, F)$ and some $K' = (G', F')$ over the same set of constants (IRIs, literals and time points) such that $K' \subseteq K$, we have the following:

$$P(K'|K) = \prod_{g_i \in G' \wedge G' \cup F' \models_t g_i} p_i \prod_{g_i \in G \setminus G'} (1 - p_i),$$

where $G' \cup F' \models_t g_i$ is temporal entailment at time point t.

4.1 Marginal Inference

An important task in probabilistic knowledge bases is computing the probability of a set of facts, i.e., given a query q and a KG K, marginal inference computes the probability of the answers of q over K. In this paper, we study temporal conjunctive queries. A temporal conjunctive query is a conjunction of a set of temporal facts.

[5] `https://www.wikidata.org/`

▶ **Definition 7.** A temporal conjunctive query q is a Horn formula of the form $q :- g_1, \ldots, g_n$ where $g_i = r_i(X_i, Y_i, t_b, t_e)$ is a temporal predicate with X_i and Y_i being temporal variables or constants; and t_b, t_e are time points or variables. Given a temporal ProbLog KG $K_p = (G_p, F_p)$ and a query q over K_p, the marginal probability of q is obtained by:

$$P(q|K_p) = \sum_{K' \subseteq K_p} P(q|K') \cdot P(K'|K_p).$$

▶ **Example 8.** Consider the following queries on the temporal facts (before coalescing) of Example 5:

1. How long is Michael Jordan's playing career? $q(t_b, t_e) :- \mathsf{playsfor}(\mathsf{MichaelJordan}, Y, t_b, t_e)$.
2. Select the teams that Michael Jordan owns and played for since 1995.
   $q(Y) :- \mathsf{playsfor}(\mathsf{MichaelJordan}, Y, t_b, t_e), \mathsf{owns}(\mathsf{MichaelJordan}, Y, t'_b, t'_e))$.

In probabilistic databases and statistical relational learning, often the probabilities of queries are computed by grounding, i.e., by replacing all the variables in the queries using constants in the database. The grounding is used to generate a propositional sentence (lineage of a query) for exact inference or a graphical model for approximate computation. Similarly, temporal conjunctive queries can be grounded by evaluating queries using the techniques from temporal databases and instantiating the variables in the queries using their answers. This results in a set of ground queries, the probability of which can be computed using ProbLog. Instead, in this paper, we evaluate temporal conjunctive queries by rewriting them in ProbLog. Temporal conjunctive queries require checking interval intersection to determine the overlap of intervals in the query predicates. In order to do this, we rewrite queries. Here, we consider only the following case and leave out the rest as a future work.

One predicate query: queries that contain only a single temporal predicate (when the size of the body of the query is one), i.e., $q(W) :- r(X, Y, t_b, t_e)$, with $W \subseteq \{X, Y\}$. Here, we consider the rewriting of queries with non-temporal variable projections. Hence, q can be rewritten $q_r(W)$ by using a self join (by rewriting the same predicate with different temporal variables) as shown below:

$$\begin{aligned} q(W) &:- r(X, Y, t_b, t_e) \\ q_r(W) &:- r(X, Y, t_b, t_e), r(X, Y, t'_b, t'_e), \mathsf{overlaps}(t_b, t_e, t'_b, t'_e). \\ \mathsf{overlaps}(t_b, t_e, t'_b, t'_e) &:- t_b \leq t'_e,\ t'_b \leq t_e. \end{aligned}$$

We use the predicate overlaps() to check if the intersection of the intervals is non-empty.

4.2 MPE inference

In addition to marginal inference, we can compute most probable explanations over temporal ProbLog KGs. MPE queries are one of the most important tasks in probabilistic reasoning. These queries are useful for computing the most probable temporal KG of a probabilistic temporal KG. Formally, given a rewritten temporal conjunctive query q, some evidence e (set of temporal facts), and a temporal ProbLog KG $K_p = (G_p, F_p)$, the most likely temporal KG of q is obtained by: $\text{argmax}_q P(q|e)$.

5 Probabilistic Temporal KGs in PSL

Since PSL like ProbLog uses Horn clauses to model KBs, we present a compact description of probabilistic temporal KGs in PSL. On the other hand, ProbLog is based on Sato's

distributional semantics and PSL is defined over hinge-loss Markov random fields. A PSL knowledge base consists of a set of (weighted) Horn rules and a set of soft evidence facts. Soft refers to probabilities. Note that while in PSL the weight of the rules can be a positive real number, in this paper, we assume the weights to be between 0 and 1. This weight conversion can be done, for instance, by the *Logit* function. We make this assumption in order to perform experiments on the same datasets for ProbLog and PSL.

A temporal PSL knowledge graph $K_{psl} = (G_{psl}, F_{psl})$ contains a set of (temporal) facts $G_{psl} = \{(g_1, w_1), \ldots, (g_n, w_n)\}$ and a set of (temporal) inference rules $F_{psl} = \{(F_1, w_1), \ldots, (F_m, w_m)\}$. Given a temporal PSL knowledge graph K_{psl}, and a set of deduction rules F_{psl}, the semantics of K_{psl} is given based on a probability density function. Formally, for a given $K_{psl} = (G_{psl}, F_{psl})$ and some K'_{psl} over the same signature, the probability density function $P(K'_{psl})$ is given by:

$$P(K'_{psl}) = \begin{cases} Z^{-1} \exp\left[\sum_{\{(g_i,w_i)\in G : K'_{psl} \models_t g_i\}} w_i (d_{g_i}(K'_{psl}))^p\right] & \text{if } K'_{psl} \models_t K_{psl} \\ 0 \qquad \text{otherwise} \end{cases}$$

where Z is the normalization constant of the probability density function P; w_i is the weight of the temporal fact g_i; $d_{g_i}(K'_{psl})$ is the distance to satisfaction of g_i in K'_{psl}; and p is a loss function. Note that in MPE inference Z is not computed. The most relevant reasoning task in PSL is MPE inference which is defined in the same way as in ProbLog.

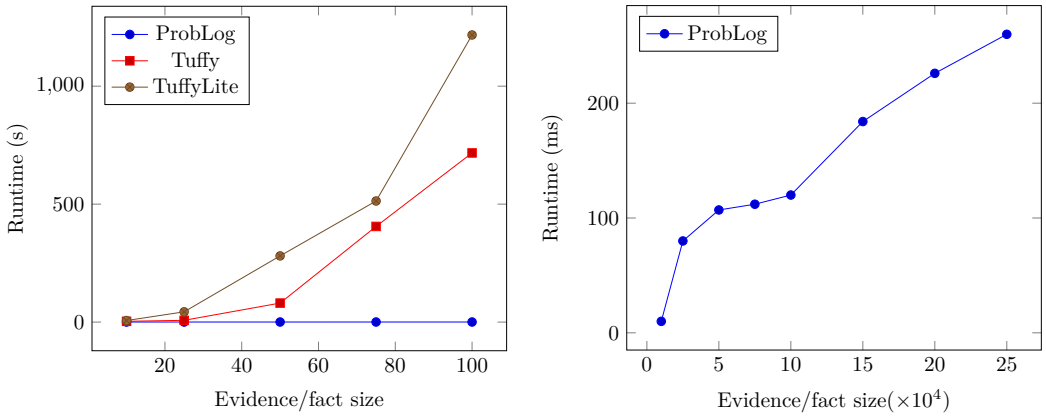
6 Experiments

We conducted two different experiments: (i) marginal inference and (ii) MPE inference. For both experiments, we carried out performance test in terms of running times by comparing four state-of-the-art solvers. We ran the experiments on a 2GHz 24-core processor with 386GB of RAM running Debian 8.

- **Tools.** We used *four* different tools for our experiments: ProbLog, PSL, Tuffy [23], and TuffyLite [21]. Tuffy and TuffyLite are based on Markov logic networks (MLNs – attaches weights to first order formulas). For marginal inference experiments, we employ ProbLog, Tuffy, and TuffyLite whereas for MPE inference, we use PSL and ProbLog. Note that PSL does not support marginal inference.
- **Temporal rules.** We designed 40 different temporal inference rules (definite clauses) based on the fluents (time-varying relations) in Wikidata.
- **Evidence.** We used as evidence different size fragments of Wikidata knowledge graph. In particular, we extract a part of the KG that contains structured temporal information (obtained from various sources using open information extraction). We extracted over 6.3 million temporal facts from Wikidata. We extracted temporal facts for various fluent relations including: *playsFor*, *educatedAt*, *memberOf*, *occupation*, *spouse*, and so on.

6.1 Marginal Temporal Query Evaluation

In this experiment, we test the scalability of marginal temporal query evaluation. We carried out the experiments using ProbLog, Tuffy, and TuffyLite. We found out that Tuffy and TuffyLite hardly scale when the size (arity) of the predicates is 3 or more (we stopped the execution after one hour timeout). On one occasion, while running Tuffy on Wikidata, we



(a). Tuffy, TuffyLite, and ProbLog. (b). ProbLog.

Figure 2 Marginal inference: runtime comparison over fixed query and varying data size.

noticed that its grounded database is 400GB large, thereby our execution eventually ran out of memory. To elaborate, the ground size in Tuffy and TuffyLite can be extremely large, for instance, for a test data size of 75 temporal facts and 40 inference rules, the grounding database contains 4,112,784 tuples. Comparatively, ProbLog uses a different grounding technique that reduces that ground size reasonably. Therefore, it is recommended to use ProbLog for marginal inference tasks in probabilistic temporal KGs when the background knowledge (inference rules) can be expressed as Horn rules. The results of the experiment are shown in Fig. 2. The reported runtimes are averaged over 5 runs for all of the solvers. As it can be seen in Fig. 2(a), due to scalability, we could only test Tuffy and TuffyLite over 10 to 100 facts. While the runtime of ProbLog increases linearly with data, the runtime of Tuffy and TuffyLite is nonlinear; it increases sharply as the size of data increases. If the inference rules of a knowledge graph can be expressed by Horn rules, then it is advisable to use ProbLog because it scales much better than Markov logic solvers (Tuffy and TuffyLite). On the other hand, if expressivity is required at the expense of scalability, Tuffy and TuffyLite can be chosen. In Fig. 2(b), we report the runtimes of marginal inference for ProbLog on large datasets ($\times 10^4$ magnitude).

6.2 MPE Inference

In this experiment, we compare the running times of ProbLog and PSL on different data sizes. Due to intractability of inference in Markov logic, we exclude Tuffy and TuffyLite. The runtimes, averaged over 5 runs, are reported in Fig. 3. As can be seen, ProbLog is faster than PSL. In addition, while the runtime of ProbLog increases linearly with respect to the datasize, the runtime of PSL does not increase linearly with respect to the size of the input data. This is due to each added incorrect temporal fact might be involved in a conflict resulting in a non trivial optimization problem. Furthermore, contrary to PSL, ProbLog handles well the temporal predicates that are used to test the overlap of temporal intervals.

7 Related Work

Temporal databases. Temporal databases have been extensively studied (see surveys [24, 28]). However, relatively few works exist on probabilistic temporal databases [11, 8]. A relational database is used to model and query temporal data, integrity constraints and deduction rules

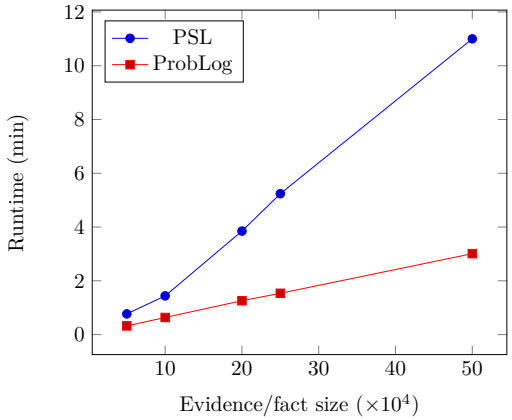


Figure 3 MPE inference: runtime comparison of ProbLog and PSL over fixed query and varying data size.

can also be specified [11]. However, these rules must be deterministic (unweighted) unlike what we do here. On the otherhand, contrary to this study where we use the valid time model, uncertain spatio-temporal databases focus on stochastically modelling trajectories through space and time (see [14] for instance).

Query evaluation in probabilistic databases is an active area of research [17, 31, 7, 29, 12]. With respect to temporal query evaluation over a probabilistic temporal knowledge base, to the best of our knowledge, there are two important studies [5] and [11]. While the former focuses on MPE inference, we study here marginal inference and deal with the problem of temporal coalescing. The later deals with marginal inference, the difference with this work are the following: (i) we consider weighted inference rules and constraints, (ii) we propose coalescing for temporal KGs, and (iii) we introduce rewriting for the coalescing of queries. In another study [13], the authors proposed an approach for resolving temporal conflicts in RDF knowledge bases. The idea is to use first-order logic Horn formulas with temporal predicates to express temporal and non-temporal constraints. However, these approaches are limited to a small set of temporal patterns and only allow for uncertainty in facts. Moreover, extending KGs using open domain information extraction, will often also lead to uncertainty about the correctness of schema information; a large variety of temporal inference rules and constraints, some of which will be domain specific, can also be the subject of uncertainty. Finally, Chen and Wang [6] debug erroneous facts by using a set of functional constraints although they do not deal with numerical and temporal facts at the same time.

8 Conclusion

Temporal reasoning is indispensable as advances in open information extraction has guided the automatic construction of temporal knowledge graphs. To perform temporal reasoning, one has to take care of the temporal scopes of facts. In addition, coalescing is necessary to prohibit errors and compact query answers. In this work, we addressed these issues. We provided theoretical as well as experimental results based on ProbLog and PSL.

A possible line of future work is to address scalability issues for ProbLog and PSL. Besides, coalescing needs to be addressed for other query operators such as union, negation, and selection.

Acknowledgements. We thank Janina Luitz for her helpful comments.

References

1 Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. Dbpedia: A nucleus for a web of open data. *The semantic web*, pages 722–735, 2007.

2 Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. arXiv:1505.04406 [cs.LG], 2015.

3 Michael H. Böhlen, Richard T. Snodgrass, and Michael D. Soo. Coalescing in temporal databases. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 180–191, 1996.

4 Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka Jr, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, volume 5, page 3, 2010.

5 Melisachew Wudage Chekol, Giuseppe Pirrò, Joerg Schoenfisch, and Heiner Stuckenschmidt. Marrying uncertainty and time in knowledge graphs. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 88–94, 2017.

6 Yang Chen and Daisy Zhe Wang. Knowledge expansion over probabilistic knowledge bases. In *SIGMOD*, pages 649–660. ACM, 2014.

7 Yang Chen, Daisy Zhe Wang, and Sean Goldberg. Scalekb: scalable learning and inference over large knowledge bases. *The VLDB Journal*, 25(6):893–918, 2016.

8 Alex Dekhtyar, Robert Ross, and VS Subrahmanian. Probabilistic temporal databases, i: algebra. *ACM Transactions on Database Systems (TODS)*, 26(1):41–95, 2001.

9 Anton Dignös, Michael H Böhlen, and Johann Gamper. Temporal alignment. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 433–444. ACM, 2012.

10 Xin Dong, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang. Knowledge Vault: A Web-Scale Approach to Probabilistic Knowledge Fusion. In *SIGKDD*, pages 601–610, 2014.

11 Maximilian Dylla, Iris Miliaraki, and Martin Theobald. A temporal-probabilistic database model for information extraction. *Proc. of the VLDB Endowment*, 6(14):1810–1821, 2013.

12 Maximilian Dylla, Iris Miliaraki, and Michael Theobald. Top-k query processing in probabilistic databases with non-materialized views. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 122–133. IEEE, 2013.

13 Maximilian Dylla, Mauro Sozio, and Martin Theobald. Resolving Temporal Conflicts in Inconsistent RDF Knowledge Bases. In *BTW*, pages 474–493, 2011.

14 Tobias Emrich, Hans-Peter Kriegel, Nikos Mamoulis, Matthias Renz, and Andreas Zufle. Querying uncertain spatio-temporal data. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 354–365. IEEE, 2012.

15 Anthony Fader, Stephen Soderland, and Oren Etzioni. Identifying relations for open information extraction. In *Proceedings of the Conference of Empirical Methods in Natural Language Processing (EMNLP '11)*, Edinburgh, Scotland, UK, July 27-31 2011.

16 Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *The VLDB Journal*, 24(6):707–730, 2015.

17 Eric Gribkoff and Dan Suciu. Slimshot: In-database probabilistic inference for knowledge bases. *PVLDB*, 9(7):552–563, 2016.

18 Claudio Gutierrez, Carlos Hurtado, and Alejandro Vaisman. Temporal RDF. In *Proc. of European Semantic Web Conference*, pages 93–107, 2005.

19 Patrick Hayes. RDF Semantics. W3C Recommendation, 2004.

20 Johannes Hoffart, Fabian M Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard De Melo, and Gerhard Weikum. Yago2: exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th international conference companion on World wide web*, pages 229–232. ACM, 2011.

21 Tushar Khot, Niranjan Balasubramanian, Eric Gribkoff, Ashish Sabharwal, Peter Clark, and Oren Etzioni. Markov logic networks for natural language question answering. *arXiv preprint arXiv:1507.03045*, 2015.

22 Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Theory and Practice of Logic Programming*, 11(2-3):235–262, 2011.

23 Feng Niu, Christopher Ré, AnHai Doan, and Jude Shavlik. Tuffy: Scaling up statistical inference in markov logic networks using an rdbms. *Proc. of the VLDB Endowment*, 4(6):373–384, 2011.

24 Gultekin Ozsoyoglu and Richard T Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

25 Luc De Raedt, Anton Dries, Ingo Thon, Guy Van den Broeck, and Mathias Verbeke. Inducing probabilistic relational rules from probabilistic examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1835–1843, 2015.

26 Stefan Schoenmackers, Oren Etzioni, Daniel S Weld, and Jesse Davis. Learning first-order horn clauses from web text. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1088–1098. Association for Computational Linguistics, 2010.

27 Jaeho Shin, Sen Wu, Feiran Wang, Christopher De Sa, Ce Zhang, and Christopher Ré. Incremental knowledge base construction using deepdive. *Proceedings of the VLDB Endowment*, 8(11):1310–1321, 2015.

28 Richard T Snodgrass. Temporal databases. In *Theories and methods of spatio-temporal reasoning in geographic space*, pages 22–64. Springer, 1992.

29 Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. Probabilistic databases. *Synthesis Lectures on Data Management*, 3(2):1–180, 2011.

30 Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

31 Hong Zhu, Caicai Zhang, Zhongsheng Cao, and Ruiming Tang. On efficient conditioning of probabilistic relational databases. *Knowl.-Based Syst.*, 92:112–126, 2016.

Logic Programming with Max-Clique and its Application to Graph Coloring (Tool Description)*

Michael Codish[1], Michael Frank[2], Amit Metodi[3], and Morad Muslimany[4]

1 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`mcodish@cs.bgu.ac.il`

2 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`frankm@cs.bgu.ac.il`

3 **Cadence Design Systems, Israel**
`ametodi@cadence.com`

4 **Department of Computer Science, Ben-Gurion University of the Negev, Beer-Sheva, Israel**
`moradm@cs.bgu.ac.il`

Abstract

This paper presents `pl-cliquer`, a Prolog interface to the `cliquer` tool for the maximum clique problem. Using `pl-cliquer` facilitates a programming style that allows logic programs to integrate with other tools such as: Boolean satisfiability solvers, finite domain constraint solvers, and graph isomorphism tools. We illustrate this programming style to solve the Graph Coloring problem, applying a symmetry break that derives from finding a maximum clique in the input graph. We present an experimentation of the resulting Graph Coloring solver on two benchmarks, one from the graph coloring community and the other from the examination timetabling community. The implementation of `pl-cliquer` consists of two components: A lightweight C interface, connecting `cliquer`'s C library and Prolog, and a Prolog module which loads the library. The complete tool is available as a SWI-Prolog module.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Logic Programming, Constraints, Maximum Clique

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.5

1 Introduction

The maximum clique problem, which is about finding the largest set of pairwise adjacent vertices in a graph, has been studied extensively both in theory [18, 10, 16, 2, 3, 14, 15, 5, 33] and in practice [29, 28, 19, 6, 7]. Several tools have been developed recently, which tackle the maximum clique problem [28, 21, 20, 29]. One of these tools is `cliquer` [27], which uses an exact branch-and-bound algorithm in the search for maximum cliques. `cliquer` consists of a collection of C routines, which can be compiled to either a standalone executable or a shared library.

* Supported by the Israel Science Foundation grant 182/13 and by the Lynn and William Frankel Center for Computer Science

The maximum clique problem is related to several other well-known graph problems. The chromatic number of any graph is bound from below by its maximum clique size, and the chromatic number of perfect graphs is identical to its maximum clique size. The size of the largest independent set in any graph G is identical to the size of the maximum clique in the complement graph of G (i.e., the graph in which u and v are adjacent if and only if they are not adjacent in G). The fast detection of cliques may also assist in the search for Ramsey graphs, which are characterized by their clique sizes. Further applications of maximum cliques can be found in bioinformatics (e.g., to infer evolutionary trees [11] and predict protein structures [32]), and in the analysis of random processes, where maximum cliques are sought in a dependency graph [13].

Integrating a (maximum) clique finding tool into a Logic Programming tool-chain has the potential to benefit search when solving hard graph problems. Such a tool can be used, for example, as a preprocessing step for instances of the graph coloring problem. After preprocessing, instances are processed by a constraint solver, or a SAT solver to find a solution. Moreover, using a tool-chain fitted for Prolog leads to new logic programming styles, such as those presented in [9], which integrates SAT solvers with Prolog, or [12] which integrates the `nauty` graph automorphism library with Prolog.

In this paper we demonstrate how a maximum clique tool can be integrated with Prolog, resulting in a library for SWI-Prolog [34] containing predicates that find maximum cliques in graphs. Moreover, we demonstrate how this library is integrated with an entire tool-chain written for Prolog. This tool-chain includes a collection of problem solving tools such as SAT and CSP solvers (e.g., [9]), finite-domain constraint compilers (e.g., [25, 26]), as well as a collection of additional symmetry breaking predicates which allow us to detect and prune equivalent solutions. We also observe that our tool-chain, augmented by `cliquer`, is competitive for graph coloring with results reported in the literature, and in some cases obtains previously unreported solutions.

The rest of this paper is structured as follows. Section 2 introduces the basic definitions and notations used throughout. Section 3 describes and illustrates the use of the `pl-cliquer` interface. Section 4 introduces the graph coloring problem, as well as a graph coloring solver, which is extended with optimizations based on the identification of a maximum clique. This example demonstrates how `pl-cliquer` augments an existing tool-chain. Section 5 defines the exam timetabling problem, as an application of the graph coloring problem. Sections 4 and 5 also present selected results of an experimentation using the standard graph coloring and exam timetabling benchmarks. Section 6 contains some technical details on `pl-cliquer`, and Section 7 concludes. Appendices A and B present results on the full benchmarks.

2 Preliminaries

A graph $G = (V, E)$ consists of a set of vertices $V = [n] = \{\ 1, \ldots, n\ \}$ and a set of edges $E \subseteq V \times V$. In the context of the tools that we present in this paper, it is natural to consider only simple graphs. Meaning, we assume that graphs are undirected, and there are no self loops or multiple edges. The degree of a vertex $v \in V$ is denoted by $deg(v)$ and it is equal to the number of neighbors v has. The maximum degree of a graph $G = (V, E)$ is denoted $\Delta(G)$ and it is equal to the maximum over all vertex degrees. In this paper we represent graphs as Boolean adjacency matrices, a list of `N` length-`N` lists. We also make use of the DIMACS file format for graph representation, which contains the line `e i j` for every edge (i, j) of the graph, such that $i < j$.

A function $w \colon V \to \mathbb{N} \setminus \{0\}$, which assigns positive weights to the vertices of the graph is called a positive weight function. As an abuse of notation, we denote the weight of a set of

vertices $U \subseteq V$ by $w(U) = \sum_{u \in U} w(u)$. For the sake of simplicity, throughout this paper, we assume that vertex weights are 1, thereby identifying the weight of U with its size.

A clique $C \subseteq V$ of $G = (V, E)$ is a set of vertices that are pairwise connected, meaning that if $u, v \in C$ then $(u, v) \in E$. If $|C| = k$ we call C a k-clique. The clique number of a graph G is denoted $\omega(G)$ and is the number of vertices in a largest clique of G. A clique $C \subseteq V$ such that $|C| = \omega(G)$ is called a maximum clique. The max-clique problem is about finding a maximum clique in a given graph G. The max-weighted-clique problem is about finding a clique $C \subseteq V$ such that the weight of C is maximal. The max-clique problem is known to be NP-Hard [18].

A vertex k-coloring of a graph $G = (V, E)$, for $k \in \mathbb{N}$, is a mapping $c \colon V \to [k]$ ($[k] = \{ 1, \ldots, k \}$) such that $(u, v) \in E$ implies that $c(u) \neq c(v)$. The chromatic number of a graph G is denoted $\chi(G)$ and it is the smallest number k such that the vertices of G are k-colorable. The graph coloring problem is about finding a k-coloring of a given graph G. The minimum graph coloring problem is about finding a k-coloring of a given graph G such that $k = \chi(G)$. The graph coloring problem is known to be NP-Complete, and the minimum graph coloring problem is NP-Hard [18].

3 Interfacing Prolog with `cliquer`'s C library

The `pl-cliquer` interface is implemented using the foreign language interface of SWI-Prolog [34]. The C library of `cliquer` is linked against corresponding C code written for Prolog, which contains the low-level Prolog predicates connecting `cliquer` with Prolog. These low-level predicates are wrapped by a Prolog module, which extends their functionality and provides five high-level predicates. The five high-level predicates, available through the module are:

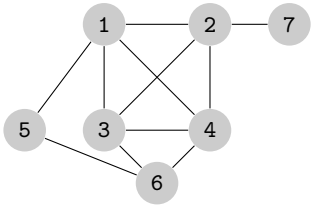
1. `graph_read_dimacs_file/5`
2. `clique_find_single/4`
3. `clique_find_multi/5`
4. `clique_find_n_sols/6`
5. `clique_print_all/6`

In the following we describe these in more detail and present several usage examples.

3.1 The `clique_read_dimacs_file/5` predicate

The call `graph_read_dimacs_file(DIMACS, NVert, Weights, Matrix, Options)` applies to convert a graph represented in the standard DIMACS format to a Prolog representation as a Boolean adjacency matrix (list of lists). Figure 1 illustrates an example graph with seven vertices, its DIMACS representation, and its corresponding adjacency matrix. If the DIMACS representation resides in the file `example.dimacs` then the following call:

```
?- graph_read_dimacs_file('example.dimacs', NVert, Weights, Matrix, []).
Matrix = [[0,1,1,1,1,0,0],
          [1,0,1,1,0,0,1],
          [1,1,0,1,0,1,0],
          [1,1,1,0,0,1,0],
          [1,0,0,0,0,1,0],
          [0,0,1,1,1,0,0],
          [0,1,0,0,0,0,0]],
NVert = 7,
Weights = [1,1,1,1,1,1,1]
```




```
p edge 7 11
e 1 2
e 1 3
e 1 4
e 1 5
e 2 3
e 2 4
e 2 7
e 3 4
e 3 6
e 4 6
e 5 6
```

```
[[0,1,1,1,1,0,0],
 [1,0,1,1,0,0,1],
 [1,1,0,1,0,1,0],
 [1,1,1,0,0,1,0],
 [1,0,0,0,0,1,0],
 [0,0,1,1,1,0,0],
 [0,1,0,0,0,0,0]]
```

Figure 1 An example graph (left), its DIMACS representation (middle), and corresponding adjacency matrix (right).

will bind (as indicated) the variable `Matrix` to the matrix depicted also on the right of Figure 1, the variable `NVert` to the number of vertices in the graph, i.e., 7, and the variable `Weights` to a list of ones (since the vertices of this graph are without weights). The last argument (an empty list in the example) specifies a list of options which may contain the options `edge(Value)` and `non_edge(Non)`. These respectively determine the symbols used to represent edges and non-edges in the adjacency matrix `Matrix`. The empty list indicates default settings. By default, edges are represented by the symbol 1 and non-edges by the symbol 0.

3.2 The `clique_find_single/4` predicate

The `clique_find_single/4` predicate provides an interface to the `cliquer` routine by the same name. It finds a single clique in the input graph. The predicate takes the form `clique_find_single(NVert, Graph, Clique, Options)`. The first argument indicates the number of vertices in the graph, the second argument is an adjacency matrix representing the graph, and the third argument is the output clique. The last argument is a list of options which fine-tune the behavior of `pl-cliquer`, constraining the size and weight of the sought after clique.

`Options` may include `min_weight(Min)` and `max_weight(Max)` limiting the clique weight to be between `Min` and `Max`. In order to obtain a maximum clique (which is the default behavior) both `Min` and `Max` should be 0. Other options are `maximal(Maximal)`, where `Maximal` is a Boolean specifying whether only maximal cliques should be found, `static_ordering(Order)` where `Order` is a permutation of $\{\ 1 \ldots n\ \}$ specifying the order in which the n vertices of the graph are backtracked over by `cliquer`'s algorithm. The option `weights(Weights)` specifies a list of n `Weights` associated with the n vertices of the graph (by default all weights are 1). The empty list indicates default settings.

For example, calling `clique_find_single(NVert, Graph, Clique, Options)` with the graph from Figure 1, unifies `Clique` with the vertices `[1,2,3,4]`, which are easily verified as the maximum clique. Notice that during the runtime of `cliquer` additional information is printed to screen, which indicates the current workload and the maximal clique found so far.

```
?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] |  ... ],
   clique_find_single(7, Graph, Clique, []).


  2/7 (max  1)  0.00 s  (0.00 s/round)
  4/7 (max  2)  0.00 s  (0.00 s/round)
  5/7 (max  3)  0.00 s  (0.00 s/round)
  7/7 (max  4)  0.00 s  (0.00 s/round)
  size=4 (max 7) 0 1 2 3

Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Clique = [1, 2, 3, 4].
```

3.3 The `clique_find_n_sols/6` predicate

The call `clique_find_n_sols(MaxSols, NVert, Graph, Sols, Total, Opts)` allows to find several cliques in the input graph. The first argument, `MaxSols`, indicates the maximal number of cliques to be found. The second argument `NVert` indicates the number of vertices in the graph, and third argument is an adjacency matrix representing the graph. The fourth argument `Sols`, is unified with the cliques that are found in the graph, and the fifth argument is unified with the number of cliques found. The last argument is the option list, which is identical to the one for `clique_find_single/4`.

For example, calling `clique_find_n_sols/6`, with the graph from Figure 1, with options indicating that at most ten cliques of weight 3 and 4 should be found, also indicating these cliques may not be maximal – returns a result that implies that six such cliques exist, five cliques of weight 3 and one clique of weight 4, and they are: `[[1, 2, 3], [2, 3, 4], [1, 2, 3, 4], [1, 3, 4], [1, 2, 4], [3, 4, 6]]`.

```
?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] | ... ],
   NVert = 7, MaxSols = 10,
   Options = [min_weight(3), max_weight(4), maximal(false)],
   clique_find_n_sols(MaxSols, NVert, Graph, Sols, Total, Options).

Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Sols = [[1, 2, 3], [2, 3, 4], [1, 2, 3, 4], [1, 3, 4], [1, 2, 4], [3, 4, 6]],
Total = 6.
```

3.4 The `clique_find_multi/5` predicate

The `clique_find_multi/5` is similar in nature to `clique_find_n_sols/6`, except that it backtracks over the cliques that are found instead of collecting them in a list. This predicate takes the form `clique_find_multi(MaxSols, NVert, Graph, Sol, Opts)`, with parameters identical in description to those of `clique_find_n_sols/6` except that `Sol` will be unified with a single clique, which will change upon backtracking.

For example, calling `clique_find_multi/5`, with the graph from Figure 1, and options indicating that at most three cliques with weights between 3 and 4 should be found, such that these cliques may not be maximal – the call returns a result that implies that at least three such cliques exist: two cliques of weight 3 and one clique of weight 4, and they are: `[1, 2, 3]`, `[2, 3, 4]` and `[1, 2, 3, 4]`.

```
?- Graph =  [[0,1,1,1,1,0,0], [1,0,1,1,0,0,1] | ... ],
   NVert = 7, MaxSols = 3,
   Options = [min_weight(3), max_weight(4), maximal(false)],
   clique_find_multi(MaxSols, NVert, Graph, Sol, Total, Options).


Graph = [[0, 1, 1, 1, 1, 0, 0] | ... ],
Sol = [1, 2, 3] ;
Sol = [2, 3, 4] ;
Sol = [1, 2, 3, 4] ;
false.
```

3.5 The `clique_print_all/6` predicate

The `clique_print_all/6` predicate is primarily intended for debugging and exploring, and it will print all the cliques of a graph which comply to certain constraints. The predicate takes the form `clique_print_all(NVert, Min, Max, Maximal, Graph, Total)`. For example, the following call to `clique_print_all/6` will print all of the cliques in the example graph of Figure 1, which contain at least two vertices and at most three vertices, and unify `Total` with the total number of lines printed.

```
 ?- NVert = 7, Min = 2, Max = 3, Maximal = false,
    Graph = [[0,1,1,1,1,0,0],
             [1,0,1,1,0,0,1],
             [1,1,0,1,0,1,0],
             [1,1,1,0,0,1,0],
             [1,0,0,0,0,1,0],
             [0,0,1,1,1,0,0],
             [0,1,0,0,0,0,0]],
    clique_print_all(NVert, Min, Max, Maximal, Graph, Total).
[1, 2]
[2, 3]
[1, 2, 3]
[1, 3]
[3, 4]
[2, 3, 4]
[1, 3, 4]
[2, 4]
[1, 2, 4]
[1, 4]
[1, 5]
[5, 6]
[4, 6]
[3, 4, 6]
[3, 6]
[2, 7]
Total = 16.
```

The predicate `clique_print_all/6` does not print out the graph, or the edge list. This is because in many cases, the graph is very large, and printing these details provides no constructive effect. Moreover, while the list of edges is available to `cliquer` when parsing the graph, it is not available to the interface of `pl-cliquer`. Nevertheless, one may extract a list of graph edges using an auxiliary predicate included with the `pl-cliquer` source code.

4 Solving the Graph Coloring Problem

The graph coloring problem has been studied vigorously, for both theoretical and practical purposes. Some of the real world applications of the graph coloring problem include: timetabling problems (e.g., [22]), frequency assignment (e.g., [1]) and register allocation (e.g., [8]). We demonstrate, using a logic program tool-chain augmented by `pl-cliquer`, a solution for the graph (vertex) coloring problem, with an application for the minimum examination timetabling schedule problem. We demonstrate how `pl-cliquer` integrates with an existing tool-chain, all specified as part of the Prolog programming process. We compare our graph coloring application to previous work using to two standard benchmark sets [7, 17], one from the graph coloring community, and the other from the exam timetabling community.

In the graph coloring problem we are given a graph $G = ([n], E)$ and a natural number $k > 0$, and we seek a labeling $c\colon [n] \to [k]$ of the graph vertices such that $(i, j) \in E \implies c(i) \neq c(j)$. In the minimum graph coloring problem, we seek the smallest k for which such a labeling exists. Both the graph coloring problem and the minimum graph coloring problem are known to be NP-Hard [18].

In practical scenarios, solving the graph coloring problem has often involved formulating it as an integer linear program [4] or as a constraint model [31]. We solve the graph coloring problem by modeling it using a constraint language and then applying the finite-domain constraint compiler `BEE` [25, 26], which stands for **B**en-Gurion University **E**qui-propagation **E**ncoder (written in Prolog), to encode it to an instance of Boolean satisfiability which is then solved using an underlying SAT solver (through its Prolog interface [9]). In the configuration for this paper we use CryptoMiniSAT v2.5.1 as the underlying SAT solver.

When describing our approach we distinguish between the constraint model for the decision problem: given a graph G and a number k — does there exist a vertex coloring with at most k colors? and the optimization problem — given a graph G what is the smallest number k for which there exists a vertex coloring with at most k colors? To model the optimization problem we apply the minimization option of the `BEE` solver which incrementally refines the number k for which the corresponding decision problem has a solution. The remainder of this section focuses on implementing the graph coloring decision problem, on which we later perform the minimization.

4.1 The Constraint Model for Graph Coloring

The basic constraint model is straightforward: for a graph $G = (V, E)$ and a given number (of colors) k, each vertex $u \in V$ is affiliated with a finite domain integer variable I_v taking a value in the range $[1, \ldots, k]$ representing its color. For each edge $(u, v) \in E$, a constraint $I_u \neq I_v$ is added to the model, forcing distinct colors between adjacent vertices.

The following `graphColoring/3` predicate lists the high level Prolog code which implements this encoding for the graph coloring problem. Given a Boolean adjacency matrix `M` for a graph with `N` vertices, we represent a coloring of that graph as a list of `N` values, such that the value in position `I` of the list is the color of vertex `I`. The `graphColoring/3` predicate takes the following form: `graphColoring(Graph, KColors, Coloring)`. The first parameter is the adjacency matrix for the graph to be colored, the second parameter is the number of colors, and the last parameter will be unified with the coloring.

The call in the first line of the definition of `graphColoring/3` generates a constraint model that is satisfiable if and only if `Graph` has a coloring with `KColors` colors. It also generates a `Map` which relates the integer variables (the colors per vertex) with their Boolean

```
% Initial Coloring
[A,B,C,D,E,F,G]

% Coloring Declaration
new_int(A,1,5)
new_int(B,1,5)
new_int(C,1,5)
new_int(D,1,5)
new_int(E,1,5)
new_int(F,1,5)
new_int(G,1,5)
```

```
% Different Colors
int_neq(A,B)
int_neq(A,C)
int_neq(A,D)
int_neq(A,E)
int_neq(B,C)
int_neq(B,D)
int_neq(B,G)
int_neq(C,D)
int_neq(C,F)
int_neq(D,F)
int_neq(E,F)
```

Figure 2 Example of the constraint model for the graph in Figure 1.

(bit-blasted) representation. The second line is a call to `BEE` which compiles the constraint model to CNF, and the third line contains a call to the underlying SAT solver. The last line of `graphColoring/3` translates the coloring from `BEE`'s (bit-blasted) representation of integers to that of Prolog.

```
graphColoring(Graph, KColors, Coloring) :-
    encode(coloring(Graph, KColors), Map, Constr),
    bCompile(Constr, CNF),
    sat(CNF),
    decode(Map, Coloring).
```

As an example, consider the graph in Figure 1. A call to `graphColoring(Graph, 5, Coloring)` would result in finding a coloring of that graph with 5 colors, and unifying `Coloring` with it. The first line of `graphColoring/3` will generate the constraint model, the second and third lines will compile and solve it. Figure 2 illustrates the constraint model for the example graph in Figure 1. The left column details the initial (unknown) coloring generated by the encoding process, as well as the variable declarations, and the right column lists the constraints forbidding adjacent vertices from having the same color. After solving the problem, in line 4 of `graphColoring/3` the coloring is translated back to Prolog values, and `Coloring` is unified with `[1,2,3,4,5,1,3]`, for example.

