Technical Communications of the 34th International Conference on Logic Programming

ICLP 2018, July 14–17, 2018 ,Oxford, United Kingdom

Edited by

Alessandro Dal Palu'
Paul Tarau
Neda Saeedloei
Paul Fodor

OASICS

Editors

Alessandro Dal Palu'
Università di Parma, Italy
`alessandro.dalpalu@unipr.it`

Paul Tarau
University of North Texas, USA
`Paul.Tarau@unt.edu`

Neda Saeedloei
Southern Illinois University Carbondale, USA
`neda@cs.siu.edu`

Paul Fodor
Stony Brook University, USA
`pfodor@cs.stonybrook.edu`

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.

Contents

ICLP 2018: Technical Communications

ICLP 2018: Doctoral Consortium

◼ Preface

This volume contains the Technical Communications and the Doctoral Consortium papers of the 34-th International Conference on Logic Programming (ICLP 2018), held in Oxford, United Kingdom, from July 14th to July 17th, 2018.

ICLP 2018 was part of the Federated Logic Conference 2018, (FLOC 2018), as the premier conference on *foundations and applications of logic programming*, including but not restricted to answer-set programming, non-monotonic reasoning, unification and constraints based logic languages, constraint handling rules, argumentation logics, deductive databases, description logics, inductive and co-inductive logic programming.

Contributions to ICLP are sought in all areas of logic programming, including:

- **Foundations**: semantics, execution algorithms, formal models.
- **Implementation**: virtual machines, compilation, memory management, parallel execution, foreign interfaces.
- **Language Design**: inference engines, type systems, concurrency and distribution, modules, metaprogramming, relations to object-oriented and functional programming, logic-based domain-specific languages.
- **Software-Development Techniques**: declarative algorithms and data structures, design patterns, debugging, testing, profiling, execution visualization.
- **Transformation and Analysis**: assertions, type and mode inference, partial evaluation, abstract interpretation, program transformations.
- **Applications and Synergies**: interaction with SAT, SMT and CSP solvers, logic programming techniques for type inference and theorem proving, horn-clause analysis, knowledge representation, cognitive computing, artificial intelligence, natural language processing, information retrieval, web programming, education, computational life sciences, computational mathematics.

Three kinds of submissions were accepted:
- *Technical papers*, which include technically sound, innovative ideas that can advance the state of logic programming;
- *Application papers*, which describe interesting application domains;
- *System and tool papers*, which emphasize novelty, practicality. usability, and availability of the systems and tools.

ICLP implemented 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 this special issue of TPLP. The TCs comprise papers which the Program Committee (PC) 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 have received 63 submissions of abstracts, of which 49 resulted in full submissions. The Program Chairs, acting as guest editors of the special issue, organized the refereeing process, which was undertaken by the PC with the support of external reviewers. Each paper was reviewed by at least three referees who provided detailed written evaluations. This enabled a list of papers to be short-listed as candidates for rapid communication. The authors of these papers revised their submissions in light of the reviewers' suggestions, and

all these papers were subject to a second round of reviewing. Of these candidates papers, 25 were accepted as rapid communications, to appear in the special issue. In addition, the PC recommended 15 papers to be accepted as TCs, of which 14 were also presented at the conference (1 was withdrawn). We would like to thank the organizers of these affiliated events for their contributions to the conference as a whole. We are also deeply indebted to the 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 were:

Mario Alviano	Hassan Aït-Kaci	Marcello Balduccini
Mutsunori Banbara	Pedro Cabalar	Mats Carlsson
Manuel Carro	Michael Codish	Alessandro Dal Palù
Marina De Vos	Thomas Eiter	Esra Erdem
Thom Frühwirth	Marco Gavanelli	Martin Gebser
Gopal Gupta	Michael Hanus	Amelia Harrison
Manuel Hermenegildo	Tomi Janhunen	Angelika Kimmig
Ekaterina Komendantskaya	Nicola Leone	Michael Leuschel
Yuliya Lierler	Vladimir Lifschitz	Barry O'Sullivan
David Pearce	Enrico Pontelli	Ricardo Rocha
Chiaki Sakama	Vitor Santos Costa	Tom Schrijvers
Tran Cao Son	Theresa Swift	Peter Szeredi
Mirek Truszczyński	German Vidal	Jan Wielemaker
Stefan Woltran	Roland Yap	Jia-Huai You
Neng-Fa Zhou		

The external reviewers were:

Weronika T. Adrian	Sandra Alves	Joaquín Arias
João Barbosa	Zhuo Chen	Md Solimul Chowdhury
Carmine Dodaro	Gregory Duck	Wolfgang Faber
František Farka	Mário Florido	Michael Frank
Daniel Gall	Gregory Gelfond	Jurriaan Hage
Markus Hecher	Arash Karimi	Emily Leblanc
Jan Maly	Fumio Mizoguchi	Eric Monfroy
Michael Morak	Falco Nogatz	Adrian Palacios
Javier Romero	Elmer Salazar	Zeynep Saribatur
Sebastian Schellhorn	Peter Schüller	Farhad Shakerin
Nada Sharaf	Jon Sneyers	Finn Teegen
Pedro Vasconcelos	Alicia Villanueva	Yisong Wang
Philipp Wanko	Fangkai Yang	

The 14th Doctoral Consortium (DC) on Logic Programming was held in conjunction with ICLP 2018 and FLoC 2018. It attracts Ph.D. students in the area of Logic Programming Languages from different backgrounds (e.g. theoretical, implementation, application) and encourages a constructive and fruitful advising. Topics included: theoretical foundations of logic and constraint logic programming, sequential and parallel implementation technologies, static and dynamic analysis, abstract interpretation, compilation technology, verification, logic-based paradigms (e.g., answer set programming, concurrent logic programming, inductive logic programming) and innovative applications of logic programming. This year the Doctoral Consortium accepted ten papers in the areas described above: 5 in Logical Systems, 1 in Implementations and 4 in Applications of logic programming. We warmly thank all

student authors, supervisors, referees, co-chairs, members of the program committee and the organizing team that made the Doctoral Consortium greatly successful.

The accepted papers were:

- Yi Wang. Probabalistic Action Language pBC+
- Emily Leblanc. Explaining Actual Causation via Reasoning about Actions and Change
- Zhun Yang. Translating P-log, LP^{MLN}, LPOD, and CR-Prolog2 into Standard Answer Set Programs
- Frantisek Farka. Proof-relevant resolution for elaboration of programming languages
- Arindam Mitra. The Learning-Knowledge-Reasoning Paradigm For Natural Language Understanding and Question Answering
- Richard Taupe. Speeding Up Lazy-Grounding Answer Set Solving
- Tiantian Gao. Knowledge Acquisition and Question Answering via Controlled Natural Language
- Van Nguyen. Natural Language Generation From Ontologies
- Filipe Gouveia, Ines Lynce and Pedro T. Monteiro. Model Revision of Logical Regulatory Networks using Logic-based Tools
- Philipp Obermeier. Scalable Robotic Intra-Logistics with Answer Set Programming

The DC Program Committee members were:

Marina De Vos, University of Bath

Fabio Fioravanti, University of Chieti-Pescara

Martin Gebser, Aalto University

Jose F. Morales, IMDEA Software Research Institute

Takehide Soh, Information Science and Technology Center, Kobe University

Frank D. Valencia LIX, Ecole Polytechnique

Neda Saeedloei, Southern Illinois University Carbondale

Paul Fodor, Stony Brook University

We would also like to express our gratitude to the full ICLP 2018 organization committee, namely Marco Gavanelli who acted as general chair; Stefan Woltran, who served as workshop chair; Enrico Pontelli, who acted as publicity chair and designed the web pages; Paul Fodor and Neda Saeedloei, who jointly chaired the Doctoral Program of ICLP; and Paul Fodor, 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 to the personnel at Schloss Dagstuhl-Leibniz Zentrum fur Informatik, especially Michael Wagner, for their assistance. We would also like to thank the staff of the EasyChair conference management system for helping the Program Chairs with their prompt support. We wish to thank each author of every submitted papers, since their efforts keep the conference alive and the participants to ICLP for bringing and sharing their ideas and latest developments.

Finally, we would like to thank the FLOC 2018 conference general chair: Moshe Y. Vardi and to the FLOC 2018 co-chairs Daniel Kroening and Marta Kwiatkowska for their help and guidance to make ICLP part of this outstanding scientific event.

Alessandro Dal Palù

Paul Fodor

Neda Saeedloei

Paul Tarau

Epistemic Logic Programs with World View Constraints

Patrick Thor Kahl
Space and Naval Warfare Systems Center Atlantic, North Charleston, SC, USA
patrick.kahl@navy.mil

Anthony P. Leclerc
Space and Naval Warfare Systems Center Atlantic, North Charleston, SC, USA; and
College of Charleston, Charleston, SC, USA
anthony.leclerc@navy.mil, leclerca@cofc.edu

── Abstract ──────────────────────────────
An epistemic logic program is a set of rules written in the language of Epistemic Specifications, an extension of the language of answer set programming that provides for more powerful introspective reasoning through the use of modal operators K and M. We propose adding a new construct to Epistemic Specifications called a *world view constraint* that provides a universal device for expressing global constraints in the various versions of the language. We further propose the use of subjective literals (literals preceded by K or M) in rule heads as syntactic sugar for world view constraints. Additionally, we provide an algorithm for finding the world views of such programs.

1 Introduction

The language of *Epistemic Specifications* extends answer set programming (ASP) by adding modal operators K ("known") and M ("may be true"). It was introduced by Gelfond [15] after observing a need for more powerful introspective reasoning than that offered by ASP alone. A program written in this language is called an *epistemic logic program* (ELP), with semantics defined using the notion of a *world view* – a collection of sets of literals (*belief sets*), analogous to answer sets of an ASP program. Recent interest has led to a succession of proposed semantics [17, 21, 12, 35, 44] advocating differing perspectives with respect to the meaning of connectives and intended world views of programs. This clash of intuition is only one aspect of the problem as defining a semantics that facilitates understanding and yet accurately reflects intuition appears to be quite difficult as discussed in Section 2.

In this paper, we don't try to resolve the clash; instead, we focus on the important problem of modeling knowledge using purely epistemic constraints. With the original semantics, such constraints could be used to eliminate possible worlds. As will be shown, this property was lost with the more recent semantics. This leads to substantial difficulties in modeling knowledge. Thus, in an attempt to facilitate ELP development in the midst of language evolution, we propose extending the language with a syntactic construct called a *world view constraint* (WVC) to distinguish certain constraints as global. WVCs are universal – immune by design to the various devices (e.g., maximality requirements) used to tweak the semantics.

As an introductory example, let us look at a simple epistemic logic program that features a purely epistemic constraint:

p or q.

$\leftarrow$ not K p. % purely epistemic constraint

The second rule is purely epistemic in that its body consists solely of a subjective literal whose interpretation is global in the sense that its truth value depends on the entire collection of belief sets in some possible world view rather than some current (local) working set. To be precise, the truth value of not K p depends on whether p is in all belief sets of some possible world view under consideration. For example, considering $W = \{\{p\}, \{q\}\}$ as a possible world view, not K p evaluates to *true*, thus violating the constraint.

As W is the only possible world view that is consistent with the rest of the program (i.e., the first rule), this program has no world view under the original semantics [15]. However, under most of the recently proposed semantics [21, 12, 35], its world view is $\{\{p\}\}$ – a result that some may consider unexpected. To achieve the same result as that of the original semantics, we propose replacing the second rule with the following:

$\overset{\text{wv}}{\Longleftarrow}$ not K p. % world view constraint

which results in no world view for any of the proposed ELP semantics (if extended with our new construct). This motivating example and others are discussed in Sections 3 and 5.

The paper is organized as follows. We begin with a summary of related work in development of the language semantics and ELP solvers. Next we discuss the use of constraints in both ASP and Epistemic Specifications, providing motivational argument for the introduction of WVCs. We then present the syntax and semantics of the extended language. We follow with examples demonstrating its use. Finally, we give an algorithm for computing the world views of an ELP with WVCs, and close with suggestions for related extensions.

2   Background and Related Work

With his good friend and colleague Vladimir Lifschitz, the foundations for what we now call *answer set programming (ASP)* had been laid down in the seminal works of Michael Gelfond [19, 20] by 1991. It seems strange in hindsight that, in the same year, a far less known language called *Epistemic Specifications* was proposed by Gelfond [15] in an attempt to address an observed inadequacy in the expressiveness of its better known predecessor. Gelfond noticed that the following ASP program does not entail a required interview for a scholarship applicant whose eligibility is not able to be established:

% rules for scholarship eligibility at a certain college where S represents a scholarship applicant

$eligible(S) \leftarrow highGPA(S).$

$eligible(S) \leftarrow fairGPA(S),\ minority(S).$

$\neg eligible(S) \leftarrow \neg highGPA(S),\ \neg fairGPA(S).$

% ASP attempt to express that an interview is required if applicant eligibility can't be determined

$interview(S) \leftarrow$ not $eligible(S),$ not $\neg eligible(S).$

% applicant data

$fairGPA(mike)$ or $highGPA(mike).$

The program correctly reflects that Mike's eligibility can not be determined, but its answer sets, $\{fairGPA(mike), interview(mike)\}$ and $\{highGPA(mike), eligible(mike)\}$, do not conclude that an interview is required since only one contains $interview(mike)$.

Gelfond's solution was to extend the language by adding modal operator K ("known") and changing the fourth rule above as follows:

% updated rule to express interview requirement using modal operator K

$interview(S) \leftarrow$ `not` K $eligible(S)$, `not` K $\neg eligible(S)$.

The updated rule says that $interview(S)$ is to be believed if both $eligible(S)$ and $\neg eligible(S)$ are each *not known* (i.e., not in all belief sets of the world view). The program has world view $\{\{fairGPA(mike), interview(mike)\}, \{highGPA(mike), eligible(mike), interview(mike)\}\}$ with its belief sets both containing $interview(mike)$; thus, the required interview is entailed.

Although the language of Epistemic Specifications was revised in the first years of its introduction, after 1994 [5, 16] (referred to hereafter as *ES1994*) little concerning its semantics was seen in the literature for almost two decades. In the intervening years before 2011, Chen [10] proposed *GOL*, a generalization of Levesque's *logic of only knowing* ($\mathcal{OL}$) [28], that "covers Gelfond's important notion of Epistemic Specifications." Preda [33] proposed an alternative to Epistemic Specifications using multiple levels of negation (perhaps a precursor to the 2016 Shen-Eiter proposal discussed later). Wang and Yan Zhang [41] offered another alternative, proposing an epistemic extension to Pearce's *equilibrium logic of here-and-there* [32]. Efforts in 2011 by Faber & Woltran [13] and Truszczyński [39] mark the beginning of a renewed interest in Epistemic Specifications.

With the resurgence of interest, Gelfond felt an update to Epistemic Specifications was needed. His proposal [17] (referred to hereafter as *ES2011*) specifically addressed unintended world views due to recursion through modal operator K, as exemplified here:

$p \leftarrow$ K p.

Under ES1994 semantics, this program has two world views, $\{\{\}\}$ and $\{\{p\}\}$. Under ES2011 semantics, only the first is a world view, which is arguably more intuitive. It was observed, however, that unintended world views due to recursion through modal operator M remain, as demonstrated by the following one-line program:

$p \leftarrow$ M p.

Under ES2011 semantics the program has two world views, $\{\{\}\}$ and $\{\{p\}\}$. This result did not seem intuitive. Following Gelfond's lead, Kahl et al. [22, 21] proposed another update (referred to hereafter as *ES2014*) to address the issue, with semantics supporting only the latter world view.

It was suggested by Fariñas del Cerro et al. [12] that there remain unintended world views for certain programs with ES2014 semantics, particularly the following:

$p \leftarrow$ M q, `not` q.
$q \leftarrow$ M p, `not` p.

Per ES2014, the program has two world views, $\{\{\}\}$ and $\{\{p\}, \{q\}\}$, of which the first, they argue, seems unintended. Their notion of *autoepistemic equilibrium models* (AEEMs) attempts to address this concern with a new epistemic extension of equilibrium logic that includes a maximality condition on epistemic equilibrium models. Using AEEMs successfully eliminates $\{\{\}\}$ from the above program's world views.

Shen and Eiter [35] offered another update to the semantics, albeit using different syntactic notation, that focused on resolving unintended world views due to:

- *epistemic circular justification* in which a literal is considered *true* solely on the assumption that it is in all belief sets (i.e., belief in ℓ is justified only by K ℓ); and
- not satisfying the *property of knowledge minimization with epistemic negation.*

The *property of knowledge minimization with epistemic negation* is based on a maximality requirement on a *guess* (i.e., a set of *epistemic negations* – equivalent to subjective literals of the forms not K ℓ and M ℓ – considered *true* within the program under consideration) for its associated collection of belief sets to be a world view, all other conditions being satisfied. In [23], the authors provided a revision of ES2014 semantics by adding this maximality requirement (referred to hereafter as *ES2016*).

Following suit, Zhizheng Zhang [45] updated his semantics for *answer set programming with graded modality* (ASP$^{\mathsf{GM}}$) by adding a maximality condition in line with Shen and Eiter. With some syntactic liberty, Epistemic Specifications can be viewed as a proper subset of ASP$^{\mathsf{GM}}$ with ASP$^{\mathsf{GM}}$ allowing for expressing a lower and upper bound on the number of belief sets containing a specified literal within a world view. (We will revisit ASP$^{\mathsf{GM}}$ in Section 7.)

Recently, Yan Zhang and Yuanlin Zhang [44] offered a different semantics for ELPs, with a stricter view on *circular justification.* To illustrate, they argue that the program

$$p \leftarrow \text{M } p.$$

should have the world view $\{\{\}\}$ rather than $\{\{p\}\}$ as they do not consider circular justification of p as sufficient reason to accept the latter. To them, justification for M p being *true* requires that *belief in p is forced in some belief set of the rational agent.*[1] Others argue that M p is equivalent to not K not p and that the *rationality principle* (which states that a rational agent should believe only what it is forced to believe) favors *not knowing* (not K) over *knowing* (K), so $\{\{p\}\}$ is the preferred world view. In contrast, Zhang & Zhang use this same principle to argue against $\{\{p\}\}$ since the *possibility* of p is not viewed as enough by itself to force belief in p.

Regardless of differing views, it appears there remains room for improvement. As one example, in [35] the *problem of unintended world views due to recursion through* M is defined as a semantics for which "its world views do not satisfy the *property of knowledge minimization with epistemic negation.*" Use of this definition avoids the question of whether, based on intuition, a program has unintended world views. Consider again the program

$$p \leftarrow \text{M } q, \text{ not } q.$$
$$q \leftarrow \text{M } p, \text{ not } p.$$

for which $\{\{p\}, \{q\}\}$ is the only world view per this knowledge minimization property. Adding

$$r \leftarrow \text{M } p, \text{ M } q.$$

results in the world view $\{\{p, r\}, \{q, r\}\}$, as one might expect. But now if we add the rule

$$s \leftarrow \text{K } r.$$

we get *two* world views: $\{\{p, r, s\}, \{q, r, s\}\}$ and $\{\{\}\}$. In lieu of the other results, this seems unintuitive in spite of following the *property of knowledge minimization with epistemic negation.* We believe this demonstrates the difficulty in defining an intuitive semantics.

---

[1] See the notion of an *externally-supported M-cycle* in [22].

In conjunction with development of the language, there has been concomitant development of tools for finding world views. Attempts at developing a solver or inference engine include `ELMO` by Watson [42], `sismodels` by Balduccini [3], `Wviews` by Kelly [24, 25, 40] using Yan Zhang's algorithm [43], `ESmodels` by Zhizheng Zhang et al. [34, 46], `ELPS` by Balai [1, 2], `ELPsolve` by the authors [23], `EP-ASP` by Son et al. [26, 36], `EHEX` by Strasser [37], and `selp` by Bichler et al. [6, 7]. A thorough discussion of these tools is left for another paper [27]. It deserves note, however, that all extant solvers use an ASP solver for backend processing, and as ASP solver development has matured, ELP solver development has slowly followed.

3 Motivation for World View Constraints

It is well known (see, for example, Proposition 2 in [30]) that constraints (headless rules) in an ASP program have the net effect of, at most, ruling out certain answer sets from the program (modulo its constraints). To illustrate, consider the following ASP program:

p or q.

$p \leftarrow q$.

which has one answer set $\{p\}$. If we add the constraint

$\leftarrow p$, $\texttt{not } q$.

the resulting program has no answer set since $\{p\}$ violates this constraint.

With Epistemic Specifications, constraints can have an additive or subtractive effect on belief sets or entire world views. Consider, for example, the following ELP:

p or q.

$r \leftarrow \mathrm{M} \, q$.

with world view $\{\{p, r\}, \{q, r\}\}$. If we add the constraint

$\leftarrow q$.

the resulting program has world view $\{\{p\}\}$. Let's look at another example:

p or q.

$r \leftarrow \mathrm{M} \, p$.

s or $t \leftarrow \mathrm{K} \, p$.

This program has a single world view, $\{\{p, r\}, \{q, r\}\}$. If we add the constraint

$\leftarrow \mathrm{M} \, p, \ \mathrm{M} \, q$.

the resulting program has two world views per ES2016: $\{\{p, r, s\}, \{p, r, t\}\}$ and $\{\{q\}\}$.

The previous examples illustrate potential differences in the effect of constraints on an ELP compared to an ASP program. The last may also show how constraints can be a possible source of confusion with respect to world views.[2] Consider another example:

p or q.

$\leftarrow \texttt{not } \mathrm{K} \, p$.

---

[2] Should the program even have a world view? Under the earlier ES1994 semantics, it does not!

Under ES2016 semantics, its world view is $\{\{p\}\}$; however, under the original semantics [15], the program has no world view. This raises the question:

Which result is intended?

If the intent of the constraint is to rule out *world views* that do not contain p in every belief set, then the latter (from the original semantics) would seem correct. Under the later semantics, the net effect of the constraint is to eliminate *belief sets* that would otherwise result in a world view that violates the constraint.

For ES2014 semantics, it was shown in [21] that, in general, to eliminate *world views* that do not contain p in every belief set (and not simply eliminate belief sets from a world view that would otherwise not meet this requirement), two constraints are required instead of the one given above, resulting here in the following program:

$$p \ \texttt{or} \ q.$$
$$\leftarrow p, \ \texttt{not} \ \mathrm{K} \ p.$$
$$\leftarrow \texttt{not} \ \mathrm{M} \ p.$$

So with ES2014 semantics we now have a program with no world view. The same is true in this case with the later ES2016 semantics; however, the new maximality requirement in ES2016 means such "tricks" won't work for all programs. Consider the following:

$$p \leftarrow \mathrm{M} \ q, \ \texttt{not} \ q.$$
$$q \leftarrow \mathrm{M} \ p, \ \texttt{not} \ p.$$
$$r \leftarrow \mathrm{M} \ p, \ \mathrm{M} \ q.$$

Under ES2014 semantics, $\{\{\}\}$ and $\{\{p,r\},\{q,r\}\}$ are the world views. Per ES2016, only the latter is a world view. If we now add the constraint

$$\leftarrow \mathrm{K} \ r.$$

the resulting program has world view $\{\{\}\}$, which is not an ES2016 world view without this constraint. We observe that there does not appear to be a general way to simply rule out world views under ES2016 semantics. This observation leads to our thesis.

As the semantics of Epistemic Specifications has evolved to address unintended world views and support intuition with respect to certain programs, we believe the added complexity has had a negative side effect with respect to intuitive understanding of certain other programs – particularly those involving constraints with subjective literals. Thus, in an attempt to facilitate correct problem encoding/program development in line with intuition, we propose a new language construct called a *world view constraint* (WVC) and introduce symbol $\overset{\text{\tiny wv}}{\leftarrow}$ read as "it is not a world view (if...)" for use in forming a WVC. For example,

$$\overset{\text{\tiny wv}}{\leftarrow} \ \mathrm{K} \ p.$$

is read "it is not a world view if p is known" and means (informally) that any world view satisfying $\mathrm{K} \ p$ is ruled out from the set of world views of the program under consideration. This is analogous to how constraints affect answer sets in ASP, though at the world view level for Epistemic Specifications.

Syntax and Semantics

For the purpose of demonstrating the use of WVCs, we first define the syntax and semantics for two versions of the language: ES2014 and ES2016. We direct the reader to the papers referenced earlier for information on other versions of Epistemic Specifications. We present our proposal for extending the language in Section 4.3, and follow by suggesting a means of expressing the bounds for the grounding of variables within the context of the new constructs.

In general, the syntax and semantics of the language of Epistemic Specifications follow that of ASP with the notable addition of modal operators K and M, plus the new notion of a *world view* which is a collection of *belief sets* analogous to answer sets. We assume familiarity with ASP [4, 8, 14, 18, 29]. We use $\mathrm{AS}(\mathcal{P})$ to denote the set of answer sets of ASP program $\mathcal{P}$. We use symbol $\models$ for *satisfies* and $\not\models$ for *does not satisfy*.

4.1 Syntax [ES2014 and ES2016]

An epistemic logic program is a set of rules of the form

$$\ell_1 \ \texttt{or} \ \dots \ \texttt{or} \ \ell_k \leftarrow e_1, \ \dots, \ e_n.$$

where $k \geq 0$, $n \geq 0$, each ℓ_i is a *literal* (an atom or a classically-/strongly-negated atom; called an *objective literal* when needed to avoid ambiguity), and each e_i is a literal or a *subjective literal* (a literal immediately preceded by K or M) possibly preceded by `not` (default negation). As in ASP, a rule having an objective/subjective literal with a variable term is a shorthand for all ground instantiations of the rule. By $body(R)$ we denote the set $\{e_1, \dots, e_n\}$ from the body of rule R.

4.2 Semantics

▶ **Definition 1** (When a Subjective Literal Is Satisfied). Let W be a non-empty set of consistent sets of ground literals, and ℓ be a ground literal.
- $W \models \mathrm{K}\,\ell$ if $\forall A \in W : \ell \in A$.
- $W \models \texttt{not}\,\mathrm{K}\,\ell$ if $\exists A \in W : \ell \notin A$.
- $W \models \mathrm{M}\,\ell$ if $\exists A \in W : \ell \in A$.
- $W \models \texttt{not}\,\mathrm{M}\,\ell$ if $\forall A \in W : \ell \notin A$.

▶ **Definition 2** (Modal Reduct). Let Π be a ground epistemic logic program, W be a non-empty set of consistent sets of ground literals, and ℓ be a ground literal. We denote by Π^W the *modal reduct of* Π *with respect to* W defined as the ASP program[3] obtained from Π by replacing/removing subjective literals in rule bodies or deleting associated rules per the following table:

subjective literal φ	**if $W \models \varphi$ then...**	**if $W \not\models \varphi$ then...**
$\mathrm{K}\,\ell$	replace $\mathrm{K}\,\ell$ with ℓ	delete rule containing $\mathrm{K}\,\ell$
$\texttt{not}\,\mathrm{K}\,\ell$	remove $\texttt{not}\,\mathrm{K}\,\ell$	replace $\texttt{not}\,\mathrm{K}\,\ell$ with $\texttt{not}\,\ell$
$\mathrm{M}\,\ell$	remove $\mathrm{M}\,\ell$	replace $\mathrm{M}\,\ell$ with $\texttt{not}\,\texttt{not}\,\ell$
$\texttt{not}\,\mathrm{M}\,\ell$	replace $\texttt{not}\,\mathrm{M}\,\ell$ with $\texttt{not}\,\ell$	delete rule containing $\texttt{not}\,\mathrm{M}\,\ell$

▶ **Definition 3** (World View under ES2014 Semantics). Let Π be a ground epistemic logic program and W be a non-empty set of consistent sets of literals. W is a *world view* of Π under ES2014 semantics if $W = \mathrm{AS}(\Pi^W)$.

---

[3] with nested expressions of the form `not not` ℓ as defined in [30]

▶ **Definition 4** (Epistemic Negations[4]). Let Π be a ground epistemic logic program, W be a non-empty set of consistent sets of literals, and ℓ be a ground literal. We denote by $E_P(\Pi)$ the set of distinct subjective literals appearing (regardless of being negated) in Π, each taking the form of $\texttt{not K}\ell$ or $\text{M}\ell$ (referred to as *epistemic negations*) as follows:

$$E_P(\Pi) = \{\ \texttt{not K}\ell : K\ell \text{ appears in } \Pi\ \} \cup \{\ M\ell : M\ell \text{ appears in } \Pi\ \}.$$

In context with Π, we use Φ to denote a subset of $E_P(\Pi)$, and denote by Φ_W the subset of epistemic negations in $E_P(\Pi)$ that are satisfied by W; i.e., $\Phi_W = \{\ \varphi\ :\ \varphi \in E_P(\Pi) \wedge W \models \varphi\ \}$.

▶ **Definition 5** (World View under ES2016 Semantics). Let Π be a ground epistemic logic program and W be a non-empty set of consistent sets of literals. W is a *world view* of Π under ES2016 semantics if:
(1) $W = \text{AS}(\Pi^W)$; and (2) there is no W' such that $W' = \text{AS}(\Pi^{W'})$ and $\Phi_{W'} \supset \Phi_W$.[5]

4.3   World View Constraints and World View Rules

We extend the language of Epistemic Specifications by introducing a *world view constraint* as a construct for restricting the world views of an ELP, and a *world view rule* as a syntactic device for specifying a world view constraint in an effort to facilitate problem encoding/program development. The syntax and semantics of ES2016 are assumed here for the core ELP, though the definitions should work with other language versions.

4.3.1   World View Constraints

A *world view constraint* (WVC) is an epistemic logic program rule of the form

$$\overset{\texttt{wv}}{\Leftarrow}\ s_1,\ ...,\ s_n.$$

where each s_i is a (possibly negated) subjective literal.[6]

▶ **Definition 6** (When a World View Constraint Is Violated). Let W be a non-empty set of consistent sets of ground literals, and C be a ground WVC of the form $\overset{\texttt{wv}}{\Leftarrow} s_1,\ ...,\ s_n$. We say that W *violates* C if $\forall s_i \in body(C) : W \models s_i$.[7]

▶ **Definition 7** (Semantics of an ELP with WVCs). Let Π be a ground ELP with WVCs such that $\Pi = \Pi_0 \cup \Pi_{\texttt{wvc}}$ where $\Pi_{\texttt{wvc}}$ is the set of all WVCs in Π and $\Pi_0 = \Pi \setminus \Pi_{\texttt{wvc}}$ (i.e., the part of the program without WVCs). Let W be a non-empty set of consistent sets of ground literals. W is a *world view* of Π if:
(1) W is a world view of Π_0; and (2) W does not violate any rule in $\Pi_{\texttt{wvc}}$.

Returning to our example, let Π be the following program, partitioned as shown:

$$\left.\begin{array}{l} p \leftarrow \text{M } q,\ \texttt{not } q. \\ q \leftarrow \text{M } p,\ \texttt{not } p. \\ r \leftarrow \text{M } p,\ \text{M } q. \end{array}\right\} \Pi_0$$

$$\left.\overset{\texttt{wv}}{\Leftarrow}\ \text{K } r.\qquad\quad\right\} \Pi_{\texttt{wvc}}$$

Per ES2016 semantics, Π_0 has one world view $W = \{\{p, r\}, \{q, r\}\}$, but by our definition W violates the WVC in $\Pi_{\texttt{wvc}}$ since $W \models \text{K } r$; hence, Π has no world view.

---

[4] introduced in [35] using a different syntax
[5] The *maximality requirement* on Φ_W comes from the *general epistemic semantics* of Shen and Eiter [35].
[6] A negated subjective literal is of the form $\texttt{not K } \ell$ or the form $\texttt{not M } \ell$ in ES2016 syntax.
[7] Likewise, we say that W *satisfies* C (i.e., $W \models C$) if $\exists s_i \in body(C) : W \not\models s_i$.

4.3.2 World View Rules and World View Facts

A *world view rule* (WVR) is an epistemic logic program rule of the form

$$s_1 \ \text{or} \ ... \ \text{or} \ s_k \ \overset{\text{\tiny wv}}{\Leftarrow} \ s_{k+1}, \ ..., \ s_n.$$

where each s_i is a (possibly negated) subjective literal. We define a WVR as follows:

$$s_1 \ \text{or} \ ... \ \text{or} \ s_k \ \overset{\text{\tiny wv}}{\Leftarrow} \ s_{k+1}, \ ..., \ s_n. \ \overset{\text{def}}{=} \ \overset{\text{\tiny wv}}{\Leftarrow} \ \text{not} \ s_1, \ ..., \ \text{not} \ s_k, \ s_{k+1}, \ ..., \ s_n.$$

where $\text{not} \ \text{not} \ \varphi \equiv \varphi$ for a subjective literal φ. A WVR is thus syntactic sugar for a WVC.

Similar to a *fact* in ASP, the $\overset{\text{\tiny wv}}{\Leftarrow}$ symbol can be omitted from a WVR with no body. We refer to such rules as *world view facts*, or *WV facts*,[8] and use below in our example:

$p \leftarrow \text{M} \ q, \ \text{not} \ q.$

$q \leftarrow \text{M} \ p, \ \text{not} \ p.$

$r \leftarrow \text{M} \ p, \ \text{M} \ q.$

$\text{not} \ \text{K} \ r. \quad$ % equivalent to $\overset{\text{\tiny wv}}{\Leftarrow} \ \text{K} \ r.$

Note that with these definitions, any WVC can be written as a WVR, or equivalently as a WV fact. To demonstrate, the following three rules are all strongly equivalent:

$\overset{\text{\tiny wv}}{\Leftarrow} \ \text{K} \ p, \ \text{not} \ \text{K} \ q, \ \text{M} \ r, \ \text{not} \ \text{M} \ s. \qquad$ % expressed here as a WVC

$\text{not} \ \text{K} \ p \ \text{or} \ \text{K} \ q \ \overset{\text{\tiny wv}}{\Leftarrow} \ \text{M} \ r, \ \text{not} \ \text{M} \ s. \qquad$ % expressed here as a WVR

$\text{not} \ \text{K} \ p \ \text{or} \ \text{K} \ q \ \text{or} \ \text{not} \ \text{M} \ r \ \text{or} \ \text{M} \ s. \qquad$ % expressed here as a WV fact

4.4 Grounding Concerns

The issue of grounding an ELP received attention by both Kelly [24] and Cui et al. [11]. In [22], Kahl proposed an ELP solver algorithm that first creates a corresponding ASP program from the ungrounded ELP, and then uses an ASP grounder to determine the associated ground terms. This requires the rules in the ELP to be *safe* in the sense that any variable term appearing in a rule has a corresponding *positive literal* (either an objective literal or a subjective literal of the form $\text{K} \ \ell$) in the body with the same variable term.

Having only subjective literals of the form $\text{K} \ \ell$ available for rule safety is too restrictive for WVCs. One could argue that the use of a *sorted signature*, such as in an *epistemic logic program with sorts* [2], would suffice if rule safety were the only issue; however, being able to limit the grounding of variable terms to less than the full range of their acceptable domains is key to abstraction. Without such capability, flexibility and elaboration tolerance suffer. To address the practical need of having a reasonable way to express limits on the domain of a variable term in a WVC, we propose an extended syntax for a WV fact as follows:

$$s_1 \ \text{or} \ ... \ \text{or} \ s_m \leftarrow d_1, \ ..., \ d_n.$$

where each s_i is a (possibly negated) subjective literal, and each d_i is a *domain atom*[9] – also referred to as a *domain predicate* [38] – or a *comparison atom* (typically expressed using an infix "built-in" predicate; e.g., $X \neq a$). The body is used here only to determine the

---

[8] In addition to being a notational convenience, solver developers can avoid introducing a new token for the $\overset{\text{\tiny wv}}{\Leftarrow}$ symbol since any WVC can be expressed as a (possibly disjunctive) WV fact.

[9] The associated *ground* domain atoms are understood to be the same in every belief set.

appropriate grounding of variable terms in the head of the rule. The use of the $\leftarrow$ symbol is intentional as the body is not (after grounding and translation) part of any WVC.[10] The program rules below demonstrate the use of this extended syntax:

```
% domain atoms
d_x(a).   d_x(b).
d_y(0).   d_y(1).   d_y(2).   d_y(3).
% WV fact using the extended syntax
not K p(X,Y)  or  M q(X) ← d_x(X), d_y(Y), Y < 2.
```

Grounding[11] the last rule results in four WV facts:

```
not K p(a,0)  or  M q(a).      not K p(b,0)  or  M q(b).
not K p(a,1)  or  M q(a).      not K p(b,1)  or  M q(b).
```

5   Examples and Simplifications

Henceforth, ES2016 extended with WVCs is assumed unless stated otherwise.

5.1   Epistemic Conformant Planning Module

The *epistemic conformant planning module*[12] for ES2014 with a sorted signature is as follows:

$$occurs(A, S) \leftarrow \text{M } occurs(A, S), \ S < n.$$
$$\neg occurs(A_2, S) \leftarrow occurs(A_1, S), \ A_1 \neq A_2.$$
$$success \leftarrow goal(n).$$
$$\leftarrow success, \ \texttt{not K } success.$$
$$\leftarrow \texttt{not M } success.$$

where constant $n \in \mathbb{N}$ represents the plan horizon, variables A, A_1, and A_2 range over actions, and variable S ranges over integral time steps where $0 \leq S \leq n$. The last two rules are constraints that together (as discussed in Section 3) rule out world views that do not satisfy K $success$. With the proposed extension, we can replace these two constraints with one WVC that is succinct, intuitive, and easier to understand than the original pair of constraints:

$$\overset{\text{\tiny WV}}{\Longleftarrow} \ \texttt{not K } success.$$

This is also relevant in that the proof of correctness for solving conformant planning problems encoded using the original epistemic conformant planning module (with the other elements of this methodology) depends in part on the two constraints ruling out world views that do not satisfy K $success$; however, that part of the proof is not valid for ES2016 semantics. Using the proposed WVC instead of the two original constraints elucidates this for both semantics.

---

[10] It also fits well with the idea that the solver developer need not introduce a new token for the $\overset{\text{\tiny WV}}{\Longleftarrow}$ symbol.
[11] to include forward propagation with removal of body literals that are always *true* and removal of any rule where a body literal is always *false* (so-called "smart" grounding)
[12] See [21] for details on the use of ELPs to solve conformant planning problems using this module.

5.2   Autonomous Control

Consider an exploratory robot operating on Mars with a round-trip communication delay of 30 minutes with Earth. Although an Earth operator may receive a continuous stream of data from the robot, the data is already 15 minutes old when received, and any instruction sent will not be received by the robot for another 15 minutes. As Thomas Ormston [31] of the European Space Agency put it, "there's a lot that can happen in half an hour on Mars." It is important, for example, that the robot does not fall off a cliff. Though intermittent goals may be provided from Earth, some autonomous control is needed for the robot to move at a reasonable pace. We envision as part of the on-board control system[13] of the robot an epistemic planning component that uses information about the terrain and observable surroundings to help form and select a plan to get to a specified goal. Included in rules used to plan could be WVCs as follows:

$\overset{\text{wv}}{\Leftarrow}$ M *likelihood_of_falling_off_a_cliff*(*high*).
$\overset{\text{wv}}{\Leftarrow}$ M *likelihood_of_falling_off_a_cliff*(*moderate*).

These would prevent selecting a plan where the possibility of falling off a cliff is high/moderate.

5.3   Subsumption and Simplification

In the table below are subjective literal forms that can subsume others in a rule body.[14]

subsumer	subsumed
	M ℓ
K ℓ	not M $\bar{\ell}$
	not K $\bar{\ell}$
M ℓ	not K $\bar{\ell}$
not M ℓ	not K ℓ

For example:
$\overset{\text{wv}}{\Leftarrow}$ K p, M p, not M $\neg p$, not K $\neg p$. $\equiv \overset{\text{wv}}{\Leftarrow}$ K p.