4.2 Throwing `pl-cliquer` into the mix

It is often the case that fast detection and iteration of cliques can be used to simplify and break symmetries in graph related problems [33, 6, 24]. In the case of graph coloring, when coloring the graph $G = (V, E)$, a clique $C \subseteq V$ must correspond to a set of vertices which are labeled by different colors, since C contains a set of mutually adjacent vertices. Given a clique $C \subseteq V$ of the graph, it is possible to arbitrarily label the vertices of C with $|C|$ different colors, thereby breaking symmetries and establishing a lower bound on $\chi(G)$.

The following `graphColoring/3` predicate is augmented with a preprocessing step which applies `pl-cliquer` in order to find a maximum clique of the input graph. This clique is passed to the encoder, to be used by it during the generation of the constraint model. The first and second lines of the predicate find a maximum clique in `Graph` and unify it with `MaxClique`. The remaining lines of code follow the same outline as in `graphColoring/3` described in Section 4.1, with the exception that `MaxClique` is taken into account as part of the encoding process.

```
% Partial Coloring
[1,2,3,4,E,F,G]

% Coloring Declaration
new_int(E,1,5)
new_int(F,1,5)
new_int(G,1,5)

% Different Colors
int_neq(1,E)
int_neq(2,G)
int_neq(3,F)
int_neq(4,F)
int_neq(E,F)
```

Figure 3 Example of the constraint model for the graph in Figure 1 with a partial coloring.

```
graphColoring(Graph, KColors, Coloring) :-
    length(Graph, NVert),
    clique_find_single(NVert, Graph, MaxClique, []),
    encode(coloring(Graph, KColors, MaxClique), Map, Constr),
    bCompile(Constr, CNF),
    sat(CNF),
    decode(Map, Coloring).
```

The encoding process initially assigns colors $\{1,\ldots,|\texttt{MaxClique}|\}$ to the vertices of `MaxClique`. The remaining vertices are associated with finite-domain variables taking values between $\{1,\ldots,\texttt{KColor}\}$, representing their color. Finally, constraints are added which prevent adjacent vertices from sharing a color.

As an example, consider the graph in Figure 1. A call to the augmented predicate `graphColoring(Graph, 5, Coloring)` would result in finding a coloring of that graph with 5 colors, and unifying `Coloring` with it. The first line of `graphColoring/3` determines the number of vertices in the graph. The second line of `graphColoring/3` finds a maximum clique and unifies `MaxClique` with a list of its vertices. For example, given the graph in Figure 1, the solver might unify `MaxClique` with `[1,2,3,4]`. The third line generates the constraint model, and the fourth line compiles the constraint model to CNF. The fifth line calls a SAT solver and the sixth line translates the result to Prolog values, resulting in either a coloring for the graph, or an unsatisfiable result, implying the graph can not be colored by `KColor` colors. Notice that since the maximum clique is known, the encoding process may introduce a partial coloring of the graph, which substantially reduces the constraint model, because constraints involving clique vertices may now be omitted. Figure 3 illustrates the constraint model for the example graph in Figure 1. The left column details the partial coloring generated by the encoding process, as well as the variable declarations, and the right column lists the constraints forbidding adjacent vertices from having the same color.

4.3 Additional Optimizations & Results

In this section we mention additional optimizations that can be made, when solving the graph coloring problem, given that the maximum clique of the graph is known. So far, the graph coloring solver is composed of: (1) a preprocessing step which embeds the colors of a maximum clique, and (2) a constraint model which is translated to CNF and solved by a SAT solver. In addition to (1) and (2), we have also implemented, as a secondary preprocessing step, the optimizations discussed in [23]. These optimizations reduce symmetries, as well as the instance size leading to an improved constraint model. Once solved, the improved model is passed through a postprocessing step, also described in [23] in order to obtain the final coloring. For the sake of brevity, we refer the interested reader to the relevant paper [23] for a complete description of the optimizations. We tested this method on the DIMACS coloring instances, which were introduced in the Second DIMACS Implementation Challenge [17].

Table 1 Satisfiable Dimacs Instances Results.

Instance	k	with `cliquer` time	with `cliquer` status	without `cliquer` time	without `cliquer` status
anna.col	11	0.02	sat(`BEE`)	0.06	sat
david.col	11	0.02	sat(`BEE`)	0.05	sat
DSJC125.1.col	5	0.06	sat	0.03	sat
DSJR500.1.col	12	0.07	sat	0.31	sat
DSJR500.5.col	122	4661.28	*memory*	∞	*memory*
fpsol2.i.1.col	65	0.03	sat(`BEE`)	4.13	sat
fpsol2.i.2.col	30	0.07	sat	1.31	sat
fpsol2.i.3.col	30	0.07	sat	1.31	sat
games120.col	9	0.05	sat	0.07	sat
huck.col	11	0.02	sat(`BEE`)	0.04	sat
inithx.i.1.col	54	0.12	sat	4.38	sat
inithx.i.2.col	31	0.3	sat	2.03	sat
inithx.i.3.col	31	0.33	sat	2.2	sat
jean.col	10	0.02	sat(`BEE`)	0.03	sat
le450_15a.col	15	1.6	sat	0.78	sat
le450_15b.col	15	0.76	sat	0.68	sat
le450_25a.col	25	0.57	sat	1.12	sat
le450_25b.col	25	0.7	sat	1.13	sat
le450_5a.col	5	0.18	sat	0.22	sat
le450_5b.col	5	0.18	sat	0.23	sat
le450_5c.col	5	0.26	sat	0.36	sat
le450_5d.col	5	0.27	sat	0.34	sat
miles1000.col	42	0.02	sat(`BEE`)	1.29	sat
miles1500.col	73	0.02	sat(`BEE`)	3.83	sat
miles250.col	8	0.02	sat(`BEE`)	0.04	sat
miles500.col	20	0.02	sat(`BEE`)	0.23	sat
miles750.col	31	0.03	sat	0.69	sat
mulsol.i.1.col	49	0.02	sat(`BEE`)	0.98	sat
mulsol.i.2.col	31	0.13	sat	0.62	sat
mulsol.i.3.col	31	0.13	sat	0.64	sat
mulsol.i.4.col	31	0.14	sat	0.64	sat
mulsol.i.5.col	31	0.14	sat	0.63	sat
myciel3.col	4	0.03	sat	0.01	sat
myciel4.col	5	0.03	sat	0.01	sat
myciel5.col	6	0.03	sat	0.01	sat
myciel6.col	7	0.04	sat	0.04	sat
myciel7.col	8	0.09	sat	0.11	sat
queen5_5.col	5	0.03	sat	0.01	sat
queen6_6.col	7	0.03	sat	0.04	sat
queen7_7.col	7	0.04	sat	0.04	sat
queen8_12.col	12	0.1	sat	0.15	sat
queen8_8.col	9	0.92	sat	0.24	sat
queen9_9.col	10	6.98	sat	14.42	sat
queen10_10.col	11	21638.0	sat	2168.42	sat
school1.col	14	66.91	sat	1.34	sat
school1_nsh.col	14	27.08	sat	1.06	sat
zeroin.i.1.col	49	0.02	sat(`BEE`)	1.05	sat
zeroin.i.2.col	30	0.03	sat	0.56	sat
zeroin.i.3.col	30	0.03	sat	0.56	sat

Selected results for this set of benchmarks are given in Tables 1 and 2, a more complete set of results can be found in (B).

Table 1 compares solving times for the satisfiable instances of the DIMACS benchmarks with and without our augmented tool-chain. Table 2 similarly compares the solving times of the unsatisfiable instances. Both tables share the same structure: the first column lists the instance name and the second column lists the coloring size we seek. In Table 1 these values k correspond to the best known colorings described in the literature [23, 24]. In Table 2 these correspond to corresponding values $k-1$, the largest values for which there does not exist a coloring. The third and fifth columns detail the solving times with and without our augmented tool-chain, and the fourth and sixth columns detail the means by which the instance was solved. The means by which an instance was solved may be one of the following: (a) sat(`BEE`) – which means that `BEE` solved the constraints without calling the

Table 2 Unsatisfiable Dimacs Instances Results.

Instance	k	with `cliquer` time	with `cliquer` status	without `cliquer` time	without `cliquer` status
anna.col	10	0.01	unsat(`cliquer`)	1.63	unsat
david.col	10	0.01	unsat(`cliquer`)	0.79	unsat
DSJC125.1.col	4	0.03	unsat(`BEE`)	0.03	unsat
DSJR500.1.col	11	0.02	unsat(`cliquer`)	4.79	unsat
DSJR500.5.col	121	4597.34	unsat(`cliquer`)	∞	*memory*
fpsol2.i.1.col	64	0.02	unsat(`cliquer`)	∞	*timeout*
fpsol2.i.2.col	29	0.02	unsat(`cliquer`)	∞	*timeout*
fpsol2.i.3.col	29	0.02	unsat(`cliquer`)	∞	*timeout*
games120.col	8	0.01	unsat(`cliquer`)	0.82	unsat
huck.col	10	0.01	unsat(`cliquer`)	0.52	unsat
inithx.i.1.col	53	0.05	unsat(`cliquer`)	∞	*timeout*
inithx.i.2.col	30	0.05	unsat(`cliquer`)	∞	*timeout*
inithx.i.3.col	30	0.05	unsat(`cliquer`)	∞	*timeout*
jean.col	9	0.01	unsat(`cliquer`)	0.29	unsat
le450_15a.col	14	0.02	unsat(`cliquer`)	535.52	unsat
le450_15b.col	14	0.02	unsat(`cliquer`)	432.15	unsat
le450_25a.col	24	0.02	unsat(`cliquer`)	∞	*timeout*
le450_25b.col	24	0.02	unsat(`cliquer`)	∞	*timeout*
le450_5a.col	4	0.01	unsat(`cliquer`)	0.18	unsat
le450_5b.col	4	0.01	unsat(`cliquer`)	0.18	unsat
le450_5c.col	4	0.01	unsat(`cliquer`)	0.3	unsat
le450_5d.col	4	0.01	unsat(`cliquer`)	0.32	unsat
miles1000.col	41	0.01	unsat(`cliquer`)	∞	*timeout*
miles1500.col	72	0.01	unsat(`cliquer`)	∞	*timeout*
miles250.col	7	0.01	unsat(`cliquer`)	0.1	unsat
miles500.col	19	0.01	unsat(`cliquer`)	∞	*timeout*
miles750.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.1.col	48	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.2.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.3.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.4.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
mulsol.i.5.col	30	0.01	unsat(`cliquer`)	∞	*timeout*
myciel3.col	3	0.03	unsat	0.01	unsat
myciel4.col	4	0.03	unsat	0.03	unsat
myciel5.col	5	0.87	unsat	109.25	unsat
myciel6.col	6	22970.5	unsat	∞	*timeout*
myciel7.col	7	∞	*timeout*	∞	*timeout*
queen5_5.col	4	0.01	unsat(`cliquer`)	0.01	unsat
queen6_6.col	6	0.03	unsat	1.68	unsat
queen7_7.col	6	0.01	unsat(`cliquer`)	0.04	unsat
queen8_12.col	11	0.01	unsat(`cliquer`)	9.09	unsat
queen8_8.col	8	15.93	unsat	∞	*timeout*
queen9_9.col	9	2295.93	unsat	∞	*timeout*
queen10_10.col	10	∞	*timeout*	∞	*timeout*
school1.col	13	66.39	unsat(`cliquer`)	592.95	unsat
school1_nsh.col	13	26.6	unsat(`cliquer`)	793.89	unsat
zeroin.i.1.col	48	0.01	unsat(`cliquer`)	∞	*timeout*
zeroin.i.2.col	29	0.01	unsat(`cliquer`)	∞	*timeout*
zeroin.i.3.col	29	0.01	unsat(`cliquer`)	∞	*timeout*

SAT solver, or (b) sat – which means that the SAT solver was called, or (c) `unsat` – which means the SAT solver was called and returned with an unsatisfiable result, or (d) unsat(`BEE`) – meaning that the `BEE` constraint compiler determined during its compilation stages that the instance is unsatisfiable, or (e) unsat(`cliquer`) – meaning that `cliquer` determined that the instance is unsatisfiable, or (f) memory – meaning that the computer ran out of memory while solving this instance, or (g) timeout – meaning that the computation for this instance did not terminate within 24 hours. All times are listed in seconds.

Table 1 illustrates that little improvement is gained from using `cliquer` when solving satisfiable instances with $k = \chi(G)$. In fact, solving times substantially increase for three satisfiable instances when `cliquer` is applied as part of a preprocessing step. Table 2 illustrates that when `cliquer` is incorporated to solve unsatisfiable instances where $k =$

Table 3 Satisfiable Toronto Instances Results.

		with `cliquer`		without `cliquer`	
Instance	k (lit)	time	status	time	status
hec-s-92	17 (17)	0.08	sat	0.17	sat
sta-f-83	13 (13)	0.03	sat	0.15	sat
yor-f-83	18 (19)	0.46	sat	0.93	sat
ute-s-92	10 (10)	0.16	sat	0.13	sat
ear-f-83	22 (22)	0.3	sat	0.62	sat
tre-s-92	20 (20)	0.51	sat	1.023	sat
lse-f-91	17 (17)	0.13	sat	0.62	sat
kfu-s-93	19 (19)	0.4	sat	0.98	sat
rye-s-93	21 (21)	0.61	sat	1.67	sat
car-f-92	27 (28)	2.08	sat	6.95	sat
uta-s-92	29 (30)	3.7	sat	4.45	sat
car-s-91	27 (27)	1580.47	sat	∞	*timeout*
pur-s-93	31 (36)	6.71	sat	32.03	sat

$\chi(G)-1$, solving times are considerably improved, often making the difference between being able to determine a result or not.

5 Exam Timetabling: An Application of Graph Coloring

The examination timetabling problem is about scheduling exams to a set of sequential timeslots, in such a way that conflicting exams are not scheduled to the same timeslot. An instance of the examination timetabling problem specifies n, the total number of exams, a set $D \subseteq [n] \times [n]$ of conflicting exams, in such a way that conflicts are symmetric (i.e., $(i,j) \in D \iff (j,i) \in D$), as well as a number of timeslots, m. We seek to find a legal schedule of size m, which is a tuple $S = \langle t_1, \ldots, t_n \rangle$ describing a mapping from exams to timeslots such that t_i is the timeslot of exam i. Each timeslot takes a value in the set $\{\ 1, \ldots, m\ \}$, and for every pair of conflicting exams i and j — the exams are not scheduled to the same timeslot (i.e., $t_i \neq t_j$).

The examination timetabling problem is reducible to the graph coloring problem by considering the *conflict graph* derived from the problem instance. This is the graph with vertices corresponding to courses and edges induced by the constraint set D such that (i,j) is an edge of the graph if and only if $(i,j) \in D$. If the graph is m-colorable, then the coloring can be taken to be a legal schedule of size m.

In the minimum examination timetabling problem, we seek the minimum m for which there exists a legal schedule. The minimum examination timetabling problem is equivalently reducible to the minimum graph-coloring problem.

We tested our approach on the Toronto timetabling instances, which were introduced by Carter *et al.,* [7]. Selected results for this set of benchmarks are given in Tables 3 and 4, the complete set of results can be found in Appendix A Table 3 lists instances for which satisfiable results were found, illustrating the coloring size that was found using our augmented tool-chain. The table follows the same description as that of Table 1, except that the second column lists the optimal coloring size we found as well as the best known coloring we found in literature [30]. Table 4 lists the corresponding unsatisfiable instances which prove that the colorings found in Table 3 are optimal. The columns of this table follow the same description as those of Table 2.

Table 3 illustrates the improvement gained by using `cliquer` when solving satisfiable instances. The table lists four instances for which the best known colorings are improved, while colorings for the remaining instances match the best known results from literature. Moreover, Table 4 illustrates that when `cliquer` is incorporated to solve unsatisfiable instances, solving

Table 4 Unsatisfiable Toronto Instances Results.

		with `cliquer`		without `cliquer`	
Instance	k (lit)	time	status	time	status
hec-s-92	16	0.01	unsat(`cliquer`)	3736.02	unsat
sta-f-83	12	0.01	unsat(`cliquer`)	25.43	unsat
yor-f-83	17	0.01	unsat(`cliquer`)	∞	*timeout*
ute-s-92	9	0.1	unsat(`cliquer`)	0.65	unsat
ear-f-83	21	0.24	unsat	∞	*timeout*
tre-s-92	19	0.01	unsat(`cliquer`)	∞	*timeout*
lse-f-91	16	0.01	unsat(`cliquer`)	3718.11	unsat
kfu-s-93	18	0.02	unsat(`cliquer`)	∞	*timeout*
rye-s-93	20	0.03	unsat(`cliquer`)	∞	*timeout*
car-f-92	26	50.2	unsat	∞	*timeout*
car-s-91	26	∞	*timeout*	∞	*timeout*
uta-s-92	28	8.18	unsat	∞	*timeout*
pur-s-93	30	4.71	unsat	∞	*timeout*

times are considerably improved, often making the difference between solvable and unsolvable instances. Also notice that, to the best of our knowledge, the chromatic numbers of Toronto instances were not previously reported in the literature [30].

6 Technical Details

The package containing `pl-cliquer` is available for download from the `pl-cliquer` homepage at: `https://www.cs.bgu.ac.il/~frankm/plcliquer/`. The package contains a `README` file, which contains usage and installation instructions, as well as an `examples` directory containing the examples discussed in this paper. The C code for `pl-cliquer` may be found in the `src` directory. Also in the `src` directory are the module files for `pl-cliquer`.

`pl-cliquer` was compiled and tested on Debian Linux and Ubuntu Linux using the 7.x.x branch of SWI-Prolog. Note that `pl-cliquer` should compile and run on any architecture where `cliquer` will compile and run.

7 Conclusions

We have presented, and made available, a Prolog interface to the core components of the `cliquer` clique-finding tool [27]. The principle contribution of this paper is in the utility of the tool which we expect to be widely used. The tool provides a "drop in" clique finding utility, through which Prolog programs which address graph related problems may apply `cliquer` natively, through Prolog, as part of the solving process. Cliques may be generated, subject to programmer selected constraints on size, maximality etc., and may be generated deterministically or non-deterministically. Additionally, we illustrate in Prolog the standard approach to implement a graph coloring solver. The experiments we report on indicate that our tool-chain, augmented with `pl-cliquer`, is on par with results reported in the literature [30, 23, 24].

References

1 Karen I Aardal, Stan PM Van Hoesel, Arie MCA Koster, Carlo Mannino, and Antonio Sassano. Models and solution techniques for frequency assignment problems. *Annals of Operations Research*, 153(1):79–129, 2007.

2 Noga Alon and Ravi B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987. `doi:10.1007/BF02579196`.

3 Béla Bollobás. Complete subgraphs are elusive. *Journal of Combinatorial Theory, Series B*, 21(1):1 – 7, 1976. doi:10.1016/0095-8956(76)90021-6.

4 Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo. *The Maximum Clique Problem*, pages 1–74. Springer US, Boston, MA, 1999. doi:10.1007/978-1-4757-3023-4_1.

5 R. L. Brooks. On colouring the nodes of a network. *Mathematical Proceedings of the Cambridge Philosophical Society*, 37(2):194–197, 004 1941. doi:10.1017/S030500410002168X.

6 M W Carter and D G Johnson. Extended clique initialisation in examination timetabling. *Journal of the Operational Research Society*, 52(5):538–544, 2001. doi:10.1057/palgrave.jors.2601115.

7 Michael W. Carter, Gilbert Laporte, and Sau Yan Lee. Examination timetabling: Algorithmic strategies and applications. *Journal of the Operational Research Society*, 47(3):373–383, 1996. doi:10.1057/jors.1996.37.

8 Gregory J Chaitin, Marc A Auslander, Ashok K Chandra, John Cocke, Martin E Hopkins, and Peter W Markstein. Register allocation via coloring. *Computer languages*, 6(1):47–57, 1981.

9 Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Logic programming with satisfiability. *TPLP*, 8(1):121–128, 2008. doi:10.1017/S1471068407003146.

10 Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM. doi:10.1145/800157.805047.

11 William H. E. Day and David Sankoff. Computational complexity of inferring phylogenies by compatibility. *Systematic Zoology*, 35(2):224–229, 1986. URL: http://www.jstor.org/stable/2413432.

12 Michael Frank and Michael Codish. Logic programming with graph automorphism: Integrating nauty with Prolog (tool description). *Theory and Practice of Logic Programming*, 16(5-6):688–702, 009 2016. doi:10.1017/S1471068416000223.

13 Ove Frank and David Strauss. Markov graphs. *Journal of the American Statistical Association*, 81(395):832–842, 1986. URL: http://www.jstor.org/stable/2289017.

14 M. R. Garey and D. S. Johnson. " strong " NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, July 1978. doi:10.1145/322077.322090.

15 Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Math.*, 182(1):105–142, 1999. doi:10.1007/BF02392825.

16 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512 – 530, 2001. doi:10.1006/jcss.2001.1774.

17 David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993*. American Mathematical Society, Boston, MA, USA, 1996.

18 Richard M Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

19 Ciaran McCreesh, Samba Ndojh Ndiaye, Patrick Prosser, and Christine Solnon. Clique and constraint models for maximum common (connected) subgraph problems. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 350–368. Springer, 2016. doi:10.1007/978-3-319-44953-1_23.

20 Ciaran McCreesh and Patrick Prosser. Multi-threading a state-of-the-art maximum clique algorithm. *Algorithms*, 6(4):618–635, 2013. doi:10.3390/a6040618.

21 Ciaran McCreesh and Patrick Prosser. A parallel branch and bound algorithm for the maximum labelled clique problem. *Optimization Letters*, 9(5):949–960, 2015. doi:10.1007/s11590-014-0837-4.

22 Nirbhay K Mehta. The application of a graph coloring method to an examination scheduling problem. *Interfaces*, 11(5):57–65, 1981.

23 Isabel Méndez-Díaz and Paula Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006.

24 Isabel Méndez-Díaz and Paula Zabala. A cutting plane algorithm for graph coloring. *Discrete Applied Mathematics*, 156(2):159–179, 2008.

25 Amit Metodi and Michael Codish. Compiling finite domain constraints to SAT with BEE. *TPLP*, 12(4-5):465–483, 2012. doi:10.1017/S1471068412000130.

26 Amit Metodi, Michael Codish, and Peter J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013. doi:10.1613/jair.3809.

27 Sampo Niskanen and Patric R. J. Östergård. Cliquer user's guide. Technical Report version 1.0, Communications Laboratory, Helsinki University of Technology, Espoo, Technical Report T48, 2003. URL: https://users.aalto.fi/~pat/cliquer.html.

28 Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, August 2002. doi:10.1016/S0166-218X(01)00290-6.

29 Patrick Prosser. Exact algorithms for maximum clique: A computational study. *Algorithms*, 5(4):545–587, 2012. doi:10.3390/a5040545.

30 Rong Qu, Edmund K Burke, Barry McCollum, LT Merlot, and Sau Y Lee. A survey of search methodologies and automated system development for examination timetabling. *Journal of scheduling*, 12(1):55–89, 2009.

31 Jean-Charles Régin. *Using Constraint Programming to Solve the Maximum Clique Problem*, pages 634–648. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/978-3-540-45193-8_43.

32 Ram Samudrala and John Moult. A graph-theoretic algorithm for comparative modeling of protein structure. *Journal of Molecular Biology*, 279(1):287 – 302, 1998. doi:10.1006/jmbi.1998.1689.

33 D. J. A. Welsh and M. B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85, 1967. doi:10.1093/comjnl/10.1.85.

34 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

A Toronto Instances

Tables 5 and 6 describe in more detail the results obtained for satisfiable and unsatisfiable instances of the Toronto benchmarks respectively. The first column lists the instance name, the second column lists the best coloring found as well as the previously best known coloring from literature [30], the third column lists the time it took `cliquer` to find a maximum clique in the graph, the fourth column lists the time it took `BEE` to compile the constraints to CNF, The fifth and sixth column detail the size of the CNF in terms of clauses and number of variables, the seventh column lists the time it took the SAT solver to obtain a result, and the last column detail the reason for that result. The values in column seven are the same as the status column of Table 1.

Table 5 Satisfiable Toronto Instances Results.

Instance	k (lit)	`cliquer`	`BEE`	#clauses	#vars	sat	Status
hec-s-92	17 (17)	0.01	0.05	6601	590	0.02	sat
sta-f-83	13 (13)	0.01	0.01	735	175	0.01	sat
yor-f-83	18 (19)	0.01	0.37	37569	1812	0.08	sat
ute-s-92	10 (10)	0.1	0.04	9100	770	0.02	sat
ear-f-83	22 (22)	0.01	0.22	23159	1639	0.07	sat
tre-s-92	20 (20)	0.01	0.36	55756	2881	0.14	sat
lse-f-91	17 (17)	0.01	0.1	17478	1418	0.02	sat
kfu-s-93	19 (19)	0.02	0.23	48300	2807	0.15	sat
rye-s-93	21 (21)	0.03	0.42	69838	4160	0.16	sat
car-f-92	27 (28)	0.2	1.15	179749	7634	0.73	sat
uta-s-92	29 (30)	0.1	2.1	338007	11490	1.5	sat
car-s-91	27 (27)	0.06	1.93	407155	12672	1578.48	sat
pur-s-93	31 (36)	0.5	2.85	1098306	39613	3.36	sat

Table 6 Unsatisfiable Toronto Instances Results.

Instance	k	`cliquer`	`BEE`	#clauses	#vars	sat	Status
hec-s-92	16	0.01	–	–	–	–	unsat(cliquer)
sta-f-83	12	0.01	–	–	–	–	unsat(cliquer)
yor-f-83	17	0.01	–	–	–	–	unsat(cliquer)
ute-s-92	9	0.1	–	–	–	–	unsat(cliquer)
ear-f-83	21	0.01	0.2	13841	1210	0.03	unsat
tre-s-92	19	0.01	–	–	–	–	unsat(cliquer)
lse-f-91	16	0.01	–	–	–	–	unsat(cliquer)
kfu-s-93	18	0.02	–	–	–	–	unsat(cliquer)
rye-s-93	20	0.03	–	–	–	–	unsat(cliquer)
car-f-92	26	0.2	1.12	159981	7138	48.88	unsat
car-s-91	26	0.06	1.09	373479	12053	∞	*timeout*
uta-s-92	28	0.1	1.5	310668	10933	6.58	unsat
pur-s-93	30	0.5	2.77	1005456	37849	1.44	unsat

B Dimacs Instances

Tables 7 and 8 describe in more detail the results obtained for satisfiable and unsatisfiable instances of the DIMACS benchmarks respectively. The tables description follows the description of Tables 5 and 6.

Table 7 Satisfiable Dimacs Instances Results.

Instance	k	`cliquer`	`BEE`	#clauses	#vars	sat	Status
anna.col	11	0.01	0.01	–	–	–	sat(BEE)
david.col	11	0.01	0.01	–	–	–	sat(BEE)
DSJC125.1.col	5	0.01	0.02	4017	553	0.03	sat
DSJR500.1.col	12	0.02	0.02	14928	1209	0.03	sat
DSJR500.5.col	122	4661.28	–	–	–	–	*memory*
fpsol2.i.1.col	65	0.02	0.01	–	–	–	sat(BEE)
fpsol2.i.2.col	30	0.02	0.04	9654	894	0.01	sat
fpsol2.i.3.col	30	0.02	0.04	9654	894	0.01	sat
games120.col	9	0.01	0.03	8820	1051	0.01	sat
huck.col	11	0.01	0.01	–	–	–	sat(BEE)
inithx.i.1.col	54	0.05	0.05	17248	1192	0.02	sat
inithx.i.2.col	31	0.05	0.13	103315	3705	0.12	sat
inithx.i.3.col	31	0.05	0.15	103315	3705	0.13	sat
jean.col	10	0.01	0.01	–	–	–	sat(BEE)
le450_15a.col	15	0.02	0.34	127561	6102	1.24	sat
le450_15b.col	15	0.02	0.32	117393	5879	0.42	sat
le450_25a.col	25	0.02	0.33	155049	6290	0.22	sat
le450_25b.col	25	0.02	0.34	190394	7420	0.33	sat
le450_5a.col	5	0.01	0.12	29092	2088	0.05	sat
le450_5b.col	5	0.01	0.11	30023	2117	0.06	sat
le450_5c.col	5	0.01	0.19	42630	1992	0.06	sat
le450_5d.col	5	0.01	0.19	44927	2051	0.07	sat
miles1000.col	42	0.01	0.01	–	–	–	sat(BEE)
miles1500.col	73	0.01	0.01	–	–	–	sat(BEE)
miles250.col	8	0.01	0.01	–	–	–	sat(BEE)
miles500.col	20	0.01	0.01	–	–	–	sat(BEE)
miles750.col	31	0.01	0.01	15	7	0.01	sat
mulsol.i.1.col	49	0.01	0.01	–	–	–	sat(BEE)
mulsol.i.2.col	31	0.01	0.06	41878	1574	0.06	sat
mulsol.i.3.col	31	0.01	0.06	41878	1574	0.06	sat
mulsol.i.4.col	31	0.01	0.06	44293	1642	0.07	sat
mulsol.i.5.col	31	0.01	0.06	43042	1608	0.07	sat
myciel3.col	4	0.01	0.01	83	30	0.01	sat
myciel4.col	5	0.01	0.01	393	91	0.01	sat
myciel5.col	6	0.01	0.01	1570	240	0.01	sat
myciel6.col	7	0.01	0.02	5736	589	0.01	sat
myciel7.col	8	0.01	0.06	19929	1386	0.02	sat
queen5_5.col	5	0.01	0.01	266	31	0.01	sat
queen6_6.col	7	0.01	0.01	1584	138	0.01	sat
queen7_7.col	7	0.01	0.02	2566	192	0.01	sat
queen8_12.col	12	0.01	0.07	19816	872	0.02	sat
queen8_8.col	9	0.01	0.03	7314	380	0.88	sat
queen9_9.col	10	0.01	0.06	12026	547	6.91	sat
queen10_10.col	11	0.01	0.09	21322	856	21637.9	sat
school1.col	14	66.39	0.51	42	16	0.01	sat
school1_nsh.col	14	26.6	0.47	107	34	0.01	sat
zeroin.i.1.col	49	0.01	0.01	–	–	–	sat(BEE)
zeroin.i.2.col	30	0.01	0.01	734	139	0.01	sat
zeroin.i.3.col	30	0.01	0.01	734	139	0.01	sat

Table 8 Unsatisfiable Dimacs Instances Results.

Instance	k	`cliquer`	`BEE`	#clauses	#vars	sat	Status
anna.col	10	0.01	–	–	–	–	unsat(cliquer)
david.col	10	0.01	–	–	–	–	unsat(cliquer)
DSJC125.1.col	4	0.01	0.02	–	–	–	unsat(BEE)
DSJR500.1.col	11	0.02	–	–	–	–	unsat(cliquer)
DSJR500.5.col	121	4597.34	–	–	–	–	unsat(cliquer)
fpsol2.i.1.col	64	0.02	–	–	–	–	unsat(cliquer)
fpsol2.i.2.col	29	0.02	–	–	–	–	unsat(cliquer)
fpsol2.i.3.col	29	0.02	–	–	–	–	unsat(cliquer)
games120.col	8	0.01	–	–	–	–	unsat(cliquer)
huck.col	10	0.01	–	–	–	–	unsat(cliquer)
inithx.i.1.col	53	0.05	–	–	–	–	unsat(cliquer)
inithx.i.2.col	30	0.05	–	–	–	–	unsat(cliquer)
inithx.i.3.col	30	0.05	–	–	–	–	unsat(cliquer)
jean.col	9	0.01	–	–	–	–	unsat(cliquer)
le450_15a.col	14	0.02	–	–	–	–	unsat(cliquer)
le450_15b.col	14	0.02	–	–	–	–	unsat(cliquer)
le450_25a.col	24	0.02	–	–	–	–	unsat(cliquer)
le450_25b.col	24	0.02	–	–	–	–	unsat(cliquer)
le450_5a.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5b.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5c.col	4	0.01	–	–	–	–	unsat(cliquer)
le450_5d.col	4	0.01	–	–	–	–	unsat(cliquer)
miles1000.col	41	0.01	–	–	–	–	unsat(cliquer)
miles1500.col	72	0.01	–	–	–	–	unsat(cliquer)
miles250.col	7	0.01	–	–	–	–	unsat(cliquer)
miles500.col	19	0.01	–	–	–	–	unsat(cliquer)
miles750.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.1.col	48	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.2.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.3.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.4.col	30	0.01	–	–	–	–	unsat(cliquer)
mulsol.i.5.col	30	0.01	–	–	–	–	unsat(cliquer)
myciel3.col	3	0.01	0.01	37	15	0.01	unsat
myciel4.col	4	0.01	0.01	267	70	0.01	unsat
myciel5.col	5	0.01	0.01	1170	195	0.85	unsat
myciel6.col	6	0.01	0.02	4548	496	22970.47	unsat
myciel7.col	7	0.01	0.06	16499	1197	∞	*timeout*
queen5_5.col	4	0.01	–	–	–	–	unsat(cliquer)
queen6_6.col	6	0.01	0.01	1070	108	0.01	unsat
queen7_7.col	6	0.01	–	–	–	–	unsat(cliquer)
queen8_12.col	11	0.01	–	–	–	–	unsat(cliquer)
queen8_8.col	8	0.01	0.04	5838	324	15.88	unsat
queen9_9.col	9	0.01	0.06	9859	474	2295.86	unsat
queen10_10.col	10	0.01	0.08	18110	716	∞	*timeout*
school1.col	13	66.39	–	–	–	–	unsat(cliquer)
school1_nsh.col	13	26.6	–	–	–	–	unsat(cliquer)
zeroin.i.1.col	48	0.01	–	–	–	–	unsat(cliquer)
zeroin.i.2.col	29	0.01	–	–	–	–	unsat(cliquer)
zeroin.i.3.col	29	0.01	–	–	–	–	unsat(cliquer)

Semantic Versioning Checking in a Declarative Package Manager

Michael Hanus

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

Abstract

Semantic versioning is a principle to associate version numbers to different software releases in a meaningful manner. The correct use of version numbers is important in software package systems where packages depend on other packages with specific releases. When patch or minor version numbers are incremented, the API is unchanged or extended, respectively, but the semantics of the operations should not be affected (apart from bug fixes). Although many software package management systems assumes this principle, they do not check it or perform only simple syntactic signature checks. In this paper we show that more substantive and fully automatic checks are possible for declarative languages. We extend a package manager for the functional logic language Curry with features to check the semantic equivalence of two different versions of a software package. For this purpose, we combine CurryCheck, a tool for automated property testing, with program analysis techniques in order to ensure the termination of the checker even in case of possibly non-terminating operations defined in some package. As a result, we obtain a software package manager which checks semantic versioning and, thus, supports a reliable and also specification-based development of software packages.

1998 ACM Subject Classification D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases functional logic programming, semantic versioning, program testing

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.6

1 Motivation

Contemporary software systems are complex and based on many components. To structure such systems and support the re-use of components in different software systems, software packages with well-defined APIs (application programming interfaces) are used. A software package consists of one or more modules and is used as a building block of a larger system. Hence, a software system or even a complex package depends on other packages. Since packages develop over time, e.g., new functionality is added, more efficient implementations are developed, or the usage of operations (i.e., the API) is changed, it is important to use appropriate versions of packages. Finding them and manage these dependencies is often called "dependency hell." As a solution to this problem, package managers use version numbers associated to package releases and allow to express such dependencies as relations on version numbers.

Semantic versioning is a recommendation to associate meaningful version numbers to software packages. In the semantic versioning standard,[1] each version number consists of major, minor, and patch number, separated by dots, and an optional pre-release specifier

[1] http://www.semver.org

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei; Article No. 6; pp. 6:1–6:16

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

consisting of alphanumeric characters and hyphens appended with a hyphen (and optional build metadata, which we do not consider here). For instance, `0.1.2` and `1.2.3-alpha.2` are valid version numbers. Furthermore, an ordering is defined on version numbers where major, minor, and patch numbers are compared in lexicographic order and pre-releases are considered unstable so that they are smaller than their non-pre-release versions. For instance, `0.1.1` < `0.1.2` < `0.3.1` < `1.1.2-alpha` < `1.1.2`. Furthermore, semantic versioning requires that the major version number is incremented when the API functionality of a package is changed, the minor version number is incremented when new API functionality is added and existing API operations are backward compatible, and the patch version number is incremented when the API functionality is unchanged (only bug fixes, code refactorings, code improvements, etc).

The advantage of semantic versioning is an increased flexibility to choose packages when building larger software systems. For instance, if package `A` requires some functionality which has been introduced in version `2.3.1` of package `B`, one can specify that `A` depends on `B` in a version greater than or equal to `2.3.1` but less than `3.0.0`. Thanks to semantics versioning, a package manager can choose newer versions of `B` (as long as they are smaller than `3.0.0`), when they become available, in order to build `A` without dependency problems.

However, semantic versioning requires the semantic compatibility of two packages with identical major version numbers (apart from new operations or operations with bug fixes). Since this property is obviously undecidable in general, the developer is responsible for this semantic compatibility so that this is not checked in contemporary package management systems. Improving this situation is the objective of the work described in this paper. Due to the absence of side effects in declarative (functional, logic) programming languages, one can easily write repeatable test suites. Tests parameterized over some arguments are also called *properties*. Property-based testing automates the checking of properties by random or systematic generation of test inputs. It has been introduced with the QuickCheck tool [12] for the functional language Haskell and adapted to other languages, like PrologCheck [2] for Prolog, PropEr [26] for the concurrent functional language Erlang, or EasyCheck [11] and CurryCheck [17] for the functional logic language Curry.

In order to check the semantic equivalence of a unary operation f defined in versions v_1 and v_2 of some package, a first approach is renaming the definitions of f in these packages to f_{v_1} and f_{v_2}, respectively, and checking the property $\forall x.f_{v_1}(x) = f_{v_2}(x)$, which is called *computed result equivalence* in [9].[2] Ideally, one should prove this property. Since fully automatic proof techniques are available only for limited domains, we propose to use property-based testing instead. Although this method is incomplete in general, in practice it is quite successful if the generated input data is well distributed (which is a goal of all property-based test tools). Unfortunately, the brute-force testing of the equivalence of all operations, as described above, does not yield an automatic checker for semantic versioning, since it might not terminate if some operations are non-terminating. Moreover, declarative languages like Haskell or Curry are based on lazy evaluation to enable optimal computations and modularity by stream-based programming [20]. Hence, operations might also compute infinite results that cannot be compared in a finite amount of time. Therefore, we propose to combine property-based testing with program analysis techniques in order to ensure the termination of property testing. In general, operations which might not terminate are excluded from equivalence checking. In

[2] This property is a necessary but not sufficient condition to ensure semantic equivalence in functional logic programs [7]. Since we do not intend to provide a faithful method for semantic versioning checking but use a testing-based approach to detect inconsistencies, we use this simplified property.

order to check operations which compute infinite data structures, e.g., stream generators, we analyze the "productivity" of these operations, i.e., a property which ensures that partial results are produced after a finite amount of time, and check finite approximations of their results.