$\overset{\text{wv}}{\Leftarrow}$ M p, not K $\neg p$. $\equiv \overset{\text{wv}}{\Leftarrow}$ M p.
$\overset{\text{wv}}{\Leftarrow}$ not M p, not K p. $\equiv \overset{\text{wv}}{\Leftarrow}$ not M p.

With world view constraints, subsumption can also occur across multiple rules, perhaps most easily seen using the WV fact form. Consider the following pair of WV facts:

K p.   M p.

The subsumer-subsumed list above applies to pairs of non-disjunctive WV facts. Any world view satisfying the first rule must satisfy the second; thus, the second rule can be removed.
Identifying tautologies can also help in program simplification. For example, WV fact

K p or not K p.

is worthless and can be removed. For a more complex example, consider the following rules:

M q or K p.   M q or not K p.

With respect to any world view of a program containing this pair, either Kp or not Kp will be satisfied (but not both), so these two rules can be reduced to the one rule: M q.

---

[13] Details of such a control system are beyond the scope of this paper and left to the reader's imagination.
[14] The symbol $\bar{\ell}$ in the table indicates the logical complement of (ground) objective literal ℓ; e.g., if $\ell = \neg p$ then $\bar{\ell} = p$. Logical subsumption follows from Definition 1 and the definition of a world view.

6   Algorithm for Computing World Views of an ELP with WVCs

The following is a generic algorithm for finding the world views of an ELP with WVCs:

Generic Algorithm
INPUT: Π (a ground ELP with WVCs)
1. partition Π into $\Pi_{\mathtt{wvc}}$ (the WVCs of Π) and $\Pi_0 = \Pi \setminus \Pi_{\mathtt{wvc}}$
2. use your favorite ELP solver to find the world views of Π_0
3. eliminate any world view of Π_0 that violates a WVC of $\Pi_{\mathtt{wvc}}$
OUTPUT: remaining world views of Π_0 not eliminated in Step 3

For those interested in implementing a solver, we now provide a more detailed algorithm. Details of an algorithm to compute the world views of an ELP under ES2016 semantics are given in [23]. We use a simplified version, modified to handle WVCs. Although we provided a grounding strategy for WVCs in Section 4.4, for brevity, the input is assumed ground.

Notation: From a ground ELP with WVCs $\Pi = \Pi_0 \cup \Pi_{\mathtt{wvc}}$, ASP program Π_0' is created as a modal reduct framework to aid in computing the world views of Π_0. For each literal ℓ appearing in an epistemic negation of the form $\mathtt{not}\ \mathrm{K}\ \ell$ in $E_P(\Pi_0)$, fresh atoms k_ℓ, $k0\_\ell$, $k1\_\ell$ are created by prefixing ℓ with $k\_$, $k0\_$, and $k1\_$ (respectively), and substituting 2 for $\neg$ if ℓ is a classically-/strongly-negated atom. Likewise, for ℓ appearing in an epistemic negation of the form $\mathrm{M}\ \ell$ in $E_P(\Pi_0)$, fresh atoms m_ℓ, $m0\_\ell$, $m1\_\ell$ are created. These fresh atoms are referred to as *k-/m-atoms*, or, allowing for negated forms, *k-/m-literals*. For example, given an epistemic negation of the form $\mathtt{not}\ \mathrm{K}\ \ell$, if $\ell = p(a)$ then k_ℓ denotes $k\_p(a)$, but if $\ell = \neg p(a)$ then k_ℓ denotes $k\_2p(a)$. Fresh atoms k_ℓ (in negated form) and m_ℓ are used as substitutes for $\mathrm{K}\ \ell$ and $\mathrm{M}\ \ell$, respectively, in the ASP representation of the modal reduct of Π with respect to a potential world view. The intended meaning of $k1\_\ell$ is "$\mathrm{K}\ \ell$ is *true*"; $k0\_\ell$ means "$\mathrm{K}\ \ell$ is *false*"; $m1\_\ell$ means "$\mathrm{M}\ \ell$ is *true*"; and $m0\_\ell$ means "$\mathrm{M}\ \ell$ is *false*". Additionally, given a set W of sets of literals (including k-/m-literals), we use $W_{\setminus km}$ to denote W *modulo k-/m-literals* (i.e., the result of removing all k-/m-literals from sets in W).

The algorithm uses a "guess and check" method to compute the world views of Π_0. Each guess corresponds to a set of truth value assignments for the elements of $E_P(\Pi_0)$. A systematic approach is used, starting with the guess corresponding to the elements of $E_P(\Pi_0)$ being all *true*, working down by increasing the number of *false* elements by 1 at each successive level. Each computed world view of Π_0 is checked to ensure no WVC in $\Pi_{\mathtt{wvc}}$ is violated before it is considered a world view of Π. Any guess for which the epistemic negations assigned as *true* are a subset of those for a guess associated with a previously computed Π_0 world view will be filtered out. The order of computation and subsequent filtering enforces the maximality requirement of ES2016 semantics. (For ES2014, remove this filtering; also, computation order w.r.t. guesses is irrelevant.)

The algorithm iterates through all relevant guesses, one-guess-at-a-time, requiring (in general) computing answer sets of up to 2^n ASP programs where $n = |E_P(\Pi_0)|$. This is inefficient but relatively easy to understand. A more complex algorithm may involve including multiple guesses in each ASP program (at the expense of the need for aggregating computed answer sets) and parallelization. See [23] for a solver that uses this approach. Steps that handle WVCs can be applied there, as well as to other approaches, such as the one in [36].

Since we start with a proven algorithm for computing world views of Π_0, correctness of the algorithm is clear from the definitions and semantics of an ELP with WVCs given herein. We note that filtering out guesses that would violate WVCs *during* the computation of world views (rather than filtering out world views as a post-processing step as proposed here) could prune the search if $E_P(\Pi_0) \cap E_P(\Pi_{\mathtt{wvc}}) \neq \emptyset$, but would (in general) not be correct

per ES2016 semantics. (That approach may be useful for an ES2014 solver.) If, however, there are epistemic negations in $\Pi_{\tt wvc}$ that are not in Π_0, **the search space of guesses is pruned significantly from what it might be without WVCs**, assuming those from $\Pi_{\tt wvc}$ would otherwise be included in other rules.

Finally, as the algorithm simply checks if world views of Π_0 violate the WVCs in $\Pi_{\tt wvc}$, the effective complexity of solving Π is the same as for solving an *equivalent*[15] ELP Π_2 (without WVCs), assuming Π_2 differs only in constraints, with $|E_P(\Pi)| \leq |E_P(\Pi_2)|$.

Algorithm 1. [Computing the World Views of an ELP with WVCs]

INPUT: a ground ELP with WVCs Π	OUTPUT: the world views of Π

1. **Program Partition**: Partition Π into $\Pi_{\tt wvc}$ (the WVCs of Π) and $\Pi_0 = \Pi \setminus \Pi_{\tt wvc}$.

2. **Translation**: Create ASP program[16] Π_0' from Π_0 by:
   - leaving rules without subjective literals unchanged;
   - otherwise, replacing subjective literals and adding new rules per the following table:

subj. lit. φ	replace φ with	add rules
$\mathrm{K}\,\ell$	$\mathtt{not}\ \neg k\_\ell,\ \ell$	$\neg k\_\ell \leftarrow k0\_\ell.$
$\mathtt{not}\ \mathrm{K}\,\ell$	$\neg k\_\ell$	$\neg k\_\ell \leftarrow k1\_\ell,\ \mathtt{not}\ \ell.$
$\mathrm{M}\,\ell$	m_ℓ	$m\_\ell \leftarrow m1\_\ell.$
$\mathtt{not}\ \mathrm{M}\,\ell$	$\mathtt{not}\ m\_\ell$	$m\_\ell \leftarrow m0\_\ell,\ \mathtt{not}\ \mathtt{not}\ \ell.$

3. **Guess & Check:** Repeat (a)-(c) until all relevant guesses are generated and checked.

   a. **Generate Guess:** For each iteration, generate a guess Φ, starting with $\Phi = E_P(\Pi_0)$ for the first iteration and moving on in *popcount* order[17] for further iterations, filtering out any guess that is a subset of a guess associated with a previously found world view of Π_0. Create Π_0'' by appending to Π_0' the ASP representation of Φ (i.e., k-/m-atoms as facts corresponding to the epistemic negations in Φ) as follows:

   $$\Pi_0'' = \Pi_0' \cup \{k0\_\ell. \mid \mathtt{not}\ \mathrm{K}\,\ell \in \Phi\} \cup \{k1\_\ell. \mid \mathtt{not}\ \mathrm{K}\,\ell \in E_P(\Pi_0) \wedge \mathtt{not}\ \mathrm{K}\,\ell \notin \Phi\}$$
   $$\cup \{m1\_\ell. \mid \mathrm{M}\,\ell \in \Phi\} \qquad \cup \{m0\_\ell. \mid \mathrm{M}\,\ell \in E_P(\Pi_0) \wedge \mathrm{M}\,\ell \notin \Phi\}.$$

   b. **Compute Answer Sets:** Use an ASP solver to compute the answer sets of Π_0''.

   c. **Check:** If Π_0'' is consistent, let W be the collection of answer sets computed in (b). Verify the following conditions:
      - if $k1\_\ell$ is in the sets of W, then ℓ is in every set of W;
      - if $k0\_\ell$ is in the sets of W, then ℓ is missing from at least one set of W;
      - if $m1\_\ell$ is in the sets of W, then ℓ is in at least one set of W; and
      - if $m0\_\ell$ is in the sets of W, then ℓ is missing from every set of W.

   $W_{\backslash km}$ is a world view of Π_0 if the conditions are met. $\boldsymbol{W_{\backslash km}}$ **is a world view of $\boldsymbol{\Pi}$ if $\boldsymbol{W_{\backslash km}}$ is a world view of $\boldsymbol{\Pi_0}$ and $\boldsymbol{W_{\backslash km}}$ doesn't violate any WVC in $\boldsymbol{\Pi_{\tt wvc}}$.**

---

[15] We note there may not be a straightforward equivalent program without WVCs under ES2016 semantics.
[16] with nested expressions of the form $\mathtt{not}\ \mathtt{not}\ \ell$ as defined in [30]
[17] guess size ($|\Phi|$) will be reduced by one after exhausting all guesses of the current size

7   Conclusions and Future Work

World view constraints provide a straightforward device for encoding restrictions on the world views of an ELP, allowing the specification of high-level conditions that must not be violated. They do not fix all semantics issues, but WVCs retain consistent meaning in all.

WVCs can also be a useful addition to languages extending Epistemic Specifications, such as ASP$^{\mathsf{GM}}$ [45]. Subjective literals in ASP$^{\mathsf{GM}}$ have the form $\mathrm{M}_{[lb:ub]}\ell$ where $lb, ub \in \mathbb{N}$, $lb \leq ub$, and ℓ is a literal. For $\mathrm{M}_{[lb:ub]}\ell$ to be satisfied by a world view, the number of belief sets containing ℓ must be in the closed range $[lb, ub]$. To indicate no upper bound, ub can be omitted. The definitions in Section 4.3.1 can be used to extend ASP$^{\mathsf{GM}}$ with WVCs; however, support for negated subjective literals needs to be added for the later definitions to apply.

For future work, we would like to incorporate the notion of *weak* WVCs into Epistemic Specifications in a manner analogous to *weak constraints* [9] in ASP, but at the world view level. In principle, these would function like normal WVCs unless the program is inconsistent, where they would be systematically relaxed (perhaps in order by given weight/level) until consistency or exhaustion. Ergo, we introduce symbol $\overset{\mathrm{\scriptscriptstyle WV}}{\Leftarrow}$ and suggest the following syntax:

$$\overset{\mathrm{\scriptscriptstyle WV}}{\Leftarrow} s_1, \ ..., \ s_n. \ [w@l]$$

where $n > 0$, each s_i is a (possibly negated) subjective literal, and both w and l are non-negative integers representing *weight* and *level* values, respectively. Returning to the Martian robot example of Section 5.2, it may be more appropriate for planning to use the following:

$\overset{\mathrm{\scriptscriptstyle WV}}{\Leftarrow} \mathrm{M} \ likelihood\_of\_falling\_off\_a\_cliff(high).$

$\overset{\mathrm{\scriptscriptstyle WV}}{\Leftarrow} \mathrm{M} \ likelihood\_of\_falling\_off\_a\_cliff(moderate). \ [1@0]$

These rules express a preference for plans where likelihood of falling off a cliff is neither high nor moderate, but if none exist, moderate likelihood is accepted by relaxing the weak WVC.

---- **References** ----

1   Evgenii Balai. `ELPS`, 2015. Texas Tech. URL: `https://github.com/iensen/elps/wiki/`.

2   Evgenii Balai and Patrick Kahl. Epistemic Logic Programs with Sorts. In Daniela Inclezan and Marco Maratea, editors, *Proc. 7th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2014)*, 2014. URL: `https://sites.google.com/site/aspocp2014/paper_4.pdf`.

3   Marcello Balduccini. `sismodels`, 2001. See `http://www.mbal.tk/` for more information.

4   Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving.* Cambridge University Press, New York, NY, USA, 2003.

5   Chitta Baral and Michael Gelfond. Logic Programming and Knowledge Representation. *J. Log. Program.*, 19/20:73–148, 1994. `doi:10.1016/0743-1066(94)90025-6`.

6   Manuel Bichler, Michael Morak, and Stefan Woltran. `selp`, 2018. URL: `http://dbai.tuwien.ac.at/proj/selp/`.

7   Manuel Bichler, Michael Morak, and Stefan Woltran. Single-Shot Epistemic Logic Program Solving. In Jérôme Lang, editor, *Proc. 27th Intl. Joint Conf. on AI (IJCAI 2018)* [to appear], 2018. URL: `http://dbai.tuwien.ac.at/proj/selp/ijcai2018.pdf`.

8   Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. `doi:10.1145/2043174.2043195`.

9   Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Strong and Weak Constraints in Disjunctive Datalog. In Jürgen Dix, Ulrich Furbach, and Anil Nerode, editors, *Proc. 4th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'97)*, volume 1265 of *LNCS*, pages 2–17. Springer, 1997. `doi:10.1007/3-540-63255-7_2`.

10   Jianhua Chen. The Generalized Logic of only Knowing (GOL) that Covers the Notion of Epistemic Specifications. *J. Log. Comput.*, 7(2):159–174, 1997. `doi:10.1093/logcom/7.2.159`.

11   Rongcun Cui, Zhizheng Zhang, and Kaikai Zhao. ESParser: An epistemic specification grounder. In James P. Delgrande and Wolfgang Faber, editors, *Proc. 1st Intl. Conf. on Computer Science and Service System (CSSS 2012)*, pages 1823–1827. IEEE Computer Society CPS, 2012. `doi:10.1109/CSSS.2012.454`.

12   Luis Fariñas del Cerro, Andreas Herzig, and Ezgi Iraz Su. Epistemic Equilibrium Logic. In Qiang Yang and Michael Wooldridge, editors, *Proc. 24th Intl. Joint Conf. on AI (IJCAI 2015)*, pages 2964–2970. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/419`.

13   Wolfgang Faber and Stefan Woltran. Manifold Answer-Set Programs and Their Applications. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *LNCS*, pages 44–63. Springer, 2011. `doi:10.1007/978-3-642-20832-4_4`.

14   Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on AI and ML. Morgan and Claypool, 2012.

15   Michael Gelfond. Strong Introspection. In Thomas L. Dean and Kathleen McKeown, editors, *Proc. 9th National Conf. on Artificial Intelligence (AAAI-91)*, pages 386–391. AAAI/MIT Press, 1991. URL: `http://www.aaai.org/Papers/AAAI/1991/AAAI91-060.pdf`.

16   Michael Gelfond. Logic Programming and Reasoning with Incomplete Information. *Ann. Math. Artif. Intell.*, 12(1-2):89–116, 1994. `doi:10.1007/BF01530762`.

17   Michael Gelfond. New Semantics for Epistemic Specifications. In James P. Delgrande and Wolfgang Faber, editors, *Proc. 11th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *LNCS*, pages 260–265. Springer, 2011. `doi:10.1007/978-3-642-20895-9_29`.

18   Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014. `doi:10.1017/CBO9781139342124`.

19   Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proc. 5th Intl. Conf. and Symposium on Logic Programming (ICLP/SLP 1988)*, pages 1070–1080. MIT Press, 1988.

20   Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Comput.*, 9(3/4):365–386, 1991. `doi:10.1007/BF03037169`.

21   Patrick Kahl, Richard Watson, Evgenii Balai, Michael Gelfond, and Yuanlin Zhang. The Language of Epistemic Specifications (Refined) Including a Prototype Solver. *Journal of Logic and Computation*, 2015. `doi:10.1093/logcom/exv065`.

22   Patrick Thor Kahl. *Refining the Semantics for Epistemic Logic Programs*. PhD thesis, Texas Tech, Lubbock, TX, USA, May 2014. URL: `http://hdl.handle.net/2346/58710`.

23   Patrick Thor Kahl, Anthony P. Leclerc, and Tran Cao Son. A Parallel Memory-efficient Epistemic Logic Program Solver: Harder, Better, Faster. In Bart Bogaerts and Amelia Harrison, editors, *Proc. 9th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2016)*, 2016. URL: `https://arxiv.org/abs/1608.06910`.

24   Michael Kelly. Wviews: A World View Solver for Epistemic Logic Programs. Honours thesis, University of Western Sydney, October 2007.

25   Michael Kelly. `Wviews`, 2018. URL: `https://github.com/galactose/wviews`.

26   Tiep Le and Tran Cao Son. `EP-ASP`, 2017. NMSU. URL: `https://github.com/tiep/EP-ASP`.

27   Anthony P. Leclerc and Patrick Thor Kahl. A Survey of Advances in Epistemic Logic Program Solvers. In Jorge Fandinno and Johannes K. Fichte, editors, *Proc. 11th Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2018)* [to appear], 2018.

28   Hector J. Levesque. All I Know: A Study in Autoepistemic Logic. *Artif. Intell.*, 42(2-3):263–309, 1990. `doi:10.1016/0004-3702(90)90056-6`.

29   Vladimir Lifschitz. What Is Answer Set Programming? In Dieter Fox and Carla P. Gomes, editors, *Proc. 23rd AAAI Conf. on Artificial Intelligence (AAAI 2008)*, pages 1594–1597, 2008.

30   Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested Expressions in Logic Programs. *Ann. Math. Artif. Intell.*, 25(3-4):369–389, 1999. `doi:10.1023/A:1018978005636`.

31   Thomas Ormston. Time delay between Mars and Earth. In: ESA's *Mars Express* blog. URL: `http://blogs.esa.int/mex/2012/08/05/time-delay-between-mars-and-earth/`.

32   David Pearce. A New Logical Characterisation of Stable Models and Answer Sets. In Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski, editors, *Proc. Non-Monotonic Extensions of Logic Programming (NMELP 1996)*, volume 1216 of *LNCS*, pages 57–70. Springer, 1996. `doi:10.1007/BFb0023801`.

33   Mircea Preda. Modeling Epistemic Knowledge in Logic Programs with Negation as Failure. In Dimitris Dranidis and Illias Sakellariou, editors, *Proc. 3rd South-East European Workshop on Formal Methods (SEEFM'07)*. SEERC, 2007.

34   SEU. ESmodels, 2015. URL: `http://cse.seu.edu.cn/people/seu_zzz/indexe.htm`.

35   Yi-Dong Shen and Thomas Eiter. Evaluating epistemic negation in answer set programming. *Artif. Intell.*, 237:115–135, 2016. `doi:10.1016/j.artint.2016.04.004`.

36   Tran Cao Son, Tiep Le, Patrick Kahl, and Anthony Leclerc. On Computing World Views of Epistemic Logic Programs. In Carles Sierra, editor, *Proc. 26th Intl. Joint Conf. on AI (IJCAI 2017)*, pages 1269–1275, 2017. `doi:10.24963/ijcai.2017/176`.

37   Anton Strasser. `EHEX`, 2018. TU Wien. URL: `https://github.com/hexhex/ehex`.

38   Tommi Syrjänen and Ilkka Niemelä. The Smodels System. In Thomas Eiter, Wolfgang Faber, and Miroslaw Truszczyński, editors, *Proc. 6th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2001)*, volume 2173 of *LNCS*, pages 434–438. Springer, 2001. `doi:10.1007/3-540-45402-0_38`.

39   Miroslaw Truszczyński. Revisiting Epistemic Specifications. In Marcello Balduccini and Tran Cao Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning - Essays Dedicated to Michael Gelfond on the Occasion of His 65th Birthday*, volume 6565 of *LNCS*, pages 315–333. Springer, 2011. `doi:10.1007/978-3-642-20832-4_20`.

40   UWS. `Wviews`, 2007. URL: `http://staff.scem.uws.edu.au/~yan/Wviews.html`.

41   Kewen Wang and Yan Zhang. Nested Epistemic Logic Programs. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Proc. 8th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2005)*, volume 3662 of *LNCS*, pages 279–290. Springer, 2005. `doi:10.1007/11546207_22`.

42   Richard Glenn Watson. *An Inference Engine for Epistemic Specifications*. Master's thesis, UTEP, El Paso, TX, USA, May 1994.

43   Yan Zhang. Computational Properties of Epistemic Logic Programs. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *Proc. 10th Intl. Conf. on Principles of Knowledge Representation and Reasoning*, pages 308–317. AAAI Press, 2006.

44   Yan Zhang and Yuanlin Zhang. Epistemic Specifications and Conformant Planning. In Roman Barták, Thomas Leo McCluskey, and Enrico Pontelli, editors, *Proc. 2017 Workshop on Knowledge-based Techniques for Problem Solving and Reasoning (KnowProS 2017)*, 2017.

45 Zhizheng Zhang. Answer Set Programming with Graded Modality. In Marcello Balduccini and Tomi Janhunen, editors, *Proc. 14th Intl. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *LNCS*, pages 205–211. Springer, 2017. `doi:10.1007/978-3-319-61660-5_18`.

46 Zhizheng Zhang, Kaikai Zhao, and Rongcun Cui. ESmodels: An inference engine of epistemic specifications. In *Proc. 25th Intl. Conf. on Tools with Artificial Intelligence (ICTAI 2013)*, pages 769–774. IEEE Computer Society, 2013. `doi:10.1109/ICTAI.2013.118`.

Cumulative Scoring-Based Induction of Default Theories

Farhad Shakerin
The University of Texas at Dallas, Texas, USA
fxs130430@utdallas.edu
 https://orcid.org/0000-0002-1825-0097

Gopal Gupta
The University of Texas at Dallas, Texas, USA
gupta@utdallas.edu

─── **Abstract** ───

Significant research has been conducted in recent years to extend Inductive Logic Programming (ILP) methods to induce a more expressive class of logic programs such as answer set programs. The methods proposed perform an exhaustive search for the correct hypothesis. Thus, they are sound but not scalable to real-life datasets. Lack of scalability and inability to deal with noisy data in real-life datasets restricts their applicability. In contrast, top-down ILP algorithms such as FOIL, can easily guide the search using heuristics and tolerate noise. They also scale up very well, due to the greedy nature of search for best hypothesis. However, in some cases despite having ample positive and negative examples, heuristics fail to direct the search in the correct direction. In this paper, we introduce the FOLD 2.0 algorithm – an enhanced version of our recently developed algorithm called FOLD. Our original FOLD algorithm automates the inductive learning of default theories. The enhancements presented here preserve the greedy nature of hypothesis search during clause specialization. These enhancements also avoid being stuck in local optima – a major pitfall of FOIL-like algorithms. Experiments that we report in this paper, suggest a significant improvement in terms of accuracy and expressiveness of the class of induced hypotheses. To the best of our knowledge, our FOLD 2.0 algorithm is the first heuristic based, scalable, and noise-resilient ILP system to induce answer set programs.

1 Introduction

Statistical machine learning methods produce models that are not comprehensible for humans because they are algebraic solutions to optimization problems such as risk minimization or data likelihood maximization. These methods do not produce any intuitive description of the learned model. Lack of intuitive descriptions makes it hard for users to understand and verify the underlying rules that govern the model. Also, these methods cannot produce a justification for a prediction they compute for a new data sample. Additionally, extending prior knowledge (background knowledge) in these methods, requires the entire model to be relearned by adding new features to its *feature vector*. A feature vector is essentially *propositional* representation of data in statistical machine learning. In case of missing features, statistical methods such as Expectation Maximization (EM) algorithm are applied to fill the

absent feature(s) with an average estimate that would maximize the likelihood of present features. This is fundamentally different from the human thought process that relies on common-sense reasoning. Humans generally do not directly perform probabilistic reasoning in the absence of information. Instead, most of the time human reasoning relies on learning default rules and exceptions.

Default Logic [15] is a *non-monotonic* logic to formalize reasoning with default assumptions. Normal logic programs provide a simple and practical formalism for expressing default rules. A default rule of the form $\frac{\alpha_1 \wedge ... \wedge \alpha_m : \neg\beta_{m+1}, ..., \neg\beta_n}{\gamma}$ can be formalized as the following normal logic program:

$$\gamma \leftarrow \alpha_1, ..., \alpha_m, not\ \beta_{m+1}, ..., not\ \beta_n$$

where γ, αs and βs are positive predicates.

Inductive Logic Programming (ILP) [9] is a sub-field of machine learning that mines data presented in the form of Horn clauses to learn hypotheses also as Horn clauses. However, Horn clause ILP is not expressive enough to induce default theories. Therefore, in order to learn default theories, an algorithm should be able to efficiently deal with *negation-as-failure* and normal logic programs [16].

Many researchers have tried to extend Horn ILP into richer non-monotonic logic formalisms. A survey of extending Horn clause based ILP to non-monotonic logics can be found in the work by Sakama [16]. He also proposes algorithms to learn from the answer set of a *categorical* normal logic program. He extends his algorithms in a framework called *brave* induction [17]. Law et. al. realized that this framework is not expressive enough to induce programs that solve practical problems such as combinatorial problems and proposed the ILASP system [4]. ASPAL [1] system is also an effort in this direction. Both ILASP and ASPAL encode the ILP instance as an ASP program and then they use an ASP solver to perform the exhaustive search of the correct hypothesis. This approach suffers from lack of scalability due to this exhaustive search. More discussion of advantages of our work presented in this paper *vis a vis* these earlier efforts is reported in Section 6.

The previous ILP systems are characterized as either bottom-up or top-down depending on the direction they guide the search. A bottom-up ILP system, such as Progol [10], builds most-specific clauses from the training examples. It is best suited for incremental learning from a few examples. In contrast, a top-down approach, such as the well-known FOIL algorithm [13], starts with the most-general clauses and then specializes them. It is better suited for large-scale datasets with noise, since the search is guided by heuristics [23].

In [20] we introduced an algorithm called FOLD that learns default theories in the form of stratified normal logic programs[1]. The default theories induced by FOLD, as well as the background knowledge used, is assumed to follow the stable model semantics [3]. FOLD extends the FOIL algorithm. FOLD can tolerate noise but it is not sound (i.e., there is no guarantee that the heuristic would always direct the search in the right direction). The *information gain* heuristic used in FOLD (that has been inherited from FOIL), has been extensively compared to other search heuristics in decision-tree induction [7]. There seems to be a general consensus that it is hard to improve the heuristic such that it would always select the correct literal to expand the current clause in specialization. The blame rests mainly on getting stuck in local optima, i.e, choosing a literal producing maximum information gain at a particular step that does not lead to a global optimum.

---

[1] Note that FOLD has been recently extended by us to learn arbitrary answer set programs, i.e., non-stratified ones too [19]; discussion of this extension is beyond the scope of this paper.

Similarly, in multi-relational datasets, a common case is that of a literal that has zero information gain but needs to be included in the learned theory. Heuristics-based algorithms will reject such a literal. Quinlan in [12] introduces *determinate literals* and suggests to add them all at once to the current clause to create a potential path towards a correct hypothesis. FOIL then requires a post pruning phase to remove the unnecessary literals. This approach cannot trivially be extended to the case of default theories where determinate literals may appear in composite *abnormality* predicates and FOIL's language bias simply does not allow negated composite literals.

In this paper we present an algorithm called FOLD 2.0 which avoids being trapped in local optima and adds determinate literals while inducing default theories. We make the following novel contributions:

- We propose a new "cumulative" scoring function which replaces the original scoring function (called *information gain*). Our experiments show a significant improvement in terms of our algorithm's accuracy.
- We also extend FOLD with determinate literals. This extension enables FOLD to learn a broader class of hypotheses that, to the best of our knowledge, no other ILP system is able to induce. Finally, we apply our algorithm in variety of different domains including *kinship* and *legal* as well as UCI benchmark datasets to show how FOLD 2.0, significantly improves our algorithm's predictive power.

Rest of the paper is organized as follows: Section 2 presents background material. Section 3 introduces the FOLD algorithm. Section 4 presents the "cumulative" scoring function and determinate literals in FOLD 2.0. Section 5 presents our experiments and results. Section 6 discusses related research and Section 7 presents conclusions along with future research directions.

2 Background

Our original learning algorithm for inducing answer set programs, called FOLD (First Order Learning of Default rules) [20], is itself an extension of the well known FOIL algorithm. FOIL is a top-down ILP algorithm which follows a *sequential covering* approach to induce a hypothesis. The FOIL algorithm is summarized in Algorithm 1. This algorithm repeatedly searches for clauses that score best with respect to a subset of positive and negative examples, a current hypothesis and a heuristic called *information gain* (IG). The FOIL algorithm learns a target predicate that has to be specified. Essentially, the target predicate appears as the head of the learned goal clause that FOIL aims to learn. A typical *stopping criterion* for the outer loop is determined as the coverage of all positive examples. Similarly, it can be specified as exclusion of all negative examples in the inner loop. The function $covers(\hat{c}, E^+, B)$ returns a set of examples in E^+ implied by the hypothesis $\hat{c} \cup B$.

The inner loop searches for a clause with the highest information gain using a general-to-specific hill-climbing search. To specialize a given clause c, a refinement operator ρ under θ-subsumption [11] is employed. The most general clause is {p($X_1, ..., X_n$) :- true.}, where the predicate p/n is the target and each X_i is a variable. The refinement operator specializes the current clause {h :- $b_1, \ldots, b_n$.}. This is realized by adding a new literal l to the clause, which yields the following: {h :- $b_1, \ldots, b_n$, l}. The heuristic based search uses *information gain*. In FOIL, information gain for a given clause is calculated as follows [8]:

$$IG(L, R) = t \left(log_2 \frac{p_1}{p_1 + n_1} - log_2 \frac{p_0}{p_0 + n_0} \right) \tag{1}$$

Algorithm 1 Overview of the FOIL algorithm.

Input: $goal, B, E^+, E^-$
Output: Hypothesis H
 1: Initialize $H \leftarrow \emptyset$
 2: **while not**(*stopping criterion*) **do**
 3:      $c \leftarrow \{\texttt{goal :- true.}\}$
 4:      **while not**(*stopping criterion*) **do**
 5:          **for** all $c' \in \rho(c)$ **do**
 6:              *compute score*$(E^+, E^-, H \cup \{c'\}, B)$
 7:          **end for**
 8:          let $\hat{c}$ be the $c' \in \rho(c)$ with the best score
 9:          $c \leftarrow \hat{c}$
10:      **end while**
11:      add $\hat{c}$ to H
12:      $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
13: **end while**

---

where L is the candidate literal to add to rule R, p_0 is the number of positive bindings of R, n_0 is the number of negative bindings of R, p_1 is the number of positive bindings of $R + L$, n_1 is the number of negative bindings of $R + L$, t is the number of positive bindings of R also covered by $R + L$.

FOIL handles negated literals in a naive way by adding the literal *not L* to the set of specialization candidate literals for any existing candidate L. This approach leads to learning predicates that do not capture the concept accurately as shown in the following example:

▶ **Example 1.** B, E^+ are background knowledge and positive examples respectively under *Closed World Assumption*, and the target predicate is `fly`.

$$B: \quad \texttt{bird(X) :- penguin(X).} \quad \texttt{bird(tweety).} \quad \texttt{bird(et).}$$
$$\texttt{cat(kitty).} \quad \texttt{penguin(polly).}$$
$$E^+: \quad \texttt{fly(tweety).} \quad \texttt{fly(et).}$$

The FOIL algorithm would learn the following rule:

```
fly(X) :- not cat(X), not penguin(X).
```

which does not yield a constructive definition. The best theory in this example is as follows:

```
fly(X):- bird(X), not penguin(X).
```

which FOIL fails to discover.

3   FOLD Algorithm

The intuition behind FOLD algorithm is to learn a concept in terms of a default and possibly multiple exceptions (and exceptions to exceptions, and so on). Thus, in the bird example given above, we would like to learn the rule that `X` flies if it is a bird and not a penguin, rather than that all non-cats and non-penguins can fly. FOLD tries first to learn the default by specializing a general rule of the form $\{\texttt{goal}(V_1, ..., V_n) \texttt{ :- true.}\}$ with positive literals. As in FOIL, each specialization must rule out some already covered negative examples without

significantly decreasing the number of positive examples covered. Unlike FOIL, no negative literal is used at this stage. Once the IG becomes zero, this process stops. At this point, if any negative example is still covered, they must be either noisy data or exceptions to the current hypothesis. Exceptions are separated from noise via distinguishable patterns in negative examples [21]. In other words, exceptions can be learned by swapping of positive and negative examples and calling the same algorithm recursively. This swapping of positive and negative examples and then recursively calling the algorithm again can continue, so that we can learn exceptions to exceptions, and so on. Each time a rule is discovered for exceptions, a new predicate $ab(V_1, ..., V_n)$ is introduced. To avoid name collisions, FOLD appends a unique number at the end of the string "ab" to guarantee the uniqueness of invented predicates. It turns out that the outlier data samples are covered neither as default nor as exceptions. If outliers are present, FOLD identifies and enumerates them to make sure that the algorithm converges. This ability to separate exceptions from noise allows FOLD (and FOLD 2.0, introduced later) pinpoint noise more accurately. This is in contrast to FOIL, where exceptions and noisy data are clubbed together. Details can be found in [20].

Algorithm 2 shows a high level implementation of the FOLD algorithm. In lines 1-8, function FOLD, serves like the FOIL outer loop. In line 3, FOLD starts with the most general clause (e.g. `fly(X) :- true`). In line 4, this clause is refined by calling the function $SPECIALIZE$. In lines 5-6, set of positive examples and set of discovered clauses are updated to reflect the newly discovered clause.

In lines 9-29, the function $SPECIALIZE$ is shown. It serves like the FOIL inner loop. In line 12, by calling the function ADD_BEST_LITERAL the "best" positive literal is chosen and the best IG as well as the corresponding clause is returned. In lines 13-24, depending on the IG value, either the positive literal is accepted or the EXCEPTION function is called. If, at the very first iteration, IG becomes zero, then a clause that just enumerates the positive examples is produced. A flag called $first\_iteration$ is used to differentiate the first iteration. In lines 26-27, the sets of positive and negative examples are updated to reflect the changes of the current clause. In line 19, the EXCEPTION function is called while swapping E^+ and E^-.

In line 31, the "best" positive literal that covers more positive examples and fewer negative examples is selected. Again, note the current positive examples are really the negative examples and in the EXCEPTION function, we try to find the rule(s) governing the exception. In line 33, FOLD is recursively called to extract this rule(s). In line 34, a new `ab` predicate is introduced and at lines 35-36 it is associated with the body of the rule(s) found by the recurring FOLD function call at line 33. Finally, at line 38, default and exception are combined together to form a single clause.

Now, we illustrate how FOLD discovers the above set of clauses given $E^+ = \{tweety, et\}$ and $E^- = \{polly, kitty\}$ and the goal `fly(X)`. By calling FOLD, at line 2 while loop, the clause $\{$`fly(X) :- true.`$\}$ is specialized. Inside the $SPECIALIZE$ function, at line 12, the literal `bird(X)` is selected to add to the current clause, to get the clause $\hat{c} = $ `fly(X) :- bird(X)`, which happens to have the greatest IG among $\{$`bird,penguin,cat`$\}$. Then, at lines 26-27 the following updates are performed: $E^+ = \{\}$, $E^- = \{polly\}$. A negative example $polly$, a penguin is still covered. In the next iteration, $SPECIALIZE$ fails to introduce a positive literal to rule it out since the best IG in this case is zero. Therefore, the EXCEPTION function is called by swapping the E^+, E^-. Now, FOLD is recursively called to learn a rule for $E^+ = \{polly\}$, $E^- = \{\}$. The recursive call (line 33), returns $\{$`fly(X) :- penguin(X)`$\}$ as the exception. In line 34, a new predicate `ab0` is introduced and at lines 35-37 the clause $\{$`ab0(X) :- penguin(X)`$\}$ is created and added to the set of

---

Algorithm 2 FOLD Algorithm

Input: $target, B, E^+, E^-$
Output: $D = \{c_1, ..., c_n\}$ ▷ defaults' clauses
$\quad AB = \{ab_1, ..., ab_m\}$ ▷ exceptions/abnormal clauses
 1: **function** $\text{FOLD}(E^+, E^-)$
 2: $\quad$ **while** $(|E^+| > 0)$ **do**
 3: $\quad\quad$ $c \leftarrow (target :\text{-} \ true.)$
 4: $\quad\quad$ $\hat{c} \leftarrow \text{SPECIALIZE}(c, E^+, E^-)$
 5: $\quad\quad$ $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
 6: $\quad\quad$ $D \leftarrow D \cup \{\hat{c}\}$
 7: $\quad$ **end while**
 8: **end function**
 9: **function** $\text{SPECIALIZE}(c, E^+, E^-)$
10: $\quad$ **while** $|E^-| > 0 \wedge c.length < max\_rule\_length$ **do**
11: $\quad\quad$ $(c_{def}, \hat{IG}) \leftarrow \text{ADD\_BEST\_LITERAL}(c, E^+, E^-)$
12: $\quad\quad$ **if** $\hat{IG} > 0$ **then**
13: $\quad\quad\quad$ $\hat{c} \leftarrow c_{def}$
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ $\hat{c} \leftarrow \text{EXCEPTION}(c, E^-, E^+)$
16: $\quad\quad\quad$ **if** $\hat{c} == null$ **then**
17: $\quad\quad\quad\quad$ $\hat{c} \leftarrow enumerate(c, E^+)$
18: $\quad\quad\quad$ **end if**
19: $\quad\quad$ **end if**
20: $\quad\quad$ $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
21: $\quad\quad$ $E^- \leftarrow covers(\hat{c}, E^-, B)$
22: $\quad$ **end while**
23: **end function**
24: **function** $\text{EXCEPTION}(c_{def}, E^+, E^-)$
25: $\quad$ $\hat{IG} \leftarrow \text{ADD\_BEST\_LITERAL}(c, E^+, E^-)$
26: $\quad$ **if** $\hat{IG} > 0$ **then**
27: $\quad\quad$ $c\_set \leftarrow \text{FOLD}(E^+, E^-)$
28: $\quad\quad$ $c\_ab \leftarrow generate\_next\_ab\_predicate()$
29: $\quad\quad$ **for each** $c \in c\_set$ **do**
30: $\quad\quad\quad$ $AB \leftarrow AB \cup \{c\_ab :\text{-} \ bodyof(c)\}$
31: $\quad\quad$ **end for**
32: $\quad\quad$ $\hat{c} \leftarrow (headof(c_{def}) :\text{-} \ bodyof(c), \mathbf{not}(c\_ab))$
33: $\quad$ **else**
34: $\quad\quad$ $\hat{c} \leftarrow null$
35: $\quad$ **end if**
36: **end function**

---

invented abnormalities, namely, AB. In line 38, the negated exception (i.e `not ab0(X)`) and the default rule's body (i.e `bird(X)`) are compiled together to form the following theory:

```
fly(X) :- bird(X), not ab0(X).
ab0(X) :- penguin(X).
```

More detailed examples can be found in [20].

■ **Table 1** FOLD Execution to Discover Rule (1).