In order to use these ideas in practice, we integrated this kind of semantic versioning checking into CPM [25], a new package management system for the functional logic language Curry. In this way, we obtain a software package manager which checks semantic versioning and, thus, supports a reliable and also specification-based development of software packages.

This paper is structured as follows. In the next section we briefly survey functional logic programming and features of Curry. Sections 3 and 4 discuss the main features of property-based testing and the Curry package manager CPM. The integration of semantic versioning checking into CPM is shown in Section 5. The techniques to check also possibly non-terminating operations are introduced in Section 6 and their implementation is discussed in Section 7. Before we conclude, we show in Section 8 an important application of our approach: the specification-based development of software systems.

2 Functional Logic Programming and Curry

In this section we briefly review some features of functional logic programming and Curry that are relevant for this paper. More details can be found in surveys on functional logic programming [6, 16] and in the language report [19].

Functional logic languages [6, 16] integrate the most important features of functional and logic languages in order to provide a variety of programming concepts. They support functional programming concepts like higher-order functions and lazy evaluation as well as logic programming concepts like non-deterministic search and computing with partial information. The declarative multi-paradigm language Curry [19] is a functional logic language with advanced programming concepts.

The syntax of Curry is close to Haskell [27], 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. $\alpha \to \beta$ denotes the type of all functions mapping elements of type α into elements of type β (where β can also be a functional type, i.e., functional types are "curried"). The application of an operation f to e is denoted by juxtaposition ("$f\ e$").

In addition to Haskell, Curry allows *free* (*logic*) *variables* in rules and initial expressions. Function calls with free variables are evaluated by a possibly non-deterministic instantiation of demanded arguments.

▶ **Example 1.** The following simple program shows the functional and logic features of Curry. It defines the well-known list concatenation and an operation that returns some element of a list having at least two occurrences:

```
(++) :: [a] → [a] → [a]        someDup :: [a] → a
[]     ++ ys = ys              someDup xs | xs == _++[x]++_++[x]++_
(x:xs) ++ ys = x : (xs ++ ys)             = x    where x free
```

Since "`++`" can be called with free variables in arguments, the condition in the rule of `someDup` is solved by instantiating `x` and the anonymous free variables "`_`" to appropriate values (i.e., expressions without defined functions) before reducing the function calls. This corresponds to narrowing [28], but Curry narrows with possibly non-most-general unifiers to ensure the optimality of computations [3]. Curry is a non-strict language, i.e., derivation steps are performed at outermost positions, which supports computations with infinite data structures [20]. We do not recapitulate the details of the operational semantics which can be

found in [1]. When we later consider evaluations of expressions, we denote by "$\to$" the one step outermost derivation relation and by "$\overset{*}{\to}$" its reflexive-transitive closure.

Note that `someDup` is a *non-deterministic operation* since it might deliver more than one result for a given argument, e.g., the evaluation of `someDup [1,2,2,1]` yields the values `1` and `2`. Non-deterministic operations, which can formally be interpreted as mappings from values into sets of values [14], are an important feature of contemporary functional logic languages. Hence, Curry has also a predefined *choice* operation:

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

Thus, the expression "`0 ? 1`" evaluates to `0` and `1` with the value non-deterministically chosen.

A *functional pattern* [4] is a pattern in the left-hand side of a rule containing defined operations (and not only data constructors and variables). Such a pattern abbreviates the set of all standard patterns to which the functional pattern can be evaluated (by narrowing). For instance, we can rewrite the definition of `someDup` as

```
someDup (_++[x]++_++[x]++_) = x
```

Functional patterns are a powerful feature to express arbitrary selections in tree structures, e.g., as shown for processing XML documents in [15].

Curry has also features which are useful for application programming, like *set functions* [5] to encapsulate non-deterministic computations, *default rules* [8] to deal with partially specified operations and negation, and standard features from functional programming, like modules or monadic I/O. Other features are explained when they are used in the following.

3 Property-based Testing and CurryCheck

Property-based testing is a useful technique to improve the reliability of software packages. Basically, properties are Boolean expressions parameterized over input data. Concrete input data is automatically generated by property-based test tools which evaluate the properties on these inputs. For instance, QuickCheck [12], PropEr [26], or PrologCheck [2] generate test inputs in a random manner, whereas SmallCheck [29], GAST [21], or EasyCheck [11] perform a systematic enumeration of test inputs so that, for finite input domains, they can actually verify properties.

CurryCheck [17] is a property-based test tool for Curry which automates the test process. CurryCheck is based on EasyCheck and extracts and tests all properties contained in a source program. A property is a top-level entity with result type `Prop` and an arbitrary number of inputs. For instance, if we add to the program of Example 1 the property

```
concIsAssoc :: [Int]  →  [Int]  →  [Int]  →  Prop
concIsAssoc xs ys zs = (xs++ys)++zs -=- xs++(ys++zs)
```

and run CurryCheck on this program, the associativity property of list concatenation is tested by systematically enumerating lists of integers for the variables `xs`, `ys`, and `zs`. The property "`-=-`" has the type `a → a → Prop` and is satisfied if both arguments have a single identical value.

To check laws involving non-deterministic operations, one can use the property "`<~>`" which is satisfied if both arguments have identical result sets. For instance, consider the following definition of a permutation of a list (which exploits a functional pattern to select some element in the argument list):

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

The requirement that permutations do not change the list length can be expressed by the property

```
permLength xs = length (perm xs) <~> length xs
```

Since the left argument of "`<~>`" evaluates to many (identical) values, the set-based interpretation of "`<~>`" is relevant here. This is reasonable since, from a declarative programming point of view, it is irrelevant how often some result is computed.

Now consider an alternative definition of permutations which non-deterministically inserts the first element into a permutation of the remaining elements:

```
permIns []     = []
permIns (x:xs) = insert x (permIns xs)

insert x (xs++ys) = xs++[x]++ys
```

In order to check whether both definitions of permutations compute identical results, we (successfully) test the following property:

```
permSameAsPermIns xs = perm xs <~> permIns xs
```

4 CPM: The Curry Package Manager

The Curry Package Manager CPM[3] [25] is a tool to distribute and install Curry software packages and manage version dependencies between them. Essentially, a *package* consists of one or more Curry modules and a package specification, a file in JSON format containing the package's metadata. Beyond some standard fields, like author, name, or synopsis, the metadata of each package contains the version number of the package (in semantic versioning format, see above) and a list of dependency constraints. A *dependency constraint* consists of the name of another package and a disjunction of conjunctions of version relations, which are comparison operators (`<`, `<=`, `>`, `>=`, `=`) together with a version number. Conjunctions are separated by commas, and disjunctions are separated by `||`. Hence, the dependency constraint

```
"B" : ">= 2.0.0, < 3.0.0 || > 4.1.0"
```

expresses the requirement that the current package depends on package `B` with major version `2` or in a version greater than `4.1.0`.

CPM has various commands to manage the set of all packages and install and upgrade individual packages. CPM uses a central index of all known packages and their versions. A user can download a local copy of this index and also add other local packages and versions to this index. To install a package, CPM tries to resolve all dependency constraints of the current package and all dependent packages. This is a classic constraint satisfaction problem and CPM uses a lazy functional approach based on [24] to solve all dependency constraints and find appropriate package versions. If there is a solution to these constraints, CPM automatically installs all required packages. If there are several possible versions of some

[3] `http://curry-language.org/tools/cpm`

package to install, CPM uses the newest one. CPM also supports upgrading packages, i.e., to replace already installed packages by newer versions, if possible. The details of these processes are outside the scope of this paper and are described in [25].

CPM adheres to the semantic versioning standard as sketched in Section 1. Thus, if there are two versions of a package with identical major version numbers, they should have compatible APIs, i.e., all public data types and operations in the exported modules[4] occurring in both package versions must have identical type signatures and behavior, and new public operations can be added only if the minor version number is increased. CPM supports the automated checking of this principle by the `diff` command. For instance, to compare the current package to a previous version `1.2.4` of the same package, a package developer can invoke the command

```
> cpm diff 1.2.4
```

This starts a complex comparison process which is described in the next section. Depending on the outcome of this API comparison, the current package can be added to the central CPM index.

5 Semantic Versioning Checking

Semantic versioning checking is the process to compare the APIs of two versions of some package and report possible violations according to the semantic versioning standard. In our context, the API of a package is the set of all public data types and operations occurring in the exported modules of this package. To accomplish this task, the semantic versioning checker integrated in CPM performs the following steps:

1. The signatures of all API data types and operations occurring in both packages are compared. If there are any syntactic differences and the major version numbers of the packages are identical, a violation is reported.
2. If there is some API entity f occurring in version $a_1.b_1.c_1$ but not in version $a_2.b_2.c_2$, then a violation is reported if a_1 and a_2 are identical but b_1 is not greater than b_2.
3. If the major version numbers of the packages are identical, then, for all API operations occurring in both package versions, the behavior of both versions of such an operation is compared (see below for more details about this comparison). If any difference is detected, a violation is reported.

To compare the behavior of some operation f defined in versions v_1 and v_2 of some package, the code of both packages is copied and all modules of these packages (and all packages on which these packages depend) are renamed with the version number as a prefix. For instance, a module `Mod` occurring in package version `1.2.3` is copied and renamed into module `V_1_2_3_Mod`. Thus, if there is a unary operation `f` occurring in module `Mod` in package versions `1.2.3` and `1.2.4` to compare, one can access both versions of this operation by the qualified name `V_1_2_3_Mod.f` and `V_1_2_4_Mod.f`. After copying all modules, CPM generates a new "comparison" module which contains the following code:

```
import qualified V_1_2_3_Mod
import qualified V_1_2_4_Mod

check_Mod_f x = V_1_2_3_Mod.f x <~> V_1_2_4_Mod.f x
```

[4] A package specification can also declare a subset of all modules as "exported" so that only operations in these modules can be used by other packages. If this is not explicitly declared, all modules of the package are considered as exported.

If this is passed to CurryCheck and the property is satisfied for all generated test inputs, we have some confidence about the semantic equivalence of `f` in both packages (although full confidence requires the proof of a more complex property [7, 9]). This approach works under the following assumptions:

1. The input and result types of `V_1_2_3_Mod.f` and `V_1_2_4_Mod.f` are identical.
2. The operations to be compared are terminating on all input values.

Since these conditions might not be satisfied in practice, we develop (partial) solutions to it. To see an example where the first condition is not satisfied, consider the following excerpt of the library `Day` dealing with weekdays:

```
data Weekday = Monday | Tuesday | ... | Sunday

nextDay :: Weekday → Weekday
...
```

Since the type `Weekday` is locally defined, copying and renaming two versions of this library for semantic versioning checking results in two different `Weekday` types so that both versions of `nextDay` have incompatbile argument and result types. Thus, to generate a property to compare both versions, CPM generates a bijective mapping between both renamed types:

```
t_Weekday :: V_1_2_4_Day.Weekday → V_1_2_3_Day.Weekday
t_Weekday V_1_2_4_Day.Monday  = V_1_2_3_Day.Monday
t_Weekday V_1_2_4_Day.Tuesday = V_1_2_3_Day.Tuesday
...
```

This mapping must exist (otherwise, semantic versioning is syntactically violated) and it allows to compare both versions of `nextDay` with the following property:

```
check_Day_nextDay :: V_1_2_4_Day.Weekday → Prop
check_Day_nextDay x = t_Weekday (V_1_2_4_Day.nextDay x)
                      <~> V_1_2_3_Day.nextDay (t_Weekday x)
```

If our second assumption (termination of the operations to be compared) is not satisfied, the behavior checker might not terminate. Obviously, this should be avoided. Therefore, we analyze the operations to be compared before the comparison properties are generated. As a simple approach, one can approximate the termination behavior of these operations, e.g., by comparing the argument sizes in recursive calls [22]. For this purpose, we used the Curry analysis framework CASS [18] to implement a simple termination analysis which checks the arguments of direct recursive calls of an operation. If all these calls contain at least one syntactically smaller argument (since we consider only algebraic data types for this purpose, there are no infinite chains of size-decreasing values) and all dependent operations are terminating, the operation is classified as terminating. We can use this analysis to check only those operations which are definitely terminating and emit warnings about the remaining unchecked operations. Although there are many opportunities to improve the termination analyzer, it can only approximate the termination property. Therefore, CPM also accepts specific pragmas where the programmer can annotate operations as terminating. For instance, CPM will consider the following operation as terminating and, thus, includes it in semantic versioning checking:

```
{-# TERMINATE -#}
mcCarthy :: Int → Int
mcCarthy n = if n<=100 then mcCarthy (mcCarthy (n+11))
                       else n-10
```

Although this is reasonable to increase the number of operations considered in semantic versioning checking, an important class of operations is still excluded: operations that are intentionally non-terminating since they generate infinite data structures. A method to check such operations will be presented in the next section.

6 Checking Non-terminating Operations

It is well-known that lazy evaluation is a useful programming feature to increase modularity by separating producers and consumers of data [20]. Typically, data producers are operations which generate infinite structures, like the following operations which generate infinite lists of ascending integers starting from the argument:

```
ints :: Int → [Int]          ints2 :: Int → [Int]
ints n = n : ints (n+1)      ints2 n = n : ints2 (n+2)
```

Although these operations compute infinite lists of a different shape, this difference cannot be detected by the property

```
checkInts x = ints x <~> ints2 x
```

due to its non-termination. Since such operations are actually used in non-strict languages, semantic versioning checking should be supported for them in some way.

How can we state that `ints` and `ints2` have a different behavior? If we consider the computed result equivalence of operations introduced in Sect. 1, there is no difference since neither `ints` nor `ints2` evaluate to some value (an expression without operation symbols). Therefore, a simple strategy like running CurryCheck with a time limit would not show any difference in the values computed by `ints` nor `ints2`. We need another way to compare the behavior of these operations. Thus, we use a more general notion of equivalence of operations in non-strict functional logic languages proposed in [7], also called "contextual equivalence" in [9]. It expresses the idea that two operations are equivalent if they can be replaced by each other in any context without changing the produced values.

▶ **Definition 2** (Equivalent operations [7]). Let f_1, f_2 be operations of the same type. f_1 is *equivalent* to f_2 iff, for any expression E_1 and value v, E_1 evaluates to v iff E_2 evaluates to v, where E_2 is obtained from E_1 by replacing any occurrence of f_1 with f_2.

Since equivalence in this sense implies computed result equivalence, counter-examples found by the method introduced in Sect. 5 are also counter-examples to the equivalence of operations. Moreover, `ints` and `ints2` are not equivalent w.r.t. Def. 2: `head (tail (ints 0))` evaluates to `1` but `head (tail (ints2 0))` evaluates to `2`. To detect such differences, we put the operations into some context where only a finite outermost part is computed. In our example, we define an operation that limits the length of a list. Since the length should be limited with non-negative numbers, we define Peano numbers with the constructors `Z`(ero) and `S`(uccessor):

```
data Nat = Z | S Nat
```

We limit potentially infinite lists to some length provided as a `Nat` argument:

```
limitList :: Nat → [Int] → [Int]
limitList Z     _      = []
limitList (S n) []     = []
limitList (S n) (x:xs) = x : limitList n xs
```

Now we can check the observable equivalence of `ints` and `ints2` by the following property:

```
limitCheckInts n x = limitList n (ints x) <~> limitList n (ints2 x)
```

CurryCheck finds a counter-example for the input arguments `n=(S (S Z))` and `x=1`.

Since the list length limit is an input parameter to the property, this property is sufficient to detect observable differences between such infinite lists. A formal result about the soundness and completeness of limited property checking will be presented below. Before we have to discuss some conditions required for this method.

In order to ensure the termination of property checking, a depth restriction is not sufficient in general. For instance, when checking the equivalence of the operations

```
loop n = loop (n+1)              loop2 n = loop2 (n+2)
```

a depth limit would not avoid the non-terminating evaluations of `loop` and `loop2`. This is due to the fact that the evaluation of these operations do not produce a constructor-rooted term after finitely many steps. To exclude this kind of operations, we define the class of productive operations (for the sake of simplicity, we consider unary operations only, but all definitions and results can be extended to operations with more than one argument):

▶ **Definition 3** (Productive operations). An operation f is called *root-productive* if, for all values t, there is no infinite derivation

$$f\ t \to e_1 \to e_2 \to \cdots$$

where each e_i is operation-rooted. An operation f is called *productive* if it is root-productive and, for all values t and derivations $f\ t \overset{*}{\to} e$, all operations in e are productive.

For instance, `loop` and `loop2` are not productive whereas `ints` and `ints2` are productive (remember that "$\to$" denotes the outermost reduction relation of Curry). Obviously, terminating operations are productive but not vice versa.

If all operations in an expression are productive and we limit the result depth of evaluating this expression, as done in the property `limitCheckInts` above, all evaluations are terminating. Thus, if we restrict semantic versioning checking to productive operations and limit the depth of the results, it is always terminating. To formalize this method, we define a limit operation for some data type τ as follows (for the sake of simplicity, we consider monomorphic data types here; the extension to polymorphic type constructors will be discussed later):

▶ **Definition 4** (Limit operation). Let τ be some type defined by

```
data τ = C1 τ11 ... τ1n1 | ... | Ck τk1 ... τknk
```

Then the *limit operation* for type τ is defined as follows:

```
limitτ :: Nat → τ → τ
limitτ Z     _              = cτ        -- cτ is some ground value of type τ
limitτ (S n) (C1 x1 ... xn1) = C1 (limitτ11 n x1) ... (limitτ1n1 n xn1)
...
limitτ (S n) (Ck x1 ... xnk) = Ck (limitτk1 n x1) ... (limitτknk n xnk)
```

The operation `limitList` defined above is an example of a limit operation for lists of integers. The definition assumes that there is always a ground value c_τ (i.e., a constructor term without variables like a constant) of type τ. This assumption might not be satisfied for data types that do not have finite values, as

```
data ByteStream = Cons Byte ByteStream
```

In this case, there is no ground value which can be used as a result of `limitByteStream Z _`. However, we could extend this data type with a new constant

```
data ByteStream = Cons Byte ByteStream | EmptyByteStream
```

and define

```
limitByteStream Z _ = EmptyByteStream
```

Note that this data type extension is similarly to the representation of failures when compiling functional logic programs into purely functional programs [10] so that it does not change the set of computed values.

The termination of semantic versioning checking with limit operations for productive operations is a consequence of the following result:

▶ **Proposition 5.** *Let* `limit`τ *be a limit operation,* n *some* `Nat` *value, and* e *an expression of type* τ *which contains only productive operations. Then all derivations of* `(limit`τ n e`)` *are finite.*

Now we can state the soundness and completeness of checking the equivalence of operations with limit operations. Soundness means that every counter-example found by limited equivalence checking shows that the considered operations are not equivalent:

▶ **Proposition 6** (Soundness of limited equivalence checking)**.** *Let* f_1 *and* f_2 *be operations of type* $\tau \to \tau'$ *and* `limit`τ' *be a limit operation for type* τ'*. If there are values* n, x, y *such that* `limit`τ' n $(f_1\ x)$ *evaluates to* y *but* `limit`τ' n $(f_2\ x)$ *does not evaluate to* y*, then* f_1 *and* f_2 *are not equivalent.*

Completeness of equivalence checking with limit operations means that one can always find a counter-example for non-equivalent operations (if we search long enough for appropriate inputs). However, this is not the case in general due to partially defined operations. For instance, consider a slightly modified variant of the `ints` operations where the generated lists also include elements `head []` whose evaluation leads to a failure:

```
fints n = head [] : n : fints (n+1)    fints2 n = head [] : n : fints2 (n+2)
```

Since the evaluation of `(limitList` n `fints)` fails and does not produce any value for non-zero n, a counter-example to the equivalence of `fints` and `fints2` is not generated. Fortunately, generators of infinite structures are in practical programs totally defined, i.e., reducible on all ground constructor terms. For such operations, completeness is ensured:

▶ **Proposition 7** (Completeness of limited equivalence checking)**.** *Let* f_1 *and* f_2 *be totally defined operations of type* $\tau \to \tau'$ *and* `limit`τ' *be a limit operation for type* τ'*. If* f_1 *and* f_2 *are not equivalent, then there are values* n, x, y *such that* `limit`τ' n $(f_1\ x)$ *evaluates to* y *but* `limit`τ' n $(f_2\ x)$ *does not evaluate to* y*.*

7 Implementation of Semantic Versioning Checking

Based on the observations discussed so far, we can construct a fully automatic tool for semantic versioning checking as follows. Instead of comparing all operations of two versions, which might not terminate, we consider the following operations:

1. Terminating operations: Since their evaluations are finite on all input values, one can check their behavior on given inputs by comparing the sets of their result values (using the property "`<~>`" of CurryCheck).
2. Productive operations: Since they might produce infinite data structures, their result values cannot be fully compared. Instead, one can check their behavior on given inputs by comparing the results obtained by applying some limit operation to them.

By Propositions 6 and 7, this is a sound and complete method for equivalence checking for totally defined operations. The method is still applicable to partial operations but then it is not ensured that counter-examples are found. On the other hand, every implementation has to limit the number of test inputs to a finite set. Therefore, the theoretical incompleteness of property testing for partially defined operation does not cause a problem in practice.

From a practical point of view, it is more relevant to ensure the termination of the checking tool. This requires to approximate the termination and productivity properties of operations and the generation of limit operations for data types. First, we consider the latter (easier) requirement.

We have already seen the definition of limit operations for monomorphic types like list of integers. This scheme can be extended to polymorphic data types: in this case, we pass limit operations for the polymorphic argument types which are applied to polymorphic arguments. For instance, a limit operation for polymorphic lists can be defined as follows:

```
limitList :: (Nat → a → a) → Nat → [a] → [a]
limitList la Z     _      = []
limitList la (S n) []     = []
limitList la (S n) (x:xs) = la n x : limitList la n xs
```

CPM generates limit operations according to this scheme for all result types of productive operations.

Since termination and productivity are undecidable properties, we approximate these properties with a program analysis and use the Curry analysis framework CASS [18] to implement this analysis. For termination, the size-change principle [22] is a reasonable framework. We implemented only a simplified version of it where all directly recursive calls must have decreasing arguments. Although this is not as powerful as the general framework, it is a good starting to implement our approach. Of course, one can make the termination analysis more precise by implementing sophisticated termination methods or use, if free variables occur in right-hand sides, specific termination methods for functional logic programming [23].

The approximation of productivity is less explored than termination. A notion of productivity has been investigated in the area of term rewriting systems (TRSs), e.g., [13, 30]. However, the focus is different there. Productivity in TRS means that there is some reduction sequence that produces an outermost constructor in finitely many steps, whereas productivity in our sense means that *all* outermost reduction sequences cannot go on forever without producing outermost constructors, which is important to ensure the termination of our checking procedure (see Prop. 5). This difference becomes relevant for non-deterministic computations where it is not sufficient for our purpose that some computation branch produces constructors. Furthermore, terminating operations are always productive in our sense.

We approximate productivity by considering the *top-level operation calls* (*tlo*) of some operation. For each operation f, the set $tlo(f)$ is defined by (we denote by $\overline{o}$ a sequence of

objects $o_1 \ldots o_n$):

$$tlo(f) = \{g \mid \exists \text{ values } \overline{t}, \overline{s} \text{ and some derivation } f\ \overline{t} \xrightarrow{*} g\ \overline{s}\}$$

Similarly, we define the set $tlc(f)$ of *top-level calls inside constructors* as all operations occurring outermost in a constructor derived from a call to f. For instance, $tlo(\texttt{ints}) = \{\}$ and $tlc(\texttt{ints}) = \{\texttt{ints}\}$. These sets can be over-approximated by a fixpoint computation on the program rules. Then we classify an operation f as productive if

1. $f \notin tlo(f)$,
2. all operations in $tlo(f)$ and $tlc(f)$ are productive, and
3. all other operations which might occur in derivations of f are terminating.

Hence, the operation `ints` is productive (note that no other operation occur in a derivation of `ints`) whereas `loop` is not productive (since it violates the first requirement). Productive operations occurring in arguments of other operations lead to non-productive operations. To understand this strong requirement, consider the following operation (the standard operation `filter` removes all elements in the second argument list which do not satisfy the predicate provided in the first argument):

```
natsWith p = filter p (ints 0)
```

Although `ints` is productive, the productivity of `natsWith` depends on the value of its argument: if the argument is the predicate `(>0)`, it always produces outermost constructors, but if the argument is the predicate `(<0)`, it loops without producing any constructor. One might improve our weak but safe approximation for particular cases, but our current approximation is still useful in practice. If this approximation does not classify an operation as productive, the package developer can add a pragma to tell CPM that an operation is productive. For instance, one can compute the list of all prime numbers by the sieve of Eratosthenes, but the productivity depends on the fact that there are infinitely many prime numbers. Hence, Euclid would add the following pragma:

```
{-# PRODUCTIVE -#}
primes = sieve (ints 2)
  where sieve (p:xs) = p : sieve (filter (\x  → mod x p > 0) xs)
```

The effectiveness of the termination and productivity analysis depends on the programs under consideration. In order to evaluate our approach, we applied our analysis to the largest library available in Curry distributions: the standard prelude which contains the definitions of operations that are available to any Curry program. The prelude defines 126 operations (plus 30 I/O actions which are excluded from automated property checking due to the problem of guessing appropriate input values like file names, see also [17]). Our analysis shows that 112 operations are terminating, 11 operations are productive, and the remaining three operations might be non-terminating so that they should not be checked. A closer look at the latter operations shows that one operation is actually non-terminating (the prelude operation `until` which implements a loop which might not terminate), whereas two other are actually terminating but use other productive operations so that their termination cannot be shown by our criteria. Nevertheless, the precision is encouraging and there are non-terminating but productive operations that can be checked thanks to our techniques.

Note that our approach to equivalence checking for non-terminating operations is also applicable to non-deterministic operations. For instance, if we define lists of ascending integers in a non-deterministic manner by

```
ndints n = n : (ndints (n+1) ? (n+1) : ndints (n+2))
```

then our check with limit operations succeeds: although `ndints` non-deterministically evaluates to several infinite lists, all of them are identical to the list computed by `ints`.

The inclusion of non-determinism is relevant for packages that use logic programming features. Apart from this, it is also useful to support specification-based software development as discussed in the following section.

If a previous version of the package contains a bug in the implementation of some operation, it is meaningless to compare the operation of the current version against the previous version. For this purpose, there is a pragma to tell CPM to drop the checking of some operation:

```
{-# NOCOMPARE -#}
f ... = ...code with bug fixes...
```

If the current version is accepted to the CPM repository, this annotation should be removed.

8 Specification-based Software Development

The advantage of using functional logic languages like Curry as a wide-spectrum language for software development is discussed in [7]. There it is shown that functional logic programming features are useful to write comprehensive, executable specifications as well as more efficient implementations. Since specifications as well as implementations are written in the same language, specifications can be used as run-time assertions for implementations or their equivalence can be statically checked by property testing [17]. For this purpose, [7] proposed to define the specification of some operation f by some operation with the name f`'spec`. CurryCheck uses this name convention for generating and testing equivalence properties.

With the use of packages, one can structure this development process even better. The idea is to write the specification of the operations to be developed in a first version of a package, i.e., the package n`.0.0` (where n is a major version number) contains the specification. For instance, if we want to develop a package `sort` with sorting operations, we could define a specification of sorting a list in version `1.0.0` by

```
sort (xs++[x,y]++ys) | x>y = sort (xs++[y,x]++ys)
sort'default xs = xs
```

In the first rule, a functional pattern is used to select some arbitrary pair of elements that are swapped to improve the ordering of the list. The second default rule [8] is applicable when the first rule cannot be applied, i.e., when all elements are in the correct order.

Note that this specification is non-deterministic, i.e., its execution might return a sorted list more than one time. However, this is not relevant if we use it for semantic versioning checking since there we compare the result *sets* of two versions of an operation. Thus, if we implement a deterministic and more efficient sorting operation in version `1.0.1` of the package `sort`, we can use the semantic versioning checker to automatically test the new implementation against its specification.

9 Conclusions and Related Work

We have presented, to the best of our knowledge, the first semantic versioning checker that is integrated in a software package manager. In order to make the checking process fully automatic, it is necessary to ensure the termination of the checker. Therefore, the checker analyzes the termination and productivity behavior of all operations and generates appropriate properties that will be tested by CurryCheck. With these methods, most operations can be automatically checked, even operations which produce infinite data structures. Since the

checker can only approximate the run-time behavior of operations, a package developer can also insert annotations to increase the number of checked operations.

We developed this framework for the functional logic programming language Curry, but most of the ideas can also be transferred to other declarative (purely functional or purely logic) languages. Nevertheless, the use of Curry also supports the specification-based development of software since specifications can often be adequately expressed in a non-deterministic way.

Although semantic versioning is recommended in many package managers and used in software projects, there are almost no tools to help the developer to check semantic properties of different package versions. An exception is the Elm package manager[5] which performs semantic versioning checks based on purely syntactic API comparisons. Thus, it can not detect semantic differences when API types are unchanged, like replacing a decrement by an increment operation.

We have demonstrated that declarative programming in combination with property testing tools is a good basis for this task. Hence, all kinds of languages with property testing tools are appropriate for this technique, e.g., Haskell with QuickCheck [12], Prolog with PrologCheck [2], or Erlang with PropEr [26]. For a fully automatic tool that can be integrated into the infrastructure of package managers, it is important to ensure the termination of the checking process. For this purpose, one needs methods to ensure the termination of the programs under consideration. For non-strict languages, one should also provide methods to compare operations which produce infinite structures. For this purpose, we defined the notion of productive operations and a method to approximate this property. One can find similar notions in term rewriting systems (e.g., [30, 13]) but with a slightly different focus.

For future work, we plan to integrate better techniques for termination checking, since this would enlarge the class of checked operations. Furthermore, it would be interesting to add methods for equivalence checking without property testing. An obvious method is to check the structural equivalence of program code. This is useful for operations which are unchanged or only reformatted in two versions of a package. One might also use an abstract semantics to infer equivalences in functional logic programs, as done in [9]. Another idea is to combine property testing with theorem proving to develop and store proofs of properties. Initial ideas are supported by CurryCheck [17] but their integration for semantic versioning checking has to be explored.

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 C. Amaral, M. Florido, and V. Santos Costa. PrologCheck - property-based testing in Prolog. In *Proc. of the 12th International Symposium on Functional and Logic Porgramming (FLOPS 2014)*, pages 1–17. Springer LNCS 8475, 2014. doi:10.1007/978-3-319-07151-0_1.

3 S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000. doi:10.1145/347476.347484.

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. Set functions for functional logic programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declar-*

[5] http://elm-lang.org/

ative Programming (PPDP'09), pages 73–82. ACM Press, 2009. doi:10.1145/1599410.1599420.

6 S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010. doi:10.1145/1721654.1721675.

7 S. Antoy and M. Hanus. Contracts and specifications for functional logic programming. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, pages 33–47. Springer LNCS 7149, 2012. doi:10.1007/978-3-642-27694-1_4.

8 S. Antoy and M. Hanus. Default rules for Curry. *Theory and Practice of Logic Programming*, 17(2):121–147, 2017. doi:10.1017/S1471068416000168.

9 G. Bacci, M. Comini, M.A. Feliú, and A. Villanueva. Automatic synthesis of specifications for first order Curry. In *Principles and Practice of Declarative Programming (PPDP'12)*, pages 25–34. ACM Press, 2012. doi:10.1145/2370776.2370781.

10 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. doi:10.1007/978-3-642-22531-4_1.

11 J. Christiansen and S. Fischer. EasyCheck - test data for free. In *Proc. of the 9th International Symposium on Functional and Logic Programming (FLOPS 2008)*, pages 322–336. Springer LNCS 4989, 2008.

12 K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *International Conference on Functional Programming (ICFP'00)*, pages 268–279. ACM Press, 2000.

13 J. Endrullis and D. Hendriks. Lazy productivity via termination. *Theoretical Computer Science*, 412(28):3203–3225, 2011. doi:10.1016/j.tcs.2011.03.024.

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. 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. doi:10.4230/LIPIcs.ICLP.2011.198.

16 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. doi:10.1007/978-3-642-37651-1_6.

17 M. Hanus. CurryCheck: Checking properties of Curry programs. In *Proceedings of the 26th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2016)*. Springer LNCS 10184, 2016.

18 M. Hanus and F. Skrlac. A modular and generic analysis server system for functional logic programs. In *Proc. of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, pages 181–188. ACM Press, 2014. doi:10.1145/2543728.2543744.

19 M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0). Available at http://www.curry-language.org, 2016.

20 J. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming*, pages 17–42. Addison Wesley, 1990.

21 P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proc. of the 14th International Workshop on Implementation of Functional Languages*, pages 84–100. Springer LNCS 2670, 2003.

22 C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 81–92, 2001.

23 N. Nishida and G. Vidal. Termination of narrowing via termination of rewriting. *Applicable Algebra in Engineering, Communication and Computing*, 21(3):177–225, 2010. doi:10.1007/s00200-010-0122-4.

24 T. Nordin and A.P. Tolmach. Modular lazy search for constraint satisfaction problems. *Journal of Functional Programming*, 11(5):557–587, 2001. doi:10.1017/S0956796801004051.

25 J. Oberschweiber. A package manager for Curry. Master's thesis, University of Kiel, 2016.

26 M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proc. of the 10th ACM SIGPLAN Workshop on Erlang*, pages 39–50, 2011. doi:10.1145/2034654.2034663.

27 S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press, 2003.

28 U.S. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. IEEE Internat. Symposium on Logic Programming*, pages 138–151, Boston, 1985.

29 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.

30 H. Zantema and M. Raffelsieper. Proving productivity in infinite data structures. In *Proc. 21st International Conference on Rewriting Techniques and Applications (RTA 2010)*, volume 6 of *LIPIcs*, pages 401–416. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010. doi:10.4230/LIPIcs.RTA.2010.401.

Understanding Restaurant Stories Using an ASP Theory of Intentions

Daniela Inclezan[1], Qinglin Zhang[2], Marcello Balduccini[3], and Ankush Israney[4]

1 **Miami University, Oxford OH, USA**
`inclezd@miamioh.edu`
2 **Miami University, Oxford OH, USA**
`zhangq7@miamioh.edu`
3 **Drexel University, Philadelphia PA, USA**
`mb3368@drexel.edu`
4 **Drexel University, Philadelphia PA, USA**
`avi26@drexel.edu`

Abstract

The paper describes an application of logic programming to story understanding. Substantial work in this direction has been done by Erik Mueller, who focused on texts about stereotypical activities (or scripts), in particular restaurant stories. His system performed well, but could not understand texts describing exceptional scenarios. We propose addressing this problem by using a theory of intentions developed by Blount, Gelfond, and Balduccini. We present a methodology in which we model scripts as activities and employ the concept of an intentional agent to reason about both normal and exceptional scenarios.

1998 ACM Subject Classification I.2.4 Knowledge Representation Formalisms and Methods

Keywords and phrases answer set programming, story understanding, theory of intentions

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.7

1 Overview

This paper describes an application of Answer Set Prolog [3] and its extension [1] to the understanding of narratives. According to Schank and Abelson [7], stories frequently narrate episodes related to *stereotypical activities – sequences of actions normally performed in a certain order by one or more actors, according to cultural conventions.* An example of a stereotypical activity is dining in a restaurant with table service. A story mentioning a stereotypical activity is not required to state explicitly all of the actions that are part of it, as it is assumed that readers are capable of filling in the blanks with their own commonsense knowledge about the activity [7]. Consider, for instance, the following narrative:

▶ **Example 1.** *Nicole went to a vegetarian restaurant. She ordered lentil soup. The waitress set the soup in the middle of the table. Nicole enjoyed the soup. She left the restaurant.*

Norms indicate that customers do not seat themselves when there is table service, but rather wait to be seated by a waiter; they are also expected to pay for their meal. Readers are supposed to know these conventions, and thus such information is missing from the text.

Schank and Abelson [7] introduced the concept of a *script* to model stereotypical activities: a *fixed* sequence of actions that are *always* executed in a specific order. Following these ideas, Erik Mueller conducted substantial work on narratives about stereotypical activities.

He focused on restaurant stories [5] and news about terrorist incidents [4]. In the former, Mueller developed a system that can take as an input a text about a restaurant episode, process it using information extraction techniques, and demonstrate an understanding of the narrative by answering questions whose answers are not necessarily explicitly stated in the text. The system had a good accuracy but the rigidity of scripts did not allow for the correct processing of scenarios describing exceptions (e.g., waiter bringing a wrong dish). To be able to handle such scenarios, all possible exceptions of a script would have to be foreseen and encoded as new scripts by a knowledge engineer, which is an important hurdle.

In this paper, we propose a new representation methodology and reasoning approach, which makes it possible to answer, in both normal and exception scenarios, questions about events that did or did not take place. We overcome limitations in Mueller's work by abandoning the rigid script-based approach. Instead, we view characters in stories about stereotypical activities (e.g., the customer, waiter, and cook in a restaurant scenario), as BDI agents that *intend* to perform some actions in order to achieve certain goals, but may not always need to/ be able to perform these actions as soon as intended. It was instrumental for our purpose to use a *theory of intentions* developed by Blount *et al.* [2] that introduces the concept of an *activity* – a triple consisting of the activity's name, a goal, and the plan aimed at achieving it. An activity (or rather its plan) may be divided into sub-activities with their own sub-goals. In our work, each character role that is relevant to a stereotypical activity had its own activity. For instance, for the customer role in a restaurant episode, we created an activity named $c\_act(C, R, W, F)$, read as "customer C goes to restaurant R where s/he communicates to waiter W an order for food F." The plan for this activity is the sequence

$$[\ enter(C, R),\ lead\_to(W, C, \mathtt{t}),\ sit(C),\ c\_subact\_1(C, F, W),\ eat(C, F),\ c\_subact\_2(C, W),\ stand\_up(C),\ move(C, \mathtt{t}, \mathtt{entrance}),\ leave(C)\ ]$$

in which `t` stands for the customer's table. The activity's goal is $satiated\_and\_out(C)$. Modeling the customer's activity as a nested one with sub-activities allowed reasoning about a larger number of exceptional scenarios compared to its formalization as a flat activity, for instance Example 2. We introduced two sub-activities: $c\_subact\_1(C, F, W)$ – "C consults the menu and communicates an order for food F to W," and $c\_subact\_2(C, W)$ – "C asks W for the bill and pays for it." In Example 2, Nicole does not execute the actions in c_subact_2 as the goal of this sub-activity, being done with payment, is already satisfied as the meal is on the house. Instead, she performs the next action in the overall activity: $stand\_up$.

▶ **Example 2** (Serendipity). *Nicole went to a vegetarian restaurant. She ordered lentil soup. When the waitress brought her the soup, she told her that it was on the house.*

Activities were encoded in Answer Set Prolog via rules like:

$$activity(c\_act(C, R, W, F)) \leftarrow customer(C), restaurant(R), waiter(W), food(F).$$
$$comp(c\_act(C, R, W, F),\ 1,\ enter(C, R)) \leftarrow activity(c\_act(C, R, W, F)). \quad \ldots$$
$$length(c\_act(C, R, W, F),\ 9) \leftarrow activity(c\_act(C, R, W, F)).$$
$$goal(c\_act(C, R, W, F),\ satiated\_and\_out(C)) \leftarrow activity(c\_act(C, R, W, F)).$$

Blount *et al.* also introduced an architecture ($\mathcal{AIA}$) of an *intentional agent*, an agent that obeys his intentions. According to $\mathcal{AIA}$, at each time step, the agent observes the world, explains observations incompatible with its expectations (diagnosis), and determines what action to execute next (planning). $\mathcal{AIA}$ models the control strategy of an agent capable of reasoning about a wide variety of scenarios, including the serendipitous achievement of its goal by exogenous actions as in Example 2 or the realization that an ongoing activity is

futile. In contrast with $\mathcal{AIA}$, which encodes an agent's reasoning process about its own goals, intentions, and ways to achieve them, we represented the reasoning process of a (cautious) reader that learns about the actions of an intentional agent from a narrative. For instance, while an intelligent agent creates or selects its own activity to achieve a goal, in a narrative context, the reader learns about the activity that was selected by the agent from the text. As a consequence, we adapted parts of the $\mathcal{AIA}$ architecture to suit our purposes.

Stories about stereotypical activities do not mention all actions that occur, as they rely on the reader's background knowledge. As a result, the reader needs to fill the story time line with new time points (and thus construct what we call a *reasoning time line*) to accommodate physical and mental actions not mentioned in the text. We complemented the reasoning module adapted from $\mathcal{AIA}$ with reasoning rules below, in which we denote story vs. reasoning time steps by predicates *story_step* and *step*, resp., and introduce predicate $map(\mathtt{s},\mathtt{i})$ to say that story step `s` is mapped into reasoning time step `i`:

$1\{map(S,I) : step(I)\}1 \quad \leftarrow \quad story\_step(S).$

$\neg map(S,I) \quad \leftarrow \quad map(S_1,I_1),\ S < S_1,\ I \geq I_1,\ story\_step(S),\ step(I).$

Observations about the occurrence of actions and values of fluents mentioned in the text, recorded using predicates *st_hpd* and *st_obs*, are translated into observations on the reasoning time line via rules of the type:

$hpd(A,V,I) \ \leftarrow \ st\_hpd(A,V,S),\ map(S,I).$

A reader may be asked questions about the story. We support yes/no, when, who, and where questions related to events. A question is represented by an atom $question(q)$, where q is a term encoding the question, e.g., $query\_occur(A)$ ("did action A occur?"), $query\_when(A)$ ("when did A occur?"). Answers are encoded by atoms $answer(q,a)$, where a is the answer. For example, $answer(occur(pay(nicole,b)),yes)$ states that the answer to question "Did Nicole pay the bill?" is yes. A positive answer about the occurrence of a specific event is encoded by the rule:

$answer(query\_occur(A),\ yes) \ \leftarrow \ physical\_action(A),\ step(I),\ occurs(A,I).$

Answering a definite "no" requires ensuring that the action did not happen at *any* step:

$maybe(A) \ \leftarrow \ physical\_action(A),\ step(I),\ \text{not } \neg occurs(A,\ I).$

$answer(query\_occur(A),\ no) \ \leftarrow \ physical\_action(A),\ step(I),$
$\qquad \text{not } answer(query\_occur(A),\ yes),\ \text{not } maybe(A).$

While we exemplified and tested our methodology on restaurant scenarios, our approach is equally applicable to other stereotypical activities. The main task for a new stereotypical activity is defining the different activities, including goals, for each relevant character role. Part of this process can be automated by starting from a rigid and centralized script learned in an unsupervised manner (e.g., [6]). Determining (sub-)goals and splitting activities into sub-activities is a more challenging problem, which deserves substantial attention.

References

1 Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In *Proceedings of Commonsense-03*, pages 9–18. AAAI Press, 2003.

2 Justin Blount, Michael Gelfond, and Marcello Balduccini. A theory of intentions for intelligent agents. In *Proceedings of LPNMR 2015*, pages 134–142, 2015.

3 Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3/4):365–386, 1991.

4 Erik T. Mueller. Understanding script-based stories using commonsense reasoning. *Cognitive Systems Research*, 5(4):307–340, 2004.

5 Erik T. Mueller. Modelling space and time in narratives about restaurants. *Literary and Linguistic Computing*, 22(1):67–84, 2007.

6 Michaela Regneri, Alexander Koller, and Manfred Pinkal. Learning script knowledge with web experiments. In *Proceedings of the ACL '10*, pages 979–988, 2010.
7 R. C. Schank and R. P. Abelson. *Scripts, Plans, Goals, and Understanding: An Inquiry into Human Knowledge Structures*. Lawrence Erlbaum, 1977.

Learning Effect Axioms via Probabilistic Logic Programming

Rolf Schwitter

Department of Computing, Macquarie University, Sydney NSW 2109, Australia
Rolf.Schwitter@mq.edu.au

Abstract

Events have effects on properties of the world; they initiate or terminate these properties at a given point in time. Reasoning about events and their effects comes naturally to us and appears to be simple, but it is actually quite difficult for a machine to work out the relationships between events and their effects. Traditionally, effect axioms are assumed to be given for a particular domain and are then used for event recognition. We show how we can automatically learn the structure of effect axioms from example interpretations in the form of short dialogue sequences and use the resulting axioms in a probabilistic version of the Event Calculus for query answering. Our approach is novel, since it can deal with uncertainty in the recognition of events as well as with uncertainty in the relationship between events and their effects. The suggested probabilistic Event Calculus dialect directly subsumes the logic-based dialect and can be used for exact as well as a for inexact inference.

1998 ACM Subject Classification D.1.6 Logic Programming

Keywords and phrases Effect Axioms, Event Calculus, Event Recognition, Probabilistic Logic Programming, Reasoning under Uncertainty

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.8

1 Introduction

The Event Calculus [9] is a logic language for representing events and their effects and provides a logical foundation for a number of reasoning tasks [13, 24]. Over the years, different versions of the Event Calculus have been successfully used in various application domains; amongst them for temporal database updates, for robot perception and for natural language understanding [12, 13]. However, many event recognition scenarios exhibit a significant amount of uncertainty, since a system may not be able to detect all events reliably and the effects of events may not always be known in advance. In order to deal with uncertainty, we have to extend the logic-based Event Calculus with probabilistic reasoning capabilities and try to learn the effects of events from example interpretations. Effect axioms are important in the context of the Event Calculus, since they specify which properties are initiated or terminated when a particular event occurs at a given point in time.

Recently, the combination of logic programming and probability under the distribution semantics [5, 23] has proven to be useful for building rich representations of domains consisting of individuals and uncertain relations between these individuals. These representations can be learned in an efficient way and used to carry out inference. The distribution semantics underlies a number of probabilistic logic languages such as PRISM [23], Independent Choice Logic [14, 15], Logic Programs with Annotated Disjunctions [27], P-log [1], and ProbLog [4, 6]. Since these languages have the same formal foundation, there exist linear transformations between them that preserve their semantics [20].

In this paper, we investigate how the language of Logic Programs with Annotated Disjunctions (LPAD) and its implementation in the *cplint* framework of programs for reasoning with probabilistic logic programs [22] can be used to learn the structure of probabilistic effect axioms for a dialect of the Event Calculus.

Recently, Skarlatidis and colleagues introduced two probabilistic dialects of the Event Calculus for event recognition, in particular for the detection of short-term activities in video frames. Their first dialect, Prob-EC [25], is based on probabilistic logic programming [8] and handles noise in the input data. Input events are assumed to be independent and are associated with detection probabilities. Their second dialect, MLN-EC [26], is based on Markov logic networks [16] and does not make any independence assumption of input events. Our probabilistic dialect of the Event Calculus is related to Prob-EC in the sense that it is based on the distribution semantics. However, we focus in our work not only on the processing of uncertain events but also on the learning of the structure and parameters of effect axioms from example interpretations, an issue that has not been addressed by Skarlatidis and colleagues.

The rest of this paper is structured as follows: In Section 2, we introduce our dialect of the logic-based Event Calculus, followed by a brief introduction to probabilistic logic programming in Section 3. In Section 4, we reformulate the logic-based Event Calculus as a probabilistic logic program where the events and the effect axioms are annotated with probabilities. In Section 5, we discuss how we can learn the structure of these effect axioms from positive and negative interpretations that are available in the form of short dialogue sequences. In Section 6, we present our experiments and show that the proposed probabilistic dialect is an elaboration-tolerant version of the logic-based Event Calculus. In Section 7, we summarise the advantages of our approach and present our conclusion.

2 The Event Calculus

The original logic-based Event Calculus as introduced by [9] is a logic programming formalism for representing the effects of events on properties. The basic ontology of the Event Calculus consists of events, fluents, and time points. An *event* represents an action that may occur in the world; for example, a person who is arriving in the kitchen. A *fluent* represents a time-varying property that might be the effect of an event; for example, a person who is located in the kitchen (after arriving in the kitchen). A *time point* represents an instant of time and indicates when an event happens or when a fluent holds; for example, Sunday, 19-Feb-17, 23:59:00, UTC-12[1]. In the following discussion we introduce the axioms of our dialect of the Simple Event Calculus (SEC) [24]. These axioms are implemented as Prolog clauses and displayed in Listing 1.

Listing 1 The Simple Event Calculus (SEC)

```
holds_at(fluent:F, tp:T) :-                                    % SEC1
  initially(fluent:F),
  \+ clipped(tp:0, fluent:F, tp:T).

holds_at(fluent:F, tp:T2) :-                                   % SEC2
  initiated_at(fluent:F, tp:T1),
  T1 < T2,
  \+ clipped(tp:T1, fluent:F, tp:T2).
```

[1] In the following we use integers instead of POSIX time to save space in the paper.

```
clipped(tp:T1, fluent:F, tp:T3) :-                              % SEC3
  terminated_at(fluent:F, tp:T2),
  T1 < T2, T2 < T3.

initiated_at([fluent:located, pers:A, loc:B], tp:C) :-          % EAX1
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C) :-         % EAX2
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  B \= D.

initially([fluent:located, pers:bob, loc:garden]).              % SCO1

happens_at([event:arrive, pers:bob, loc:kitchen], tp:3).        % SCO2

happens_at([event:arrive, pers:bob, loc:garage], tp:5).         % SCO3
```

Axiom `SEC1` specifies that a fluent `F` initially holds at time point `T1`, if it held at time point `0`, and has not been terminated between these two time points. Axiom `SEC2` specifies that a fluent `F` holds at time point `T2`, if the fluent has been initiated at some time point `T1`, which is before `T2` and has not been clipped between `T1` and `T2`. Axiom `SEC3` states that a fluent `F` has been clipped between time point `T1` and `T3`, if the fluent has been terminated at a time point `T2` and this time point is between `T1` and `T3`. Note that according to these domain-independent axioms (`SEC1-SEC3`), a fluent does not hold at the time of the event that initiates it but at the time of the event that terminates it.

Events have effects on properties of the world; they initiate and terminate these properties at a given point in time. These effects can be described by domain-dependent effect axioms. For example, the positive effect axiom `EAX1` in Listing 1 specifies that the fluent with the name `located` involving a person `A` and a location `B` is initiated after the time point `C`, if an event occurs at a time point `C` where the person `A` arrives at the location `B`. The negative effect axiom `EAX2` in Listing 1 specifies that the fluent with the name `located` involving a person `A` and a location `D` is terminated after the time point `C`, if an event occurs at a time point `C` where the person `A` arrives at a location `B` that is different from location `D`. Finally, a scenario (`SCO1-SCO3`) is required where the initial state of the world (`SCO1`) is described as well as a sequence of events (`SCO2` and `SCO3`) that occur at subsequent time points.

This setting allows us to investigate which fluents hold at a given point in time. The logic-based SEC assumes that the effect axioms are known in advance and that there is no uncertainty in the relationships between events and effects and in the recognition of events. In the following, we assume that the effect axioms are unknown and need to be learned first from example interpretations and that the recognition of events that occur in the real world can be noisy. During the learning process, the structure of the effect axioms will be generated automatically and the resulting axioms will be annotated with probabilities. These probabilistic effect axioms can then be processed with a version of the SEC that is based on a probabilistic logic programming language which supports probabilistic reasoning.

3 Probabilistic Logic Programs (PLP)

One of most successful approaches to Probabilistic Logic Programs (PLP) is based on the distribution semantics [23]. Under the distribution semantics a probabilistic logic program defines a probability distribution over a set of normal logic programs (called *worlds*). The

probability of a query is then obtained from the joint distribution of the query and the worlds by marginalization.

While the distribution semantics underlies a number of different probabilistic languages (see [20] for an overview), Logic Programs with Annotated Disjunctions (LPAD) [27] offer the most general syntax of this family of languages. An LPAD P consists of a set of annotated disjunctive clauses C_i of the form:

$h_1 : \alpha_1; \ldots; h_n : \alpha_n \leftarrow b_1, \ldots, b_m.$

where h_i are logical atoms, α_i are real numbers, each of them standing for a probability in the interval $[0, 1]$ such that the sum of all α_i is 1, and b_i are logical literals (incl. negation as failure). The set of elements $h_i : \alpha_i$ form the head of a clause and the set of elements b_i the body. Disjunction in the head is represented by a semicolon and the atoms in the head a separated by a colon from their probabilities. Note that if $n = 1$ and $\alpha_1 = 1$, then a clause corresponds to a normal clause and the annotation can be omitted. Note also that if the sum of all α_i is smaller than 1, then an additional disjunct *null* is assumed with probability $1 - sum(\alpha_i)$. If the body of a clause is empty, then it can be omitted.

The semantics of an LPAD P is defined via its grounding. The grounding of P is obtained by replacing the variables of each clause C with the terms of the Herbrand universe $H_U(P)$ [10]. Each of these ground clauses represents a probabilistic choice between a number of non-disjunctive clauses. By selecting a head atom for each ground clause of an LPAD, we get an instance of a normal logic program. Each selection has its own probability assigned to it and the product of these probabilities induces the probability of a program instance, assuming independence among the choices made for each clause. All instances of an LPAD together define a probability distribution over a set of interpretations of the program. The probability of a particular interpretation I is then the sum of the probability of all instances for which I is a model (see [27] for details).

4 The Simple Event Calculus as a PLP

We can reformulate the logic-based SEC as a PLP and process it with the help of the PITA library [18] that runs in SWI Prolog[2]. PITA computes the probability of a query from an LPAD program by transforming the program into a normal program that contains calls to manipulate Binary Decision Diagrams as auxiliary data structures and is evaluated by Prolog. PITA was compared with ProbLog [4] and found to be fast and scalable [17]. Alternatively, we can use MCINTYRE [19], if exact inference in PITA gets too expensive. MCINTYRE performs approximate inference using Monte Carlo sampling. Both PITA and MCINTYRE are part of the *cplint* framework[3] for reasoning with probabilistic logic programs.

In contrast to the logic-based dialect of the SEC, we have to load the `pita` (or `mcintyre`) library first, initialise it with a corresponding directive and enclose the LPAD clauses for the SEC in additional directives that mark the start and end of the program as illustrated in Listing 2:

Listing 2 PITA Initialisation

```
:- use_module(library(pita)).
:- pita.
```

[2] http://www.swi-prolog.org
[3] https://github.com/friguzzi/cplint

```
:- begin_lpad.

   % The SEC with probabilistic events and probabilistic effect
   % axioms goes here.

:- end_lpad.
```

The domain-independent clauses for the probabilistic version of the SEC are the same (`SEC1-SEC3`) as those of the logic-based version introduced in Listing 1. The probabilistic effect axioms that we are going to learn (see Section 5) have the same basic form as the axioms `EAX1` and `EAX2`, but they are additionally annotated with probabilities and contain special conditions in the body of the clauses to guarantee that all variables in the head of a clause are range restricted; otherwise the distribution semantics is not well-defined for an LPAD program. In our case, the annotated effect axioms look as illustrated in Listing 3:

Listing 3 Effect Axioms with Probabilities

```
initiated_at([fluent:located, pers:A, loc:B], tp:C):0.66;'':0.34 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C):0.66;'':0.34 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  location([loc:D]).
```

Here, the value `0.66` stands for the probability of the clause to be true and the value `0.34` for the probability of the clause to be false. Note that the predicate `location/1` in the body of the second clause restricts the range of the variable `D` in the head of the clause. That means we have to make sure – as we will see in Section 5.2 – that facts for the locations are available in the background knowledge for the LPAD program that is used to learn the structure of these axioms.

In order to deal with uncertainty in the initial setting and the recognition of events, we can also add probabilities to the facts that describe the scenario (`SCO1-SCO3`); for example, to the facts `SCO2` and `SCO3` that describe the events as shown in Listing 4:

Listing 4 Events with Probabilities

```
happens_at([event:arrive, pers:bob, loc:kitchen], tp:3):0.95;'':0.05.
happens_at([event:arrive, pers:bob, loc:garage],  tp:5):0.99;'':0.01.
```

If the annotation of the probability is omitted in a clause, then its value is implicitly `1`. As we will see in Section 6, the probabilistic version of the SEC without annotated probabilities behaves exactly like the logic-based version of the SEC. In this respect, the probabilistic version of the SEC can be considered as an elaboration-tolerant extension of the logic-based version, since all modifications that are required for building the resulting probabilistic logic program are additive.

5 Learning the Structure of Effect Axioms

To learn the structure of effect axioms from positive and negative interpretations, we use a separate LPAD program together with SLIPCOVER [3], an algorithm for learning the structure and parameters of probabilistic logic programs. SLIPCOVER takes as input a set of example interpretations and a language bias that indicates which predicates are target. These interpretations must contain positive and negative examples for all predicates that

may appear in the head of a clause. SLIPCOVER learns the structure of effect axioms by first performing a beam search in the space of probabilistic clauses and then a greedy search in the space of theories. The first search step starts from a set of bottom clauses, aims at finding a set of promising clauses, and looks for good refinements of these clauses in terms of the log-likelihood of the data. The second search step starts with an empty theory and tries to add each clause for a target predicate to that theory. After each addition, the log-likelihood of the data is computed as the score of the new theory. If the value of the new theory is better than the value of the previous theory, then the clause is kept in the theory, otherwise the clause is discarded. Finally, SLIPCOVER completes a theory consisting of target predicates by adding the body predicates to the clauses and performs parameter learning on the resulting theory (for details see [3]).

Note that the refinements during this process are scored by estimating the log-likelihood of the data by running a small number of iterations of EMBLEM [2], an implementation of expectation-maximization for learning parameters that computes expectations directly on Binary Decision Diagrams.

Listing 5 LPAD Program for Learning the Structure of Effect Axioms with SLIPCOVER

```
:- use_module(library(slipcover)).                                    % 1.1
:- sc.                                                                % 1.2
:- set_sc(max_var, 4).                                                % 1.3
:- set_sc(megaex_bottom, 3).                                          % 1.4
:- set_sc(depth_bound, false).                                        % 1.5
:- set_sc(neg_ex, given).                                             % 1.6

:- begin_bg.                                                          % 2.1
   location([loc: bathroom]).                                         % 2.2
   location([loc: kitchen]).                                          % 2.3
   location([loc: living_room]).                                      % 2.4
:- end_bg.                                                            % 2.5

output(initiated_at/2).                                               % 3.1
input(happens_at/2).                                                  % 3.2
modeh(*, initiated_at([fluent: -#fl, pers: +pers, loc: +loc],         % 3.3
                      tp: +tp)).
modeb(*, happens_at([event: -#ev, pers: +pers, loc: +loc],            % 3.4
                    tp: +tp)).
determination(initiated_at/2, happens_at/2).                          % 3.5

initiated_at(ID, [fluent:F2, pers:P, loc:L2], tp:T2) :-               % 4.1
   holds_at(ID, [fluent:_F1, pers:P, loc:_L1], tp:T1),
   happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
   T1 < T2,
   holds_at(ID, [fluent:F2, pers:P, loc:L2], tp:T3),
   T2 < T3.

neg(initiated_at(ID, [fluent:F2, pers:P, loc:_L], tp:T2)) :-          % 4.2
   holds_at(ID, [fluent:_F1, pers:P, loc:_L1], tp:T1),
   happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
   T1 < T2,
   neg(holds_at(ID, [fluent:F2, pers:P, loc:L2], tp:T3)),
   T2 < T3.
```

```
begin(model(f1)).                                                    % 5.1
   holds_at([fluent:located, pers:mary, loc:bathroom], tp:0).
   happens_at([event:arrive, pers:mary, loc:kitchen], tp:1).
   holds_at([fluent:located, pers:mary, loc:kitchen], tp:2).
end(model(f1)).

begin(model(f2)).                                                    % 5.2
   holds_at([fluent:located, pers:emma, loc:living_room], tp:3).
   happens_at([event:arrive, pers:emma, loc:bathroom], tp:4).
   neg(holds_at([fluent:located, pers:emma, loc:bathroom], tp:5)).
end(model(f2)).

begin(model(f3)).                                                    % 5.3
   holds_at([fluent:located, pers:sue, loc:kitchen], tp:6).
   happens_at([event:arrive, pers:sue, loc:living_room], tp:7).
   holds_at([fluent:located, pers:sue, loc:living_room], tp:8).
end(model(f3)).

fold(train, [f1, f2, f3]).                                           % 5.4

learn_effect_axioms(C) :-                                            % 5.5
  induce([train], C).
```

In our case, the LPAD program for structure learning consists of five parts: (1) a preamble, (2) background knowledge for type definitions, (3) language bias information, (4) clauses for finding examples, and (5) example interpretations (= models). In the following subsections, we discuss these parts in more detail.

5.1 Preamble

In the preamble of the program, the SLIPCOVER library is loaded (1.1) and initialised (1.2), and the relevant parameters are set (1.3-1.6). In our case, these parameters are: the maximum number of distinct variables that can occur in a clause to be learned (1.3); the number of examples on which to build the bottom clauses (1.4); the depth of the derivation which is unbound in our case, and therefore `false` (1.5); and the availability of negative examples in the interpretations (1.6).

5.2 Background Knowledge

The background knowledge specifies the kind of knowledge that is valid for all interpretations of a clause. This knowledge is enclosed in a begin directive (2.1) and an end directive (2.5). In our case, it contains type definitions for different locations (2.2-2.4) that are required to restrict the range of variables in the clauses to be learned. These are predicates that may be used for constructing the body of a clause. As we will see in Section 5.5, these type definitions are automatically derived from the same dialogue sequences that are used to construct the example interpretations.

5.3 Language Bias Information

The language bias specifies the accepted structure of the clauses to be learned and helps to guide the construction of the refinements for the resulting theory. The language bias is

expressed in terms of: (a) predicate declarations, (b) mode declarations, (c) type specifications, and (d) determination statements.

(a) **Predicate declarations** take the form of output predicates (3.1) or input predicates (3.2). Output predicates are declared as `output/1` and specify those predicates whose atoms one intends to predict. Input predicates are declared as `input/1` and specify those predicates whose atoms one is not interested in predicting but which should occur in the body of a hypothesized clause.

(b) **Mode declarations** are used to guide the process of constructing a generalisation from example interpretations and to constrain the search space for the resulting clauses. We distinguish between head mode declarations (3.3) and body mode declarations (3.4). A head mode declaration (`modeh(n, atom)`) specifies the atoms that can occur in the head of a clause and a body mode declaration (`modeb(n, atom)`) those atoms that can occur in the body of a clause. The argument `n`, the recall, is either an integer ($n \geq 1$) or an asterisk (`*`) and indicates how many atoms for the predicate specification are retained in the bottom clause during a saturation step. The asterisk stands for all those atoms that are found; otherwise the indicated number of atoms is randomly chosen.

(c) **Type specifications** have, in our case, the form `+type`, `-type`, or `-#type`, and specify that the argument should be either an input variable (`+`) of that type, an output variable (`-`) of that type, or a constant (`-#`) of that type. For example, the argument of the form `-#fl` in the head mode declaration of (3.3) stands for the name of a fluent, and the argument of the form `-#ev` in the body mode declaration of (3.4) stands for the name of an event, and symbols prefixed with `+` and `-` for input and output variables.

(d) **Determination statements** such as (3.5) are required by SLIPCOVER and declare which predicates can occur in the body of a particular clause.

In addition to the specification of the language bias for the positive effect axiom (`initiated_at/2`) in Listing 5, we present below in Listing 6 the specification for the negative effect axiom (`terminated_at/2`), since the successful construction of this negative effect axiom depends on some important additions:

Listing 6 Negative Effect Axiom

```
output(terminated_at/2).
input(happens_at/2)

modeh(*, terminated_at([fluent: -#fl, pers: +pers, loc: +loc2],
                       tp: +tp)).
modeb(*, happens_at([event: -#ev, pers: +pers, loc: -loc1], tp: +tp)).
modeb(*, location([loc: +loc2])).

determination(terminated_at/2, happens_at/2).
determination(terminated_at/2, location/1).

lookahead_cons_var(location([loc:_L2]),
                   [happens_at([event:_E, pers:_P, loc:_L1], tp:_T)]).
lookahead_cons_var(happens_at([event:_E, pers:_P, loc:_L1], tp:_T),
                   [location([loc:_L2])]).
```

Here, the two determination statements indicate that the predicate `happens_at/2` as well as the predicate `location/1` can appear in the body of the negative effect axiom. Additionally, we have to specify a lookahead that enforces that whenever one of these two predicates is added to the body of the clause during refinement, then also the other predicate needs to be added to that body.

5.4 Program Clauses for Finding Examples

The program clauses for finding examples (4.1 + 4.2) in Listing 5 contain those predicates that are used during search. These clauses are not part of the background knowledge and therefore contain an additional argument (`ID`) that is used to identify the relevant example interpretations (models). We encode the search for finding examples intensionally as the clauses in Listing 5 illustrate. This representation completes the interpretations by generating positive and negative examples for the positive effect axiom (`initiated_at/2`) using the predefined predicate `neg/1` in the clause (4.2). To complete our discussion, we show below in Listing 7 the clauses for finding examples for the negative effect axiom (`terminated_at/2`):

Listing 7 Finding Negative Effect Axiom

```
terminated_at(ID, [fluent:F1, pers:P, loc:L1], tp:T2) :-
  holds_at(ID, [fluent:F1, pers:P, loc:L1], tp:T1),
  happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
  T1 < T2,
  holds_at(ID, [fluent:_F2, pers:P, loc:L2], tp:T3),
  T2 < T3.

neg(terminated_at(ID, [fluent:F1, pers:P, loc:L1], tp:T2)) :-
  holds_at(ID, [fluent:F1, pers:P, loc:L1], tp:T1),
  happens_at(ID, [event:_E, pers:P, loc:L2], tp:T2),
  T1 < T2,
  neg(holds_at(ID, [fluent:_F2, pers:P, loc:L2], tp:T3)),
  T2 < T3.
```

The interesting thing to note here is that the variable `L1` for the location becomes available via the example interpretation, but the predicate `location/1` that is used to restrict the range of this variable comes from the background information and is enforced via the lookahead predicate discussed in the previous section.

5.5 Example Interpretations

In our case the example interpretations (5.1-5.3) in Listing 5 are derived from short dialogue sequences. We experimented with dialogue sequences consisting of a state sentence, followed by an event sentence, followed by a question that results in a positive or negative answer. These dialogue sequences are similar to the data used in the dialogue-based language learning dataset [28], and look in our case as follows:

S.1.1 Mary is located in the bathroom.
S.1.2 Mary arrives in the kitchen.
S.1.3 Where is Mary?
S.1.4 In the kitchen.

S.2.1 Emma is located in the living room.
S.2.2 Emma arrives in the bathroom.
S.2.3 Is Emma in the bathroom?
S.2.4 No.

These dialogues are parsed into dependency structures and automatically translated into the corresponding Event Calculus representation. For this purpose, we used the Stanford Parser [11] that generates for each sentence (or answer fragment) a dependency structure.

This dependency structure is then translated with the help of a simple ontology into the corresponding Event Calculus notation. The ontology contains, among other things, the information that Mary is a person and that a bathroom is a location. The dialogue sequence S.1.1-S.1.4, for example, is first translated into four dependency structures as illustrated on the left-hand side of Listing 8:

Listing 8 From Dependency Structures to Event Calculus Representation

```
DEPENDENCY STRUCTURES                EVENT CALCULUS REPRESENTATION

% D.1.1
nsubjpass(located-3, Mary-1)         holds_at([fluent:located,
auxpass(located-3, is-2)                       pers:mary,
root(ROOT-0, located-3)                        loc:bathroom], tp:1).
case(bathroom-6, in-4)
det(bathroom-6, the-5)               location([loc:bathroom]).
nmod(located-3, bathroom-6)

% D.1.2
nsubj(arrives-2, Mary-1)             happens_at([event:arrive,
root(ROOT-0, arrives-2)                         pers:mary,
case(kitchen-5, in-3)                           loc:kitchen], tp:2).
det(kitchen-5, the-4)
nmod(arrives-2, kitchen-5)           location([loc:kitchen]).

% D.1.3
advmod(is-2, Where-1)
root(ROOT-0, is-2)
nsubj(is-2, Mary-3)                  holds_at([fluent:located,
                                               pers:mary,
% D.1.4                                        loc:kitchen], tp:3).
case(kitchen-3, In-1)
det(kitchen-3, the-2)
root(ROOT-0, kitchen-3)
```

For example, the dependency structure D.1.1 for the sentence S.1.1 is translated into a `holds_at/2` predicate, since the sentence describes a fluent. The translation of the dependency structure D.1.2 for sentence S.1.2 results in a `happens_at/2` predicate, since the sentence describes an event. Finally, the translation of the two dependency structures D.1.3 + D.1.4 for the question-answer pair in S.1.3 + S.1.4 introduce a positive `holds_at/2` predicate, since the question and the answer together confirm that a particular fluent holds. Note that this dialogue sequence is also used to derive together with the help of the ontology the factual information that bathroom and kitchen are locations.

In our LPAD program for structure learning in Listing 5, the derived example interpretations (5.1-5.3) are initiated by predicates of the form `begin(model(<name>))` and terminated by predicates of the form `end(model(<name>))` and the relevant background information is added to the background section (2.1-2.5) of the program.

Note that each example interpretation may contain an additional fact of the form `prob(P)` that assigns a probability `P` to the interpretation. This probability may be used to reflect the confidence of the parser in a particular interpretation. If this probability is omitted, then the probability of each interpretation is considered equal to `1/n` where `n` is the total number of interpretations (for details see [22]).

Finally, we have to specify how the example interpretations are divided in folds for taining (5.4), before we can perform parameter learning on the training fold as illustrated in (5.5).

6 Experiments

In order to get more realistic probabilities for the effect axioms that we discussed in this paper, we learned the structure of these axioms with the help of 50 example dialogues. This resulted in the following probabilities for the two effect axioms as shown in Listing 9:

Listing 9 Effect Axioms with Probabilities Derived from Example Dialogues

```
initiated_at([fluent:located, pers:A, loc:B], tp:C):0.87;'':0.13 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C).

terminated_at([fluent:located, pers:A, loc:D], tp:C):0.87;'':0.13 :-
  happens_at([event:arrive, pers:A, loc:B], tp:C),
  location([loc:D]).
```

We used the same domain-independent axioms (`SEC1-SEC3`) for our experiments as in the logic-based version of the SEC in Listing 1, but added the following background axioms in Listing 10 to the probabilistic version of the program to deal with the range requirement for variables of the learned clauses under the distribution semantics:

Listing 10 Background Axioms

```
location([loc: garage]).
location([loc: garden]).
location([loc: kitchen]).
```

To test the probabilistic dialect of the SEC, we conducted three experiments using PITA and MCINTYRE for reasoning and the queries shown in Listing 11 + 12. In the first experiment A, we removed all probabilities from the axioms and executed the queries; in the second experiment B, we used the effect axioms annotated with the learned probabilities to answer the queries; and finally in the third experiment C, we used the annotated effect axioms together with probabilities for noisy event occurrences and executed the queries. We then run the same three experiments using MCINTYRE and sampled the answer for each query 100 times. This sampling process returns the estimated probability that a sample is true (i.e., that a sample succeeds).

Listing 11 Test Queries used to Evaluate the SEC with PITA

```
test_ec_pita(Num, [P1, P2, P3, P4, P5, P6, P7]) :-
  prob( holds_at([fluent:located, pers:bob, loc:garden],  tp:1, P1 ),
  prob( holds_at([fluent:located, pers:bob, loc:garden],  tp:2, P2 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:3, P3 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:4, P4 ),
  prob( holds_at([fluent:located, pers:bob, loc:kitchen], tp:5, P5 ),
  prob( holds_at([fluent:located, pers:bob, loc:garage],  tp:5, P6 ),
  prob( holds_at([fluent:located, pers:bob, loc:garage],  tp:6, P7 ).
```

Listing 12 Test Queries used to Evaluate the SEC with MCINTYRE

```
test_ec_mcintyre([P1, P2, P3, P4, P5, P6, P7]) :-
  mc_sample( holds_at([fluent:located, pers:bob, loc:garden],  tp:1),
             100, P1 ),
```

```
mc_sample( holds_at([fluent:located, pers:bob, loc:garden],  tp:2),
           100, P2 ),
mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:3),
           100, P3 ),
mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:4),
           100, P4 ),
mc_sample( holds_at([fluent:located, pers:bob, loc:kitchen], tp:5),
           100, P5 ),
mc_sample( holds_at([fluent:located, pers:bob, loc:garage],  tp:5),
           100, P6 ),
mc_sample( holds_at([fluent:located, pers:bob, loc:garage],  tp:6),
           100, P7 ).
```

6.1 Experiment A

For the first experiment, we removed the probabilities of the learned effect axioms and automatically combined them with the domain-independent axioms (`SEC1-SEC3`) and the axioms of the original scenario (`SCO1-SCO3`), and then executed the queries in Listing 11 and 12 using the two inference modules PITA (`P`) and MCINTYRE (`M`). This resulted in the following answers:

```
(P) Probs = [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0]
(M) Probs = [1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0]
```

Here the value `1.0` stands for "true" and the value `0.0` for "false" and the answers are the same as we get for the logic-based version of the SEC. Note that by definition a fluent does not hold at the time point of the event that initiates it; therefore, the probability for the answer of the third query and the sixth query in Listing 11 + 12 is `0.0`.

6.2 Experiment B

In this experiment, we used the probabilistic effect axioms that we learned with the help of our 50 training examples. This gives the following results for our test queries:

```
(P) Probs = [1.0, 1.0, 0.0, 0.87, 0.87, 0.0, 0.87]
(M) Probs = [1.0, 1.0, 0.0, 0.85, 0.91, 0.0, 0.87]
```

Note that the first three answers and the sixth answer are the same as before, answer four, five and seven show the probabilistic effect axioms at work. Since MCINTYRE uses inexact inference and relies on sampling, the results for each run show some variation. We observe a standard deviation of about 0.034 for the relevant answers when we run each query ten times and use 100 samples for each query (see Listing 12).

6.3 Experiment C

For our third experiment, we used again the probabilistic effect axioms and added probabilities to the events to deal with a situation where we have noise in the recognition of events:

```
happens_at([event:arrive,pers:bob,loc:kitchen],tp:3):0.95;'':0.05.
happens_at([event:arrive,pers:bob,loc:garage],tp:5):0.99;'':0.01.
```

Running our test queries under these uncertain conditions gives the following results:

```
(P) Probs = [1.0, 1.0, 0.0, 0.8265, 0.8265, 0.0, 0.8613]
(M) Probs = [1.0, 1.0, 0.0, 0.84, 0.82, 0.0, 0.84]
```

As expected, uncertainty in event recognition lowers the probability for the relevant answers, and there is of course some variation under inexact inference for each run.

7 Conclusion

In this paper we showed how we can automatically learn the structure and parameters of probabilistic effect axioms for the Simple Event Calculus (SEC) from positive and negative example interpretations stated as short dialogue sequences in natural language. We used the *cplint* framework for this task that provides libraries for structure and parameter learning and for answering queries with exact and inexact inference. The example dialogues that are used for learning the structure of the probabilistic logic program are parsed into dependency structures and then further translated into the Event Calculus notation with the help of a simple ontology. The novelty of our approach is that we can not only process uncertainty in event recognition but also learn the structure of effect axioms and combine these two sources of uncertainty to successfully answer queries under this probabilistic setting. Interestingly, our extension of the logic-based version of the SEC is completely elaboration-tolerant in the sense that the probabilistic version fully includes the logic-based version. This makes it possible to use the probabilistic version of the SEC in the traditional way as well as when we have to deal with uncertainty in the observed world. In the future, we would like to extend the probabilistic version of the SEC to deal – among other things – with concurrent actions and continuous change.

Acknowledgements. I would like to thank Fabrizio Riguzzi for his valuable help with the implementation of the algorithm for learning the effect axioms.

References

1 Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic reasoning with answer sets. In *Theory and Practice of Logic Programming*, 9(1), pp. 57–144, 2009.

2 Elena Bellodi and Fabrizio Riguzzi. Expectation Maximization over binary decision diagrams for probabilistic logic programs. In *Intelligent Data Analysis*, 17(2), pp. 343–363, 2013.

3 Elena Bellodi and Fabrizio Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. In *Theory and Practice of Logic Programming*, 15(2), pp. 169–212, 2015.

4 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic Prolog and its application in link discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, pp. 2462–2467, 2007.

5 Luc De Raedt and Angelika Kimmig. Probabilistic (logic) programming concepts. In *Machine Learning*, Vol. 100, Issue 1, Springer New York LLC, pp. 5–47, 2015.

6 Anton Dries, Angelika Kimmig, Wannes Meert, Joris Renkens, Guy Van den Broeck, Jonas Vlasselaer and Luc De Raedt. ProbLog2: Probabilistic logic programming. In *Machine Learning and Knowledge Discovery in Databases*, LNCS 9286, Springer, pp. 312–315, 2015.

7 Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. Incremental learning of event definitions with Inductive Logic Programming. In *Machine Learning*, Vol. 100, Issue 2, pp. 555–585, 2015.

8 Angelika Kimmig, Bart Demoen, Luc De Raedt, Vitor Santos Costa and Ricardo Rocha. On the implementation of the probabilistic logic programming language ProbLog. In: *Theory and Practice of Logic Programming*, Vol. 11, pp. 235–262, 2011.

9 Robert Kowalski and Marek Sergot. A Logic-Based Calculus of Events. In *New Generation Computing*, Vol. 4, pp. 67–95, 1986.

10 John W. Lloyd. *Foundations of logic programming.* Second, Extended Edition. Springer-Verlag, New York, 1987.

11 Christopher, D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60, 2014.

12 Rob Miller and Murray Shanahan. Some Alternative Formulations of the Event Calculus. In *Computational Logic: Logic Programming and Beyond – Essays in Honour of Robert A. Kowaski*, LNAI 2408, Springer pp. 452–490, 2002.