Literal / Clause	uncle(V1,V2) :- true	uncle(V1,V2) :- male(V1)
parent(V1,V3)	1.44	1.01
parent(V2,V3)	1.06	**1.16**
parent(V3,V1)	1.44	1.01
sibling(V1,V3)	2.27	1.01
sibling(V3,V1)	2.27	1.01
male(V1)	**3.18**	-
female(V2)	0.34	0.50
married(V1,V3)	0.69	0
married(V2,V3)	0.34	0.50
married(V3,V1)	0.69	0
married(V3,V2)	0.34	0.5

4 The FOLD 2.0 Algorithm

4.1 Cumulative Scoring Function

The *kinship* domain is one of the initial successful applications of the FOIL algorithm [13], where the algorithm learns general rules governing social interactions and relations (particularly kinship) from a series of examples. For example, it can learn the "Uncle" relationship, given the background knowledge of "Brother", "Sister", "Father", "Mother", "Husband", "Wife" and some positive and negative examples of the concept. However, if the background knowledge only contains the primitive relationships including "Sibling", "Parent", "Married" and gender descriptors, it fails to discover the correct rule for "Uncle". As an experiment, we used an arbitrarily produced kinship dataset only containing the primitive relationships. The FOIL algorithm produced the following rules:

```
Rule (1)   uncle(A,B) :- male(A), parent(A,_), female(B).
Rule (2)   uncle(A,_) :- male(A), parent(A,B), female(B), sibling(B,_).
```

Similarly, the FOLD algorithm found incorrect rules as follows:

```
Rule (1)   uncle(V1,V2) :- male(V1), parent(V2,V3).
Rule (2)   uncle(V1,V2) :- male(V1), parent(V2,V3), female(V2).
```

Table 1 shows the *information gain* for each candidate literal while discovering Rule (1). At first iteration, the algorithm successfully finds the literal `male(V1)`, because it has the maximum gain ($IG = 3.18$). At second iteration, the literal `parent(V2,V3)` has the highest gain ($IG = 1.16$) and hence is selected. At this point, since the rule does not cover any negative example, the algorithm returns. This example characterizes a case in which the highest score does not correspond to the correct literal. The correct literal at second iteration is `sibling(V1,V3)`, whose information gain is 1.01 and it is less than the maximum.

We observed that neither increasing the number of examples nor changing the scoring function would solve this problem. As an experiment, we replaced the *information gain* with other scoring functions reported in the literature including *Matthews Correlation Coefficient* (MCC), F_β-measure [23] and the FOSSIL [2] scoring measure based on statistical correlation. They all suffer from the same problem.

A key observation is the following: as more literals are introduced, the number of positive and negative examples covered by the current clause shrinks. With fewer examples, the accuracy of heuristic decreases too. In Table 1, `sibling(V1,V3)` should have had the highest score at second iteration. At first iteration, `sibling(V1,V3)` ranks second after `male(V1)`. A simple comparison between the score of `sibling(V1,V3)` and `parent(V2,V3)` shows the former provides better coverage (exclusion) of positive (negative) examples than the latter. But the algorithm is oblivious of this information at the beginning of second iteration as it goes only by magnitude of the scoring function for the current iteration. This score becomes less and less accurate as more literals are introduced and fewer examples remain to cover. If the algorithm could remember that at first iteration, `sibling(V1,V3)` was able to cover/exclude the examples much better than `parent(V2,V3)`, it would prefer `sibling(V1,V3)` over `parent(V2,V3)`.

To concretize this, we propose the idea of keeping a *cumulative score*, i.e., to transfer a portion of past score (if one exists) to the value that the scoring function computes for current iteration. Our experiments suggest that there is not a universal optimal value that would always result in highest accuracy. In other words, the optimal value varies from a dataset to another. Thus, in order to implement the "cumulative score", we introduce a new hyperparameter[2], namely, α, whose value is decided via cross-validation of the dataset being used. In order to compute the score of each literal during the search, the *information gain* is replaced with "cumulative gain".

Formally, let R_i denote the induced rule up until iteration $i + 1$ of FOLD's inner loop execution. Thus, R_0 is the rule `{goal :- true.}`. Also, let $score_i(R_{i-1}, L)$ denote the score of literal L in clause R_{i-1} at iteration i of FOLD's inner loop execution. The "cumulative" score at iteration $i + 1$ for literal l is computed as follows:

$$score_{i+1}(R_i, L) = IG(R_i, L) + \alpha \times score_i(R_{i-1}, L)$$

If $score_i(R_{i-1}, L)$ does not exist, it is considered as zero. Also, if $IG(R_i, L) = 0$, the "cumulative" score from the past is not taken into account. Initially, the cumulative score is considered zero for all candidate literals. Table 2 shows the FOLD 2.0 algorithm's execution to learn "uncle" predicate on the same dataset. With choice of $\alpha = 0.2$, the algorithm is able to discover the following rule: `uncle(V1,V2) :- male(V1), sibling(V1,V3), parent(V3,V2).` It should also be noted that only promising literals are shown in Table 1 and 2. Next, we discuss how our FOLD 2.0 algorithm handles zero information-gain literals.

4.2   Extending FOLD with Determinate Literals

A literal in the body of a clause can serve two purposes: (i) it may contribute directly to the inclusion/exclusion of positive/negative examples respectively; or, (ii) it may contribute indirectly by introducing new variables that are used in the subsequent literals. This type of literal may or may not yield a positive score. Therefore, it is quite likely that our hill-climbing algorithm would miss them. Two main approaches have been used to take this issue into account: *determinate literals* [12] and *lookahead* technique [6]. The latter technique is not of interest to us because it does not preserve the greedy nature of search.

Determinate literals are of the form `r(X,Y)`, where `r/2` is a new literal introduced in the hypothesis' body and Y is a new variable. The literal `r/2` is determinate if, for every value

---

[2] In Machine Learning, a hyperparameter is a parameter whose value is set before the learning process begins.

■ **Table 2** FOLD 2.0 Execution with Cumulative Score.

Literal / Clause	uncle(V1,V2).	uncle(V1,V2):- male(V1)	uncle(V1,V2):-male(V1), sibling(V1,V3)
parent(V1,V3)	1.44	1.30	0
parent(V2,V3)	1.06	1.38	0
parent(V3,V2)	0	0	**2.49**
parent(V3,V1)	1.44	1.30	0
parent(V2,V4)	-	-	0.83
sibling(V1,V3)	2.27	**1.47**	-
sibling(V3,V1)	2.27	1.47	1.15
male(V1)	**3.18**	-	-
female(V2)	0.34	0.57	0
female(V3)	-	-	1.15
married(V1,V3)	0.69	0	0
married(V2,V3)	0.34	0.57	0
married(V3,V1)	0.69	0	0
married(V3,V2)	0.34	0.57	0
married(V2,V4)	-	-	1.24
married(V4,V2)	-	-	1.24

of X, there is at most one value for Y, when the hypothesis' head is unified with positive examples. Determinate literals are not contributing directly to the learning process, but they are needed as they influence the literals chosen in the future. Since their inclusion in the hypothesis is computationally inexpensive, the FOIL algorithm adds them to the hypothesis simultaneously. In Section 2 we showed why the naive handling of negation in FOIL would not work in case of non-monotonic logic programs. Another issue with FOIL's handling of negated literals arises when we deal with *determinate literals*. Whenever a combination of a determinate and a gainful literal attempts to find a pattern in the negative examples, the FOIL algorithm fails to discover it because FOIL prohibits conjunction of negations in its language bias to prevent search space explosion. However, by introducing the abnormality predicates and recursively swapping positive and negative examples, FOLD makes inductive learning of such default theories possible.

The FOLD algorithm always selects literals with positive information gain first. Next, if some negative examples are still covered and no gainful literal exists, it would swap the current positive examples with current negative examples and recursively calls itself to learn the exceptions. To accommodate determinate literals in FOLD 2.0, we make the following modification to FOLD. In the SPECIALIZE function, right before swapping the examples and making the recursive call to the FOLD function (see Algorithm 3), we try the current rule for a second time. By adding determinate literals and iterating again, we hope that a positive gainful literal will be discovered. Next, if that choice does not exclude the negative examples, FOLD 2.0 swaps the examples and recursively calls itself. A nice property of this recursive approach is that the determinate literals might be added inside the exception finding routine to induce a composite abnormality predicate. Neither FOIL nor FOLD could induce such hypotheses. The following example shows how this is handled in the FOLD 2.0 algorithm.

▶ **Example 2.** In United States immigration system, student visa holders are classified as F1(student) and F2(student's spouse). F1 and F2 status remains valid until a student graduates. The spouse of such an individual maintains a valid status, as long as that individual is a student. Table 3 shows a dataset for this domain. In this dataset, it turns out that $married(V1, V2)$ is a determinate literal and essential to the final hypothesis. If we

---

Algorithm 3 Overview of FOLD 2.0 Algorithm + Determinate Literals

---

Input: $goal, B, E^+, E^-$
Output: $D = \{c_1, ..., c_n\}, AB = \{ab_1, ..., ab_m\}$
 1: **function** SPECIALIZE(c, E^+, E^-)
 2:     $determinate\_added \leftarrow false$
 3:     **while** $(size(E^-) > 0)$ **do**
 4:         $(c_{def}, \hat{IG}) \leftarrow$ ADD_BEST_LITERAL(c, E^+, E^-)
 5:         **if** $\hat{IG} \leq 0$ **then**
 6:             **if** $determinate\_added == false$ **then**
 7:                 $c \leftarrow ADD\_DETERMINATE\_LITERAL(c, E^+, E^-)$
 8:                 $determinate\_added \leftarrow true$
 9:             **else**
10:                 $\hat{c} \leftarrow$ EXCEPTION(c, E^-, E^+)
11:                 **if** $\hat{c} == null$ **then**
12:                     $\hat{c} \leftarrow enumerate(c, E^+)$
13:                 **end if**
14:             **end if**
15:         **else**
16:             $E^+ \leftarrow E^+ \setminus covers(\hat{c}, E^+, B)$
17:             $E^- \leftarrow covers(\hat{c}, E^-, B)$
18:         **end if**
19:     **end while**
20: **end function**

---

run the FOLD 2.0 algorithm, it would produce the following hypothesis:

```
Default rule(1):   valid(V1) :- student(V1), not ab1(V1).
Default rule(2):   valid(V1) :- class(V1,f2), not ab2(V1).
Exception(1) :     ab1(V1) :- graduated(V1).
Exception(2) :     ab2(V1) :- married(V1,V2), graduated(V2).
```

In this example default rule(1) as well as rules for its exception are discovered first. This rule (rule(1)) takes care of students who have not graduated yet. Then, while discovering rule(2), after choosing the only gainful literal, i.e., `class(V1,f2)`, the algorithm is recursively called on the exception part. It turns out that there is no gainful literal that covers the now positive examples (previously negative examples). The only determinate literal in this example is `married(V1,V2)`, which is added at this point. This is followed by FOLD 2.0 finding a gainful literal, i.e., `graduated(V2)`, and then returning the default rule(2). At this point, all positive examples are covered and the algorithm terminates. Default rule(2) takes care of the class of F2 visa holders whose spouse is a student unless they have graduated. The Algorithm 3 shows the changes necessary to the FOLD algorithm in order to handle determinate literals.

5   Experiments and results

In this section we present our experiments on UCI benchmark datasets [5]. Table 4 summarizes an accuracy-based comparison between Aleph [21], FOLD [20] and FOLD 2.0. We report a significant improvement just by picking up an optimal value for α via cross-validation. In these experiments we picked $\alpha \in \{0, 0.2, 0.5, 0.8, 1\}$.

Table 3 Valid Student Visa Dataset.

B				E$^+$	E$^-$
class(p1,f2).	class(p7,f1).	student(p3).	married(p1,p2).	valid(p1).	valid(p4).
class(p2,f1).	class(p8,f1).	student(p4).	married(p5,p6).	valid(p2).	valid(p5).
class(p3,f1).	class(p9,f2).	student(p6).	married(p9,p10).	valid(p3).	valid(p6).
class(p4,f1).	class(p10,f1).	student(p7).	graduated(p4).	valid(p7).	valid(p8).
class(p5,f2).		student(p8).	graduated(p6).	valid(p9).	
class(p6,f1).		student(p10).	graduated(p8).	valid(p10).	

ILP algorithms usually achieve lower accuracy compared to state-of-the-art statistical methods such as SVM. But in case of "Post Operative" dataset, for instance, our FOLD 2.0 algorithm outperforms SVM, whose accuracy is only 67% [18]. Next, we show in detail how FOLD 2.0 achieves higher accuracy in case of Moral Reasoner dataset. Moral Reasoner is a rule-based model that qualitatively simulates moral reasoning. The model was intended to simulate how an ordinary person, down to about age five, reasons about harm-doing. The Horn-clause theory has been provided along with 202 instances that were used in [22]. The top-level predicate to predict is `guilty/1`. We encourage the interested reader to refer to [5] for more details. Our goal is to learn the moral reasoning behavior from examples and check how close it is to the Horn-clause theory reported in [22].

First, we run FOLD 2.0 algorithm with $\alpha = 0$. This literally turns off the "cumulative score" feature. The algorithm would return the following set of rules:

```
Rule(1) guilty(V1) :- severity(V1,1), external_force(V1,n),
                  benefit_victim(V1,0),intervening_contribution(V1,n).
Rule(2) guilty(V1) :- severity(V1,1), external_force(V1,n),
                  benefit_victim(V1,0),foresee_intervention(V1,y).
Rule(3) guilty(V1) :- someone_else_cause_harm(V1,y),achieve_goal(V1,n),
                  control_perpetrator(V1,y), foresee_intervention(V1,n).
```

In the original Horn clause theory [22] there are two theories for being guilty: i) blame-worthy, ii) vicarious_blame. The following rules for `blame_worthy(X)` are reproduced from [22]:

```
blameworthy(X):- responsible(X), not justified(X), severity_harm(X,H),
                 benefit_victim(X,L), H > L.
responsible(X):- cause(X), not accident(X), external_force(X,n),
                 not intervening_cause(X).
intervening_cause(X) :- intervening_contribution(X,y),
                        forsee_intervention(X).
```

Rule(1) and Rule(2), that FOLD 2.0 learns, together build the blameworthy definition of the original theory. The predicates `severity_harm` and `benefit_victim` occur in Rule(1) and Rule(2). It should be noted that due to the nature of the provided examples, FOLD 2.0 comes up with a more specific version compared to the original theory reported in [22]. In addition, instead of learning the predicate `responsible(X)`, our algorithm learns its body literals. The predicate `cause(X)` does not appear in the hypothesis because it is implied by all positive and negative examples, one way or another. The predicate `not intervening_cause(X)` appears in our hypothesis due to application of *De Morgan's law* and flipping yes and no in

■ **Table 4** Performance Results on UCI Benchmark Datasets.

Dataset	Accuracy (%)			α
	Aleph	FOLD	FOLD 2.0	
Labor	85	94	100	0.5
Post-op	62	65	78	1
Bridges	89	90	93	1
Credit-g	70	78	84	0.5
Moral	96	96	100	0.2

the second arguments. The rest of the guilty cases fall into the category of `vicarious_blame` below:

```
vicarious_blame(X):- vicarious(X),      vicarious(X) :-
        not justified(X),                   someone_else_cause_harm(X,y),
        severity_harm(X,H),                 outrank_perpetrator(X,y),
        benefit_victim(X,L), H > L.         control_perpetrator(X,y).
```

There is a discrepancy in Rule(3), compared to the corresponding `vicarious_blame` in the original theory. However, by setting the cumulative score parameter $\alpha = 0.2$, FOLD 2.0 would produce the following set of rules:

```
Rule(1):                                 Rule(2):
guilty(V1) :- severity_harm(V1,1),           guilty(V1) :-
    external_force(V1,n),                    severity_harm(V1,1),
    benefit_victim(V1,0),                    external_force(V1,n),
    intervening_contribution(V1,n).          benefit_victim(V1,0),
                                             foresee_intervention(V1,y).
Rule(3):
guilty(V1) :- severity_harm(V1,1), benefit_victim(V1,0),
              someone_else_cause_harm(V1,y),outrank_perpetrator(V1,y),
              control_perpetrator(V1,y).
```

Rule(1) and Rule(2) are generated in FOLD 2.0 as before. However, Rule(3) perfectly matches that of the original theory which our FOLD algorithm would have not been able to discover without "cumulative score". Note that the cumulative score heuristics is quite general and can be used to enhance any machine learning algorithm that relies on the concept of information gain. In particular, it can be used to improve the FOIL algorithm itself.

6 Related Work

A survey of non-monotonic ILP work can be found in [16]. Sakama also introduces an algorithm to induce rules from answer sets. His approach may yield premature generalizations that include redundant negative literals. We skip the illustrative example due to lack of space, however, the reader can refer to [20]. ASPAL [1] is another ILP system capable of producing non-monotonic logic programs. It encodes ILP problem as an ASP program. XHAIL [14] is another ILP system that heavily uses abductive logic programming to search for the best hypothesis. Both ASPAL and XHAIL systems can only learn hypotheses that have a single stable model. ILASP [4] is the successor of ASPAL. It can learn hypotheses that have multiple stable models by employing brave induction [17]. All of these systems perform an exhaustive search to find the correct hypothesis. Therefore, they are not scalable to real-life datasets. They also have a restricted language bias to avoid the explosion of

search space of hypotheses. This overly restricted language bias does not allow them to learn new predicates, thus keeping them from inducing sophisticated default theories with nested or composite abnormalities that our FOLD 2.0 algorithm can induce. For instance consider the following example, a default theory with abnormality predicate represented as conjunction of two other predicates, namely `s(X) and r(X)`.

```
p(X) :- q(X), not ab(X).
ab(X) :- s(X), r(X).
```

Our algorithm has advantages over the above mentioned systems: It follows a greedy top-down approach and therefore it is better suited for larger datasets and noisy data. Also, it can invent new predicates [19], distinguish noise from exceptions, and learn nested levels of exceptions.

7 Conclusion and Future Work

In this paper we presented *cumulative score*-based heuristic to guide the search for best hypothesis in a top-down non-monotonic ILP setting. The main feature of this heuristic is that it avoids being trapped in local optima during clause specialization search. This results in significant improvement in the accuracy of induced hypotheses. This heuristic is quite general and can be used to enhance any machine learning algorithm that relies on the concept of information gain. In particular, it can be used to improve the FOIL algorithm itself. We used it in this paper to extend our FOLD algorithm to obtain the FOLD 2.0 algorithm for learning answer set programs. FOLD 2.0 performs significantly better than our FOLD algorithm [20], where the FOLD algorithm itself produces better results than previous systems such as FOIL and ALEPH. We also showed how determinate literals can be adapted to identifying patterns in negative examples after the swapping of positive and negative examples in FOLD. Note that while determinate literals were introduced in the FOIL algorithm, their use in FOIL was limited to only positive literals. Generalizing the use of determinate literals in FOLD 2.0, enables us to induce hypotheses that no other non-monotonic ILP system is able to induce.

There are three main avenues for future work: (i) handling large datasets using methods similar to QuickFoil [23]. In QuickFoil, all the operations of FOIL are performed in a database engine. Such an implementation, along with pruning techniques and query optimization tricks, can make the FOLD 2.0 training phase much faster. (ii) FOLD 2.0 learns function-free answer set programs. We plan to investigate extending the language bias towards accommodating functions. (iii) Combining statistical methods such as SVM with FOLD 2.0 to increase accuracy as well as providing explanation for the behavior of models produced by SVM.

References

1   Domenico Corapi, Alessandra Russo, and Emil Lupu. Inductive Logic Programming in Answer Set Programming. In Stephen Muggleton, Alireza Tamaddoni-Nezhad, and Francesca A. Lisi, editors, *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, volume 7207 of *Lecture Notes in Computer Science*, pages 91–97. Springer, 2011. `doi:10.1007/978-3-642-31951-8_12`.

2   Johannes Fürnkranz. FOSSIL: A robust relational learner. In Francesco Bergadano and Luc De Raedt, editors, *Machine Learning: ECML-94*, pages 122–137, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

3 Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

4 Mark Law, Alessandra Russo, and Krysia Broda. Inductive Learning of Answer Set Programs. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *LNCS*, pages 311–325. Springer, 2014. `doi: 10.1007/978-3-319-11558-0_22`.