13 Erik T. Mueller. *Commonsense Reasoning, An Event Calculus Based Approach.* 2nd Edition, Morgan Kaufmann/Elesevier, 2015.

14 David Poole. The independent choice logic for modelling multiple agents under uncertainty. In *Artificial Intelligence* Vol. 94, pp. 7–56, 1997.

15 David Poole. The independent choice logic and beyond. In L. De Raedt, P. Frasconi, K. Kersting, and S. Muggleton (eds.), *Probabilistic Inductive Logic Programming: Theory and Application*, LNAI Vol. 4911, Springer, pp. 222–243, 2008.

16 Matthew Richardson and Pedro Domingos. Markov logic networks. In *Machine Learning*, Vol. 62, Issue 1, pp. 107–136, 2006.

17 Fabrizio Riguzzi and Terrance Swift. An extended semantics for logic programs with annotated disjunctions and its efficient implementation. In *Italian Conference on Computational Logic.* CEUR Workshop Proceedings, Vol. 598. Sun SITE Central Europe, 2010.

18 Fabrizio Riguzzi and Terrance Swift. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. In *Theory and Practice of Logic Programming, 27th International Conference on Logic Programming (ICLP'11) Special Issue*, 11(4-5), pp. 433–449, 2011.

19 Fabrizio Riguzzi. MCINTYRE: A Monte Carlo system for probabilistic logic programming. In: *Fundamenta Informaticae*, 124(4), pp. 521–541, 2013.

20 Fabrizio Riguzzi, Elena Bellodi, and Riccardo Zese. A History of Probabilistic Inductive Logic Programming. In *Frontiers in Robotics and AI*, 18. September 2014.

21 Fabrizio Riguzzi and Terrance Swift. Probabilistic logic programming under the distribution semantics. In M. Kifer and Y. A. Liu, (eds), *Declarative Logic Programming: Theory, Systems, and Applications*, LNCS. Springer, 2016.

22 Fabrizio Riguzzi. cplint Manual. SWI-Prolog Version. July 4, 2017.

23 Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In L. Stearling (ed.), *12th International Conference on Logic Programming*, Cambridge: MIT Press, pp. 715–729, 1995.

24 Murray Shanahan. The Event Calculus Explained. In M.J. Wooldridge and M. Veloso (eds), *Artificial Intelligence Today*, LNAI, Vol. 1600, Springer, pp. 409–430, 1999.

25 Anastasios Skarlatidis, Alexander Artikis, Jason Filippou, Georgios Paliouras. A Probabilistic Logic Programming Event Calculus. In *Theory and Practice of Logic Programming*, Vol. 15, No. 2, pp. 213–245, 2015.

26 Anastasios Skarlatidis, Georgios Paliouras, Alexander Artikis, and George A. Vouros. Probabilistic Event Calculus for Event Recognition. In *ACM Transactions on Computational Logic*, Vol. 16, No. 2, Article 11, 2015.

27 Joost Vennekens, Sofie Verbaeten, and Maurice Bruynooghe. Logic programs with annotated disjunctions. In *International Conference on Logic Programming* LNCS 3131, Berlin: Springer, pp. 195–209, 2004.

28 Jason Weston. Dialog-based Language Learning. Facebook AI Research. In *arXiv:1604.06045v7*, 24th October 2016.

Towards Run-time Checks Simplification via Term Hiding*†

Nataliia Stulova[1], José F. Morales[2], and Manuel V. Hermenegildo[3]

1 **IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain**
nataliia.stulova@imdea.org
2 **IMDEA Software Institute, Madrid, Spain**
josef.morales@imdea.org
3 **IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain**
manuel.hermenegildo@upm.es

Abstract

Flexibility in term creation and manipulation in dynamic languages can threaten safety of data processing. To counter this issue expensive run-time checks are often added to programs to ensure safety of operations, yet they incur impractically high overheads. While such overheads can be greatly reduced with static analysis, the gains depend strongly on the quality of the information inferred.

We propose a technique for improving term shape inference during static program analysis, that exploits term visibility rules of the underlying module system. We also describe an improved run-time checking approach that takes advantage of the proposed mechanisms to achieve large reductions in overhead. While the approach is general and system-independent, we present and evaluate it for concreteness in the context of the Ciao assertion language and combined static/dynamic checking framework. One of its benefits is that it does not introduce the need to switch the language to (static) type systems, which is known to change the semantics in languages like Prolog, while allowing for reductions in overhead closer to those of static languages.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.3 Studies of Program Constructs, F.3.2 Semantics of Programming Languages

Keywords and phrases Module Systems, Implementation; Run-time Checking, Assertion-based Debugging and Validation, Static Analysis

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.9

1 Brief Overview

Modular programming has become widely adopted due to the benefits it provides in code reuse and for structuring data flow between program components. A tightly related concept is the principle of *information hiding* that allows concealing the concrete implementation details behind a well-defined interface and thus allows for cleaner abstractions. In different

* In [2] we provide full details on this work.
† This research has been partially funded by Spanish MINECO project TIN2015-67522-C3-1-R *TRACES*, and Madrid Region program M141047003 *N-GREENS*.

programming languages these concepts are implemented in different ways, some examples being the encapsulation mechanism of classes adopted in object-oriented programming and opaque data types. In the (constraint) logic programming context, most mature language implementations incorporate module systems, which are either *predicate-based* (where predicate symbol visibility is controlled by the module import-export rules but functor symbols are public) or *atom-based* (where both predicate and functor symbol visibility is controlled by the module import-export rules).

We propose a hybrid predicate-based module system [2] that offers an optional hiding mechanism for selected functor symbols, providing more fine-grained term visibility control. The proposed module system is still strict in the sense that it disallows breaking predicate or term visibility rules by bypassing the module interfaces. The hiding mechanism allow programmers to restrict the visibility of some terms to the module where they are defined, thus both making the concrete implementation details opaque to other modules and providing guarantees that all data terms with such shapes may only be constructed by the predicates of that particular module. Our motivation comes from the reusable library scenario, i.e., the case of analyzing, verifying, and compiling a library for general use, without access to the client code or analysis information on it. This includes for example the important case of servers accessed via remote procedure calls.

The need for mechanisms for controlling term visibility is in particular prominent in the context of assuring safety of data access and manipulation in untyped programming languages. One of the most attractive features of untyped languages for programmers is the flexibility they offer in term creation and manipulation. However, with such power comes the responsibility of ensuring correctness in the manipulation of data, and this is specially relevant when data can come from unknown clients. A popular solution for ensuring safety is to enhance the language with optional assertions that allow specifying correctness conditions both at the public module interface and for the private internal module routines [1]. These assertions can be checked dynamically by adding run-time checks to the program, but this can also introduce overheads that are in many cases impractical. Such overheads can be greatly reduced with static analysis, but the gains then depend strongly on the quality of the analysis information inferred. Unfortunately, in the reusable library setting shape/type analyses are necessarily imprecise, since in this context the unknown clients can fake data that is really intended to be internal to the library. Ensuring safety then requires sanitizing input data with potentially expensive run-time checks.

In order to reduce the checking cost, we present a technique that, using the combination of term hiding and the strict visibility rules in the module system, enhances the inference of shape information during static program analysis. By restricting some functors to the scope of a module it becomes possible to reason statically about whether the data shapes that are built with these functors are *hidden* and *visible* to the other modules with respect to the module interface. We will further refer to all possible terms that may exist outside a module m as its *escaping terms*. In [2] we provide an algorithm to compute an over-approximation of the set of all escaping terms from a module for a given set of functor hiding declarations.

▶ **Example 1.** Let `point/1` be a hidden functor in a module `m1` that exports a single predicate `p/1` which constructs a term `point(1)`:

```
:- module(m1, [p/1]).          % module interface
:- hide point/1.               % hidden functor
p(A) :- A = point(B), B = 1.
```

There is no success substitution for `p/1` where variables can be bound to some `point(_)` more general than `point(1)`. The same applies to any possible substitution in any derivation

in programs that are composed with this module. Without term hiding, this is impossible to ensure (without client knowledge) since any module could define any `point(_)` terms (e.g., `point([_,_])`, `point(coord(_,_,_))`). In this simplified example `point(1)` is the *escaping* term of module `m1`.

Note that hidden functor symbols are essential to reason *compositionally* about the flow of data in a program composed of *reusable* libraries. This is analogous to the reasoning about the semantics of the predicates in a module, which requires the predicate symbols to be local. The information about escaping terms obtained by the static analysis can then be used to replace the original run-time checks with their optimized versions while preserving the safety guarantees the original checks provide. These optimized, or *shallow* versions of properties are weakened forms that are semantically equivalent to the original ones in the context of the possible program executions, and are cheaper to execute (e.g., requiring asymptotically fewer steps). Shallow run-time checking consists in using *shallow* versions of properties in the run-time checks for the calls across module boundaries.

▶ **Example 2.** Assume that the set of escaping terms of m contains `point(1)` and it does not contain the more general `point(_)`. Consider the property `intpoint(point(X)) :- int(X)`. Checking `intpoint(A)` at any program point outside m must check first that `A` is instantiated to `point(X)` and that `X` is instantiated to an integer (`int(X)`). However, the escaping terms show that it is not possible for a variable to be bound to `point(X)` without `X=1`. Thus, the latter check is redundant. We can compute the optimized – or *shallow* – version of `intpoint/1` in the context of all execution points external to m as `intpoint(point(_))`.

Note that since the argument(s) inside, e.g., the first level of a term can be arbitrarily large the savings from this technique can also be unbounded. In our work we show experimentally that for practical programs and settings, thanks to the term creation safety guarantees provided by the module system, it is possible to reduce the run-time overhead for the calls across module boundaries by several orders of magnitude. Together, the combination of these techniques with traditional static analysis brings improvements in the number and cost of the run-time checks that allow providing equivalent guarantees to those of statically-typed approaches, at similar run-time cost, but without imposing on programs the restrictions of being well typed.

For concreteness, we use in this work the relevant parts of the Ciao system [1]: the module system, the assertion language –which allows providing optional program specifications with various kinds of information, such as modes, (regular) types, or non-determinism–, and the verification framework, that combines static and dynamic checking. However, our results are general and can be applied to other languages.

References

1 M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J. F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. CoRR abs/1102.5497 [cs.PL]. doi:10.1017/S1471068411000457.

2 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Term Hiding and its Impact on Run-time Check Simplification. Technical Report CLIP-1/2017.0, The CLIP Lab, May 2017. CoRR abs/1705.06662 [cs.PL].

A Hitchhiker's Guide to Reinventing a Prolog Machine

Paul Tarau

Department of Computer Science and Engineering, University of North Texas, Denton, USA
paul.tarau@unt.edu

Abstract

We take a fresh, "clean-room" look at implementing Prolog by deriving its translation to an executable representation and its execution algorithm from a simple Horn Clause meta-interpreter.

The resulting design has some interesting properties. The heap representation of terms and the abstract machine instruction encodings are the same. No dedicated code area is used as the code is placed directly on the heap. Unification and indexing operations are orthogonal. Filtering of matching clauses happens without building new structures on the heap. Variables in function and predicate symbol positions are handled with no performance penalty. A simple English-like syntax is used as an intermediate representation for clauses and goals and the same simple syntax can be used by programmers directly as an alternative to classic Prolog syntax. Solutions of (multiple) logic engines are exposed as answer streams that can be combined through typical functional programming patterns, with flexibility to stop, resume, encapsulate and interleave executions. Performance of a basic interpreter implementing our design is within a factor of 2 of a highly optimized compiled WAM-based system using the same host language.

To help placing our design on the fairly rich map of Prolog systems, we discuss similarities to existing Prolog abstract machines, with emphasis on separating necessary commonalities from arbitrary implementation choices.

1998 ACM Subject Classification D.3 Programming Languages, D.3.4 Processors

Keywords and phrases Prolog abstract machines, heap representation of terms and code, immutable goal stacks, natural language syntax for clauses, answer streams, multi-argument indexing algorithm

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.10

1 Introduction

Forgetting how to implement a Prolog system is as hard as learning how to build one. While contaminated with episodic memories acquired through the not so few Prolog systems that we have built in the past [17, 15, 14, 16], a fresh, "clean-room" attempt is described here to reinvent a Prolog machine by deriving it from the intuitions gleaned from the execution algorithm of a simplified two-clause meta-interpreter.

But why would one do this, when more than three decades of Prolog implementation seem to have fully saturated the search space of available implementation choices? Some of the details will unfold as our story progresses through the next sections, but an important reason is that we felt that existing systems, while possibly peeking out in terms of performance [5, 3, 11, 6, 24] or overall environment convenience and system usability [22, 3, 6], have left interesting implementation choices unexplored. Another reason is that the natural chain of concepts leading the execution mechanism of SLD-resolution to an efficient low-level implementation has stayed often in a "no-man's land" between theoretical work exploring

the foundations of logic programming languages and implementation work focussed mostly on refining the gold standard set by the Warren Abstract machine [21, 1, 19].

While comparisons to WAM-based systems abound to help placing in context the alternatives we propose, the paper can also be seen as a self-contained shortcut allowing a "hitchhiker" to Prolog implementation to get the core of an efficient-enough Prolog system up and running in a few days. However, given the space constraints of the paper, our step-by-step derivation process will leave out some routine elements well-known to people modestly familiar with prolog implementation.

The resulting design, implemented in an easily translatable to **C** subset of Java, around 1000 lines, is available at `http://www.cse.unt.edu/~tarau/research/2016/prologEngine.zip`. We refer to it for details that space constraints will force us to summarize or omit. More intuitions and background, for the reader less familiar with Prolog implementation, are available as the recording `https://www.youtube.com/watch?v=SRYAMt8iQSw&list=PLJq3XDLIJkib2h2fObomdFRZrQeJg4UIW` of our VMSS'2016 invited tutorial.

It covers only Horn-Clause Prolog, as this is usually a good first step to evaluate the basic implementation choices and performance characteristics of a Prolog Machine. It does not cover orthogonal implementation issues like garbage collection, unification of cyclical terms, constraints, tabling, built-ins or host language interface, as our focus is on the inner workings of the Prolog machine seen as an unification+backtracking+indexing engine.

We will start by sketching here some features that are not typically shared with most existing Prolog systems:

- a shared representation of executable code and terms on the heap
- a simple English-like intermediate representation used as our "assembler language"
- the packaging of solutions as answer streams
- the decoupling of indexing and unification instructions
- interpreted, but fast-enough execution.

The paper is organized as follows. Section 2 derives (informally) the main lines of our design from a simple Horn Clause meta-interpreter and an equational representation of Prolog terms. Section 3 describes an execution-ready heap representation of Prolog clauses that also serves as instruction set for our abstract machine. Section 4 explains the execution algorithm seen as iterated unfolding of the first goal in the body of a clause, the generation of answer streams and the main run-time data structures. Section 5 overviews a generic indexing mechanism, orthogonal to the unification-based execution algorithm, designed as an add-on to the iterated unfolding interpreter loop. Section 6 shows some preliminary performance results. Section 7 overviews related work. Section 8 concludes the paper.

2 Distilling the "essence" of Prolog's execution algorithm

When seen through the eyes of a meta-interpreter for the Horn Clause subset of Prolog, the execution algorithm is astonishingly simple. We will next expand step-by-step the intuitions behind its implicit operations and derive an equational form that we will use to guide subsequent refinements into a self-contained interpreter.

2.1 Our starting point: a simplified Horn Clause meta-interpreter

We will start by inventing a simpler meta-interpreter than the usual one, with a bit of help from a convenient clause representation. The meta-interpreter `metaint/1` uses a (difference)-list view of prolog clauses.

```
metaint([]).
metaint([G|Gs]):-
   cls([G|Bs],Gs),
   metaint(Bs).
```

Clauses are represented as facts of the form cls/2 with the first argument representing the head of the clause followed by a (possibly empty) list of body goals and terminated with a variable returned also as the second argument of cls/2.

```
cls([
     add(0,X,X)
        |Tail],Tail).
cls([
     add(s(X),Y,s(Z)), add(X,Y,Z)
        |Tail],Tail).
cls([
     goal(R), add(s(s(0)),s(s(0)),R)
        |Tail],Tail).
```

The actual content of the clauses, that we will use as our running example is marked with %1, %2 and %3. As one can verify with any Prolog system, it runs as expected when computing the successor arithmetic equivalent of $2 + 2 = 4$.

```
?- metaint([goal(R)]).
R = s(s(s(s(0)))) .
```

2.2 The equational form of terms and clauses

This flattened form of Prolog clauses is well-known. We will use the term *equation* to denote an unification step, typically between a variable and a compound term or constant. An equation like

```
T=add(s(X),Y,s(Z))
```

can be rewritten as a conjunction of 3 equations as follows

```
T=add(SX,Y,SZ),SX=s(X),SZ=s(Z)
```

When applying this to a clause like

```
C=[add(s(X),Y,s(Z)), add(X,Y,Z)]
```

it can be transformed to a conjunction derived from each member of the list

```
C=[H,B],H=add(SX,Y,SZ),SX=s(X),SZ=s(Z), B=add(X,Y,Z)
```

The list of variables (`[H,B]` in this case) can be seen as a toplevel skeleton abstracting away the main components of a Horn clause: the variable referencing the head followed by 0 or more references to the elements of the conjunction forming the body of the clause. One can see that a Prolog clause can be decomposed into a sequence of such equations, which, if executed as unification steps, build back the representation of the clause.

2.3 The "English-like equivalent" of the equational form

As the recursive tree structure of a Prolog term has been flattened, it makes sense to express it as an equivalent "controlled natural language" sentence. Note that we use here "natural

language" with a grain of salt, as we are talking about a severely restricted form of controlled English.

```
add SX Y SZ if SX holds s X and SZ holds s Z and add X Y Z.
```

Note that we keep the Prolog convention for the uppercase (or "`_`") as the fist character of variable names and the correspondence between the keywords "`if`" and "`and`" to Prolog's "`:-`" clause neck and "`,`" conjunction symbols. Note also the correspondence between the keyword "`holds`" and the use of Prolog's "`=`" to express a unification operation between a variable and a flattened Prolog term. The toplevel skeleton of the clause can be kept implicit as it is easily recoverable.

We can consider this syntax our "assembler language" to be read in directly by the loader of a runtime system, as well as the virtual machine *code* generated by a simple compiler translating Prolog clauses to it. A tokenizer splitting into words sentences delimited by "`.`" is all that is needed to complete a parser for this restricted English-style "assembler language".

2.4 A small expressiveness lift: allowing variables in function and predicate symbol positions

Let us observe right away that our flat natural syntax allows the use of variables in function and predicate symbol position as in

```
Someone likes beer if Someone likes fries and Someone drinks alcohol.
```

corresponding to a Prolog syntax involving variables in predicate positions as in

```
Someone(likes, beer):-Someone(likes, fries),Someone(drinks, alcohol).
```

This suggests dropping this Prolog restriction for a form of higher order syntax with a first order semantics giving to our intermediate language a (touch of) the capabilities of HiLog [4]. While this example can be seen as an indirect way to support the use if `likes` as an infix operator the use in

```
call(Operation,10,20,Result)
```

as

```
Operation 10 20 Result
```

hints about more interesting uses, with `Operation` working as a variable in predicate symbol position.

An easy way to make precise the semantics of such programs, is to think about them as a single freshly named conventional Prolog predicate for each arity, with a first argument denoting the name of the predicate, a variable standing for it, or standing for a compound term. This is basically the same semantics as Hilog, [4], known to be translatable to equivalent first order programs.

As a convenient notational improvement, we can instruct our parser to expand

```
Xs lists a b c
```

to

```
Xs holds list a _0 and _0 holds list b _1 and _1 holds list c nil
```

with the new keywords "`list`" representing the list constructor and "`nil`" representing the empty list.

3 The heap representation as the executable code

Derived from the equational representation of Prolog terms, our natural language form of the clauses is ready to leave Prolog for a conventional implementation language that does not provide, like our two clause meta-interpreter did, unification, recursion and backtracking for free. *It is basically an array representation with variables on the left side of our equations turned into array indices pointing to compound terms at higher addresses in the same array.*

For convenience to both the readers and the writer of this paper, we have picked a subset of Java, trivially translatable to **C** that does not make use of object oriented features, while benefitting from simplicity of automated memory management and safety coming from things like polymorphic types and index checking.

3.1 The tag system

We will instruct our tokenizer to recognize variables, symbols and (small) integers as primitive data types. As we develop a Java-based interpreter, we represent our Prolog terms top-down (as first described in [8]). Java's primitive `int` type is used for tagged words[1].

We will instruct our parser to extract as much information as possible by marking each word with a relevant tag (that will also be seen as a WAM-like instruction by the execution algorithm). We will use the following 3-bit tags:

- V=0 marking the first occurrence of a variable in a clause
- U=1 marking a second, third etc. occurrence of a variable
- R=2 marking a reference to an array slice representing a subterm
- C=3 marking the index in the symbol table of a constant (identifier or any other data object not fitting in a word)
- N=4 marking a small integer
- A=5 marking the arity of the array slice holding a flattened term (of size 1 + the number of arguments, to also make room for the "function symbol" - that could be an atom or a variable)

To ensure fast inlining in a language like Java we make most of our methods `final private static` - with similar annotations available in **C**. For clarity, we omit these annotations from our code snippets. To emulate the existence of distinct types for tagged words and their content we flip the sign when tagging and untagging:

```
int tag(int tag, int word) {return -((word << 3) + tag);}
```

```
int detag(int word) {return -word >> 3;}
```

```
int tagOf(int word) {return -word & 7;}
```

The minus sign marking `word` is meant to trigger an index error at the smallest mis-step when a method would confuse an address use and a value use of an `int`. The same technique would help catching such errors in **C**. As the cost of this operation is virtually 0, it is not worth making it a debug-only option.

Note that we will ensure that the Java compiler does as much inline expansion as possible by coding in a "**C**-friendly" style, avoiding inheritance and declaring most methods as `static`, `private` and `final`.

[1] In a **C** implementation one might want to choose `long long` instead of `int` to take advantage of the 64 bit address space.

3.2 The top-down representation of terms

Our top-down representation of Prolog terms closely follows our natural language-style assembler language syntax. After the clause

```
add(s(X),Y,s(Z)):-add(X,Y,Z).
```

compiles to

```
add _0 Y _1 and _0 holds s X and _1 holds s Z if add X Y Z .
```

it is represented on the heap (starting in this case at address 5 and shown here marked with lower case tags and colons) as follows:

```
[5]a:4 [6]c:add [7]r:10 [8]v:8 [9]r:13 [10]a:2 [11]c:s [12]v:12
[13]a:2 [14]c:s [15]v:15 [16]a:4 [17]c:add [18]u:12 [19]u:8 [20]u:15
```

Note the distinct tags of first occurrences (tagged "`v:`") and subsequent occurrences of variables (tagged "`u:`"). References (tagged "`r:`") always point to arrays starting with their length marked with tag "`a:`". As length information in kept in a separate word, cells tagged as array length contain the arity of the corresponding function symbol incremented by `1`.

The skeleton of the clause in the previous example is

```
r:5 :- [r:16]
```

as the head of this clause starts at address `5` and its (one-element) body follows at address `16`.

3.3 Clauses as descriptors of heap cells

The parser places the cells composing a clause directly to the heap, creating a *prototype clause* directly usable for its execution at runtime. At the same time, a descriptor (defined by the small class `Clause`) is created and collected to the array called "`clauses`" by the parser. An object of type `Clause` (that one would mimic with a `struct` in **C**), contains the following fields:

- `int base`: the base of the heap where the cells for the clause start
- `int len`: the length of the code of the clause i.e., number of the heap cells the clause occupies
- `int neck`: the length of the head and thus the offset where the first body element starts (or the end of the clause if none)
- `int[] gs`: the toplevel skeleton of a clause containing references to the location of its head and then body elements
- `int[] xs`: the index vector containing dereferenced constants, numbers or array sizes as extracted from the outermost term of the head of the clause, with 0 values marking variable positions.

As a side note, this is not a structure-sharing representation, as at runtime the heap representation of the clauses will be copied via a fast relocation algorithm.

4 Execution as iterated clause unfolding

As the meta-interpreter in section 2 shows it, Prolog's execution algorithm can be seen as iterated unfolding of a goal with heads of matching clauses. If unification is successful, we extend the list of goals with the elements of the body of the clause, to be solved first. Thus, indexing, meant to speed-up the selection of matching clauses, is orthogonal to the core

unification and goal reduction algorithm. Given also that we do not assume anymore that predicate symbols are non-variables, it makes sense to design indexing as a distinct algorithm, while ensuring that there's a convenient way to plug it in as a refinement of our iterated unfolding mechanism.

4.1 Unification, trailing and pre-unification clause filtering

In a way similar to the WAM's unification instructions, our relatively rich tag system reduces significantly the need to call the full unification algorithm. If one ignores the WAM's indexing instructions and avoids implementing an AND-stack by binarization [18] or by placing terms directly on the heap, the remaining unification instructions can be seen as closely corresponding to the tags of the cells on the heap, identified in our case with the "code" segment of the clauses.

We will look first at some low-level aspects of unification, that tend to be among the most frequently called operations of a Prolog machine.

4.1.1 Dereferencing

The function `deref` walks, as usual in a Prolog implementation, through variable references. We ensure that the compiler can inline it, and inline as well the functions `isVAR` and `getRef` that it calls, with `final` declarations. We put here their "low-level" code snippets (most likely well-known to experienced Prolog implementors) as they are in the "innermost loop" of the system.

```
int deref(int x) {
  while (isVAR(x)) {
    int r = getRef(x);
    if (r == x) break;
    x = r;
  }
  return x;
}
```

```
boolean isVAR(int x) {return tagOf(x) < 2;}

int getRef(int x) {return heap[detag(x)];}
```

4.1.2 The pre-unification step: detecting matching clauses without copying to the heap

Independently of indexing, one can filter matching clauses by comparing the outermost array of the current goal with the outermost array of a clause head.

Interestingly, as a *prototype of each clause is already placed on the heap at loading and linking time*, one could tentatively unify it with the goal and then undo the bindings. On success, one would then redo the unification while progressively copying to the heap the subterms of the head that need to be newly created.

But even better, we can emulate WAM's registers as a copy of the outermost array of the goal element we are working with, holding dereferenced elements in it.

This "register"-array can be used to reject clauses that mismatch it in positions holding symbols, numbers or references to array-lengths. We can use for this the prototype of a clause

head without starting to place new terms on the heap. At the same time, dereferencing is avoided when working with material from the heap-represented clauses, as our tags will tell us that first occurrences of variables do not need it at all, and that other variable-to-variable-references need it exactly once as a `getRef` step.

4.1.3 Unification

As we have unfolded to registers the outermost array of the current goal (corresponding to a predicate's arguments) we will start by unifying them, when needed, with the corresponding arguments of a matching clause. A dynamically growing and shrinking `int` stack is used to emulate recursion by the otherwise standard unification algorithm. Given that we actually start by unifying the arguments of the outermost terms we split our algorithm in two methods, `unify_args` and `unify` that take turns pushing tasks on the stack and popping them off as they explore the structure of the two Prolog terms.

4.1.4 Trailing

As usual, variables at higher addresses are bound to those at lower addresses on the heap and after binding, variables are trailed when lower than the heap level corresponding to the current goal.

4.2 Fast linear term relocation

While the WAM's instructions (that need as well to be decoded by an interpreter loop in non-native implementations) spend effort on deciding which (new) terms are created on the heap ("write" mode) and which (old) terms are reused from below ("read" mode), we bet instead on a very fast relocation loop that speculatively places the clause head (including its subterms) on the heap. This "single instruction multiple data" operation would likely benefit from parallel execution simply by the presence of multiple arithmetic units in modern CPUs, or even more significantly, via a CUDA or OpenCL GPU implementation, especially if copies are speculatively created in parallel, based on predicted future uses.

As our variable and reference codes that need relocation (`V,U,R`) are `0,1` and `2`, we ensure that the `relocate` method is inlined by the Java compiler by defining it "`static private final`". Similar inlining would occur in today's **C** compilers.

```
int relocate(int b, int cell) {
  return tagOf(cell) < 3 ? cell + b : cell;
}
```

Note that we compute the relocation offset ahead of time, once we know the difference between its source and its target, i.e., when the process for selecting matching clauses starts. To relocate a slice `<from,to>` from our *prototype clause*, placed on the heap ahead of time by the parser, we use another potentially inlineable method, `pushCells`:

```
void pushCells(int b, int from, int to,int base) {
  ensureSize(to - from);
  for (int i = from; i < to; i++) {
    push(relocate(b, heap[base + i]));
  }
}
```

As our heap is a dynamic array, we check ahead of time if it would overflow with `ensureSize` to avoid testing if expansion is needed for each cell.

New terms are built on the heap by the relocation loop in two stages: first the clause head (including its subterms) and then, if unification succeeds, also the body. The method `pushHead` copies and relocates the head of clause at (precomputed) offset `b` from the prototype clause on the heap to the higher area where it is ready for unification with the current goal.

```
int pushHead(int b, Clause C) {
    pushCells(b, 0, C.neck, C.base);
    int head = C.gs[0];
    return relocate(b, head);
}
```

As the code reuse coming from distinguishing between a "read" and a "write" mode during head-unification only increases heap usage by a small percentage (as most clauses are rather body heavy than head-heavy) we trade it for copying the complete head.

On the other hand, we only copy the body once unification succeeds, by calling the method `pushBody` that also relies on the precomputed relocation offset `b`.

```
  int[] pushBody(int b, int head, Clause C) {
    pushCells(b, C.neck, C.len, C.base);
    int l = C.gs.length;
    int[] gs = new int[l];
    gs[0] = head;
    for (int k = 1; k < l; k++) {
      int cell = C.gs[k];
      gs[k] = relocate(b, cell);
    }
    return gs;
  }
```

Note also that we relocate the skeleton `gs` starting with the address of the first goal so it is ready to be merged in the immutable list of goals. At the end, an interesting assertion holds: *the heap is a stream of successive clauses connected among them by variable bindings.* And one can devise a mechanism where, based on profiling, these contiguous heap slices, corresponding to clauses, are created in advance, speculatively, on separate threads.

While we have not implemented it yet, we mention here an *alternative algorithm*, that like the WAM, only creates new terms originating form the head of the clause when needed, while also ensuring that term creation happens all at once, and only if full unification succeeds.

We start with the pre-unification matching, followed on success with full unification, but happening on the *prototype of the clause* located at a lower address on the heap. We take care to trail *every* variable binding. Then we scan the trail and handle the following two situations:

1. if a variable located in the prototype clause area points to a goal already on the heap we collect it for future unbinding, as we want to clear the prototype from all bindings, for reuse
2. if the binding comes from the goals already on the heap, above the prototype clauses area, we copy to the heap the subterm it points to, relocate the variable to point to it, while making sure to collect the variable for later placement on the trail as now it will point upwards.

An interesting aspect of this alternative is that one mimics Prolog's resolution step on the pre-built prototype and it trades some extra work on the trail, in exchange for less work and some space efficiency on the heap.

4.3 Stretching out the Spine: the immutable goal stack

A *Spine* can be seen as a runtime abstraction of a `Clause` instance, collecting the information needed for the execution of the goals originating from it. Implemented as the small methodless `Spine` class, it declares the following fields:

- `int hd`: head of the clause
- `int base`: base of the heap where the clause starts
- `IntList gs`: immutable list of the locations of the goal elements accumulated by unfolding clauses so far
- `int ttop`: top of the trail as it was when this clause got unified
- `int k`: index of the last clause the top goal of the `Spine` has tried to match so far
- `int[] regs`: dereferenced goal registers
- `int[] xs`: index elements based on `regs`

Like the meta-interpreter we have started with, a spine extends the goal stack with the contribution of a given clause's body. Note that (most of the) goal elements on this immutable list are shared among alternative branches. In a way, the illusion that the complete goal stack is local to each `Spine` instance is helpful as it matches the way they look in our simplified meta-interpreter from section 2, but has virtually no space overhead compared to a global procedurally managed goal stack as most of its (immutable) tail is safely shared among Spines.

4.4 The interpreter loop: yielding an answer and ready to resume

Our main interpreter loop starts from a `Spine` and works though a stream of answers, returned to the caller one at a time, until the `spines` stack is empty, when it returns null, signaling that no more answers are available.

```
Spine yield() {
  while (!spines.isEmpty()) {
    Spine G = spines.peek();
    if (hasClauses(G)) {
      if (hasGoals(G)) {
        Spine C = unfold(G);
        if (C != null) {
          if (!hasGoals(C)) return C; // return answer
          else spines.push(C);
        } else popSpine(); // no matches
      } else unwindTrail(G.ttop); // no more goals in G
    } else  popSpine(); // no clauses left
  }
  return null;
}
```

The active component of a `Spine` is the topmost goal in the immutable goal stack `gs` contained in the `Spine`.

When no goals are left to solve, a computed answer is yield, encapsulated in a `Spine` that can be used by the caller to resume execution.

When there are no more matching clauses for a given goal, the topmost `Spine` is popped off. An empty `Spine` stack indicates the end of the execution signaled to the caller by returning `null`.

A key element in the interpreter loop is to ensure that after an `Engine` yields an answer, it can, if asked to, resume execution and work on computing more answers.

To achieve this, the class `Engine` defines in the method `ask()`. A variable "`query`" of type `Spine`, contains the top of the trail as it was before evaluation of the last goal, up to where bindings of the variables will have to be undone, before resuming execution. It also unpacks the actual answer term (by calling the method `exportTerm`) to a tree representation of a term, consisting of recursively embedded arrays hosting as leaves, an external representation of symbols, numbers and variables.

```
Object ask() {
    query = yield();
    if (null == query) return null;
    int res = answer(query.ttop).hd;
    Object R = exportTerm(res);
    unwindTrail(query.ttop);
    return R;
}
```

4.5 Playing with answer streams

We model our answer streams to match Java 8's stream API [9], although as a more expressive alternative (*interactors*), that made it in our Prolog implementations starting with [13], can be considered instead.

A reason for choosing the Java 8 stream API is that it allows elegant embedding in cluster and cloud configurations using high-level functional programming constructs like `map`, `fold` and `filter` as well as automatic parallelization of complex data-flows as provided by frameworks like `Apache Flink`.

To encapsulate our answer streams in a Java 8 `stream`, a special iterator-like interface called `Spliterator` is used [9]. The work is done by the `tryAdvance` method which yields answers while they are not equal to `null`, and terminates the `stream` otherwise.

```
public boolean tryAdvance(Consumer<Object> action) {
  Object R = ask();
  boolean ok = null != R;
  if (ok) action.accept(R);
  return ok;
}
```

Three more methods are required by the interface, mostly to specify when to stop the stream and that the stream is ordered and sequential.

```
public Spliterator<Object> trySplit() {
  return null; // nothing to do here as we do not want to split our answer stream
}

public int characteristics() { // answers are ordered and possibly infinitely many
  return (Spliterator.ORDERED | Spliterator.NONNULL) & ~Spliterator.SIZED;
}

public long estimateSize() { // a way to approximate infinitely many
    return Long.MAX_VALUE;
}
public boolean tryAdvance(Consumer<Object> action) {
    Object R = ask();
    boolean ok = null != R;
```

```
    if (ok) action.accept(R);
    return ok;
}
```

Once the `Spliterator` interface is implemented, the stream of answers encapsulating this engine is created with second argument `false`, specifying that it is not a parallel stream.

```
public Stream<Object> stream() {return StreamSupport.stream(this, false);}
```

5 Multi-argument indexing: a modular add-on

The indexing algorithm is designed as an independent add-on to be plugged into the main Prolog engine. For each argument position in the head of a clause (up to a maximum that can be specified by the programmer) it associates to each indexable element (symbol, number or arity) the set of clauses where the indexable element occurs in that argument position. For deep indexing, the argument position can be generalized to be the integer sequence defining the path leading to the indexable element in a compound term. The clauses having variables in an indexed argument position are also collected in a separate set for each argument position.

Sets of clause numbers associated to each (tagged) indexable element are supported by an `IntMap` implemented as a fast `int`-to-`int` hash table (using linear probing). An `IntMap` is associated to each indexable element by a `HashMap`. The `HashMap`s are placed into an array indexed by the argument position to which they apply. When looking for the clauses matching an element of the list of goals to solve, for an indexing element x occurring in position i, we fetch the set $C_{x,i}$ of clauses associated to it. If V_i denotes the set of clauses having variables in position i, then any of them can also unify with our goal element. Thus, we would need to compute the union of the sets $C_{x,i}$ and V_i for each position i, and then intersect them to obtain the set of matching clauses. We will not actually compute the unions, however. Instead, for each element of the set of clauses corresponding to the "predicate name" (position 0), we retain only those which are either in $C_{x,i}$ or in V_i for each $i > 0$. We do the same for each element for the set V_0 of clauses having variables in predicate positions (if any). Finally, we sort the resulting set of clause numbers and hand it over to the main Prolog engine for unification and possible unfolding in case of success.

Two interesting special cases can benefit from custom variants of the algorithm.

For very small programs or programs having predicates with fewer clauses than the bit size of a long (64), the `IntMap` can be collapse to a `long` made to work as a *bit set*. Alternatively, given our fast pre-unification filtering one can bypass indexing altogether, below a threshold.

For very large programs challenging overall memory capacity, a more compact sparse bit set implementation like [23] or a Bloom filter-based set [2] would replace our `IntMap`-based set, except for the first "predicate name" position, needed to enumerate the potential matches. In this case, the probability of false positives can be fine-tuned as needed, while keeping in mind that false positives will be anyway quickly eliminated by our pre-unification head-matching step. Finally, especially for very large programs, one might want to compute the set of matching clauses lazily, using the Java 8 streams API [9].

Implementation of indexing for large fact databases is a combination of *ordered set intersection* and *lazy execution* as provided by the Java 8 stream API. The intersection of the sets of clauses associated (via `LinkedHashSets`, to preserve order) to each argument position (or more generally a path to a deeper indexable component) is implemented as a stream filtering operation. As sizes of matching sets are known in advance, the initial stream

Table 1 Timings and number of logical inferences (as counted by SWI-Prolog) on 4 small Prolog programs.

System	11 queens	perms of 11 + nrev	sudoku 4x4	metaint perms
our interpreter	5.710s	5.622s	3.500s	16.556s
Lean Prolog	3.991s	5.780s	3.270s	11.559s
Styla	13.164s	14.069s	22.196s	37.800s
SWI-Prolog	1.835s	2.620s	1.336s	4.872s
LIPS	7,278,988	7,128,483	9,261,376	6,651,000

is created from the smallest set of matching clauses, which is then filtered with the others ordered by size. As stream specifications are mainly promises to perform operations when needed, clauses are extracted from the intersection one at a time and then passed to the logic engine for pre-unification and eventual heap construction and full unification. It also makes sense to speed-up these operations by declaring the streams parallel.

6 Some basic performance tests

We prototyped our design as a small, slightly more than 1000 lines of generously commented Java program. However, as a more natural target for a system developed around it would use **C**, we have stayed away from Java's object oriented features by using a large `Engine` class hosting all the data areas and a few small classes like `Clause` and `Spine` that can be easily mapped to **C** `structs`. While implemented as an interpreter, our preliminary tests (Table 1) indicate, somewhat surprisingly, that performance is close, (within a factor of **2**) to our Java-based systems like Jinni and Lean Prolog that use a (fairly optimized) WAM-based instruction set and a factor of **2**-4 from **C**-based SWI-Prolog. While this an order of magnitude slower than today's **C** based Prologs the "apples-to-apples" comparison with our fast WAM-based Prolog implemented in the same language - Java - is a more accurate indication that this lightweight interpreter design is actually quite fast and likely to be within a factor of 2 to 3 of today's optimized WAM-based Prologs if implemented in **C**. The program `11 queens` computes (without printing them out) all the solutions of the 11-queens problem. `Sudoku 4x4` does the same for a reduced Sudoku solver and `perms of 11+nrev` computes the unique permutation that is equal to the reversed sequence of " numbers computed by the naive reverse predicate. The fourth program, `metaint perms` is a variant of the second program, run this time via the two clause meta-interpreter that we have started from, in section 2, to derive our execution algorithm.

For a more conclusive performance comparison, future work is planned on first deriving a WAM-like compiled instruction set from our interpreter. Also, we expect that a **C**-based system, even if kept as an interpreter, is likely to boost performance slightly above slower compiled systems like SWI-Prolog, from which our Java-based interpreter is within a factor of 2-4 on the Horn Clause subset, for small programs.

7 Related work

The closest Prolog implementation is our own Styla system [16], a Scala-based interpreter, itself a derivative of our Java-based Kernel Prolog [15] system. They both use a clause unfolding interpreter along the lines of [12], but contrary to our current design, they rely heavily on high-level features of the implementation language, including an object-oriented

term hierarchy and a unification algorithm distributed over various term sub-types. First-class, "resumable" Prolog engines have been present in our systems since [13] and there's some renewed interest in them (as reflected by a few dozen recent messages in comp.lang.prolog in September 2015) as well as in a similar, thread-based model used in SWI-Prolog's Pengines [7].

Locating function symbols and arity in separate words is different to the typical "symbol+arity" used in most other Prolog systems we are aware of. The translation from HiLog [4] to a first order form, using `apply/N` predicates, is similar to our natural language assembler, where (unnamed) arrays of various lengths are essentially the same thing as HiLog's `apply/N` predicate wrappers. Consequently, work on a first-order semantics for HiLog [4] also covers our natural assembler programs.

There are some clear commonalities with the WAM [1] and more closely with the BinWAM variant of it [17] where transformation to binary clauses implicitly emulates a form of goal stacking. In a way, we also end up emulating something close to the WAM's registers that are saved in our `Spine` stack, itself playing a role similar to the WAM's OR-stack. Placing them on a stack, rather than mapping them to a register vector, is also similar to B-Prolog's stack frame-based representation [24].

The major commonality with the BinWAM [17], not present in the WAM, is that the implicit goal-stacking of the BinWAM is replaced here with an explicit and immutable goal stack seen as present in each `Spine`. As most (except the few topmost) goal elements are shared among `Spines`, the overall time and space costs are comparable with a mutable goal stack managed by saving in our Spines pointers to its top. Another commonality with a BinWAM-optimization (as implemented in BinProlog) is that arguments placed in registers are dereferenced once and then matched against several clause candidates. On the other hand, the WAMs instruction set ensures subterms are only built on the heap as needed, while our interpreter tries instead to eliminate up front the non-matching clauses by borrowing pre-unification information from a prototype clause at a lower heap location before unification is called. On unification success, the complete body of the clause is relocated, and, as this heap-to-heap copy is quite fast, we do not get a significant performance hit as a result. Thus, some of the unexplored implementation choices that materialized through our "from scratch" design steps result in comparable performance, while staying intuitively closer to the view offered by the two clause meta-interpreter, that we have started with in section 2.

Another feature not present in typical WAM-based Prolog systems, is the decoupling of indexing and unification instructions, although one might argue that the same philosophy is motivating the just-in-time indexing schemes of YAP [5] and SWI-Prolog [22] and the user defined indexing of [20]. In fact, when generalized to arbitrary paths, reaching constants occurring deep in a term, our indexing algorithm has more in common with the ones used in theorem proving systems like [10], than with the WAM's tightly interleaved indexing instructions [1]. We believe that besides separating naturally independent concerns, decoupling indexing favors deployment scenarios where the Prolog code is distributed on a cluster or cloud, and fetched as needed. In this case, indexing might need to happen at a different site than the one where the call is made.

8 Conclusions

We hope that by trying to forget as much as we could about the long polished art of Prolog implementation, we have obtained a genuinely more intuitive view of Prolog's execution algorithm. By deriving our Prolog machine as naively as possible, from a two line meta-interpreter, we have captured the necessary step-by-step transformations that one needs to implement in a procedural language that mimics it.

In the process, we have lifted some restrictions on Prolog syntax, like the need for the function or predicate symbol to be a constant, and we have decoupled the indexing algorithm from the main execution mechanism of our Prolog machine. We have also proposed a natural language style, human readable intermediate language that can be loaded directly by the runtime system using a minimalistic tokenizer and parser. The code and the heap representation became one and the same. And the interpreter based on our design was able to get close enough (within a factor of two but often less) to optimized compiled code as shown by our preliminary performance tests. With only slightly more than 1000 lines of Java code, we believe that future ports of this design can help with the embedding of logic programming languages as lightweight software or hardware components.

References

1. H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.
2. Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13:422–426, 1970.
3. Mats Carlson and Per Mildner. SICStus Prolog – The first 25 years. *Theory and Practice of Logic Programming*, 12:35–66, 1 2012. doi:10.1017/S1471068411000482.
4. W. Chen, M. Kifer, and D.S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In E.L. Lusk and R.A. Overbeek, editors, *1st North American Conf. Logic Programming*, pages 1090–1114, Cleveland, OH, 1989. MIT Press.
5. Vitor Santos Costa, Ricardo Rocha, and Luis Damas. The YAP Prolog system. *Theory and Practice of Logic Programming*, 12:5–34, 1 2012. doi:10.1017/S1471068411000512.
6. M. V. Hermenegildo, F. Bueno, M. Carro, P. Lopez-Garcia, E. Mera, J. F. Morales, and G. Puebla. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12:219–252, 1 2012. doi:10.1017/S1471068411000457.
7. Torbjörn Lager and Jan Wielemaker. Pengines: Web logic programming made easy. *TPLP*, 14(4-5):539–552, 2014. doi:10.1017/S1471068414000192.
8. Micha Meier. Compilation of compound terms in Prolog. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 63–79, Cambridge, Massachusetts London, England, 1990. MIT Press.
9. Oracle Corp. Java 8 Streams package. URL: https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.
10. Alexandre Riazanov and Andrei Voronkov. *Efficient Instance Retrieval with Standard and Relational Path Indexing*, pages 380–396. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. doi:10.1007/978-3-540-45085-6_34.
11. Terrance Swift and S. Warren, David. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming*, 12:157–187, 1 2012. doi:10.1017/S1471068411000500.
12. Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48, Berlin Heidelberg, 1999. Springer. ISBN 3-540-65463-1.
13. Paul Tarau. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In John Lloyd, editor, *Computational Logic–CL 2000: First International Conference*, London, UK, July 2000. LNCS 1861, Springer-Verlag.
14. Paul Tarau. Jinni Prolog: a Java-based Prolog compiler and runtime system , May 2012. https://code.google.com/archive/p/jinniprolog/.
15. Paul Tarau. Kernel Prolog: a Java-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy, May 2012. https://code.google.com/archive/p/kernel-prolog/.

16 Paul Tarau. Styla: a Lightweight Scala-based Prolog Interpreter Based on a Pure Object Oriented Term Hierarchy, May 2012. https://code.google.com/archive/p/styla//.

17 Paul Tarau. The BinProlog Experience: Architecture and Implementation Choices for Continuation Passing Prolog and First-Class Logic Engines. *Theory and Practice of Logic Programming*, 12(1-2):97–126, 2012.

18 Paul Tarau and Michel Boyer. Elementary Logic Programs. In P. Deransart and J. Maluszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 159–173, Berlin Heidelberg, August 1990. Springer.

19 Peter Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19(20):385–441, 1994.

20 David Vaz, Vítor Santos Costa, and Michel Ferreira. User defined indexing. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 372–386, Berlin, Heidelberg, 2009. Springer-Verlag. doi:10.1007/978-3-642-02846-5_31.

21 D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

22 Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjorn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12:67–96, 1 2012. doi:10.1017/S1471068411000494.

23 Brett Wooldridge. Sparse Bit Set, 2016. https://github.com/brettwooldridge/SparseBitSet.

24 Neng-Fa Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12:189–218, 1 2012. doi:10.1017/S1471068411000445.

Efficient Declarative Solutions in Picat for Optimal Multi-Agent Pathfinding

Neng-Fa Zhou[1] and Roman Barták[2]

1 **CUNY Brooklyn College and Graduate Center, New York, USA**
nzhou@sci.brooklyn.cuny.edu
2 **Charles University, Praha, Czech Republic**
bartak@ktiml.mff.cuni.cz

Abstract

The multi-agent pathfinding (MAPF) problem has attracted considerable attention because of its relation to practical applications. The majority of solutions for MAPF are algorithmic. Recently, declarative solutions that reduce MAPF to encodings for off-the-shelf solvers have achieved remarkable success. We present a constraint-based declarative model for MAPF, together with its implementation in Picat, which uses SAT and MIP. We consider both the makespan and the sum-of-costs objectives, and propose a preprocessing technique for improving the performance of the model. Experimental results show that the implementation using SAT is highly competitive. We also analyze the high performance of the SAT solution by relating it to the SAT encoding algorithms that are used in the Picat compiler.

1998 ACM Subject Classification I.2.5 Programming Languages and Software (D.3.2)

Keywords and phrases Multi-agent Path Finding, SAT, MIP, Picat

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.11

1 Brief Overview

The multi-agent pathfinding (MAPF) problem amounts to finding a plan for agents to move within a graph from their starting locations to their destinations, such that no agents collide with each other at any time. While MAPF can be solved suboptimally in polynomial time [4], the optimization version with the objective of minimizing the makespan or the sum-of-costs is NP-hard [9, 13]. MAPF has been intensively studied, because the problem occurs in various forms in practical applications, such as robotics and games [1, 7], and the problem also provides a platform for studying search algorithms [6, 8, 12].

Recently, studies of MAPF have proposed using declarative models that rely on off-the-shelf solvers to find solutions. These solvers include CSP (Constraint Satisfaction Problems) [5], SAT (Satisfiability) [10, 11], ASP (Answer Set Programming) [2], and MIP (Mixed Integer Programming) [14]. Declarative models are easy to implement and maintain, can easily be altered for other variants, and are amenable to new domain-specific constraints. SAT-based MAPF solutions are especially promising; they have been shown to be competitive with some well-designed heuristic search algorithms [11].

All of the constraint-based models follow the planning-as-satisfiability approach [3], which finds a sequence of states of a bounded length, where the first state corresponds to the initial state, the last state satisfies the goal condition, and each pair of successive states constitutes a valid action. An efficient declarative solution requires a good model of variables and constraints, a fast solver, and a decent encoding of the model for the solver.

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei; Article No. 11; pp. 11:1–11:2

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We give a constraint-based declarative model for MAPF. The model is very natural; it uses a Boolean variable to indicate whether an agent occupies a vertex of the graph in a state, and uses constraints to ensure the validity of all of the states and state transitions. The basic model minimizes the makespan objective. This model is easily extended to deal with the sum-of-costs objective. We adapt a preprocessing technique for eliminating some of the variables in the model that can never be set to 1. The model is implemented in Picat [15], a general-purpose language that provides several tools for modeling and solving combinatorial problems. Experiments with the SAT and MIP modules show that the SAT solution is more competitive than the MIP solution. A comparison with ASP also reveals the high performance of the SAT solution. We also analyze the performance of the SAT solution by relating it to the encoding algorithms used in the Picat SAT compiler.

References

1 Kurt M. Dresner and Peter Stone. A multiagent approach to autonomous intersection management. *J. Artif. Intell. Res. (JAIR)*, 31:591–656, 2008.

2 Esra Erdem, Doga Gizem Kisa, Umut Öztok, and Peter Schüller. A general formal framework for pathfinding problems with multiple agents. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

3 Henry A. Kautz and Bart Selman. Planning as satisfiability. In *ECAI*, pages 359–363, 1992.

4 Gabriele Röger and Malte Helmert. Non-optimal multi-agent pathfinding is solved (since 1984). In *SOCS*, 2012.

5 Malcolm Ryan. Constraint-based multi-robot path planning. In *IEEE International Conference on Robotics and Automation, ICRA*, pages 922–928, 2010.

6 Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. The increasing cost tree search for optimal multi-agent pathfinding. *Artif. Intell.*, 195:470–495, 2013.

7 David Silver. Cooperative pathfinding. In *Proceedings of the First Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 117–122, 2005.

8 Trevor Scott Standley and Richard E. Korf. Complete algorithms for cooperative pathfinding problems. In *IJCAI*, pages 668–673, 2011.

9 Pavel Surynek. An optimization variant of multi-robot path planning is intractable. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

10 Pavel Surynek. A simple approach to solving cooperative path-finding as propositional satisfiability works well. In *PRICAI*, pages 827–833, 2014.

11 Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski. Efficient SAT approach to multi-agent path finding under the sum of costs objective. In *ECAI*, pages 810–818, 2016.

12 Ko-Hsin Cindy Wang and Adi Botea. Fast and memory-efficient multi-agent pathfinding. In *ICAPS*, pages 380–387, 2008.

13 Jingjin Yu and Steven M. LaValle. Structure and intractability of optimal multi-robot path planning on graphs. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

14 Jingjin Yu and Steven M. LaValle. Optimal multi-robot path planning on graphs: Complete algorithms and effective heuristics. *CoRR*, abs/1507.03290, 2015.

15 Neng-Fa Zhou, Håkan Kjellerstrand, and Jonathan Fruhman. *Constraint Solving and Planning with Picat*. Springer, 2015.

Treewidth in Non-Ground Answer Set Solving and Alliance Problems in Graphs*

Bernhard Bliem

TU Wien, Vienna, Austria
bliem@dbai.tuwien.ac.at

Abstract

To solve hard problems efficiently via answer set programming (ASP), a promising approach is to take advantage of the fact that real-world instances of many hard problems exhibit small treewidth. Algorithms that exploit this have already been proposed – however, they suffer from an enormous overhead. In the thesis, we present improvements in the algorithmic methodology for leveraging bounded treewidth that are especially targeted toward problems involving subset minimization. This can be useful for many problems at the second level of the polynomial hierarchy like solving disjunctive ground ASP. Moreover, we define classes of non-ground ASP programs such that grounding such a program together with input facts does not lead to an excessive increase in treewidth of the resulting ground program when compared to the treewidth of the input. This allows ASP users to take advantage of the fact that state-of-the-art ASP solvers perform better on ground programs of small treewidth. Finally, we resolve several open questions on the complexity of alliance problems in graphs. In particular, we settle the long-standing open questions of the complexity of the Secure Set problem and whether the Defensive Alliance problem is fixed-parameter tractable when parameterized by treewidth.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases answer set programming, treewidth, secure set, defensive alliance, parameterized complexity

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.12

1 Introduction

The problem solving paradigm Answer Set Programming (ASP) [12, 30, 46, 45] has become quite popular for tackling computationally hard problems. It offers its users a very convenient declarative language that allows for succinct specifications, and there are highly efficient systems available [32, 31, 29, 2, 3, 44, 4, 51, 23].

Although ASP systems have made huge advances in performance, they still struggle with several tough problems. This is not always just an issue of computational complexity in the classical sense. Interestingly, ASP systems may perform quite well in practice on one problem whereas the performance on another problem of the same complexity can be significantly worse. Often classical complexity theory is thus only of limited help to explain ASP solving performance in practice. In such cases, it may be insightful to consider the *parameterized* complexity of the problems [20, 28, 17, 48]. This theoretical framework investigates the complexity of a problem not only in terms of the input size, but also of other parameters.

In this work, we are particularly interested in the effect of the structural parameter *treewidth* [50] on the performance of ASP solvers. Intuitively, the smaller the treewidth

* This work was supported by the Austrian Science Fund (FWF) projects P25607 and Y698.

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei; Article No. 12; pp. 12:1–12:12

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of a graph, the closer the graph resembles a tree. It is well-known that many graph problems become easy if we restrict the input to trees and it has turned out that for many important problems this even holds for the more general class of instances of bounded treewidth [5]. Luckily, it has been observed that real-world instances usually exhibit small treewidth [10, 52, 41]. Moreover, treewidth is not only relevant for graph problems. It can also be applied to instances of all kinds of problems by choosing a suitable representation of the instance as a graph. For instance, treewidth has also been considered for constraint satisfaction problems [19], where it is known under the name of "induced width" and is crucial for the performance of a technique called bucket elimination [18].

There have already been some investigations concerning treewidth and ground ASP (i.e., ASP programs without variables, also known as propositional programs) [34, 49, 26]. An important result is the algorithm from [39] for deciding whether a ground ASP program has a solution in linear time on instances of bounded treewidth. This algorithm employs a technique called *dynamic programming on tree decompositions*, which is very common for algorithms that exploit small treewidth. The algorithm from [39] has also been implemented and proposed as an alternative solver for ground ASP [47]. For certain problems, this dynamic-programming-based solver was able to outperform state-of-the-art ASP solvers if the instances had a very small treewidth and the sizes of the instances were very large.

Although the encouraging results from [47] confirmed that small treewidth can be successfully exploited for ASP solving in experimental settings, the restrictions on problems and instances that make this approach perform well were still too severe for most practical applications. The main obstacles that prevented this approach from being useful for a broad range of applications were the facts that, on the one hand, the naive dynamic programming approach involves an enormous overhead (especially in terms of memory) and, on the other hand, state-of-the-art ASP solvers often perform so well that the theoretical superiority of the dynamic programming algorithm only pays off for instances of tremendous size. In fact, experiments in [7] indicated that state-of-the-art ASP solvers are "sensitive" to the treewidth of their input in the sense that smaller treewidth strongly correlates with higher solving performance.

These issues hint at interesting research challenges. In particular, two approaches seem promising for successfully exploiting small treewidth for ASP solving in practice:

- The first research challenge is to improve the dynamic-programming-based methodology in order to avoid some of its overhead and redundant computations.

  For solving ground ASP, these issues are especially severe compared to other problems because the corresponding computational problems are even harder than NP under standard complexity-theoretic assumptions. (In fact, deciding whether a ground ASP program with disjunctions has an answer set is at the second level of the polynomial hierarchy.) This high complexity of ground ASP is mirrored in the dynamic programming algorithm [39], which uses brute force to first of all find all models of all parts of the decomposed program, and it subsequently uses brute force again *for each such partial model* to find all potential counterexamples that may cause the candidate to be discarded.

  This pattern also frequently occurs in dynamic programming algorithms for other problems that search for solutions satisfying some form of subset minimality. Besides ground ASP, this is the case, for instance, for the problem of finding subset-minimal models of a propositional formula. In general, problems involving subset-minimization are quite common in AI, and dynamic programming algorithms have been proposed for, e.g., circumscription, abduction or abstract argumentation (see [38, 35, 21]). Such algorithms typically store a great number of redundant objects because the subsets that may invalidate

a solution candidate are themselves solution candidates. Moreover, the specifications of such algorithms themselves contain redundancies because the potential counterexamples are usually manipulated in almost the same way as the solution candidates.

- The second research challenge is to solve ASP by not doing dynamic programming at all but instead exploiting small treewidth *implicitly* by relying on the assumption that state-of-the-art solvers perform better when given ground programs of small treewidth (as indicated by the experiments in [7]).
  Since problems are usually encoded in non-ground ASP, here the research objective is to investigate which non-ground encoding techniques significantly blow up the treewidth of the grounding when compared to the treewidth of the input facts.

In addition to leveraging treewidth for ASP solving, we are interested in several variants of a graph problem called Secure Set [13]. It belongs to the class of so-called *alliance problems* [42, 24, 53], which are problems that ask for groups of vertices that help each other out in a certain way. Practical applications of alliance problems include finding groups of websites that form communities [27] or distributing resources in a computer network in such a way that simultaneous requests can be satisfied [36]. Intuitively, a set S of vertices in a graph is secure if every subset of S has as least as many neighbors in S as neighbors not in S. The Secure Set problem asks whether a given graph contains a secure set at most of a certain size.

The reason why we are concerned with Secure Set is that this problem has quite interesting properties, especially for ASP researchers: Attempts of encoding this problem in ASP have resulted in very involved specifications indicating that Secure Set may require the full expressive power of ASP [1]. However, it is unfortunately unclear whether this is really necessary because its complexity has still remained unresolved although the problem has been introduced already in 2007 [13].

One of the variants of Secure Set that we consider in the proposed thesis is the Defensive Alliance problem [42, 43]. This problem has received quite some attention in the literature [24]. It is known to be NP-complete, but its complexity when parameterized by treewidth has remained open.

2 Background

We assume some familiarity with ASP; introductions can be found in [12, 30, 46, 45]. In the thesis, we study both ground ASP programs, i.e., programs without variables, but also programs utilizing the full ASP language as described in the ASP-Core-2 specification [14]. In particular, we include weak constraints and aggregates.

We only outline the syntax of a simplified version of non-ground ASP. For details and semantics, we refer to the standard [14]. An ASP program consists of *rules* of the following form:

$$h_1 \mid \ldots \mid h_k :- p_1, \ldots, p_\ell, \text{not } n_1, \ldots, \text{not } n_m.$$

The *head* of a rule r is the set denoted by $H(r) = \{h_1, \ldots, h_k\}$, the *positive body* of r is the set $B^+(r) = \{p_1, \ldots, p_\ell\}$, and the *negative body* of r is the set $B^-(r) = \{n_1, \ldots, n_m\}$. All elements of these sets are called *atoms*. An atom is either a *predicate atom* or an *aggregate atom*. Predicate atoms have the form $p(t_1, \ldots, t_j)$, where p is a *predicate* and $t_1, \ldots, t_j$ are *terms*, that is, constants or variables. A predicate is called *extensional* in a program Π if it only occurs in rule bodies of Π. An atom is called *extensional* in Π if it is a predicate atom over an extensional predicate. We omit a definition of aggregate atoms but only mention

that they allow us to, e.g., compute a sum of integers or count the cardinality of a set. For details, we refer to [14]. In addition to rules as defined above, we allow for weak constraints, which are special kinds of rules and have the following form:

$$:\sim p_1, \ldots, p_\ell, \text{not } n_1, \ldots, \text{not } n_m.\ [w, t_1, \ldots, t_q]$$

The intuition is that if an answer set violates a ground instantiation of such a weak constraint, then this incurs a penalty of w to the cost of the solution. (Without the terms $t_1, \ldots, t_k$, each weight w would in fact only be counted once in the cost of a solution, hence $t_1, \ldots, t_k$ can be specified for counting the same weight multiple times.) Again, we omit details and refer to [14] instead.

To solve a non-ground ASP program, ASP systems usually first invoke a *grounder* that transforms a program into a set of ground rules. The answer sets of the original program are the *stable models* (as defined in [33]) of the resulting ground program.

A "naive" grounder blindly instantiates variables by all possible ground terms. Grounders in practice, on the other hand, employ sophisticated techniques in order to keep the resulting ground program as small as possible. As these techniques differ between systems, we define a simplified notion of grounding that is easier to study. For a meaningful investigation of the relationship between the treewidth of the input and the treewidth of the grounding, we need to assume that the grounder still performs some basic simplifications. These simplifications are so basic that they can be assumed to be implemented by all reasonable grounders. The intuition is that a rule from the "naive" grounding is omitted in our grounding whenever its positive body contains an atom that cannot possibly be derived.

▶ **Definition 1.** Let Π be a non-ground ASP program, let Π^+ denote the positive program obtained from Π by removing all negated atoms and replacing disjunctions with conjunctions (i.e., splitting disjunctive into normal rules), and let M^+ be the unique minimal model of Π^+. The *grounding* of Π, denoted by $\mathrm{gr}(\Pi)$, is such that, for every substitution s from variables to constants, $s(r) \in \mathrm{gr}(\Pi)$ iff $s(B^+(r)) \subseteq M^+$.

In our work, we are interested in the treewidth of ground ASP programs. For this, we represent programs as graphs as follows.

▶ **Definition 2.** The *primal graph* of a ground ASP program Π is an undirected graph whose vertices are the atoms in Π and there is an edge between two atoms if they appear together in a rule in Π. The *treewidth* of a ground ASP program Π is the treewidth of its primal graph.

Deciding whether a disjunctive ground ASP program has a stable model is Σ_2^{P}-complete in general [22], and it can be done in linear time for ground programs of bounded treewidth [34]. Treewidth is an important parameter studied in the context of *parameterized complexity theory*. Here, decision problems consist not only of an instance and a yes-no question, but additionally of a parameter of the instance. For introductions, we refer to [20, 28, 17, 48]. The central notion of tractability is called *fixed-parameter tractability*.

▶ **Definition 3.** A problem is *fixed-parameter tractable* (FPT) w.r.t. a parameter k of the instances if it admits an algorithm that runs in time $\mathcal{O}(f(k) \cdot n^c)$, where f is an arbitrary computable function that only depends on k, n is the input size and c is an arbitrary constant. We call such an algorithm an *FPT algorithm*.

Note that the factor $f(k)$ in this running time may be exponential in the parameter k, but if k is bounded by a constant, then the algorithm runs in polynomial time. Importantly, the degree c of the polynomial must be a constant and may not depend on the parameter, otherwise the algorithm is not considered FPT.

Dynamic programming on tree decompositions is perhaps the most common technique for obtaining FPT algorithms when the parameter is treewidth. It is employed in the algorithm for solving ground ASP in [39], for instance. The basic idea is the following: Given a graph G, a tree decomposition of G is a tree whose nodes correspond to subgraphs of G according to certain conditions. If the treewidth of a graph is bounded by a constant, then we can find (in linear time) a tree decomposition whose nodes correspond to subgraphs of constant size [11]. We can then solve many problems by first applying brute force at each subgraph in order to solve a subproblem corresponding to this subgraph and then trying to combine the obtained partial solutions. Due to the bound on the treewidth, we can afford this brute force approach because each of the considered subgraphs has bounded size. Formal definitions and examples of this technique can be found in, e.g., [48].

We now define secure sets in graphs [13]. For this, we use the notation $N_G[S]$ to denote the closed neighborhood of a subset S of the vertices of a graph G; that is, $N_G[S]$ contains the vertices in S itself and the vertices that are adjacent to an element of S.

▶ **Definition 4.** Let G be a graph and S be a subset of its vertices. We call S *secure in* G if $|N_G[X] \cap S| \geqslant |N_G[X] \setminus S|$ holds for every $X \subseteq S$.

Intuitively, we can regard a neighbor of X as a "good" neighbor if it is also in S, and as a "bad" neighbor otherwise. Now X is a counterexample to S being secure if X has more bad neighbors than good ones.

3 Contributions

Our contributions can be arranged in three groups: First, we present improvements in the dynamic programming methodology; second, we define non-ground ASP classes that can be shown to preserve bounded treewidth of the input in grounding; third, we provide complexity results and algorithms for alliance problems in graphs.

3.1 Improvements in the Dynamic Programming Methodology

We present an improved dynamic programming methodology for problems that involve subset minimization. Specifically, for any problem P whose solutions are exactly the subset-minimal solutions of some base problem B, we formalize how a dynamic programming algorithm for B can automatically be transformed into a dynamic programming algorithm for P. We prove that the resulting algorithm runs in linear time on instances of bounded treewidth if the base algorithm does. Moreover, we prove that the resulting algorithm is correct if the base algorithm is correct and, intuitively, it only computes partial solutions that do not "revoke decisions" made by associated partial solutions further down in the tree decomposition. The resulting algorithm has two advantages compared to solving P directly in a naive way: first, it is usually easier to specify because we only need to design an algorithm for the base problem and need not care about subset minimization; second, it is potentially more efficient because it stores fewer redundant items.

Indeed, this methodology has been empirically shown to lead to significant performance benefits for several problems [6]. An improved version of the classical dynamic programming algorithm for ground ASP has been implemented using these ideas [25] and proved to be significantly more efficient than the algorithm from [39]. Our result formalizes the common scheme that underlies these algorithms. We thus provide a formal framework that makes it possible to transfer the mentioned optimizations easily to other problems. Thereby we make the impressive performance benefits that have been reported in [6, 25] accessible to

Listing 1 A guarded ASP encoding for checking whether a given set S (declared using predicate `s`) is secure in a given graph (declared using predicates `v` and `e`). The guards of rules are underlined.

```
% Guess a subset X of S.
x(S) | nx(S) :- s(S).
% Neighbors of X are "good" if they are in S, otherwise they are "bad".
neighbor(V) :- x(X), e(X,V).
neighbor(X) :- x(X), v(X).
good(V)     :- neighbor(V), s(V).
bad(V)      :- neighbor(V), v(V), not s(V).
% If X has more bad neighbors than good ones, S is not secure.
% We use the following weak constraints to determine this by summing up.
:~ v(V), good(V). [1,V]  % Add 1 for each good neighbor.
:~ v(V), bad(V). [-1,V]  % Subtract 1 for each bad neighbor.
```

algorithm designers working on related problems. This is primarily useful for problems on the second level of the polynomial hierarchy as subset minimization is a recurring theme in many such problems.

3.2 Non-Ground ASP Classes that Preserve Bounded Treewidth

We define non-ground ASP classes for which grounding, according to Definition 1, preserves bounded treewidth of the input. By restricting the syntax of non-ground ASP, we define two classes of programs called *guarded* and *connection-guarded* programs [7]. Guarded programs guarantee that the treewidth of any fixed program after grounding stays small whenever the treewidth of the input facts is small. We formally prove this property and show that, despite their restrictions, guarded programs can still express problems that are complete for the second level of the polynomial hierarchy.

Connection-guarded programs are even more expressive than guarded programs. We show that the treewidth of any fixed connection-guarded program after grounding is small whenever the treewidth *and the maximum degree* of (a graph representation of) the input facts is small.

These results bring us closer to the goal of implicitly taking advantage of the apparent sensitivity to treewidth exhibited by modern ASP solvers because they give us insight into what happens to the treewidth of the input during grounding. Thus, by writing a program in guarded ASP, we can be sure that the grounder does not destroy the property of bounded treewidth. In the case of connection-guarded ASP, the same holds for the combination of treewidth and maximum degree.

▶ **Example 5.** The ASP encoding in Listing 1 can be used for deciding whether a given set S of vertices is secure in a given graph G. It guesses a subset X of S and uses weak constraints in such a way that the cost of each answer set is exactly $|N_G[X] \cap S| - |N_G[X] \setminus S|$. If there is a subset of S that has more "bad" neighbors than "good" ones, then the program has an answer set with negative cost. The solutions of a program with weak constraints are those answer sets that minimize the cost incurred by violated weak constraints. We can thus decide whether S is secure by checking if this minimum value is negative.

The program in Listing 1 is guarded, which means that, for each rule r, all variables of r occur together in a single extensional atom of $B^+(r)$ that we call the *guard* of r. Note that, alternatively, it is also possible to check whether a set of vertices is secure without using weak constraints. For instance, we can replace the weak constraints by the "hard" constraint

`:- #sum{ 1,G : good(G); -1,B : bad(B) } >= 0.` The resulting program has an answer set if and only if S is not secure. However, the new constraint is not guarded. This means that the original program in Listing 1 generally leads to groundings of much lower treewidth and can thus be expected to perform better. Indeed, the ground instantiation of the new hard constraint would contain a linear number of atoms. Thus, the primal graph of the grounding would contain a clique of linear size and thus have linear treewidth even if the treewidth of input graphs is bounded by a constant.

In the thesis, we also present a complexity analysis of computational problems corresponding to these classes when the parameter is the treewidth of the input, the maximum degree of the input, or the combination of both. The results of this analysis show that, for any fixed guarded ASP program, answer set solving is FPT when parameterized by the treewidth of the input; moreover, for any fixed connection-guarded ASP program, answer set solving is FPT when parameterized by the combination of treewidth and maximum degree. This is not obvious because our ASP classes support weak constraints and aggregates, which are not accounted for in the FPT algorithms [39, 25] for ground ASP. Furthermore, we prove hardness results showing that for connection-guarded ASP programs *both* the treewidth and the maximum degree must be bounded for obtaining fixed-parameter tractability. We do this by presenting a connection-guarded ASP encoding of a problem that is NP-hard even if the treewidth of the instances is fixed and by presenting a guarded encoding of a problem that is Σ_2^P-hard even if the degree of the instances is fixed.

As a side-product of these investigations, we obtain *metatheorems* for proving FPT results. That is, our results on guarded ASP allow us to prove that a problem is FPT when parameterized by treewidth by simply expressing the problem in guarded ASP. We compare this metatheorem to the common approach of proving fixed-parameter tractability by expressing a problem in monadic second-order logic and invoking the well-known theorem by Courcelle [15, 16]. Similarly, we can prove that a problem is FPT when parameterized by the combination of treewidth and maximum degree by expressing the problem in connection-guarded ASP. This result is appealing because we are not aware of any metatheorems that allow us to obtain FPT results for the combination of treewidth and degree as the parameter.

3.3 Alliance Problems in Graphs

We perform a complexity analysis of alliance problems in graphs, both in the classical setting and when parameterized by treewidth. First, we settle the complexity of the Secure Set problem by proving that the problem, along with several variants, is Σ_2^P-complete (that is, at the second level of the polynomial hierarchy).

Next we turn to the complexity of Secure Set and Defensive Alliance when the problems are parameterized by treewidth. We illustrate the use of our ASP classes as FPT classification tools by presenting simple encodings for alliance problems in graphs. By encoding the NP-complete Defensive Alliance problem in connection-guarded ASP, we easily obtain the already known result that the problem is FPT when parameterized by the combination of treewidth and maximum degree. More importantly, we obtain the new result that the co-NP-complete problem of deciding whether a given set is secure in a graph is FPT for the parameter treewidth by encoding the problem in guarded ASP.

We also give several negative results. We prove that both Defensive Alliance and Secure Set, as well as several problem variants, are not FPT when parameterized by treewidth (under commonly held complexity-theoretic assumptions). These questions have been open since the problems have been introduced in 2002 and 2007, respectively. They have explicitly been stated as open problems in [40] (for Defensive Alliance) and in [37] (for Secure Set).

Despite the parameterized hardness of Secure Set, we can give at least a slightly positive result: We show that the Secure Set problem can still be solved in polynomial time for instances of bounded treewidth although the degree of the polynomial depends on the treewidth.

4 Current Status

The largest part of the research for the proposed thesis has already been done and is in the process of being integrated and written down. Most of the results have been published in conference proceedings and journals:

- The work on improving the dynamic programming methodology for problems involving subset minimization has been published in [6].
- The class of connection-guarded ASP programs, which preserves bounded treewidth of the input in grounding whenever the maximum degree is also bounded has been published in [7]. That paper neither contained the thorough complexity analysis performed in the proposed thesis nor the work on the class of guarded programs, which may be attractive because this class does not require the degree of input graphs to be bounded.
- The Σ_2^P-completeness result of the Secure Set problem has been published in [9]. An extended version [8], which is currently under review for a journal, additionally contains the parameterized complexity results. The proposed thesis extends this by results on the parameterized complexity of the Defensive Alliance problem as well.

5 Open Issues

The proposed thesis opens up several possibilities for future research:

- The class of connection-guarded ASP programs may be of interest for algorithmic purposes because it allows us to classify a problem as FPT when parameterized by treewidth plus degree. A common technique for classifying problems parameterized by treewidth as FPT is expressing them in monadic second-order logic (MSO). Our result may lead to an extension of MSO that can be used for classifying problems as FPT when the parameter is treewidth + degree.
- From a more practical perspective, it is promising to look closely into what ASP solvers and in particular their heuristics are doing when they are presented with a grounding of small treewidth. This could provide us insight into why state-of-the-art ASP solvers perform better on instances of small treewidth even though they do not "consciously" exploit this fact. With the gained understanding, we may be able to improve their performance by explicitly taking information from a tree decomposition into account during solving. This could perhaps lead to a hybrid ASP solving approach that uses classical conflict-driven clause learning in combination with techniques based on tree decompositions.
- We showed that Secure Set is not FPT when parameterized by treewidth (unless the class W[1] is equal to FPT). It would be interesting to study which additional restrictions beside bounded treewidth need to be imposed on Secure Set instances to achieve fixed-parameter tractability. In particular, we do not know whether it becomes FPT when additionally the degree is bounded.
- Regarding our polynomial-time algorithm for Secure Set on instances of bounded treewidth, it would be interesting to study if this result can be extended to instances of bounded clique-width, a parameter related to treewidth.

- Moreover, we have not considered a problem that is closely related to Defensive Alliance, namely Offensive Alliance. Possibly some of our techniques can also be applied to obtain complexity results for this problem.
- Defensive Alliance differs from Secure Set in the size of the subsets of solution candidates that need to be checked. For future work it would be interesting to study the complexity of a problem that generalizes both of them, where the size of the subsets is a parameter.
- Finally, for the parameterized hardness results that we obtained we do not have corresponding membership results. This is an obvious task for future work.

Acknowledgements. The proposed thesis was supervised by Stefan Woltran and is based on publications with significant contributions by Günther Charwat, Markus Hecher, Marius Moldovan and Michael Morak.

References

1 Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, and Stefan Woltran. Computing secure sets in graphs using answer set programming. *J. Logic Comput.*, 2015. Accepted for publication. doi:10.1093/logcom/exv060.

2 Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In Pedro Cabalar and Tran Cao Son, editors, *Proceedings of LPNMR 2013*, volume 8148 of *LNCS*, pages 54–66. Springer, 2013. doi:10.1007/978-3-642-40564-8_6.

3 Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Francesco Calimeri, Giovambattista Ianni, and Mirosław Truszczyński, editors, *Proceedings of LPNMR 2015*, volume 9345 of *LNCS*, pages 40–54. Springer, 2015. doi:10.1007/978-3-319-23264-5_5.

4 Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio Terracina. The disjunctive datalog system DLV. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers, editors, *Revised Selected Papers of Datalog 2010*, volume 6702 of *LNCS*, pages 282–301. Springer, 2011. doi:10.1007/978-3-642-24206-9_17.

5 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *Journal of Algorithms*, 12(2):308–340, 1991. doi:10.1016/0196-6774(91)90006-K.

6 Bernhard Bliem, Günther Charwat, Markus Hecher, and Stefan Woltran. D-FLATˆ2: Subset minimization in dynamic programming on tree decompositions made easy. *Fund. Inform.*, 147(1):27–61, 2016. doi:10.3233/FI-2016-1397.