5 M. Lichman. UCI machine learning repository, 2013. URL: `http://archive.ics.uci.edu/ml`.

6 Marco Lippi, Manfred Jaeger, Paolo Frasconi, and Andrea Passerini. Relational information gain. *Machine Learning*, 83(2):219–239, May 2011. `doi:10.1007/s10994-010-5194-7`.

7 John Mingers. An Empirical Comparison of Selection Measures for Decision-Tree Induction. *Machine Learning*, 3:319–342, 1989.

8 Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

9 Stephen Muggleton. Inductive Logic Programming. *New Generation Comput.*, 8(4):295–318, 1991. `doi:10.1007/BF03037089`.

10 Stephen Muggleton. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4):245–286, 1995. `doi:10.1007/BF03037227`.

11 G. D. Plotkin. A further note on inductive generalization, In machine Intelligence, volume 6, pages 101-124, 1971.

12 J. R. Quinlan. Determinate Literals in Inductive Logic Programming. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'91, pages 746–750, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. URL: `http://dl.acm.org/citation.cfm?id=1631552.1631572`.

13 J. Ross Quinlan. Learning Logical Definitions from Relations. *Machine Learning*, 5:239–266, 1990. `doi:10.1007/BF00117105`.

14 Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3):329–340, 2009. Special Issue: Abduction and Induction in Artificial Intelligence. `doi:10.1016/j.jal.2008.10.007`.

15 Raymond Reiter. A Logic for Default Reasoning. *Artif. Intell.*, 13(1-2):81–132, 1980. `doi:10.1016/0004-3702(80)90014-4`.

16 Chiaki Sakama. Induction from answer sets in nonmonotonic logic programs. *ACM Trans. Comput. Log.*, 6(2):203–231, 2005. `doi:10.1145/1055686.1055687`.

17 Chiaki Sakama and Katsumi Inoue. Brave induction: a logical framework for learning from incomplete information. *Machine Learning*, 76(1):3–35, 2009. `doi:10.1007/s10994-009-5113-y`.

18 Mathieu Serrurier and Henri Prade. Introducing possibilistic logic in ILP for dealing with exceptions. *Artificial Intelligence*, 171:939–950, 2007.

19 Farhad Shakerin and Gopal Gupta. Technical Report, Heuristic Based Induction of Answer Set Programs: From Default theories to combinatorial problems, http://arxiv.org/abs/1802.06462, 2018. URL: `http://arxiv.org/abs/1802.06462`.

20 Farhad Shakerin, Elmer Salazar, and Gopal Gupta. A new algorithm to automate inductive learning of default theories. *TPLP*, 17(5-6):1010–1026, 2017. `doi:10.1017/S1471068417000333`.

21 Ashwin Srinivasan, Stephen Muggleton, and Michael Bain. Distinguishing Exceptions from Noise in Non-Monotonic Learning,In S. Muggleton and K. Furukawa, editors, Second International Inductive Logic Programming Workshop (ILP92), 1996.

22   James Wogulis. Revising Relational Domain Theories. In Lawrence Birnbaum and Gregg
        Collins, editors, *Proceedings of the Eighth International Workshop (ML91), Northwestern
        University, Evanston, Illinois, USA*, pages 462–466. Morgan Kaufmann, 1991.
23   Qiang Zeng, Jignesh M. Patel, and David Page. QuickFOIL: Scalable Inductive Logic Pro-
        gramming. *Proc. VLDB Endow.*, 8(3):197–208, November 2014. `doi:10.14778/2735508.`
        `2735510`.

Introspecting Preferences in Answer Set Programming

Zhizheng Zhang[1]

School of Computer Science and Engineering, Southeast University, Nanjing, China
seu_zzz@seu.edu.cn

Abstract

This paper develops a logic programming language, ASP^{EP}, that extends answer set programming language with a new epistemic operator $\succcurlyeq_x$ where $x \in \{\sharp, \supseteq\}$. The operator are used between two literals in rules bodies, and thus allows for the representation of introspections of preferences in the presence of multiple belief sets: $G \succcurlyeq_\sharp F$ expresses that G is preferred to F by the cardinality of the sets, and $G \succcurlyeq_\supseteq F$ expresses G is preferred to F by the set-theoretic inclusion. We define the semantics of ASP^{EP}, explore the relation to the languages of strong introspections, and study the applications of ASP^{EP} by modeling the Monty Hall problem and the principle of majority.

1 Introduction

Preferences have extensively been studied in disciplines such as economy, operations research, psychology, philosophy, and artificial intelligence as showed in [8], [18], [2], [15], and [7] etc. In [25], von Wright defined preference as a relation between states of affairs. In formal logical languages, states of affairs are typically represented as propositions. Follow this tradition, one of the important directions in artificial intelligence is the logical representation and reasoning of preferences. Many extensions of the languages of answer set programming (ASP) have been developed for handling preferences due to the strong power of ASP in expressing defaults. Those languages provide elegant methodologies for modeling the intractable problems with defaults and preferences. Examples include the ordered logic programming [20], the logic programming with ordered disjunction [4], the answer set optimization [5][3], the prioritized logic programming [19], the CR-prolog [1], the possibilistic answer set programming [17] etc. The preferences handled in those answer set programs are used to evaluate the preferred answer sets via specifying the precedence over the rules or the literals in rules heads.

Different from the above answer set programming paradigms with preferences, our purpose in this paper is to represent introspections of preferences over propositions in the presence of multiple belief sets by proposing a new epistemic operator $\succcurlyeq_x$ where $x \in \{\sharp, \supseteq\}$. For propositions F and G, $F \succcurlyeq_\sharp G$ expresses that F is true in more belief sets than G, and can be read as *"F is more possible than G"*. And $F \succcurlyeq_\supseteq G$ expresses that F is always true in the belief sets where G is true, which tells *"F is antecedent to G"* or *"F is true whenever G is true"* etc. We first demonstrate this motivation using an example from our family life.

■ **Table 1** Combo of Attractions.

(a) Packages Information

Package	Attractions	Ages
1	a_1	Kids,teens
	a_2	Adults
	a_3	teens
2	b_1	All
	b_2	Adults
	b_3	Kids
3	c_1	All
	c_2	teens
	c_3	Kids

(b) Possible Combinations of Attractions]

Package	Combinations	Age Interest
1	$\{a_1,a_2\}$	All
	$\{a_1,a_3\}$	Kids,teens
	$\{a_2,a_3\}$	Adults,teens
2	$\{b_1,b_2\}$	All
	$\{b_1,b_3\}$	All
	$\{b_2,b_3\}$	Adults,Kids
3	$\{c_1,c_2\}$	All
	$\{c_1,c_3\}$	All
	$\{c_2,c_3\}$	teens,Kids

▶ **Example 1.** Consider three discount packages offered by an amusement resort as showed in Table 1(a)[2]. Each of them contains three attractions but only two of them are available. A family is allowed to buy at most one package in advance, and may determine which two attractions to choose according to the actual situations, such as the waiting time, physical situation, when they are in the resort. For instance, a family with a kid child and a teenage boy decide which package to buy by the following criteria: (1). *The family prefer the package that promises more opportunities for the kid child;* (2). *The parents request that their teenage boy has an attraction to visit whenever they visit an attraction*[3].

Directly, the packages information allow the family to have nine possible combinations of attractions as showed in the table 1(b).

And the family can have the following three conclusions via simple counting.

(i) Both package 2 and package 3 provide more opportunities for the kid child than package 1.

(ii) Both package 1 and package 3 guarantee that the teenage boy has an attraction to visit whenever the parents visit an attraction.

(iii) By (i) and (ii), Package 3 should be the favorite package for the family.

It is easy to get the combinations by encoding the packages information and the purchase requirements in a logic program Π_{ep} containing the following rules:

---

[2] In the tables, "All" means that there is no age limitation.
[3] To avoid the boy running around without parents.

$1\{package(1); package(2); package(3)\}1$

$2\{attraction(a_1); attraction(a_2); attraction(a_3)\}2 \leftarrow package(1)$

$2\{attraction(b_1); attraction(b_2); attraction(b_3)\}2 \leftarrow package(2)$

$2\{attraction(c_1); attraction(c_2); attraction(c_3)\}2 \leftarrow package(3)$

$age(kids) \leftarrow attraction(a_1)$

$age(adults) \leftarrow attraction(a_2)$

$age(teens) \leftarrow attraction(a_3)$

$age(all) \leftarrow attraction(b_1)$

$age(adults) \leftarrow attraction(b_2)$

$age(kids) \leftarrow attraction(b_3)$

$age(all) \leftarrow attraction(c_1)$

$age(teens) \leftarrow attraction(c_2)$

$age(kids) \leftarrow attraction(c_3)$

$age(kids) \leftarrow age(all)$

$age(teens) \leftarrow age(all)$

$age(adults) \leftarrow age(all)$

$age\_interest(X, Y) \leftarrow package(X), age(Y).$

that has exactly nine answer sets which correspond to the nine possible combinations in Table 1. We now expect to expand Π_{ep} by rules that is able to intuitively represent the criteria such that the result program is able to give the conclusions as showed in (i),(ii), and (iii). It is easy to see, for achieving the above goal, our representation and reasoning system should have an introspective ability that is able to look at the preferences over the beliefs with regard to those belief sets/answer sets.

Specifically, this paper will address the issue of introspection of preferences illustrated in the above example. We develop a logic programming language, ASP^{EP}, that extends the answer set programming language with a new epistemic operator $\succcurlyeq_x$ where $x \in \{\sharp, \supseteq\}$. In ASP^{EP}, the operator is used between two literals in rules bodies, and thus allows for the representation of introspections of preferences. Consider rules $r_\sharp$:

$prefer(X, Y, kid) \leftarrow age\_interest(X, kids) \succcurlyeq_\sharp age\_interest(Y, kids),$

$package(X), package(Y)$

and $r_\subseteq$:

$request(X) \leftarrow age\_interest(X, teens) \succcurlyeq_\supseteq age\_interest(X, adults), package(X)$

They are able to represent the criteria (1) and (2) in the motivation example respectively.

The rest of the paper is organized as follows. In the next section, we review the basic principles underlying the answer set semantics of logic programs. In section 3, we introduce syntax and semantics of ASP^{EP}. In section 4, we consider the relationship between ASP^{EP} and the strong introspection specification languages. In section 6, we explore the applications of ASP^{EP}. We conclude in section 7 with some further discussion.

2   Answer Set Programming

Throughout this paper, we assume a finite first-order signature σ that contains no function constants of positive arity. There are finitely many Herbrand interpretations of σ, each of which is finite as well. We follow the description of ASP from [14]. A logic program over σ is a collection of rules of the form

$$l_1 \ or \ ... \ or \ l_k \leftarrow l_{k+1}, ..., l_m, not \ l_{m+1}, ..., not \ l_n$$

where the ls are literals of σ, *not* is called negation as failure, *or* is epistemic disjunction. The left-hand side of a rule is called the *head* and the right-hand side is called the *body*. A rule is called a fact if its body is empty and its head contains only one literal, and a rule is called a denial if its head is empty. A logic program is called ground if it contains no variables. [14] intuitively interprets that an answer set associated with a ground logic program is a set of beliefs (collection of ground literals) and is formed by a reasoner guided by three principles:

- *Rule's Satisfiability principle*: Believe in the head of a rule if you believe in its body.
- *Consistency principle*: Do not believe in contradictions.
- *Rationality Principle*: Believe nothing you are not forced to believe.

The definition of the answer set is extended to any non-ground program by identifying it with the ground program obtained by replacing every variable with every ground term of σ. It is worthy noting that $\top$ can be removed if it is in the body of a rule, the rule can be removed from the program if $\bot$ is in its body.

3   The ASP$^{\text{EP}}$ Language

3.1   Syntax

An ASP$^{\text{EP}}$ program Π is a set of rules of the form

$$l_1 \ or \ ... \ or \ l_k \leftarrow e_1, ..., e_m, s_1, ..., s_n.$$

where $k \geq 0$, $m \geq 0$, $n \geq 0$, the ls are literals in first order logic language and are called objective literals here, es are extended literals which are 0-place connectives $\top$ and $\bot$, or objective literals possibly preceded by *not*, ss are subjective literals of the form $e \succcurlyeq_x e'$ or $e \not\succcurlyeq_x e'$ where e and e' are extended literals and $x \in \{\sharp, \supseteq\}$. The left-hand side of a rule is called the *head* and the right-hand side is called the *body*. As in usual logic programming, a rule is called a fact if its body is empty and its head contains only one literal, and a rule is called a denial if its head is empty. We use $head(r)$ to denote the set of objective literals in the head of a rule r and $body(r)$ to denote the set of extended literals and subjective literals in the body of r. Sometimes, we use $head(r) \leftarrow body(r)$ to denote a rule r. The positive body of a rule r is composed of the extended literals containing no *not* in its body. We use $body^+(r)$ to denote the positive body of r. r is said to be *safe* if each variable in it appears in the positive body of the rule. We will use $sl(\Pi)$ to denote the set of subjective literals appearing in Π.

It is clear that an ASP$^{\text{EP}}$ program containing no subjective literals is a disjunctive logic program that can be dealt with by ASP solvers like DLV [9], CLASP [10].

It is worthy of noting that, for convenient description, we will use $e \succ_x e'$ to denote the strict preference that can be expressed by the conjunction of $e \succcurlyeq_x e'$ and $e' \not\succcurlyeq_x e$, and use $e \approx_x e'$ to denote the preferential indifference that can be expressed by the conjunction of

$e \not\succ_x e'$ and $e' \not\succ_x e$, and use $e \equiv_x e'$ to denote the preferential equivalence that can be expressed by the conjunction of $e \succcurlyeq_x e'$ and $e' \succcurlyeq_x e$.

3.2 Semantics

We will restrict our definition of the semantics to ground programs. However, we admit rule schemata containing variables bearing in mind that these schemata are just convenient representations for the set of their ground instances. In the following definitions, l is used to denote a ground objective literal, e is used to denote a ground extended literal, and s is used to denote a ground subjective literal.

3.2.1 Satisfiability

Let W be a non-empty collection of consistent sets of ground objective literals, (W, w) is a pointed ASP^{EP} structure of W where $w \in W$. W is a model of a program Π if for each rule r in Π, r is satisfied by every pointed ASP^{EP} structure of W. The notion of satisfiability denoted by $\models_{\text{ep}}$ is defined below.

- $(W, w) \models_{\text{ep}} \top$
- $(W, w) \not\models_{\text{ep}} \bot$
- $(W, w) \models_{\text{ep}} l$ if $l \in w$
- $(W, w) \models_{\text{ep}} not\ l$ if $l \notin w$
- $(W, w) \models_{\text{ep}} e \succcurlyeq_\sharp e'$ if $|\{w \in W : (W, w) \models_{\text{ep}} e\}| \geq |\{v \in W : (W, v) \models_{\text{ep}} e'\}|$
- $(W, w) \models_{\text{ep}} e \succcurlyeq_\supseteq e'$ if $\{w \in W : (W, w) \models_{\text{ep}} e\} \supseteq \{v \in W : (W, v) \models_{\text{ep}} e'\}$
- $(W, w) \models_{\text{ep}} e \not\succcurlyeq_x e'$ if $(W, w) \not\models_{\text{ep}} e \succcurlyeq_x e'$, $x \in \{\sharp, \supseteq\}$

Then, for a rule r in Π, $(W, w) \models_{\text{ep}} r$ if

- $\exists l \in head(r)$: $(W, w) \models_{\text{ep}} l$, or
- $\exists t \in body(r)$: $(W, w) \not\models_{\text{ep}} t$.

The satisfiability of a subjective literal does not depend on a specific belief set w in W, hence we can simply write $W \models_{\text{ep}} s$ if $(W, w) \models_{\text{ep}} s$ and say the subjective literal s is satisfied by W, and we can simply write $W \not\models_{\text{ep}} s$ if $(W, w) \not\models_{\text{ep}} s$ and say the subjective literal s is not satisfied by W.

We consider the properties of the above satisfiability by some axioms of the strict preference relation proposed by von Wright in [25]. Let W be a non-empty collection of consistent sets of ground objective literals, the following properties of the satisfiability $\models_{ep}$ hold.

- $\succ_x$ Asymmetry. $W \models_{\text{ep}} e \succ_x e' \Longrightarrow W \models_{\text{ep}} e' \not\succ_x e$
- $\succ_\sharp$ Inescapability. $W \models_{\text{ep}} e \succ_\sharp e', W \models_{\text{ep}} e'' \not\succ_\sharp e' \Longrightarrow W \models_{\text{ep}} e \succ_\sharp e''$
- $\succ_x$ Transitivity. $W \models_{\text{ep}} e \succ_x e', W \models_{\text{ep}} e' \succ_x e'' \Longrightarrow W \models_{\text{ep}} e \succ_x e''$
- $\succ_x$ Irreflexivity. $W \models_{\text{ep}} e \not\succ_x e$
- $\approx_x$ Reflexivity. $W \models_{\text{ep}} e \approx_x e$
- $\approx_x$ Symmetry. $W \models_{\text{ep}} e \approx_x e' \Longrightarrow W \models_{\text{ep}} e' \approx_x e$
- $\approx_\sharp$ Transitivity. $W \models_{\text{ep}} e \approx_\sharp e', W \models_{\text{ep}} e' \approx_\sharp e'' \Longrightarrow W \models_{\text{ep}} e \approx_x e''$
- $\succ_\sharp$ R-Analogy. $W \models_{\text{ep}} e \succ_\sharp e', W \models_{\text{ep}} e' \approx_\sharp e'' \Longrightarrow W \models_{\text{ep}} e \succ_\sharp e''$
- $\succ_\sharp$ L-Analogy. $W \models_{\text{ep}} e \approx_\sharp e', W \models_{\text{ep}} e' \succ_\sharp e'' \Longrightarrow W \models_{\text{ep}} e' \succ_\sharp e''$

where $x \in \{\sharp, \supseteq\}$.

In addition, let W be a non-empty collection of consistent sets of ground objective literals, it is easy to find that

- $W \models_{\text{ep}} e \succcurlyeq_x e$
- $W \models_{\text{ep}} \top \succcurlyeq_x e$

- $W \models_{\text{ep}} e \succcurlyeq_x \bot$
- $W \models_{\text{ep}} e \not\succcurlyeq_\supseteq e^{not}$

where e^{not} is l if e is *not* l, and e^{not} is *not* l if e is l, and $\top^{not}$ is $\bot$, and $\bot^{not}$ is $\top$.

3.2.2   World Views

We first give the definition of *candidate world view* for disjunctive logic programs and arbitrary ASP^{EP} programs respectively. Then, we define *world view* for ASP^{EP} programs by presenting a minimizing preferences principle.

▶ **Definition 2.** Let Π be a disjunctive logic program, the candidate world view of Π is the non-empty set of all its answer sets, written as $AS(\Pi)$.

▶ **Definition 3.** Let Π be an arbitrary ASP^{EP} program, and W is a non-empty collection of consistent sets of ground objective literals in the language of Π, we use Π^W to denote the disjunctive logic program obtained by removing the epistemic operators using the following reduct laws
1. removing from Π all rules containing subjective literals not satisfied by W.
2. removing all other occurrences of subjective literals of the form $e \succcurlyeq_x e$ or $\top \succcurlyeq_x e$ or $e \succcurlyeq_x \bot$ or $e \not\succcurlyeq_\supseteq e^{not}$.
3. replacing all other occurrences of subjective literals of the form $e \succcurlyeq_x \top$ by e.
4. replacing all other occurrences of subjective literals of the form $\bot \succcurlyeq_x e$ by e^{not}.
5. replacing other occurrences of subjective literals of the form $e_1 \succcurlyeq_x e_2$ or $e_1 \not\succcurlyeq_x e_2$ by four conjunctions e_1, e_2, and e_1^{not}, e_2, and e_1, e_2^{not}, and e_1^{not}, e_2^{not} respectively.
where e^{not} is l if e is *not* l, and e^{not} is *not* l if e is l, and $\top^{not}$ is $\bot$, and $\bot^{not}$ is $\top$. Then, W is a candidate world view of Π if W is a candidate world view of Π^W.

We use $cwv(\Pi)$ to denote the set of candidate world views of an ASP^{EP} program Π. Π^W is said to be the *reduct* of Π with respect to W. Such a reduct process eliminates subjective literals so that the belief sets in the model are identified with the answer sets of the program obtained by the reduct process. The intuitive meanings of the reduct laws can be described as follows:
- The first reduct law directly comes from the notion of Rule Satisfiability and Rationality Principle in answer set programming which means if a rule's body cannot be satisfied (believed in), the rule will contribute nothing;
- The second reduct law stems from the fact $e \succcurlyeq_x e$ and $\top \succcurlyeq_x e$ and $e \succcurlyeq_x \bot$ and $e \not\succcurlyeq_\supseteq e^{not}$ are tautologies.
- The third reduct law states that, you are forced to believe e with regard to each belief set due to the fact that $e \succcurlyeq_x \top$ implies e is true with regard to each answer set and the *Rationality Principle* in ASP.
- The fourth law states that, you are forced to believe e^{not} with regard to each belief set due to the fact that $\bot \succcurlyeq_x e$ implies e is not true with regard to each answer set.
- The last law states that, both the literals e_1 and e_2 in $e_1 \succcurlyeq_x e_2$ may be true or not with regard to each belief set.

▶ **Definition 4.** Let Π be an arbitrary ASP^{EP} program, and W is a non-empty collection of consistent sets of ground objective literals in the language of Π, W is a world view of Π if it satisfies the conditions below
- $W \in cwv(\Pi)$
- Minimizing preferences principle: $\nexists V \in cwv(\Pi)(\{\bar{s}|s \in sl(\Pi) \wedge V \models_{ep} \bar{s}\} \supset \{\bar{s}|s \in sl(\Pi) \wedge W \models_{ep} \bar{s}\})$

where $\bar{s}$ is $e \succcurlyeq_x e'$ if s is $e \not\succcurlyeq_x e'$, and $\bar{s}$ is $e \not\succcurlyeq_x e'$ if s is $e \succcurlyeq_x e'$.

We use $wv(\Pi)$ to denote the set of world views of an $\mathrm{ASP}^{\mathrm{EP}}$ program Π.

▶ **Definition 5.** Let Π be an $\mathrm{ASP}^{\mathrm{EP}}$ program, a ground objective literal l is true in Π (written by $\Pi \vdash_{\mathrm{ep}} l$) if $\forall W \in wv(\Pi) \forall w \in W((W,w) \models_{\mathrm{ep}} l)$.

▶ **Example 6.** Consider $\Pi = \Pi_{\mathrm{ep}} \cup \{r_\sharp, r_\supseteq\}$ where Π_{ep} and $r_\sharp$ and $r_\supseteq$ are given in section 1. It is easy to see that Π has an unique world view containing nine belief sets:

> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(1),age\_interest(1,kids),$
> $age\_interest(1,adults),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(1),age\_interest(1,kids),$
> $age\_interest(1,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(1),age\_interest(1,adults),$
> $age\_interest(1,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(2),age\_interest(2,kids),$
> $age\_interest(2,adults),age\_interest(2,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(2),age\_interest(2,kids),$
> $age\_interest(2,adults),age\_interest(2,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(2),age\_interest(2,adults),$
> $age\_interest(2,kids),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(3),age\_interest(3,kids),$
> $age\_interest(3,adults),age\_interest(3,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),package(3),age\_interest(3,kids),$
> $age\_interest(3,adults),age\_interest(3,teens),...\}$
> $\{prefer(2,1),prefer(3,1),request(1),request(3),age\_interest(3,teens),\ age\_interest(1,kids),...\}$

Then we have $\Pi \vdash_{\mathrm{ep}} prefer(2,1)$ and $\Pi \vdash_{\mathrm{ep}} prefer(3,1)$ corresponding to the conclusion (i), and $\Pi \vdash_{\mathrm{ep}} request(3)$ and $\Pi \vdash_{\mathrm{ep}} request(1)$ corresponding to the conclusion (ii), and it is easy to verify that if we add to Π another rule:

> $buy(X) \leftarrow request(X), not\ prefer(Y,X), package(X), package(Y), X! = Y$

that states a simple ordered-based choice strategy, then we can get $\Pi \vdash_{\mathrm{ep}} buy(3)$ corresponding to the conclusion (iii) in section 1.

4 Relation to Strong Introspection Specifications

Several languages have been developed by extending the languages of answer set programming (ASP) using epistemic operators to handle introspections. The need for such extension of ASP was early recognized and addressed by Gelfond in [11], where Gelfond proposed an extension of ASP with two modal operators K and M and their negations ($\mathrm{ASP}^{\mathrm{KM}}$). Informally, K p expresses "p is known"(p is true in all belief sets of the agent), M p means "p may be true"(p is true in some belief sets of the agent). It has been proved that $\mathrm{ASP}^{\mathrm{KM}}$ is potential in dealing with some important issues in the field of knowledge representation and reasoning, for instance the correct representation of incomplete information in the presence of multiple belief sets [12], commonsense reasoning [12], formalization for conformant planning [16], and meta-reasoning [24] etc. Recently, there is increasing research in this direction to address the long-standing problems of unintended world views due to recursion through modalities

Table 2 Modal Reduct in ASP^{KM}.

subjective literal s	if $W \models_{km} s$	$W \not\models_{km} s$
Kl	replace Kl with l	delete the rule
not Kl	remove not Kl	replace not Kl with not l
Ml	remove Ml	replace Ml with not not l
not Ml	replace not Ml with not l	delete the rule

that were introduced by Gelfond [11], e.g. [13, 16, 6]. Very recently, Shen and Eiter [22] introduced general logic programs possible containing epistemic negation NOT (ASP^{NOT}), and defined its world views by minimizing the knowledge. ASP^{NOT} can not only express K p and M p formulas by *not* NOT p and NOT *not* p, but also offer a solution to the problems of unintended world views. In this section we show that ASP^{KM} logic programs in [16] where the most recent version of ASP^{KM} is defined, and a special kind of ASP^{NOT} programs can be viewed as ASP^{EP} programs.

4.1   Relation to ASP^{KM}

An ASP^{KM} program is a set of rules of the form h_1 *or* ... *or* $h_k \leftarrow b_1, ..., b_m$ where $k \geq 0$, $m \geq 0$, h_i is an objective literal, and b_i is an objective literal possible preceded by a negation as failure operator *not*, a modal operator K or M, or a combination operator *not* K or *not* M. For distinguishment, we call the world view of the ASP^{KM} program **KM-world view**. Let W be a non-empty collection of consistent sets of ground objective literals, W is a **KM-world view** of an ASP^{KM} program Π if $W = AS(\Pi_W)$ where Π_W is a disjunctive logic program obtained using *Modal Reduct* as showed in Table 2.

In ASP^{KM}, the notion of satisfiability is defined from $\models_{km}$ relationship below.

- $< W, w > \models_{km} l$ if $l \in w$
- $< W, w > \models_{km}$ *not* l if $l \notin w$
- $< W, w > \models_{km}$ Kl if $\forall v \in W : l \in v$
- $< W, w > \models_{km}$ *not* Kl if $\exists v \in W : l \notin v$
- $< W, w > \models_{km}$ Ml if $\exists v \in W : l \in v$
- $< W, w > \models_{km}$ *not* Ml if $\forall v \in W : l \notin v$

▶ **Definition 7.** Given an ASP^{KM} program Ω, an ASP^{EP} program is called a *KM-EP-Image* of Ω, denoted by $KM - EP - I(\Omega)$, if it is obtained by

- Replacing all occurrences of literals of the form $K\ l$ in Π by $l \succcurlyeq_\sharp \top$.
- Replacing all occurrences of literals of the form $M\ l$ in Π by *not* $l \not\succcurlyeq_\sharp \top$ and *not not* l[4] respectively.
- Replacing all occurrences of literals of the form *not* $K\ l$ in Π by $l \not\succcurlyeq_\sharp \top$ and *not* l respectively.
- Replacing all occurrences of literals of the form *not* $M\ l$ in Π by *not* $l \succcurlyeq_\sharp \top$.

▶ **Theorem 8.** *Let Ω be an ASP^{KM} program, and Π be the ES-EP-Image of Ω, and W be a non-empty collection of consistent sets of ground objective literals, W is a candidate world view of Π iff W is a KM-world view of Ω.*

---

[4] Here, we view *not not* l as a representation of *not* l' where we have $l' \leftarrow$ *not* l and l' is a fresh literal. It is worthwhile to note that CLINGO is able to deal with *not not*.

▶ **Example 9.** Consider an ASP$^{\text{KM}}$ program Ω: $p \leftarrow \text{M } p$. Ω has an unique KM-world view $\{\{p\}\}$. Its ES-EP-Image Π contains two rules

$$p \leftarrow not\ p \not\succcurlyeq_\sharp \top \qquad p \leftarrow\ not\ not\ p$$

Then, the reduct $\Pi^{\{\{p\}\}}$ contains five rules

$$p \leftarrow p, \top \qquad p \leftarrow not\ p, \top \qquad p \leftarrow p, \bot \qquad p \leftarrow not\ p, \bot \qquad p \leftarrow\ not\ not\ p$$

which has only one answer set $\{p\}$. While the reduct $\Pi^{\{\{\}\}}$ contains only one rule $p \leftarrow\ not\ not\ p$ which has two answer sets $\{\}$ and $\{p\}$. Then, $\{\{p\}\}$ is the unique candidate world view of Π.

4.2 Relation to ASP$^{\text{NOT}}$

Here, we consider the ASP$^{\text{NOT}}$ program that is a set of the rules of the form l_1 *or* ... *or* $l_k \leftarrow e_1, ..., e_m, s_1, ..., s_n$ where $k \geq 0$, $m \geq 0$, $n \geq 0$, l_i is an objective literal, e_i is an extended literal, s_i is a subjective literal of the form NOT e or *not* NOT e. For distinguishment, we call the world view of an ASP$^{\text{NOT}}$ program **NOT-world view**. Let W be a non-empty collection of consistent sets of ground objective literals, W is a candidate NOT-world view of an ASP$^{\text{NOT}}$ program Π if $W = AS(\Pi_W)$ where Π_W is a general logic program obtained using *Epistemic Reduct* by (1) replacing every NOT F that is satisfied by W with $\top$, and (2) replacing every NOT F that is not satisfied by W with *not* F. In ASP$^{\text{NOT}}$, the notion of satisfiability of a subjective formula NOT F is defined from $\models_{\text{NOT}}$ relationship

$$< W, w > \models_{\text{NOT}} \text{NOT } F \text{ if } \exists v \in W : v \not\models_{\text{GLP}} F$$

where the satisfaction denoted by $\models_{\text{GLP}}$ is as the satisfaction of a formula defined in general logic programming introduced in [23]. W is a NOT-world view of an ASP$^{\text{NOT}}$ program Π if it is a candidate NOT-world view satisfying maximal set of literals of the form NOT e appearing in Π.

▶ **Definition 10.** Given an ASP$^{\text{NOT}}$ program Ω, an ASP$^{\text{EP}}$ program is called a *NOT-EP-Image* of Ω, denoted by NOT-EP-I(Ω), if it is obtained by

- Replacing all occurrences of literals of the form *not NOT e* in Ω by $e \succcurlyeq_\sharp \top$.
- Replacing all occurrences of literals of the form *NOT e* in Ω by $e \not\succcurlyeq_\sharp \top$ and *not e* respectively.

▶ **Theorem 11.** *Let Ω be an ASP^{NOT} program, and Π be the NOT-EP-Image of Ω, and W be a non-empty collection of consistent sets of ground objective literals, W is a world view of Π iff W is a NOT-world view of Ω.*

▶ **Example 12.** Consider an ASP$^{\text{NOT}}$ program from [22] that contains two rules

$$innocent(john) | guilty(john) \qquad innocent(john) \leftarrow \text{NOT } guility(john)$$

Ω has an unique NOT-world view $\{\{innocent(john)\}\}$. The NOT-EP-Image of Ω has three rules

$$innocent(john) | guilty(john)$$
$$innocent(john) \leftarrow guilty(john) \not\succcurlyeq_\sharp \top$$
$$innocent(john) \leftarrow not\ guility(john)$$

and a unique world view $\{\{innocent(john)\}\}$.

5   Applications

Consider the relationship between ASP$^{\text{EP}}$ and the languages of strong introspections mentioned in section 5, ASP$^{\text{EP}}$ is potential in dealing with some important issues. In this section, we illustrate the use of ASP$^{\text{EP}}$ in modeling problems with introspective preferences.

5.1   Describing the Principle of Majority

The principle of majority (PM) is a widely used epistemic commonsense in the fields of information fusion, decision making, social choice, etc, where incomplete information usually causes multiple belief sets, and queries are usually answered by the principle of majority. For example, consider the behavior of common birds modeled by a program PM as below:

$pigeon(X) \text{ } or \text{ } raven(X) \text{ } or \text{ } swallow(X) \text{ } sparrow(X) \leftarrow commonBird(X)$

$behavior(X, migratory) \leftarrow swallow(X)$

$behavior(X, resident) \leftarrow pigeon(X)$

$behavior(X, resident) \leftarrow raven(X)$

$behavior(X, resident) \leftarrow sparrow(X)$

Then, given a fact f_t:

$commonBird(tom)$

and answer the query *behavior(tom,?)* by the principle of majority described by the following rules r_r, r_m, and r_u:

$behavior(X, resident) \leftarrow behavior(X, resident) \succ_\sharp behavior(X, migratory), bird(X)$

$behavior(X, migratory) \leftarrow behavior(X, migratory) \succ_\sharp behavior(X, resident), bird(X)$

$behavior(X, unknown) \leftarrow behavior(X, migratory) \approx_\sharp behavior(X, resident), bird(X)$

They express that *a bird X is a resident(migratory) bird if X being resident(migratory) is strictly more possible than X being migratory(resident)*, otherwise *it is unknown*. It is easy to see that the program $PM \cup \{f_t, r_r, r_m, r_u\}$ gives answer *behavior(tom, resident)* to the query, that is

$PM \cup \{f_t, r_r, r_m, r_u\} \vdash_{\text{ep}} behavior(tom, resident)$

5.2   Modeling the Monty Hall Problem

We will use ASP$^{\text{EP}}$ to solve the Monty Hall problem from [21]: *One of the three boxes labeled 1, 2, and 3 contains the keys to that new 1975 Lincoln Continental. The other two are empty. If you choose the box containing the keys, you win the car. A contestant is asked to select one of three boxes. Once the player has made a selection, Monty is obligated to open one of the remaining boxes which does not contain the key. The contestant is then asked if he would like to switch his selection to the other unopened box, or stay with his original choice.* Here is the problem:does it matters if the contentant switches? The answer is YES.

One of many solutions of the Monty Hall Problem is by arithmetic [21], where nine possible states are given as showed in Table 3, and the idea in the solution can be described naturally as: *Constestant switches if SWITCH can bring more wins than STAY, Constestant stays if STAY can bring more wins than SWITCH.*

■ **Table 3** Possible Results of MHP.

Keys are in box	Contestant choose box	Monty can open box	Contestant switches	Results
1	1	2 or 3	2 or 3	loses
1	2	3	1	wins
1	3	2	1	wins
2	1	3	2	wins
2	2	1 or 3	1 or 3	loses
2	3	1	2	wins
3	1	2	3	wins
3	2	1	3	wins
3	3	1 or 2	1 or 2	loses

Encode the definition of the problem using a disjunctive logic program MHP below.

$box(1)$

$box(2)$

$box(3)$

$1\{choose\_box(X) : box(X)\}1$

$1\{key\_in\_box(X) : box(X)\}1$

$can\_open\_box(X) \leftarrow box(X), not\ choose\_box(X), not\ key\_in\_box(X)$

$win\_by\_switch \leftarrow choose\_box(X), not\ key\_in\_box(X)$

$win\_by\_stay \leftarrow choose\_box(X), key\_in\_box(X)$

Represent the idea in the solution by two rules r_1 :

$$switch \leftarrow win\_by\_switch \succcurlyeq_\sharp win\_by\_stay, win\_by\_stay \nsucceq_\sharp win\_by\_switch$$

and r_2:

$$stay \leftarrow win\_by\_stay \succcurlyeq_\sharp win\_by\_switch, win\_by\_switch \nsucceq_\sharp win\_by\_stay$$

Then, we have the following result that gives a correct answer for the problem.

▶ **Theorem 13.** $MHP \cup \{r_1, r_2\} \vdash_{ep} switch$ and $MHP \cup \{r_1, r_2\} \nvdash_{ep} stay$.

6 Conclusion and Future Work

We present a logic programming formalism capable of reasoning that combines nonmonotonic reasoning, epistemic preferential reasoning, which is built on the existing efficient answer set solvers. This makes it an elegant way to formalize some problems with defaults and introspections of preferences.

A limitation of the work in this paper is that we do not consider the relationships between ASP$^{\text{EP}}$ and other well developed formalisms of preferences.

As a next goal, we will consider the introspection of other typs of preferences which are considered in the AI field [8, 18]. Our future work also includes the mathematical properties of ASP$^{\text{EP}}$ programs, the methodologies for modeling with ASP$^{\text{EP}}$, and the efficient solver of ASP$^{\text{EP}}$ programs.

─── **References** ───

1 Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, pages 9–18, 2003.

2 Ronen I. Brafman and Carmel Domshlak. Preference Handling - An Introductory Tutorial. *AI Magazine*, 30(1):58–86, 2009. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2114.

3 Gerhard Brewka. Answer Sets and Qualitative Optimization. *Logic Journal of the IGPL*, 14(3):413–433, 2006. doi:10.1093/jigpal/jzl017.

4 Gerhard Brewka, Ilkka Niemelä, and Tommi Syrjänen. Logic Programs with Ordered Disjunction. *Computational Intelligence*, 20(2):335–357, 2004. doi:10.1111/j.0824-7935.2004.00241.x.

5 Gerhard Brewka, Ilkka Niemelä, and Miroslaw Truszczynski. Answer Set Optimization. In *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 867–872, 2003. URL: http://ijcai.org/Proceedings/03/Papers/125.pdf.

6 Luis Fariñas Del Cerro, Andreas Herzig, and Ezgi Iraz Su. Epistemic equilibrium logic. In *Proc. 24th Int. Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 2964–2970, 2015.

7 James P. Delgrande, Torsten Schaub, Hans Tompits, and Kewen Wang. A Classification and Survey of Preference Handling Approaches in Nonmonotonic Reasoning. *Computational Intelligence*, 20(2):308–334, 2004. doi:10.1111/j.0824-7935.2004.00240.x.

8 Carmel Domshlak, Eyke Hüllermeier, Souhila Kaci, and Henri Prade. Preferences in AI: An overview. *Artif. Intell.*, 175(7-8):1037–1052, 2011. doi:10.1016/j.artint.2011.03.004.

9 Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Tina Dell'armi, and Giuseppe Ielpa. Design and Implementation of Aggregate Functions in the DLV System. *Theory Pract. Log. Program.*, 8(5-6):545–580, November 2008. doi:10.1017/S1471068408003323.

10 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven Answer Set Solving: From Theory to Practice. *Artif. Intell.*, 187–188:52–89, August 2012. doi:10.1016/j.artint.2012.04.001.

11 Michael Gelfond. Strong Introspection. In Thomas L. Dean and Kathleen McKeown, editors, *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, CA, USA, July 14-19, 1991, Volume 1.*, pages 386–391. AAAI Press / The MIT Press, 1991. URL: http://www.aaai.org/Library/AAAI/1991/aaai91-060.php.

12 Michael Gelfond. Logic Programming and Reasoning with Incomplete Information. *Ann. Math. Artif. Intell.*, 12(1-2):89–116, 1994. doi:10.1007/BF01530762.

13 Michael Gelfond. New semantics for epistemic specifications. In *Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2011.

14 Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents*. Cambridge Unversity Press, 2014.

15 Judy Goldsmith and Ulrich Junker. Preference Handling for Artificial Intelligence. *AI Magazine*, 29(4):9–12, 2008. URL: http://www.aaai.org/ojs/index.php/aimagazine/article/view/2180.