7 Bernhard Bliem, Marius Moldovan, Michael Morak, and Stefan Woltran. The impact of treewidth on ASP grounding and solving. In Carles Sierra and Fahiem Bacchus, editors, *Proceedings of IJCAI 2017*. The AAAI Press, 2017. Accepted for publication.

8 Bernhard Bliem and Stefan Woltran. Complexity of secure sets. *CoRR*, abs/1411.6549, 2014. Updated to version 3 on July 11, 2017. URL: http://arxiv.org/abs/1411.6549.

9 Bernhard Bliem and Stefan Woltran. Complexity of secure sets. In Ernst W. Mayr, editor, *Revised Papers of WG 2015*, volume 9224 of *LNCS*, pages 64–77. Springer, 2016. doi:10.1007/978-3-662-53174-7_5.

10 Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernet.*, 11(1-2):1–21, 1993.

11 Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996. doi:10.1137/S0097539793251219.

12 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

13 Robert C. Brigham, Ronald D. Dutton, and Stephen T. Hedetniemi. Security in graphs. *Discrete Appl. Math.*, 155(13):1708–1714, 2007. doi:10.1016/j.dam.2007.03.009.

14 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. https://www.mat.unical.it/aspcomp2013/ASPStandardization, 2015. Version: 2.03c.

15 Bruno Courcelle. The monadic second-order logic of graphs I: Recognizable sets of finite graphs. *Inform. and Comput.*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

16 Bruno Courcelle. The monadic second-order logic of graphs III: Tree-decompositions, minors and complexity issues. *RAIRO Theor. Inform. Appl.*, 26:257–286, 1992. doi:10.1051/ita/1992260302571.

17 Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer International Publishing, Cham, Switzerland, 2015. doi:10.1007/978-3-319-21275-3.

18 Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence*, 113(1-2):41–85, 1999. doi:10.1016/S0004-3702(99)00059-4.

19 Rina Dechter. *Constraint Processing*. Elsevier Morgan Kaufmann, Amsterdam, The Netherlands, 2003. doi:10.1016/b978-1-55860-890-0.x5000-2.

20 Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, NY, USA, 1999. doi:10.1007/978-1-4612-0515-9.

21 Wolfgang Dvořák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artificial Intelligence*, 186:1–37, 2012. doi:10.1016/j.artint.2012.03.005.

22 Thomas Eiter and Georg Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3-4):289–323, 1995. doi:10.1007/BF01536399.

23 Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels^{A} - A system for computing answer sets of logic programs with aggregates. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proceedings of LPNMR 2005*, volume 3662 of *LNCS*, pages 427–431. Springer, 2005. doi:10.1007/11546207_40.

24 Henning Fernau and Juan A. Rodríguez-Velázquez. A survey on alliances and related parameters in graphs. *Electron. J. Graph Theory Appl. (EJGTA)*, 2(1):70–86, 2014. doi:10.5614/ejgta.2014.2.1.7.

25 Johannes Klaus Fichte, Markus Hecher, Michael Morak, and Stefan Woltran. Answer set solving with bounded treewidth revisited. In Marcello Balduccini and Tomi Janhunen, editors, *Proceedings of LPNMR 2017*, volume 10377 of *LNCS*, pages 132–145. Springer, 2017. doi:10.1007/978-3-319-61660-5_13.

26 Johannes Klaus Fichte and Stefan Szeider. Backdoors to tractable answer set programming. *Artificial Intelligence*, 220:64–103, 2015. doi:10.1016/j.artint.2014.12.001.

27 Gary William Flake, Steve Lawrence, C. Lee Giles, and Frans Coetzee. Self-organization and identification of web communities. *IEEE Computer*, 35(3):66–71, 2002. doi:10.1109/2.989932.

28 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. Springer, 2006. doi:10.1007/3-540-29953-X.

29 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In Francesco Calimeri, Giovambattista Ianni, and

Mirosław Truszczyński, editors, *Proceedings of LPNMR 2015*, volume 9345 of *LNCS*, pages 368–383. Springer, 2015. doi:10.1007/978-3-319-23264-5_31.

30 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, Williston, VT, USA, 2012. doi:10.2200/S00457ED1V01Y201211AIM019.

31 Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp*: A conflict-driven answer set solver. In Chitta Baral, Gerhard Brewka, and John S. Schlipf, editors, *Proceedings of LPNMR 2007*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007. doi:10.1007/978-3-540-72200-7_23.

32 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012. doi:10.1016/j.artint.2012.04.001.

33 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of JICSLP 1988*, volume 2, pages 1070–1080. The MIT Press, 1988.

34 Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, 174(1):105–132, 2010. doi:10.1016/j.artint.2009.10.003.

35 Georg Gottlob, Reinhard Pichler, and Fang Wei. Tractable database design and datalog abduction through bounded treewidth. *Inf. Syst.*, 35(3):278–298, 2010. doi:10.1016/j.is.2009.09.003.

36 Teresa W. Haynes, Stephen T. Hedetniemi, and Michael A. Henning. Global defensive alliances in graphs. *Electron. J. Combin.*, 10, 2003. URL: http://www.combinatorics.org/Volume_10/Abstracts/v10i1r47.html.

37 Yiu Yu Ho and Ronald D. Dutton. Rooted secure sets of trees. *AKCE Int. J. Graphs Comb.*, 6(3):373–392, 2009.

38 Michael Jakl, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. Fast counting with bounded treewidth. In Iliano Cervesato, Helmut Veith, and Andrei Voronkov, editors, *Proceedings of LPAR 2008*, volume 5330 of *LNCS*, pages 436–450. Springer, 2008. doi:10.1007/978-3-540-89439-1_31.

39 Michael Jakl, Reinhard Pichler, and Stefan Woltran. Answer-set programming with bounded treewidth. In Craig Boutilier, editor, *Proceedings of IJCAI 2009*, pages 816–822. The AAAI Press, 2009.

40 Masashi Kiyomi and Yota Otachi. Alliances in graphs of bounded clique-width. *Discrete Appl. Math.*, 223:91–97, 2017. doi:10.1016/j.dam.2017.02.004.

41 András Kornai and Zsolt Tuza. Narrowness, pathwidth, and their application in natural language processing. *Discrete Appl. Math.*, 36(1):87–92, 1992. doi:10.1016/0166-218X(92)90208-R.

42 Petter Kristiansen, Sandra M. Hedetniemi, and Stephen T. Hedetniemi. Introduction to alliances in graphs. In Ilyas Cicekli, Nihan Kesim Cicekli, and Erol Gelenbe, editors, *Proceedings of ISCIS 2002*, pages 308–312. CRC Press, 2002.

43 Petter Kristiansen, Sandra M. Hedetniemi, and Stephen T. Hedetniemi. Alliances in graphs. *J. Combin. Math. Combin. Comput.*, 48:157–178, 2004.

44 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006. doi:10.1145/1149114.1149117.

45 Vladimir Lifschitz. What is answer set programming? In Dieter Fox and Carla P. Gomes, editors, *Proceedings of AAAI 2008)*, pages 1594–1597. The AAAI Press, 2008.

46 Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, pages 375–398. Springer, New York, NY, USA, 2011. doi:10.1007/978-3-642-60085-2.

47 Michael Morak, Reinhard Pichler, Stefan Rümmele, and Stefan Woltran. A dynamic-programming based ASP-solver. In Tomi Janhunen and Ilkka Niemelä, editors, *Proceedings of JELIA 2010*, volume 6341 of *LNCS*, pages 369–372. Springer, 2010. doi:10.1007/978-3-642-15675-5_34.

48 Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, Oxford, United Kingdom, 2006. doi:10.1093/acprof:oso/9780198566076.001.0001.

49 Reinhard Pichler, Stefan Rümmele, Stefan Szeider, and Stefan Woltran. Tractable answer-set programming with weight constraints: Bounded treewidth is not enough. *Theory Pract. Log. Program.*, 14(2):141–164, 2014. doi:10.1017/S1471068412000099.

50 Neil Robertson and Paul D. Seymour. Graph minors. III. Planar tree-width. *J. Combin. Theory Ser. B*, 36(1):49–64, 1984. doi:10.1016/0095-8956(84)90013-3.

51 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. doi:10.1016/S0004-3702(02)00187-X.

52 Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Inform. and Comput.*, 142(2):159–181, 1998. doi:10.1006/inco.1997.2697.

53 Ismael González Yero and Juan A. Rodríguez-Velázquez. Defensive alliances in graphs: A survey. *CoRR*, abs/1308.2096, 2013. URL: http://arxiv.org/abs/1308.2096.

Achieving High Quality Knowledge Acquisition using Controlled Natural Language

Tiantian Gao*

Department of Computer Science, Stony Brook University, Stony Brook, NY, USA
tiagao@cs.stonybrook.edu

Abstract

Controlled Natural Languages (CNLs) are efficient languages for knowledge acquisition and reasoning. They are designed as a subset of natural languages with restricted grammar while being highly expressive. CNLs are designed to be automatically translated into logical representations, which can be fed into rule engines for query and reasoning. In this work, we build a knowledge acquisition machine, called KAM, that extends Attempto Controlled English (ACE) and achieves three goals. First, KAM can identify CNL sentences that correspond to the same logical representation but expressed in various syntactical forms. Second, KAM provides a graphical user interface (GUI) that allows users to disambiguate the knowledge acquired from text and incorporates user feedback to improve knowledge acquisition quality. Third, KAM uses a paraconsistent logical framework to encode CNL sentences in order to achieve reasoning in the presence of inconsistent knowledge.

1998 ACM Subject Classification I.2.1 Applications and Expert Systems

Keywords and phrases Logic Programming, Controlled Natural Languages, Knowledge Acquisition

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.13

1 Introduction

Much of human knowledge can be represented as rules and facts, which can be used by rule engines (e.g., XSB [22], Clingo [9], IDP [5]) to conduct formal logical reasoning in order to derive new conclusions, answer questions, or explain the validity of true statements. However, rules and facts extracted from human knowledge can be very complex in the real world. This will demand domain experts to spend a lot of time on understanding the rule systems in order to write logical rules. CNLs emerge as better knowledge acquisition systems over rule systems in that they can acquire knowledge from text and represent the text in logical forms for reasoning. CNLs are designed based on natural languages, but with restricted grammar to avoid ambiguities while being highly expressive. Representative languages include ACE [7], Processable English (PENG) [24], BioQuery-CNL [6]. In general, CNL systems provide a GUI for user to enter CNL text. The language parser checks the grammar of the text and sends back suggestions for correction to the user. CNL text is then mapped into the corresponding logic programs based on the syntax and semantics of the underlying rule engine in order to perform question answering tasks.

Though the aforementioned systems have good intent of design, we found that there are several limitations in current CNL systems. First, they have limited ability to identify sentences that express the same meaning but in various syntactical forms. For instance,

* The author is co-advised by Michael Kifer and Paul Fodor from Stony Brook University.

Technical Communications of the 33rd International Conference on Logic Programming (ICLP 2017).
Editors: Ricardo Rocha, Tran Cao Son, Christopher Mears, and Neda Saeedloei; Article No. 13; pp. 13:1–13:10

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

ACE translates sentences `Mary owns a car` and `Mary is the owner of a car` into two different logical representations. As a result, if the first sentence is entered into the knowledge base, the reasoner will fail to answer the question `who is the owner of a car`. However, in the real world, it is very common that the user writes questions in a different way from the author who composes the knowledge base. Second, current CNL systems do not accept inconsistent knowledge to occur. In other words, once inconsistent information is found, the underlying rule engine will break and not be able to conduct any inference tasks. In our view, inconsistency is very likely to happen when the knowledge base is formed by merging multiple resources together. Hence, it is useful to design a paraconsistent logical framework that can reason in the presence of inconsistent knowledge. Third, there is no way for the user to edit or audit the acquired knowledge. CNL systems are not guaranteed to always return the user-expected results. As a result, it is necessary to provide a mechanism for the user to edit the acquired knowledge as opposed to re-write sentences many times in order to meet the requirement.

In this work, we design a knowledge acquisition system, KAM, that achieves three goals. First, KAM performs deep semantic analysis of English sentences and maps sentences that express the same meaning via different syntactic forms to the same standard logical representations. Second, KAM performs valid logical inferences based on the facts and rules extracted from English sentences and achieves inconsistency-tolerance for query answering. Third, KAM builds an environment to assist users with entering and disambiguating English texts. In the following parts, Section 2 shows the background knowledge of the natural language processing tools KAM uses in the knowledge acquisition process, Section 3 describes the architecture of the system, Section 4 shows the current state of research and discusses some open issues, and Section 5 concludes the paper.

2 Background

In this section, we provide the background of linguistic databases, semantic relation extraction, and word similarity measures in the field of natural language processing in order to help readers understand KAM better.

2.1 Linguistic Databases

KAM uses a lexical database, BabelNet, and a frame-relation database, FrameNet, in the process of knowledge acquisition. A lexical database is a database of words. It contains the information of part-of-speech, word sense, *synset* and semantic relations of words. WordNet [19] is one of the famous linguistic databases, where each word is defined with a list of word senses. Words that share similar meanings are grouped as a synset. Synsets are connected by semantic relations. For instance, the *hypernym* relation says that one synset is a more general concept of the other, i.e., `human` is the hypernym of `homo sapiens`. WordNet is rich in word knowledge, but it does not have enough information about named entities we encounter in every life or some specialized fields. DBPedia [1], WikiData [27], and YAGO [25] are databases of entities, where each is defined with a set of properties and the relations with other entities or with some pre-defined ontological classes. However, there is no link between an entity in an entity database like DBPedia and a concept in WordNet. As a result, there is no way to find the semantic relation between a name entity and a concept in WordNet, which is useful in many cases. BabelNet solves this problem by integrating multiple knowledge bases, including WordNet, DBPedia, Wikidata, etc. Besides, it automatically finds the mapping across different knowledge bases and therefore bridges the gap between concepts and entities.

FrameNet is a database representing entity relations using *frames*. A frame consists of a set of *frame elements* and *lexical units*. A frame element denotes an entity that serves a particular semantic role in a frame relation. Frame elements are frame-specific. Therefore, they are not shared among frames. A lexical unit indicates a target word in a sentence that triggers a frame relation. For example, the sentence `Mary works for IBM as an engineer` semantically entails the `Being_Employed` frame relation, where `work` is the lexical unit and `Mary`, `IBM`, and `engineer` represent the `Employee`, `Employer`, and `Position` frame elements respectively. In FrameNet, each lexical unit is associated with a set of *valence patterns* and *exemplar sentences*. Valence patterns show the *grammatical functions* [13] of each frame element with respect to the lexical unit. For the above sentence, `Mary` is the `external` of `work`, and `IBM` and `engineer` are the `dependent` of the prepositional modifiers of `work`. Exemplar sentences are the sample English sentences that realize the valence patterns.

In addition to FrameNet, VerbNet [23] and PropBank [14] are also databases of entity relations. VerbNet and PropBank are purely verb-oriented. Therefore, they cannot recognize noun-, adjective-, or adverb-triggered relations. Besides, since VerbNet and PropBank group verbs based on the syntactic patterns of verbs with respect to the entities, verbs that belong to the same class may not represent the same meaning. The advantage of VerbNet over FrameNet is that VerbNet assigns a WordNet synset ID to each verb. Additionally, it defines an ontology that defines the semantic restrictions for entities that can serve particular semantic roles in an entity relation. In KAM, we use FrameNet augmented with BabelNet synset IDs for each frame element and lexical unit.

2.2 Semantic Relation Extraction

Semantic relation extraction tools analyze the semantics of English sentences and extract their entailed relations. Representative tools include Ollie [18], Stanford Relation Extractor [26], LCC [16], SEMAFOR [4], and LTH [12]. Ollie is a relation extractor that extracts triples representing binary relations based on open domains. Stanford Relation Extractor and LCC, on the other hand, can only extract from a fixed set of relations. Although Ollie is flexible at extracting relations, it cannot standardize triples that represent the same semantic relation. Stanford Relation Extractor and LCC are better at relation standardization, but can work with a limited number of relations.

Compared with the aforementioned tools, SEMAFOR and LTH are FrameNet-based semantic parsers that aim to identify a large number of relations and achieve standardization. Basically, they use machine learning algorithms to train the model based on the exemplar sentences in FrameNet. Based our empirical study, SEMAFOR and LTH do not perform well enough for knowledge acquisition. Recall the sentence `Mary works for IBM as an engineer` from the previous section. SEMAFOR extracts two frames: one is `usefulness` frame triggered by `work`, where `Mary` and `for IBM` represent the `entity` and `purpose` frame elements respectively; the other one is `People_by_vocation` frame triggered by `engineer`, with no frame elements attached. The first one is wrong because `for` in this context does not express the purpose meaning. Although the second frame is correct, it does not find who holds this vocation.

In our analysis of FrameNet 1.6 data, 70.2% valence patterns have only one exemplar sentence and 12.8% valence patterns have two exemplar sentences. However, there are also valence patterns with more than 100 exemplar sentences. An uneven distribution of the exemplar sentences per valence pattern will result in an imprecise estimation of model parameters. In addition, frame elements do not have semantic restrictions, which are useful in practical cases. For instance, comparing sentences `Mary has a full-time job` and `Mary`

`has a well-paid job`, both `full-time` and `well-paid` are adjective modifiers of `job`. But, they are classified as two different frame elements: `Contract-basis` and `Compensation` respectively. Without any semantic constraints, we cannot distinguish these two frame elements based on their syntactical context.

2.3 Semantic Similarity

Semantic similarity measures the semantic closeness between a pair of synsets. In general, there are three classes of methods to compute semantic scores. The first class measures the text similarity of the glosses of between two synsets, where a gloss refers to the English description of the meaning of a word. The representative method includes Lesk [2], where the semantic score is calculated based on the degree of overlapping information between their glosses. The second class measures the distance of the synsets in WordNet. In WordNet, a synset is connected by some semantic edges (i.e., hypernym, hyponym). A simple and intuitively way to measure the semantic similarity is to compute the shortest path between two synsets in WordNet. Therefore, synsets with shorter path lengths have stronger semantic connections. Representative methods include wup [28], lch [15], jcn [11], lin [17], res [21], hso [10]. Recently, with the advancement of machine learning, we can represent a synset by a vector of arbitrary dimensions, where the synset vector is obtained by training a large set of corpus. The representative method includes NASARI [3]. Based on the vector representations of synsets, we can measure the semantic similarity by computing the cosine similarity, weighted overlaps [20] of the vectors. In KAM, we use the NASARI approach. For one thing, NASARI dataset is based on BabelNet, which is more up-to-date than WordNet. Second, NASARI approach shows better performance based on the experimental results shown in [3].

3 KAM Framework

KAM consists of two parts: supervised knowledge annotation and knowledge acquisition. Supervised knowledge annotation is designed to create a Prolog knowledge base that represents an augmented version of FrameNet data. Basically, the knowledge base includes the logical representations of frames, frame elements, lexical units, and valence patterns. Besides, each frame element is assigned with a list of BabelNet synsets that capture its definition. Users can also add new frames to the knowledge base. The Prolog knowledge base is used in knowledge acquisition, where we provide a tool that achieves the following:

1. run deep semantic analysis of controlled English text in order to ensure that different sentences that express the same meaning are mapped to the same logical representations.
2. perform valid logical inferences based on the facts and rules extracted from English sentences and achieve inconsistency-tolerance in the process of knowledge acquisition.
3. allow the user to enter controlled English text, disambiguate acquired knowledge, and perform question answering tasks

3.1 Preliminaries

First, we give a brief overview of KAM's language parser, Attempto Parsing Engine (APE), which is based on ACE grammar[1]. APE translates CNL sentences into a Discourse Representation Structure (DRS)[2], which captures the semantic meaning of the sentences. A

[1] `http://attempto.ifi.uzh.ch/site/docs/syntax_report.html`
[2] `http://attempto.ifi.uzh.ch/site/pubs/papers/drs_report_66.pdf`

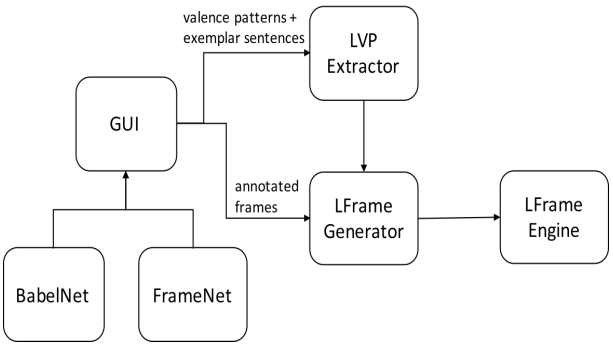


Figure 1 Supervised Knowledge Annotation.

DRS uses six pre-defined predicates to represent the semantics of a word in a sentence, including `object`, `property`, `relation`, `modifier_adv`, `modifier_pp`, `has_part`, `query`, and `predicate` predicates. For instance, the sentence `A man enters a door with a card` is represented as

```
object(A,man,countable,na,eq,1)
object(B,door,countable,na,eq,1)
object(C,card,countable,na,eq,1)
predicate(D,enter,A,B)
modifier_pp(D,with,C)
```

where the `object`-predicate denotes the head word of a noun phrase, the `predicate`-predicate represents an action, and the `modifier_pp` signifies a prepositional modifier to the action.

We define the semantic relation between two predicates as a *dependency path* that connects these two predicates via a list of variables and intermediate predicates. For the above example, `man` is the subject of the `enter` action. The semantic relation is represented as

```
predicate(D,enter,A,B) -> A -> object(A,man,countable,na,eq,1)
```

There can be more than one dependency paths that connect two predicates. For the rest of this section, we will only consider the shortest dependency path.

3.2 Supervised Knowledge Annotation

Figure 1 shows the architecture of supervised knowledge annotation. The GUI provides an environment for the user to annotate FrameNet frames and query BabelNet. Given a frame, the user is required to disambiguate each frame element name by assigning a BabelNet synset to it. For instance, in `Being_Employed` frame, `Position` is assigned with the synset `bn:00010073n` (a job in an organization) and `Employee` is assigned with the synset `bn:00030618n` (a person who is hired to perform a job). The annotated frame and frame elements are mapped into Prolog representation by LFrame Generator as

```
frame_def(Frame_Name,[
    frame_element(Frame_Element_Name, BabelNet_SID)|...])
```

Next, the user annotates each lexical unit and its exemplar sentences. Given that FrameNet exemplar sentences are written in normal English, some may not be parsed by APE. Therefore, the user needs to manually rephrase each exemplar sentence according to ACE grammar.

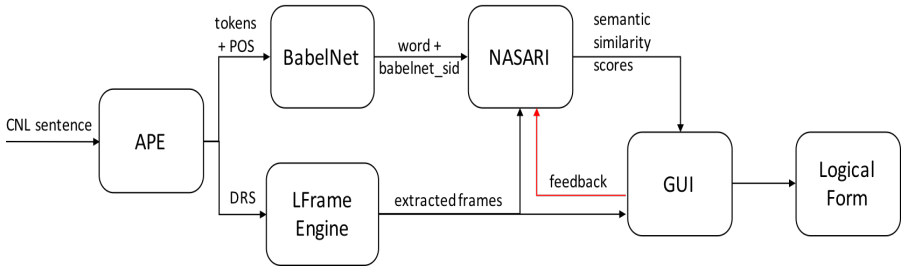


Figure 2 Knowledge Acquisition.

Besides, the user marks the lexical unit and frame elements of a sentence. The annotated lexical units and exemplar sentences are mapped into Prolog representation by LVP Extractor as

```
lvp(Lexical_Unit, Frame_Name, [
    lgf(Frame_Element_1,Dependency_Path_1)|...])
```

where it extracts dependency paths that represent the semantic relations between a lexical unit and the frame elements.

LFrame Engine uses the `frame_def` and `lvp` predicates to extract frame relations and identify frame elements from CNL sentences. Specifically, LFrame Engine applies the `lvp` to each word of a sentence to extract potential frames and frame elements, denoted as

```
frame(Frame_Name,[
    frame_element(Frame_Element_Name, Val)|...])
```

3.3 Knowledge Acquisition

Figure 2 shows the process of translating a CNL sentence into its logical form. First, APE parses the input sentence and generates the DRS and part-of-speech of each word. Second, KAM queries BabelNet and gets the synsets each word belongs to. In parallel, LFrame Engine extracts the candidate frames and frame elements from the DRS.

Next, for each candidate frame relation, KAM disambiguates the word sense of each frame element based on the frame element name. Recall from the previous subsection, each frame element name is assigned with a BabelNet synset ID that captures its definition. Here, KAM uses NASARI database to measure the semantic similarity between each synset the frame element belongs to and the frame element name. KAM chooses the synset with the highest semantic similarity score as the word sense of the frame element. The sum of the semantic scores of each frame element is defined as the score of the extracted frame relation. Finally, KAM ranks the candidate frames based on their scores. For example, given the sentence `There is a person who works in London`, LFrame Engine finds three candidate frame relations:

```
frame(Being_Employed,[frame_element(Employee, person),
    frame_element(Employer, London)])
frame(Being_Employed,[frame_element(Employee, person),
    frame_element(Position, London)])
frame(Being_Employed,[frame_element(Employee, person),
    frame_element(Place, London)])
```

KAM computes the semantic similarity scores between `person` and `Employee` (resp. `London` and `Employer`, `London` and `Position`, and `London` and `Place`) in order to disambiguate the word sense of `person` and `London` in each frame. In this case, the third frame has the highest score where person is assigned with BabelNet synset `bn:00046516n` (a human being) and London is assigned with `bn:00013179n` (the capital and largest city of England). KAM shows the ranked results to the user and asks the user to choose the one which is consistent with his/her understanding. Given that NASARI uses a statical approach to measure the semantic similarities, there could be errors in the computation. KAM allows the user to audit the result. The feedback will be recorded in order to improve the quality of semantic similarity measures in the next run.

3.4 Logical Representation

KAM represents the semantics of the frame relations in a paraconsistent logical framework, Annotated Predicate Calculus (APC) [8]. APC is a paraconsistent logical framework that deals with inconsistency. The syntax is the same as FOL except for atomic formulas of the form $p : s$, where p is an FOL atomic formula and s is a truth annotation. Truth annotations come from an arbitrary upper the Belnap's semilattice with four truth values: $\bot$, $\mathbf{t}$, $\mathbf{f}$, $\top$ where $\bot \leq \mathbf{f} \leq \top$ and $\bot \leq \mathbf{t} \leq \top$. Here, $\mathbf{t}$ and $\mathbf{f}$ denote a predicate is true and false respectively. $\bot$ denotes a predicate is neither true or false. $\top$ denotes a predicate is both true and false, which causes an inconsistency. APC is based on stable model semantics and the models are computed on Clingo. Further details of APC and its applications in natural language understanding can be found in [8]. The advantage APC provides over Answer Set Programming (ASP) systems and first-order logic is that APC allows inference in the presence of inconsistent knowledge. Besides, it captures a lot of complex features in natural language, e.g., negation, numerical constraints, reasoning by cases. For the previous sentence `There is a person who works in London`, its encoding is

```
frame(being_employed, #1) : t.
frame_element(#1, employee, #2) : t.
frame_element(#1, place, #3) : t.
object(#2, person, bn:00046516n) : t.
object(#3, london, bn:00013179n) : t.
```

where `t` is a truth annotation in APC, `#1`, `#2`, and `#3` are skolemized constants, `bn:00046516n` and `bn:00013179n` refer to BabelNet synsets.

4 Evaluation Design

Our initial step of evaluation is to test CNL sentences which describe human-related information, including a person's gender, occupation, origin, age, nationality, religious belief, and so on. We encode a set of frames such as *Being_employed*, *People_by_origin*, *People_by_religion*, *People_by_age*, *Personal_relationship* that represent the entity relations with respect to human. The testing set is constructed from Wikipedia. Given that Wikipedia provides an abundance pages about people, we extract the sentences that are related to a person's background. We evaluate both the precision and recall with respect to the testing set. Particularly, for precision, it is very likely that multiple frames are extracted for one sentence. We consider the one with the highest score as the best answer. As the next step, we will work on specific domains such as medical text, financial rules, etc. For each domain, it would require the knowledge engineer to create additional frames in order to represent the entity relations that are used there.

5 Current State of Research and Open Issues

Currently, we are working on building the prototype of the system that achieves knowledge annotation and knowledge acquisition. In the first stage, we focus on extracting logical facts from CNL sentences. We have encoded a subset of the frames in FrameNet that suffices to capture the frame relations in one domain. We also run experiments to show the power of KAM in standardizing CNL sentences to logical representations in comparison with other relation extraction tools. As the next step, we will work on extracting rules from CNL text and apply the rules and facts in question answering. Besides, we will expand the LFrame Engine to include additional frames and apply to broader domains.

6 Conclusion

In this paper, we show a novel knowledge acquisition system, KAM. First, it is a new approach in information extraction that can identify English sentences expressing the same meaning in different syntactic forms and standardize them to the same semantic representation. Second, it applies APC, a paraconsistent logical framework to encode English sentences in a logical manner to support inference in the presence of inconsistent knowledge. Third, KAM provides the users an environment to enter and disambiguate the English text and perform question answering tasks.

References

1 Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. Dbpedia: A nucleus for a web of open data. In Karl Aberer, Key-Sun Choi, Natasha Fridman Noy, Dean Allemang, Kyung-Il Lee, Lyndon J. B. Nixon, Jennifer Golbeck, Peter Mika, Diana Maynard, Riichiro Mizoguchi, Guus Schreiber, and Philippe Cudré-Mauroux, editors, *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

2 Satanjeev Banerjee and Ted Pedersen. An adapted lesk algorithm for word sense disambiguation using wordnet. In Alexander F. Gelbukh, editor, *Computational Linguistics and Intelligent Text Processing, Third International Conference, CICLing 2002, Mexico City, Mexico, February 17-23, 2002, Proceedings*, volume 2276 of *Lecture Notes in Computer Science*, pages 136–145. Springer, 2002.

3 José Camacho-Collados, Mohammad Taher Pilehvar, and Roberto Navigli. Nasari: Integrating explicit knowledge and corpus statistics for a multilingual representation of concepts and entities. *Artif. Intell.*, 240:36–64, 2016.

4 Dipanjan Das, Desai Chen, André F. T. Martins, Nathan Schneider, and Noah A. Smith. Frame-semantic parsing. *Computational Linguistics*, 40(1):9–56, 2014.

5 Broes de Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014. URL: `http://arxiv.org/abs/1401.6312`.

6 Esra Erdem, Halit Erdogan, and Umut Öztok. BIOQUERY-ASP: querying biomedical ontologies using answer set programming. In Stefano Bragaglia, Carlos Viegas Damásio, Marco Montali, Alun D. Preece, Charles J. Petrie, Mark Proctor, and Umberto Straccia, editors, *Proceedings of the 5th International RuleML2011@BRF Challenge, co-located with the 5th International Rule Symposium, Fort Lauderdale, Florida, USA, November 3-5, 2011*, volume 799 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

7 Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto controlled english for knowledge representation. In Cristina Baroglio, Piero A. Bonatti, Jan Maluszynski, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web, 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, volume 5224 of *Lecture Notes in Computer Science*, pages 104–124. Springer, 2008.

8 Tiantian Gao, Paul Fodor, and Michael Kifer. Paraconsistency and word puzzles. *TPLP*, 16(5-6):703–720, 2016.

9 M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Clingo* = ASP + control: Preliminary report. In M. Leuschel and T. Schrijvers, editors, *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, volume arXiv:1405.3694v1, 2014. Theory and Practice of Logic Programming, Online Supplement.

10 Graeme Hirst, David St-Onge, et al. Lexical chains as representations of context for the detection and correction of malapropisms. *WordNet: An electronic lexical database*, 305:305–332, 1998.

11 Jay J. Jiang and David W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *CoRR*, cmp-lg/9709008, 1997.

12 Richard Johansson and Pierre Nugues. Lth: Semantic structure extraction using nonprojective dependency trees. In *Proceedings of the 4th International Workshop on Semantic Evaluations*, SemEval '07, pages 227–230, Stroudsburg, PA, USA, 2007. Association for Computational Linguistics.

13 Chrstopher R. Johnson, Charles J. Fillmore, Miriam R.L. Petruck, Collin F. Baker, Michael J. Ellsworth, Josef Ruppenhofer, and Esther J. Wood. FrameNet: Theory and Practice, 2002.

14 Paul Kingsbury and Martha Palmer. Propbank: the next level of treebank. In *Proceedings of Treebanks and lexical Theories*, volume 3. Citeseer, 2003.

15 Claudia Leacock and Martin Chodorow. Combining local context and wordnet similarity for word sense identification. *WordNet: An electronic lexical database*, 49(2):265–283, 1998.

16 John Lehmann, Sean Monahan, Luke Nezda, Arnold Jung, and Ying Shi. LCC approaches to knowledge base population at TAC 2010. In *Proceedings of the Third Text Analysis Conference, TAC 2010, Gaithersburg, Maryland, USA, November 15-16, 2010*. NIST, 2010.

17 Dekang Lin. An information-theoretic definition of similarity. In Jude W. Shavlik, editor, *Proceedings of the Fifteenth International Conference on Machine Learning (ICML 1998), Madison, Wisconsin, USA, July 24-27, 1998*, pages 296–304. Morgan Kaufmann, 1998.

18 Mausam, Michael Schmitz, Stephen Soderland, Robert Bart, and Oren Etzioni. Open language learning for information extraction. In Jun'ichi Tsujii, James Henderson, and Marius Pasca, editors, *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 523–534. ACL, 2012.

19 George A. Miller. Wordnet: A lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

20 Mohammad Taher Pilehvar, David Jurgens, and Roberto Navigli. Align, disambiguate and walk: A unified approach for measuring semantic similarity. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 1: Long Papers*, pages 1341–1351. The Association for Computer Linguistics, 2013.

21 Philip Resnik. Using information content to evaluate semantic similarity in a taxonomy. *arXiv preprint cmp-lg/9511007*, 1995.

22 Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. In *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 442–453. ACM Press, 1994.

23 Karin Kipper Schuler. *Verbnet: A Broad-coverage, Comprehensive Verb Lexicon*. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, 2005. AAI3179808.

24 Rolf Schwitter. English as a formal specification language. In *13th International Workshop on Database and Expert Systems Applications (DEXA 2002), 2-6 September 2002, Aix-en-Provence, France*, pages 228–232. IEEE Computer Society, 2002.

25 Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*, pages 697–706. ACM, 2007.

26 Mihai Surdeanu, David McClosky, Mason R. Smith, Andrey Gusev, and Christopher D. Manning. Customizing an information extraction system to a new domain. In *Proceedings of the ACL 2011 Workshop on Relational Models of Semantics*, RELMS '11, pages 2–10, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

27 Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85, 2014.

28 Zhibiao Wu and Martha Stone Palmer. Verb semantics and lexical selection. In James Pustejovsky, editor, *32nd Annual Meeting of the Association for Computational Linguistics, 27-30 June 1994, New Mexico State University, Las Cruces, New Mexico, USA, Proceedings.*, pages 133–138. Morgan Kaufmann Publishers / ACL, 1994.

A Simple Complete Search for Logic Programming

Jason Hemann[1], Daniel P. Friedman[2], William E. Byrd[3], and Matthew Might[4]

1 **Indiana University, Bloomington, IN 47402, USA**
`jhemann@indiana.edu`
2 **Indiana University, Bloomington, IN 47402, USA**
`dfried@indiana.edu`
3 **University of Utah, Salt Lake City, UT 84112, USA**
`Will.Byrd@cs.utah.edu`
4 **University of Utah, Salt Lake City, UT 84112, USA**
`might@cs.utah.edu`

Abstract

Here, we present a family of complete interleaving depth-first search strategies for embedded, domain-specific logic languages. We derive our search family from a stream-based implementation of incomplete depth-first search. The DSL's programs' texts induce particular strategies guaranteed to be complete.

1998 ACM Subject Classification D.3.2 Language Classifications: Applicative (functional) languages, Constraint and logic languages

Keywords and phrases logic programming, streams, search, Racket, backtracking, relational programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.14

1 Introduction

A common logic language implementation technique is the shallowly-embedded, internal domain-specific language (DSL) [12, 8, 4]. In this technique, the logic-language programmer writes in the syntax of the underlying host language and the DSL's operators' behavior are described in terms of the host's semantics. Designers need implement only behaviors not supported natively by the host. For logic languages implemented in functional hosts, these may include backtracking and search, among others.

Here, we present a family of complete interleaving depth-first search strategies induced by an embedding. Each logic program's text induces a particular search strategy. Unlike most other embeddings, our operators provide a complete search without the performance penalties associated with, for example, breadth-first search [12, 8]. We improve on earlier efforts [5] by combining the hand-off of control with relation definition, and in doing so decrease the amount of interleaving while maintaining a complete search. We achieve a minimal placement of interleaving points for arbitrary relation definitions.

We host our embedding in Racket [3], but any eager language with functions as values is equally suited. We deliberately restrict ourselves to a small host language feature set. We rely chiefly on `cons` and `lambda` (λ). The data-structure interpolation operators ‘ and , are a shorthand for explicit `cons`es, and the promise and force operators we use are shallow wrappers over function creation and application.

Program

A program consists of zero or more *relations* (predicates, in Prolog parlance) and an initial *goal*. Invoking the first goal may require a call to some relation, which may itself require a call to another relation or relations, etc.

Goals

Goals are implemented as functions that take a *state* and return a *stream* of states. They consist of primitive constraints such as equality (`==`), relation invocations like `(peano q)`, and their closure under operators that perform conjunction, disjunction, and variable introduction.

State

We execute a program p by attempting an initial goal in the context of zero or more relations. The program proceeds by executing a goal in a *state*. The state contains a *substitution* and a *counter* for generating fresh variables. Every program's execution begins with an *initial state* devoid of any constraint information and a variable count 0.

Streams

Executing a goal in a state s/c (connoting a `s`ubstitution and `c`ounter pair) yields a stream. A stream takes one of three shapes. The stream may be empty, indicating the goal cannot be achieved in s/c. A stream may contain one or more resultant states. In this case, each element of the stream is a different (in terms of control flow (i.e., disjunctions); the same state may occur many times in a single stream) way to achieve that goal from s/c. Our streams are not necessarily infinite; there may be finitely many ways to achieve a goal in a given state. We call these first two shapes *mature*, whereas an *immature* stream is a delayed computation that will return a stream when forced.

The final step of running a program is to continually force the resultant stream until it yields a list of answers. Our programs are not guaranteed to terminate. The stream we get from invoking the initial goal may be *unproductive*: repeated applications of `force` will never produce an answer [11]. This is the only potential cause of non-termination; all of the other core operations in our implementation are total.

2 Implementing Depth-first Search

We now implement our interleaving search operators: `disj`, `conj`, `define-relation`, and `call/initial-state`. We omit here the syntactic equality constraint `==` and `call/fresh` (which scopes new logic variables). Interested readers should consult an extended version of this work [6].

The binary operators `disj` and `conj` act as goal combinators, and they let us to write composite goals representing the disjunction or conjunction of their arguments.

```
#| Goal × Goal → Goal |#
(define ((disj g1 g2) s/c) ($append (g1 s/c) (g2 s/c)))

#| Goal × Goal → Goal |#
(define ((conj g1 g2) s/c) ($append-map g2 (g1 s/c)))
```

We define `disj` and `conj` in terms of `$append` and `$append-map`. If we define these functions as aliases for the finite-list `append` and `append-map` functions standard to many

languages [10], our streams will always be empty or answer-bearing; in fact, they will be fully computed. The result of attempting an `==` goal must be a finite list, of length 0 or 1. If both of `disj`'s arguments are goals that produce finite lists, then the result of invoking `append` on those lists is itself a finite list. If both of `conj`'s arguments are goals that produce finite lists, then the result of invoking `append-map` with a goal and a finite list must itself be a finite list. Invoking a goal constructed from these operators in the initial state returns a list of all successful computations, computed in a depth-first, preorder traversal of the search tree generated by the program.

3 Recursion and `define-relation`

We must enrich our implementation to allow recursive relations. DFS is incomplete for computations with infinite branches. Consider the following stylized Prolog definition of the predicate `peano` that generates Peano numbers.

```
peano(N) :- N = z ; [s R], peano(R).
```

At present there are several obstacles to writing relations like `peano` that refer to themselves or one another in their definitions in our embedding. Suppose we'd used `define` to build a function that we hope would behave like a relation:

```
(define (peano n)
  (disj (== n 'z)
        (call/fresh (λ (r) (conj (== n `(s ,r))
                                 (peano r))))))
```

When we use the `peano` relation in the following program, we hope to generate some Peano numbers. We invoke `(call/fresh ...)` with an initial state. Invoking that goal creates and lexically binds a new fresh variable over the body. The body, `(peano n)`, evaluates to a goal that we pass the state `(() . 1)`. This goal is the disjunction of two subgoals. To evaluate the `disj`, we evaluate its two subgoals, and then call `$append` on the result. The first evaluates to `(((0 . z)) . 1)`, a list of one state.

```
> ((call/fresh (λ (n) (peano n)))
   '(() . 0))
```

Invoking the second of the `disj`'s subgoals however is troublesome. We again lexically scope a new variable, and invoke the goal in body with a new state, this time `(() . 2)`. The `conj` goal has two subgoals. To evaluate these, we run the first in the current state, which results in a stream. We then run the second of `conj`'s goals over each element of the resulting stream and return the result. Running this second goal begins the whole process over again. In a call-by-value host, this execution won't terminate. Simply using `define` in this manner will not suffice.

We instead introduce the `define-relation` operator. This allows us to write recursive relations; with a sequence of uses of `define-relation`, we can create mutually recursive relations. Unlike the other operators, `define-relation` is a macro.

```
(define-syntax-rule (define-relation (defname . args) g)
  (define ((defname . args) s/c) (delay/name (g s/c))))
```

Racket's `define-syntax-rule` gives a simple way to construct non-recursive macros. The first argument is a pattern that specifies how to invoke the macro. The macro's first symbol, `define-relation`, is the name of the macro we define. The second argument is

a template to be filled in with the appropriate pieces from the pattern. We do implement `define-relation` in terms of Racket's `define`.

This macro expands a name, arguments, and a goal expression to a `define` expression with the same name and number of arguments and whose body is a goal. It takes a state and returns a stream, but unlike the others we've seen before, this goal returns an immature stream. When given a state `s/c`, this goal returns a promise that evaluates the original goal `g` in the state `s/c` when forced, returning a stream. A promise that returns a stream is itself an immature stream.

`define-relation` does two useful things for us: it adds the relation name to the current namespace, and it ensures that the function implementing our relation is total. It turns out that we will *never* re-evaluate an immature stream. Unlike `delay`, `delay/name` doesn't *memoize* the result of forcing the promise, so it is like a "by name" variant of `delay`. In languages without macros, the programmer could explicitly add a delay at the top of each relation; though this has the unfortunate consequence of exposing the implementation of streams.

We implement `define-relation` as a macro, since it is critical that the expression `g` not be evaluated prematurely: we need to delay the invocation of `g` in `s/c`. Under call-by-value, a function would (prematurely) evaluate its argument and would not delay the computation.

This solves the non-termination of relation invocations. When `peano` is defined by `define-relation`, the goal `(peano n)` immediately returns an immature stream when invoked. We can also write recursive relations whose goals quite clearly will never produce answers.

```
(define-relation (unproductive n)
  (unproductive n))
```

We now redefine `$append` and `$append-map`, augmenting them with support for immature streams.

```
(define ($append $1 $2)
  (cond
    ((null? $1) $2)
    ((promise? $1) (delay/name ($append (force $1) $2)))
    (else (cons (car $1) ($append (cdr $1) $2)))))
```

If the recursive argument to `$append` is an immature stream, we return an immature stream, which, when forced, continues appending the second to the first. Likewise, in `$append-map`, when `$` is an immature stream, we return an immature stream that will continue the computation but still forcing the immature stream. Rather than `delay/name`, `force`, and `promise?`, we could have used `(λ () ...)`, procedure invocation, and `procedure?`. Using λ to construct a procedure delays evaluation, and `procedure?` would be our test for an immature stream.

```
#| Goal × Stream → Stream |#
(define ($append-map g $)
  (cond
    ((null? $) '())
    ((promise? $) (delay/name ($append-map g (force $))))
    (else ($append (g (car $)) ($append-map g (cdr $))))))
```

After these changes, we must do something special when we invoke a goal in the initial state, as this can now produce an immature stream instead of an empty or answer-bearing stream such as in the following example.

```
> ((call/fresh (λ (n) (peano n)))
   '(() . 0))
#<promise>
```

4 call/initial-state

At the very least, we would like to know if our programs are *satisfiable* or not. That is, we would hope to get at least one answer if one exists, and the empty list if there are none. The `call/initial-state` operator ensures that if we return, we return with a list of answers.

```
#| Maybe Nat+ × Goal→ Mature |#
(define (call/initial-state n g) (take n (pull (g '() . 0)))))
```

`call/initial-state` takes an argument `n` which represents the number of answers to retrieve. `n` may just be a positive natural number, in which case we return at most that many answers. Otherwise, we provide `#f`, indicating our embedding should return *all* answers. It also takes a goal as an argument. The function `pull` takes a stream as argument, and if `pull` terminates, it returns a mature stream. As streams may be unproductive, it is not always possible to produce a mature stream. As a result, `pull`, and consequently `take` and `call/initial-state`, are partial functions. These are the only partial functions in our implementation.

```
#| Stream → Mature |#
(define (pull $) (if (promise? $) (pull (force $)) $))
```

`take` receives the mature stream that is the result of `pull` and, `n`, the argument dictating whether to return all, or just the first n elements of the stream.

```
#| Maybe Nat+ × Mature → List |#
(define (take n $)
  (cond
    ((null? $) '())
    ((and n (zero? (- n 1))) (list (car $)))
    (else (cons (car $) (take (and n (- n 1)) (pull (cdr $)))))))
```

Our embedding is now capable of creating, combining, and searching for answers in infinite streams.

```
> (call/initial-state 2
    (call/fresh (λ (n) (peano n))))
'((((0 . z)) . 1) (((1 . z) (0 . (s 1))) . 2))
```

Rather than always returning a list implementation of non-deterministic choice, we either have no values, a value now (possibly more than one), or something we can search later for a value. `pull`, since it forces an actual value out of a promise, is akin to run in the delay monad. `take` bears a similar relationship to run in the list monad.

5 Interleaving, Completeness, and Search

Although we can now create and manage infinite streams, we cannot manage them as well as we'd like. Consider what happens in the following program execution:

```
> (call/initial-state 1
    (call/fresh (λ (n) (disj (unproductive n)
                             (peano n)))))
```

We wish the program to return a stream containing the `n`s for which `peano` holds and in addition the `n`s for which `unproductive` holds. We know from Section 3 that there are no `n`s for which `unproductive` holds, but infinitely many for `peano`. The stream should contain

only ns for which peano holds. It's perhaps surprising, then, to learn that this program loops infinitely.

Streams that result from using unproductive will always be, as the name suggests, unproductive. When executing the program above, such an unproductive stream will be the recursive argument $1 to $append. Unproductive streams are necessarily immature. According to our definition of $append, we always return the immature stream. When we force this immature stream, it calls $append on the forced stream value of (the delayed) $1 and $2. Since unproductive is unproductive, this process continues without ever returning any of the results from peano.

Such surprising results are not solely the consequence of goals with unproductive streams. Consider the definition of church.

```
(define-relation (church n)
  (call/fresh (λ (b) (conj (== n `(λ (s) (λ (z) ,b)))
                           (peano b)))))
```

The relation church holds for Church numerals. Using a newly created variable b, it constructs a list resembling a lambda-calculus expression whose body is the variable b. It uses peano to generate the body of the numeral. We can thus use it to generate Church numerals in a manner analogous to our use of peano. But consider the following program, wherein the resulting stream is productive, but only contains elements for which peano holds.

```
> (call/initial-state 3
    (call/fresh (λ (n) (disj (peano n)
                             (church n)))))
'((((0 . z)) . 1) (((1 . z) (0 . (s 1))) . 2)
  (((2 . z) (1 . (s 2)) (0 . (s 1))) . 3))
```

Under the default Racket printing convention, "." is suppressed when it precedes a "(". We retain the "." for legibility – Racket's current-print parameter controls this behavior.

Our implementation of $append in Section 3 induces a depth-first search. Depth-first search is the traditional search strategy of Prolog and can be implemented quite efficiently. As we've seen though, depth-first search is an *incomplete* search strategy: answers can be buried infinitely deep in a stream. The stream that results from a disj goal produces elements of the stream from the second goal only after exhausting the elements of the stream from the first.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append (force $1) $2)))))
```

As a result, even if answers exist microKanren may fail to produce them. We will remedy this weakness in $append, and provide microKanren with a simple complete search. We want microKanren to guarantee each and every answer should occur at a finite position in the stream. Fortunately, this doesn't require a significant change.

```
#| Stream × Stream → Stream |#
(define ($append $1 $2)
  (cond
    ...
    ((promise? $1) (delay/name ($append $2 (force $1))))))
```

That's it. This one change to the promise? line of $append is sufficient to make disj *fair* and to transform our search from an incomplete, depth-first search to a complete one.

Interestingly, we haven't reconstructed a particular, single complete search strategy. Instead, the search strategy of microKanren programs is program- and query-specific. The particular definitions of a program's relations, together with the goal from which it's executed, dictates the order we explore the search tree. By contrast, Spivey and Seres implement breadth-first search, also a complete search, in a language similar to microKanren [12].

Relying on non-strict evaluation simplifies their implementation; manually managing delays would make the call-by-value version less elegant than their implementation. Even excepting that, their implementation requires a somewhat more sophisticated transformation than does ours. Kiselyov et al. describe a different mechanism to achieve a complete search, but they too rely on non-strict evaluation [9]. We achieve a simpler implementation of a complete search by using the delays as markers for interleaving our streams.

6 Conclusion and Related Work

There has been extensive research on logic programming implementation [1]. Spivey and Seres's [12] present a Haskell embedding of a language quite similar to microKanren. They begin with depth-first search language, and through transformations derive an implementation of breadth-first search.

Hinze [7, 8] and Kiselyov et al. [9] implement backtracking with asymptotic performance improvements over stream-based approaches like that used in microKanren and the works cited above. These context-passing implementations are also more complicated to understand and to implement. We chose to use streams in part to more easily communicate ideas.

The fair search operators in Kiselyov et al.'s LogicT monad provide the basis of the interleaving search in earlier miniKanren implementations. The LogicT transformer augments an arbitrary monad with backtracking and control operators similar to those we use. We have access to the whole logic program in our embedding and carefully control interleaving in recursions; therefore we can use less frequent interleaving and maintain a complete search.

Our development led us to a number of interesting, still-open problems. Hinze [7] shows list-based implementations of nondeterminism to be asymptotically slower than a continuation-based "context-passing" implementation. We would like to combine our manual control of delays with a context-passing implementation à la Hinze and Kiselyov et al. [9]. Earlier work by Wand [13] and Danvy et al. [2] in relating models of backtracking has provided a starting point.

While `define-relation` is sufficient to ensure our search is complete, it in general causes more interleaving than necessary. For instance, mutually-recursive relations only need one interleaving point between them, and we don't need to interleave at all deterministic relations. We could statically "push down" the delays into the body of a relation, reducing the amount of interleaving we perform while retaining a complete search. We would also like to mechanically prove the correctness of our search with a dependently-typed implementation whose types encode our fairness properties.

Acknowledgements. We thank our reviewers, both known and anonymous, for their comments and suggestions. This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Isaac Balbin and Koenraad Lecot. *Logic Programming: A Classified Bibliography*. Springer Science & Business Media, 2012.
2. Olivier Danvy, Bernd Grobauer, and Morten Rhiger. A unifying approach to goal-directed evaluation. *New Generation Computing*, 20(1):53, 2002. URL: http://dx.doi.org/10.1007/BF03037259, doi:10.1007/BF03037259.
3. Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. http://racket-lang.org/tr1/.
4. Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. *The Reasoned Schemer*. MIT Press, Cambridge, MA, 2005.
5. Jason Hemann and Daniel P. Friedman. μKanren: A minimal functional core for relational programming. In *Scheme 13*, 2013. URL: http://schemeworkshop.org/2013/papers/HemannMuKanren2013.pdf.
6. Jason Hemann, Daniel P. Friedman, William E. Byrd, and Matthew Might. A small embedding of logic programming with a simple complete search. In *Proceedings of DLS '16*. ACM, 2016. URL: http://dx.doi.org/10.1145/2989225.2989230.
7. Ralf Hinze. Deriving backtracking monad transformers. In *ACM SIGPLAN Notices*, volume 35, pages 186–197. ACM, 2000.
8. Ralf Hinze. Prolog's control constructs in a functional setting: Axioms and implementation. *International Journal of Foundations of Computer Science*, 12(02):125–170, 2001.
9. 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, *Proceedings of the 10th ACM SIGPLAN ICFP*, pages 192–203. ACM, September 2005.
10. Olin Shivers. List Library. Scheme Request for Implementation. SRFI-1, 1999. URL: http://srfi.schemers.org/srfi-1/srfi-1.html.
11. Ben A. Sijtsma. On the productivity of recursive list definitions. *ACM Trans. Program. Lang. Syst.*, 11(4):633–649, October 1989. URL: http://doi.acm.org/10.1145/69558.69563, doi:10.1145/69558.69563.
12. JM Spivey and Silvija Seres. Embedding Prolog in Haskell. In E. Meier, editor, *Haskell 99*, 1999.
13. Mitchell Wand and Dale Vaillancourt. Relating models of backtracking. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, ICFP '04, pages 54–65, New York, NY, USA, 2004. ACM. URL: http://doi.acm.org/10.1145/1016850.1016861, doi:10.1145/1016850.1016861.

On Improving Run-time Checking in Dynamic Languages*

Nataliia Stulova†

IMDEA Software Institute, Madrid, Spain and
Universidad Politécnica de Madrid (UPM), Madrid, Spain
nataliia.stulova@imdea.org

Abstract

In order to detect incorrect program behaviors, a number of approaches have been proposed, which include a combination of language-level constructs (procedure-level annotations such as assertions/contracts, gradual types, etc.) and associated tools (such as static code analyzers and run-time verification frameworks). However, it is often the case that these constructs and tools are not used to their full extent in practice due to a number of limitations such as excessive run-time overhead and/or limited expressiveness. This issue is especially prominent in the context of dynamic languages without an underlying strong type system, such as Prolog. In our work we propose several practical solutions for minimizing the run-time overhead associated with assertion-based verification while keeping the correctness guarantees provided by run-time checks. We present the solutions in the context of the Ciao system, where a combination of an abstract interpretation-based static analyzer and run-time verification framework is available, although our proposals can be straightforwardly adapted to any other similar system.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, F.3.1 Specifying and Verifying and Reasoning about Programs, F.3.3 Studies of Program Constructs, F.3.2 Semantics of Programming Languages

Keywords and phrases Runtime Verification, Assertions, Prolog, Logic Programming

Digital Object Identifier 10.4230/OASIcs.ICLP.2017.15

1 Introduction

Detecting incorrect program behaviors is an important part of the software development life cycle. It is also a complex and tedious one, in which dynamic languages bring special challenges.

A number of techniques have been proposed to aid in the process, among which we center our attention on the use of language-level constructs to describe expected program behavior, and of associated tools to compare actual program behavior against expectations, such as static code analyzers/verifiers and run-time verification frameworks.

Approaches that fall into this category are the assertion-based frameworks used in (Constraint) Logic Programming [10, 24, 16, 19], soft/gradual typing approaches in functional

* Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft*, TIN2015-67522-C3-1-R *TRACES* and TIN2008-05624 *DOVES*, and Comunidad de Madrid TIC/1465 *PROMETIDOS-CM* and S2013/ICE-2731 *N-Greens Software*, and Madrid Region program M141047003 *N-GREENS*.

† Supervised by José F. Morales (IMDEA Software Institute, Madrid, Spain) and Manuel V. Hermenegildo (IMDEA Software Institute, Madrid, Spain and Universidad Politécnica de Madrid (UPM), Madrid, Spain)

programming [5, 34, 32] and contract-based extensions in object-oriented programming [17, 18, 11]. These tools are aimed at detecting violations of the expected behavior or certifying the absence of any such violations, and often involve a certain degree of run-time testing, specially for non-trivial properties.

In practice, however, run-time overhead often remains impractically high, specially for complex properties, such as, for example, deep data structure tests. This reduces the attractiveness of run-time checking to programmers, which may allow sporadic checking of very simple conditions, but tend to turn off run-time checking for more complex properties. Some approaches even opt for limiting the expressiveness of the assertion language in order to reduce the overhead.

Our research objective is twofold:

- First, we aim for enhancing the expressiveness of the assertion language to reflect all the features of the related programming language, including, e.g., higher-order constructs, and to do so in a way that allows the programmer to write precise program specifications while not imposing a learning or programming burden on them.
- At the same time, our goal is to efficiently check such specifications, mitigating the associated run-time overhead as much as possible without compromising the safety guarantees that the checks provide.

While our work is general and system-independent, we present it for concreteness in the context of the Ciao run-time checking framework. The Ciao model [13, 24] is well understood, and different aspects of it have been incorporated in popular (C)LP systems, such as Ciao, SWI, and XSB [14, 31, 21].

2 Current Research Results

2.1 Supporting Higher-Order Properties

Higher-order programming is a widely adopted programming style that adds flexibility to the software development process. Within the (Constraint) Logic Programming ((C)LP) paradigm, Prolog has included higher-order constructs since the early days, and there have been many other proposals for combining the first-order kernel of (C)LP with different higher-order constructs, e.g., [35, 23, 4]). Many of these proposals are currently in use in different (C)LP systems and have been found very useful in programming practice, inheriting the well-known benefits of code reuse (templates), elegance, clarity, and modularization.

When higher-order constructs are introduced in the language it becomes necessary to describe properties of arguments of predicates/procedures that are themselves also predicates/procedures. While the combination of contracts and higher-order has received some attention in functional programming [12, 9], within (C)LP the combination of higher-order with the previously mentioned assertion-based approaches has received comparatively little attention to date. Current Prolog systems simply use basic atomic types (i.e., stating simply that the argument is a `pred`, `callable`, etc.) to describe predicate-bearing variables (see Listing 1). Other approaches [1] are more oriented instead towards meta programming, describing meta-types but there is no notion of directionality (modes), and only a single pattern is allowed per predicate.

Our proposal [26] contributes towards filling this gap between higher-order (C)LP programs and assertion-based extensions for error detection and program validation. Our starting point is the Ciao assertion model, which we have enhanced with a new class of properties, "predicate properties" (or *predprops*), for which we have proposed a syntax and semantics.

Listing 1 A simple program with a higher-order predicate `min/4` that accepts a custom comparator predicate

```
:- pred min(X,Y,Cmp,Min) : callable(Cmp).

min(X,Y,P,Min) :- P(R,X,Y), R <= 0, Min = X.
min(X,Y,_,Y  ).

less( 0,A,A).              lt('=',A,A).
less(-1,A,B) :- A < B.     lt('<',A,B) :- A < B.
less( 1,_,_).              lt('>',_,_).

test_min :- min(4,2,lt,2). % lt/3 is passed, but less/3 is expected
```

Listing 2 A *predprop* `comparator` example and an *anonymous* assertion in its definition.

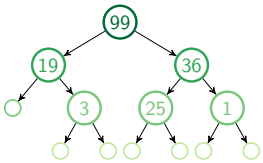
```
:- comparator(Cmp) {
  :- pred Cmp(Res,M,N) : (num(M), num(N)) => between(-1,1,Res).
}.

:- pred min(X,Y,Cmp,Min) : comparator(Cmp).
```

These new properties can be used in assertions for higher-order predicates to describe the properties of the higher-order arguments. An example of a predprop is provided in Listing 2, where an *anonymous* assertion (note the variable symbol `Cmp` in place of a predicate symbol) is used to describe a comparison predicate, that is not known at compilation time. By reusing the original assertion language syntax to describe call and success conditions of predicate-bearing arguments we allow both for better integration of these new constructs into the verification framework and at the same time lessening the burden on a programmer, who needs to provide such annotations.

Our predprop properties specify conditions for predicates that are independent of the usage context. This corresponds in functional programming to the notion of *tight* contract satisfaction [9], and it contrasts with alternative approaches such as *loose* contract satisfaction [12]. In the latter, contracts are attached to higher-order arguments by implicit function wrappers. The scope of checking is local to the function evaluation. Although this is a reasonable and pragmatic solution, we believe that our approach is more general and more amenable to combination with static verification techniques. For example, avoiding wrappers allows us to remove checks (e.g., by static analysis) without altering the program semantics. Moreover, our approach can easily support *loose* contract satisfaction, since it is straightforward in our framework to optionally include wrappers as special predprops.

2.2 Trading Memory for Speed

While having become an integral part of software development process, run-time testing can generally incur a high penalty in execution time and/or space over the standard, test-less program execution. A number of techniques have been proposed to date to reduce this overhead, including simplifying the checks at compile time via static analysis [2, 13] or reducing the frequency of checking, including for example testing only at a reduced number of points [19, 20]. Our proposal of [27] describes an approach to run-time testing that is efficient while being minimally obtrusive and remaining exhaustive. It is based on the use of memoization to cache intermediate results of check evaluation in order to avoid repeated checking of previously verified properties over the same data structure.




```
1 tree(e).
2 tree(t(L,_,R)) :- tree(L), tree(R).
3
4 t(                  ,99,                    )
5   t( ,19,         )     t(          ,36,         )
6     e     t( ,3, )         t( ,25, )     t( ,1, )
7             e   e            e    e        e   e
8
```

Figure 1 A minimalistic tree data structure implementation as a regular type `tree/1`.

Memoization has of course a long tradition in (C)LP in uses such as tabling resolution [33, 8]. Memoization has also been used in program analysis [36, 22], where tabling resolution is performed using abstract values. However, in tabling and program analysis it is call-success patterns that are usually tabled, whereas in our case the aim is to cache the results of test execution.

We concentrate our attention on checks for conformance of run-time heap structures to *regular types* [7], a useful subset of properties that are often used in assertions. An example in Fig. 1 shows a binary tree (left) and its possible implementation as a regular type together with an instance of that type describing the tree (right).

Our approach is based on the observation that run-time checks of regular types are monotonic instantiation checks, i.e., terms only become more and more instantiated with every subsequent state that the program enters.[1] We extend the Ciao run-time checking framework with a common cache, accessible to run-time checks, that stores tuples (x, t), where x is a term address and t is an identifier of a regular type. After the initial check on the term is performed, the corresponding tuple is added to the cache and for all the subsequent checks on the term, the check is replaced by a (computationally cheaper) cache lookup operation.

While studying the resulting performance of the enhanced verification framework we have observed that a straightforward use of a cache does not necessarily lead to a significant reduction in checking cost. An important issue that must be taken into account is the character of cache rewrites: as the terms grow in size cache collisions happen more often, which leads to element eviction. This in turn cancels out the advantage that using a cache gives: the ability to just lookup the regular type of some term without actually performing the check of it. However, this effect can be remedied by limiting the depth of terms that are stored in the cache (e.g., not caching terms with depth more than some n).

The idea of using memoization techniques to speed up checks has attracted some attention recently [15]. Their work (developed independently from ours) is based on adding fields to data structures to store the properties that have been checked already for such structures. In contrast, our approach has the advantage of not requiring any modifications to data structure representation, or to the checking code, program, or core run-time system.

2.3 Intertwining Compile- and Run-time Checks

A complementary approach to run-time overhead reduction consists in using static analysis to minimize the number and cost of the run-time checks that need to be placed in the program to detect incorrect program behaviors. This idea was pioneered by the Ciao system where a

[1] This is not the case of course if the language allows mutable variables and they are used in the program, but in those cases variable inmutability is tracked by the compiler / preprocessor.

number of (abstract interpretation-based) static analyses are combined in order to verify assertions to the largest extent possible at compile time, and for simplifying and reducing the number of remaining properties that need to be introduced in the program as run-time checks. However, while there has been evidence from use, there has been little systematic experimental work presented to date measuring the actual impact of analysis on reducing run-time checking overhead.

In our work [28] we generalize the existing practices as four *assertion checking modes*, each of which represents a trade-off between code annotation depth, execution time slowdown, and program behavior safety guarantees:

- *Unsafe*: no run-time checks are generated from program assertions, program execution is fast but incorrect program behaviors may stay undetected; the run-time overhead is nonexistent.
- *Client-Safe*: run-time checks are generated from the assertions of the program's interface (providing behavior guarantees for its clients), yet internal program assertions remain unchecked; the run-time overhead is taken as a minimal unavoidable one.
- *Safe-RT*: run-time checks are generated from all program assertions; this mode of checking is characterized with the strongest behavior safety guarantees yet is at the same time associated with highest check costs;
- *Safe-CT-RT*: a variation of *Safe-RT* checking mode with an additional static analysis phase, during which some of program assertions may be proven to always hold and generating run-time checks from them can be omitted; depending on the kind of analysis and the program the overhead generated by the checks from remaining assertions may be closer to that of *Client-Safe Safe-RT* modes.

We also define a transformation-based approach in order to implement each one of these modes. The differences between the transformations used in different modes are illustrated in Fig. 2. Starting with the source of the program and its assertions in the bottom-left corner, a sequence of preprocessing and compilation steps (denoted by arrows) is applied. Respective components of the verification framework that perform the source transformations are represented as hexagons.

We then concentrate on the reduction of the number of run-time tests via (abstract interpretation-based) program analysis. To this end we propose a technique that enhances analysis precision by taking into account that any assertions that cannot be proved statically will be the subject of run-time testing. In practice, it means that it is safe to make the two following assumptions for any predicate that has an assertion specifying properties that should hold on calls and success:

- The calls conditions hold after the analysis has entered the predicate definition, since either the checks for these calls conditions have already succeeded or the program has exited with error.
- The relevant success conditions hold after the predicate has exited, since, again, at this point either these success conditions have already succeeded or the program has exited with error.

2.4 Benefiting from Information Hiding

While dynamic languages offer programmers great flexibility in term creation and manipulation, for the very same reason the need for exhaustive run-time checks arises in order to guarantee the data manipulation safety and correctness. Reusable libraries, i.e., library modules that are pre-compiled independently of the client, pose special challenges in this

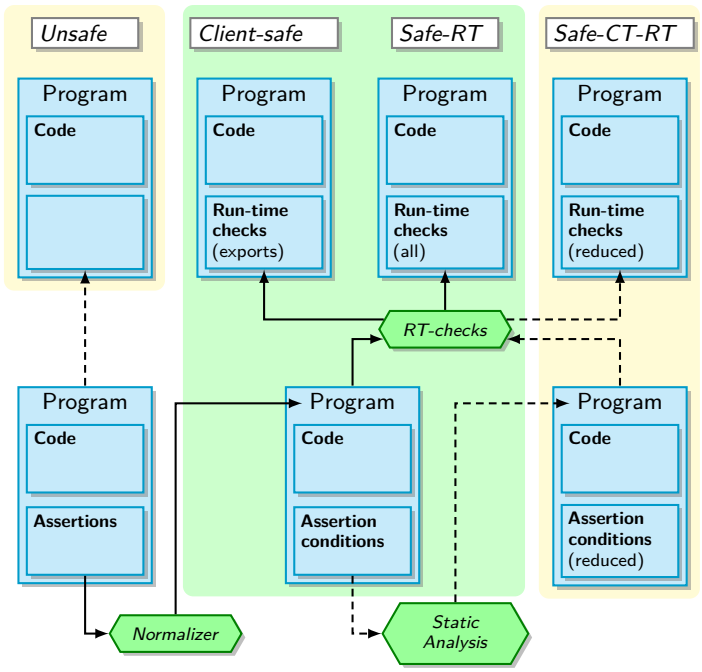


Figure 2 Source transformation differences per checking mode.

context. The key issue here is that there is virtually no control on how and where valid terms can be created, and thus it is quite common in the client-library interaction that any of these modules can create any data shape and pass it. As a consequence, (often expensive) run-time checks on the module boundaries become a necessary evil. In our work [29] we propose a possible solution to overhead reduction for run-time checks on module boundaries based on the *information hiding* principle (which is adopted in many other systems in form of encapsulation or opaque data types).

Currently, most mature Prolog implementations adopt some flavor of a module system, *predicate-based* in SWI [37], SICStus [30], YAP [25], ECLiPSe [6], and *atom-based* in XSB [31]. The difference between the two systems is the strictness of the term visibility rules: in an atom-based system local to the module terms are not visible outside it if they are not a part of the module interface. The Ciao approach [3] has until now been closer to a predicate-based module system.

We propose an extension of the predicate-based module system, that allows to specify only a subset of module terms as local (*hidden*) ones. Our argument is that in this setup we have the guarantee of the module terms' structural homogeneity, as only one module is allowed to construct/deconstruct some particular data shapes. With this there is no need to perform thorough checks of properties that verify the correctness of the data term structure at the module boundaries. Instead, it would suffice to perform checks of *shallow* versions of data shape properties: the weakened forms of the original properties that are semantically equivalent to them in the context of the possible program executions. These versions typically require asymptotically fewer execution steps and in some cases of the calls across module boundaries allow us to achieve constant run-time overhead.

To illustrate the idea behind the approach, let us consider that the `tree/1` regular type from Fig. 1 is defined in a following module `bintrees`:

```
:- module(bintrees,[tree/1,add/3,del/3,find/2]). % exported predicates

:- hide e/0.                                     % \__ hidden
:- hide t/3.                                     % /  functors

:- regtype tree/1.
tree(e).
tree(t(L,_,R)) :- tree(L), tree(R).

:- pred add(El,T0,T1) : tree(T0) => tree(T1).
add(El,T0,T1) :- ...

...  % rest of the implementation of module predicates
```

If we can assure by static analysis that none of the exported predicates of `bintrees` passes `t(_,_,_)` terms outside the module, which could allow module clients to construct arbitrary terms with this functor (e.g., `t([...],_,[...])`), then we can safely substitute the full `tree/1` property in the precondition check for `add/1` (and also in precondition checks for other exported predicates!) by its *shallow* version:

```
:- regtype shallow_tree/1.
shallow_tree(e).
shallow_tree(t(_,_,_)).

:- pred add(El,T0,T1) : shallow_tree(T0) => tree(T1).
add(El,T0,T1) :- ...
```

This way in all calls across module `bintrees` boundaries the cost of precondition checks goes down from linear in the size of the term to constant (functor correctness check).

Preliminary experimental results and details about the algorithms that perform the inference of shallow properties and conditions under which it is safe to use them in run-time checks are provided in a technical report, to which [29] refers to.

3 Conclusions and Future Work

The specification-based runtime verification approach has attracted significant interest in recent decades, both from academia and industry. As it is the case with any other open problem, finding an approach that would suit each and every system is hard, and thus opting for tailored partial solutions is more practical.

In our work we have concentrated on the peculiarities of the run-time verification task in the context of dynamic languages and their use in (C)LP systems. We have proposed an enhancement for the Ciao assertion language that allows us to capture the concrete execution contexts of higher-order terms and thus adapts the verification framework to this new use case. We have also proposed several solutions for reducing the overhead associated with run-time checks, that treat the issue from several different angles.

While for concreteness of presentation our work was carried out within the Ciao language and combined static/dynamic verification framework, our results are general and system-independent. We believe they can be straightforwardly transferred to the contexts of other declarative languages. In addition, given the advances in verification of a wide class of programming languages, including imperative languages, by translation into Horn clauses and proving properties at this level, and the fact that this approach is fully supported in the Ciao system, we argue that our results can easily be adapted to a much broader spectrum of languages.

For future work we plan to concentrate first on fully incorporating the optimization techniques described here into Ciao's run-time verification framework, as now they are available as separate system bundles. Among the issues we would also like to address more profoundly are further developments on blame assignment in the case of higher-order

assertion checking, scalability evaluation of the combination of all optimization techniques, and providing on-line demos and documentation.

Acknowledgements. We would like to thank the anonymous reviewers for providing valuable comments and suggestions that had helped to improve this paper.

References

1 C. Beierle, R. Kloos, and G. Meyer. A Pragmatic Type Concept for Prolog Supporting Polymorphism, Subtyping, and Meta-Programming. In *Proc. of the ICLP'99 Workshop on Verification of Logic Programs, Las Cruces*, Electronic Notes in Theoretical Computer Science, volume 30, issue 1. Elsevier, 2000. URL: `http://www.elsevier.nl/locate/entcs/volume30.html`.

2 F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press. URL: `ftp://cliplab.org/pub/papers/aadebug_discipldeliv.ps.gz`.

3 D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *International Conference on Computational Logic, CL2000*, number 1861 in LNAI, pages 131–148. Springer-Verlag, July 2000.

4 D. Cabeza, M. Hermenegildo, and J. Lipton. Hiord: A Type-Free Higher-Order Logic Programming Language with Predicate Abstraction. In *Ninth Asian Computing Science Conference (ASIAN'04)*, number 3321 in LNCS, pages 93–108. Springer-Verlag, December 2004.

5 Robert Cartwright and Mike Fagan. Soft Typing. In *Programming Language Design and Implementation (PLDI 1991)*, pages 278–292. SIGPLAN, ACM, 1991.

6 Cisco Systems. *ECLIPSE User Manual*, 2006.

7 P.W. Dart and J. Zobel. Efficient Run-Time Type Checking of Typed Logic Programs. *Journal of Logic Programming*, 14:31–69, October 1992.

8 S. Dietrich. *Extension Tables for Recursive Query Evaluation.* PhD thesis, Departament of Computer Science, State University of New York, 1987.

9 Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. Program. Lang. Syst.*, 33(5):16, 2011. `doi:10.1145/2039346.2039348`.

10 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.

11 Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-oriented Software*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30, Berlin, Heidelberg, 2011. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1949303.1949305`.

12 Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In Mitchell Wand and Simon L. Peyton Jones, editors, *ICFP*, pages 48–59. ACM, 2002. `doi:10.1145/581478.581484`.

13 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

14 M. V. Hermenegildo, F. Bueno, M. Carro, P. López, E. Mera, J.F. Morales, and G. Puebla. An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming*, 12(1–2):219–252, January 2012. http://arxiv.org/abs/1102.5497. doi:10.1017/S1471068411000457.

15 Emmanouil Koukoutos and Viktor Kuncak. Checking Data Structure Properties Orders of Magnitude Faster. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification*, volume 8734 of *Lecture Notes in Computer Science*, pages 263–268. Springer International Publishing, 2014. doi:10.1007/978-3-319-11164-3_22.

16 Claude Laï. Assertions with Constraints for CLP Debugging. In Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 109–120. Springer, 2000. doi:10.1007/10722311_4.

17 Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.

18 Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007. doi:10.1007/s00165-007-0026-7.

19 E. Mera, P. López-García, and M. Hermenegildo. Integrating Software Testing and Run-Time Checking in an Assertion Verification Framework. In *25th Int'l. Conference on Logic Programming (ICLP'09)*, volume 5649 of *LNCS*, pages 281–295. Springer-Verlag, July 2009.

20 E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. Profiling for Run-Time Checking of Computational Properties and Performance Debugging. In *Practical Aspects of Declarative Languages (PADL'11)*, volume 6539 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, January 2011.

21 Edison Mera and Jan Wielemaker. Porting and refactoring Prolog programs: the PROSYN case study. *TPLP*, 13(4-5-Online-Supplement), 2013.

22 K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

23 Gopalan Nadathur and Dale Miller. Higher–Order Logic Programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5. Oxford University Press, 1998.

24 G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, March 2000.

25 Vítor Santos Costa, Luís Damas, and Ricardo Rocha. The YAP Prolog System. *Theory and Practice of Logic Programming*, 2011. http://arxiv.org/abs/1102.3896v1.

26 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Assertion-based Debugging of Higher-Order (C)LP Programs. In *16th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'14)*. ACM Press, September 2014.

27 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Practical Run-time Checking via Unobtrusive Property Caching. *Theory and Practice of Logic Programming, 31st Int'l. Conference on Logic Programming (ICLP'15) Special Issue*, 15(04-05):726–741, September 2015. URL: http://arxiv.org/abs/1507.05986.

28 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Reducing the Overhead of Assertion Run-time Checks via static analysis. In *18th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'16)*, pages 90–103. ACM Press, September 2016.

29 N. Stulova, J. F. Morales, and M. V. Hermenegildo. Term Hiding and its Impact on Run-time Check Simplification (Extended Abstract). In *Proceedings of the Technical Com-*

munications of the 33rd International Conference on Logic Programming (ICLP 2017), Melbourne, Australia, August 28 - September 1, 2017. OASIcs, August 2017.

30 Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. *SICStus Prolog User's Manual*, 4.1.1 edition, December 2009. Available from `http://www.sics.se/sicstus/`.

31 Terrance Swift and David Scott Warren. XSB: Extending Prolog with Tabled Logic Programming. *TPLP*, 12(1-2):157–187, 2012.

32 Asumu Takikawa, Daniel Feltey, Earl Dean, Matthew Flatt, Robert Bruce Findler, Sam Tobin-Hochstadt, and Matthias Felleisen. Towards practical gradual typing. In John Tang Boyland, editor, *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, volume 37 of *LIPIcs*, pages 4–27. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. URL: `http://www.dagstuhl.de/dagpub/978-3-939897-86-6`, `doi:10.4230/LIPIcs.ECOOP.2015.4`.

33 H. Tamaki and M. Sato. OLD Resolution with Tabulation. In *Third International Conference on Logic Programming*, pages 84–98, London, 1986. Lecture Notes in Computer Science, Springer-Verlag.

34 Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In George C. Necula and Philip Wadler, editors, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, pages 395–406. ACM, 2008. `doi:10.1145/1328438.1328486`.

35 D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood Ltd., Chicester, England, 1982.

36 R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

37 J. Wielemaker. *The SWI-Prolog User's Manual 5.9.9*, 2010. Available from `http://www.swi-prolog.org`.