16 Patrick Kahl, Richard Watson, Michael Gelfond, and Yuanlin Zhang. A Refinement of the Language of Epistemic Specifications. *Journal of Logic and Computation*, 2015. URL: 10.1093/logcom/exv065.

17 Pascal Nicolas, Laurent Garcia, Igor Stéphan, and Claire Lefèvre. Possibilistic uncertainty handling for answer set programming. *Ann. Math. Artif. Intell.*, 47(1-2):139–181, 2006. doi:10.1007/s10472-006-9029-y.

18 Gabriella Pigozzi, Alexis Tsoukiàs, and Paolo Viappiani. Preferences in artificial intelligence. *Ann. Math. Artif. Intell.*, 77(3-4):361–401, 2016. `doi:10.1007/s10472-015-9475-5`.

19 Chiaki Sakama and Katsumi Inoue. Prioritized logic programming and its application to commonsense reasoning. *Artif. Intell.*, 123(1-2):185–222, 2000. `doi:10.1016/S0004-3702(00)00054-0`.

20 Torsten Schaub and Kewen Wang. A semantic framework for preference handling in answer set programming. *TPLP*, 3(4-5):569–607, 2003. `doi:10.1017/S1471068403001844`.

21 Steve Selvin. A problem in probability (letter to the editor). *American Statistician*, 29:67, 1975.

22 Yi-Dong Shen and Thomas Eiter. Evaluating epistemic negation in answer set programming. *Artificial Intelligence*, 237:115–135, 2016.

23 Yidong Shen, Kewen Wang, and Thomas Eiter. FLP answer set semantics without circular justifications for general logic programs. *Artificial Intelligence*, 213:1–41, 2014.

24 Mirosław Truszczyński. Revisiting epistemic specifications. In *Logic programming, knowledge representation, and nonmonotonic reasoning*, pages 315–333. Springer, 2011.

25 G. H. Von Wright. The logic of preference reconsidered. *Theory and Decision*, 3(2):140–169, 1972. `doi:10.1007/BF00141053`.

A New Proof-Theoretical Linear Semantics for CHR

Igor Stéphan

LERIA, Université d'Angers, France
igor.stephan@univ-angers.fr

Abstract

Constraint handling rules are a committed-choice language consisting of multiple-heads guarded rules that rewrite constraints into simpler ones until they are solved. We propose a new proof-theoretical declarative linear semantics for Constraint Handling Rules. We demonstrate completeness and soundness of our semantics w.r.t. operational ω_t semantics. We propose also a translation from this semantics to linear logic.

1 Introduction

CHR (for *constraint handling rules*) [9, 10, 11, 12, 13, 14] are a committed-choice language consisting of multiple-heads guarded rules that rewrite constraints into simpler ones until they are solved. CHR are a special-purpose language concerned with defining declarative constraints in the sense of *Constraint logic programming* [16, 17, 18]. CHR are a language extension that allows to introduce *user-defined* constraints, i.e. first-order predicates, as for example less-than-or-equal ($\leq$), into a given host language as Prolog, Lisp, Java or C/C++. CHR define *simplification* of user-defined constraints, which replaces constraints by simpler ones while preserving logical equivalence. For example the antisymmetry of less-than-or-equal constraint: $((X \leq Y), (Y \leq X) \Leftrightarrow (X = Y))$ where "$(X \leq Y), (Y \leq X)$" is the multiple head of the rule, X, Y are variables and "," denotes conjunction. This rule means "if constraints $(X \leq Y)$ and $(Y \leq X)$ are present then equality $(X = Y)$ is enforced and constraints are solved". CHR define also *propagation* over user-defined constraints that adds new constraints, which are logically redundant but may cause further simplifications. For example the transitivity of less-than-or-equal constraint: $((X \leq Y), (Y \leq Z) \Rightarrow (X \leq Z))$. This rule means "if constraints $(X \leq Y)$ and $(Y \leq Z)$ are present then constraint $(X \leq Z)$ is logically equivalent". CHR allow to use guards, which are sequences of host language statements. For example the reflexivity of less-than-or-equal constraint: $((X \leq Y) \Leftrightarrow (X = Y) \mid true)$ where $(X = Y)$ is a test and *true* is a reserved symbol that has for operational semantics "add nothing". This rule means "if constraint $(X \leq Y)$ is present and $(X = Y)$ is true then constraint $(X \leq Y)$ is solved". CHR finally define *simpagation* over user-defined constraints that mixes and subsumes simplification and propagation. The general schema of CHR (simpagation) rules is then $(K_1, \dots, K_n \backslash D_1, \dots, D_m \Rightarrow guard \mid G)$ with $n + m \neq 0$ and $G = B_1, \dots, B_p$ or $G = true$. Constraints $K_1, \dots, K_n$ are kept like in propagation and constraints $D_1, \dots, D_m$ are deleted like in simplification. If $n = 0$, a simpagation rule is a simplification rule, and if $m = 0$, a simpagation rule is a propagation rule. For example, the idempotency of less-than-or-equal constraint: $((X \leq Y) \backslash (X \leq Y) \Leftrightarrow true)$. This rule means "if constraint $(X \leq Y)$ is present twice, only one occurrence is kept". This last

example suggests that CHR is more about consumption than truth. CHR rules are applied on multi-sets of constraints. Repeated application of those rules on a multi-set of initial constraints incrementally solves these constraints. The committed-choice principle expresses a *don't care* nondeterminism, which leads to efficient implementations.

From the very beginning, [9, 10] gives a declarative semantics in terms of first-order classical logic: simplification rules are considered as logical equivalences and propagation rules as implications (with an equivalence-based semantics ω_e [19]). But [10] gives also a first *abstract* (or *high-order* or *theoretical*) operational semantics ω_t based on a transition system over sets (with some extensions to avoid the trivial nontermination of propagation rules [1]). The *refined* operational semantics ω_r [8] is finer than the previous one w.r.t. the classical implementations of CHR. Those operational semantics are in fact ad-hoc linear semantics [6]. In [5, 6, 4] two different proof-theoretical *intuitionistic* linear semantics for CHR are proposed based on (intuitionist) Linear Logic [15]. Those linear semantics for CHR have been extended to CHR$^\vee$[2] which introduces the *don't know* nondeterminism[1] in CHR [7].

As emphasized in [6], "Many implemented algorithms do not have a first-order classical logical reading, especially when these algorithms are deliberately non-confluent", i.e. the committed-choice matters. Moreover "Considering arbitrary derivation from a given goal, termination (and confluence) under the abstract semantics ω_t are preserved under the refined semantics ω_r, but not the other way around. While it fixes the constraint and rule order for execution, the refined operational semantics is still *nondeterministic*" [14]. But if anyone wants, for example, to compile another high level language to CHR paradigm there must be only two sources of nondeterminism: the *don't care* nondeterminism of the committed-choice and the *don't know* nondeterminism of the disjunction of CHR$^\vee$ and no other hidden nondeterminism not controllable by the programmer. But in the already defined semantics of the literature and the current implementations, there is a third source of nondeterminism due to the management of the constraints as an *unordered multi-set*: the order in which the constraints are reactivated by the wake-up-policy function[2] is left unspecified (page 68 of [14]). And there is even a forth source of nondeterminism due to the management of the multiple heads of the simpagation rules. The matching order in the application of a simpagation rule is not deterministic and we do not know which constraints from the multi-set may be chosen and kept or deleted, if more than one possibility exists (page 69 of [14]). Consider the following first-order CHR program with only one rule, which illustrates the first hidden nondeterminism:

$$(a(X), a(Y), s \Leftrightarrow true)$$

and $\{a(1), a(2), a(3), s\}$ as the store (an unordered multi-set) of constraints. The final state may be $\{a(1)\}$, $\{a(2)\}$ or $\{a(3)\}$. Even with the refined ω_r semantics, the semantics of the CHR program rests unknown. We propose in this article a new proof-theoretical linear semantics for CHR by means of a sequent calculus system in which the store is managed as a multi-set as in the ω_t semantics. This system is proved to be sound and complete w.r.t. the ω_t semantics. We propose also a second new proof-theoretical linear semantics for CHR by means of a sequent calculus system in which the store is managed as a *sequence*. This system is proved to be sound. But, more important, this system is completely deterministic and overcomes the two sources of hidden nondeterminism defined above. Finally, we propose for those two systems a translation into the Linear Logic and we prove the soundness of this translation.

---

[1] freely offered when the host language is Prolog
[2] With first-order constraints, instantiation of some variables of the constraints makes them eligible to the application of CHR rules.

Section 2 presents the needed background on Linear Logic (Subsection 2.1) and CHR syntax and semantics (Subsection 2.2). Section 3 presents our two new linear sequent calculi for CHR, the ω_l sequent calculus system in which the store is managed as a multi-set and the $\omega_l^{\otimes}$ sequent calculus system in which the store is managed as a sequence (Subsection 3.1). Those systems are then translated into the Linear Logic and we prove the soundness of this translation (Subsection 3.2). We conclude by a discussion about the possible links to *focusing proofs* of [3] and on some remarks about our two new proof-theoretical semantics for CHR.

2 Background

2.1 Linear logic

Linear Logic is a substructural logical formalism introduced in [15]. It is based on *tokens* which are built on predicate symbols and terms in the usual first-order manner. These tokens (w.r.t. atoms of classical first order logic) represent resources (w.r.t. truth). Linear Logic consumes and produces resources and is aware of their multiplicities. The linear-logic sequent calculus is based on the *sequent*, which is a pair of multi-sets of linear-logic formulas. Linear formulas are built on tokens and the following operators (we only present the useful ones for us): The symbol $\otimes$ stands for the multiplicative conjunction and is similar to conjunction of classical logic. The **1** symbol stands for the neutral of $\otimes$ and represents empty resource and corresponds to the *true* of classical logic. The symbol & stands for the additive conjunction. *a&b* represents an internal choice between *a* and *b*, it means that one can freely choose between *a* and *b* but not have *a* and *b* at the same time. The symbol $\multimap$ stands for the linear implication and apply *modus ponens* but by consuming the preconditions. The symbol **0** corresponds to the *false* of classical logic. The modality symbol ! marks the unlimited resources. The symbol $\exists$ (resp. $\forall$) stands for existential (resp. universal) first-order quantifications.

In what follows we only use the fragment of the linear-logic sequent calculus that is relevant for us in its two-sided version (F, F_1, F_2 and L some linear formulas, Γ, Γ_1, Γ_2, Δ, Δ_1, Δ_2 some multi-sets of formulas).

■ Identity rules

$$\frac{}{F \vdash F} \, I \qquad\qquad \frac{\Gamma_1 \vdash L \qquad L, \Gamma_2 \vdash \Delta}{\Gamma_1, \Gamma_2 \vdash \Delta} \, Cut$$

■ Multiplicative rules

$$\frac{\Gamma, F_1, F_2 \vdash \Delta}{\Gamma, F_1 \otimes F_2 \vdash \Delta} \, \otimes L \qquad\qquad \frac{\Gamma \vdash \Delta}{\mathbf{1}, \Gamma \vdash \Delta} \, \mathbf{1}L$$

$$\frac{\Gamma_1 \vdash \Delta_1, F_1 \qquad \Gamma_2 \vdash \Delta_2, F_2}{\Gamma_1, \Gamma_2 \vdash \Delta_1, \Delta_2, F_1 \otimes F_2} \, \otimes R \qquad \frac{\Gamma_1 \vdash F_1, \Delta_1 \qquad \Gamma_2, F_2 \vdash \Delta_2}{\Gamma_1, \Gamma_2, F_1 \multimap F_2 \vdash \Delta_1, \Delta_2} \, \multimap L$$

■ Additive rules

$$\frac{\Gamma, F_1 \vdash \Delta}{\Gamma, F_1 \& F_2 \vdash \Delta} \, \& L_1 \qquad\qquad \frac{\Gamma, F_2 \vdash \Delta}{\Gamma, F_1 \& F_2 \vdash \Delta} \, \& L_2$$

■ Quantifier rules (t is a term)

$$\frac{\Gamma, [x \leftarrow t](F) \vdash \Delta}{\Gamma, (\forall x \, F) \vdash \Delta} \, \forall L \qquad\qquad \frac{\Gamma, [x \leftarrow y](F) \vdash \Delta}{\Gamma, (\exists x \, F) \vdash \Delta} \, \exists L$$

The usual proviso for the $\exists$L rule is assumed: the variable y must not be free in the formulas of the sequent conclusion of the inference rule.

- Exponential rules

$$\frac{\Gamma, !F, !F \ \vdash \ \Delta}{\Gamma, !F \ \vdash \ \Delta} \ !C \qquad\qquad \frac{\Gamma, F \ \vdash \ \Delta}{\Gamma, !F \ \vdash \ \Delta} \ !D \qquad\qquad \frac{\Gamma \ \vdash \ \Delta}{\Gamma, !F \ \vdash \ \Delta} \ !W$$

A proof tree is a finite labeled tree whose nodes are labeled with sequents such that every sequent node is the consequence of its direct children according to one of the inference rules of the calculus. A proof tree is a *linear proof* if all its leaves are axioms (i.e. instances of the Identity rule I).

2.2 CHR language and its semantics

In this article, we consider a first-order CHR program as an intensional version of the grounded corresponding propositional program with respect to its Herbrand universe based on the function and constant symbols of the program. A constraint is a predicate symbol with elements of the Herbrand universe as arguments. With this point of view, we omit the guard and there is no need of equivalence relation between variables. Moreover, there is no need for a wake up rule since there is no more variable to be woken up in the store of constraints.

2.2.1 The syntax

The CHR formalism is defined as follows : a CHR rule is a rule of the form ($K_1, \ldots, K_m$, $D_1, \ldots, D_n, B_1, \ldots, B_p$ some constraints):

- [Simpagation rule] $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)$ with $n > 0$, $m > 0$ or
- [Propagation rule] $(K_1, \ldots, K_m \Rightarrow B)$ with $m > 0$ or
- [Simplification rule] $(D_1, \ldots, D_n \Leftrightarrow B)$ with $n > 0$

and $B = B_1, \ldots, B_p$ with $p > 0$ or *true* or *false* (two reserved symbols).

2.2.2 The operational ω_t semantics

An *identified* constraint $A\#i$ is a constraint A with some unique integer i, its identity. Function *const*, resp. *id*, gets from an identified constraint its constraint, resp. identity: $const(A\#i) = A$, resp. $id(A\#i) = i$. The *id* function and *const* are extended to sequences, sets and multi-sets of identified constraints in the obvious manner. An *execution state* is a tuple $\langle \Omega, S, H \rangle_c$ where Ω (the current *goal*) stands for a multi-set of constraints to be executed, S (the current *store*) stands for a multi-set of identified constraints, H (the current *propagation history*) stands for a set of words, each recording the name of a rule and identities of identified constraints, c stands for a counter that represents the next free integer which can be used to number an identified constraint. For an initial goal Ω, the initial state is $\langle \Omega, \emptyset, \emptyset \rangle_1$. The operational semantics ω_t is based on the following two transitions, which map a state to an other state (symbol $\uplus$ stands for union of multi-sets):

- [*Introduce*] $\langle \{A\} \uplus \Omega, S, H \rangle_c \rightsquigarrow_t \langle \Omega, \{A\#c\} \uplus S, H \rangle_{c+1}$
- [*Apply*] $\langle \Omega, K_\# \uplus D_\# \uplus S, H \rangle_c \rightsquigarrow_t \langle B \uplus \Omega, K_\# \uplus S, H \uplus \{r.i_1 \ldots i_m\} \rangle_c$ where there exists a simpagation rule $r@(K \backslash D \Leftrightarrow B)$ such that $K_\# = \{K_1\#i_1, \ldots, K_m\#i_m\}$ and $D_\# = \{D_1\#i_{m+1}, \ldots, D_n\#i_{m+n}\}$ and $K_1, \ldots, K_m = K$ and $D_1, \ldots, D_n = D$ and $r.i_1 \ldots i_m \notin H$ ($r.i_1 \ldots i_m$ is the *identity* of the instantiated rule and r is a name for the rule).

The [*Introduce*] transition transports a constraint from the goal to the store and associates an identity to this constraint. A CHR rule $(K \backslash D \Leftrightarrow B)$ is *applicable* if the head of the rule (considered as a multi-set) $K \uplus D$ is a subset of the multi-set $const(S)$ of the constraints of the store S. If a CHR rule $(K \backslash D \Leftrightarrow B)$ is applicable then the CHR rule is *applied*: [*Apply*] transition removes identified constraints $D_1 \# i_{m+1}, \ldots, D_n \# i_{m+n}$ from the store and adds the constraints of B to the goal. If $B = true$ nothing is added to the goal. This can only be done if the CHR rules has not already been fired with the same identity in order to forbid trivial loops. In the [*Apply*] transition, if $B = false$ there is no transition at all. The transitions are non-deterministically applied until either no more transition is applicable (a *successful derivation*), or $B = false$ (a *failed derivation*). In both cases a *final state* has been reached.

▶ **Example 1.** Consider the following first-order CHR program of the introduction with only one rule

$$(a(X), a(Y), s \Leftrightarrow true)$$

and $\{a(1), a(2), a(3), s\}$ as the store of constraints.

We give an ω_t derivation:

$$
\begin{array}{rcl}
 & & \langle\{a(1), a(2), a(3), s\}, \emptyset, \emptyset\rangle_1 \\
[Introduce] & \rightsquigarrow_t & \langle\{a(2), a(3), s\}, \{a(1)\#1\}, \emptyset\rangle_2 \\
[Introduce] & \rightsquigarrow_t & \langle\{a(2), s\}, \{a(1)\#1, a(3)\#2\}, \emptyset\rangle_3 \\
[Introduce] & \rightsquigarrow_t & \langle\{a(2)\}, \{a(1)\#1, a(3)\#2, s\#3\}, \emptyset\rangle_4 \\
[Apply] & \rightsquigarrow_t & \langle\{a(2)\}, \emptyset, \{r.1.2.3\}\rangle_4 \\
[Introduce] & \rightsquigarrow_t & \langle\emptyset, \{a(2)\#4\}, \{r.1.2.3\}\rangle_5
\end{array}
$$

The store in the final state is $\{a(2)\}$ but may be $\{a(1)\}$ or $\{a(3)\}$ since the order of [*Introduce*] derivation steps is arbitrary.

The semantics of this program is only clear if we consider its extensional version with the grounded rules in this (arbitrary) order:

$$(a(1), a(2), s \Leftrightarrow true) \qquad (a(1), a(3), s \Leftrightarrow true) \qquad (a(2), a(3), s \Leftrightarrow true)$$

and the initial store (a sequence) of constraints as, for example, $a(1), a(2), a(3), s$. When the constraint s is considered the constraints $a(1)$, $a(2)$ and $a(3)$ are already in the store of constraints. The first rule is tried and the matching of its multiple head with the store of constraints is a success. Since it is a simplification rule, the constraints $a(1)$ and $a(2)$ are deleted from the store of constraints. The final store of constraints is then $\{a(3)\}$.

The operational ω_r semantics. There exists a *refined* operational semantics ω_r [8] which considers the goal as a sequence instead of a multi-set. This semantics is very closed to the way it is usually implemented. It also uses a transition system with identified constraints, identities and propagation history. The operational semantics ω_t is based on six transitions which map a state to another state.

Linear-logic semantics of [6, 5]. This linear-logic semantics is directly inspired by the classical first-order logic semantics: goals (and stores of constraints) are translated to multiplicative conjunctions, simpagation rules $(K \backslash D \Leftrightarrow B)$ to the linear-logic formulas: $!(K \otimes D) \multimap (K \otimes B)$ and a CHR program to a large conjunction of linear-logic formulas. We denote by $(.)^L$ the above translation. A CHR program P has a computation with initial store S_0 and final store S_n if and only if $(P)^L \vdash ((S_0)^L \multimap (S_n)^L)$.

Axiomatic linear semantics of [5, 7].   The *axiomatic linear semantics* is based on the cut-rule of the linear logic and proper axioms: each CHR rule of the program is interpreted as an axiom. A goal is solved if there exists a linear proof of *true* in a linear-logic sequent calculus augmented by the proper axioms.

None of the previous semantics offers a semantics for the example of the introduction since they all manage the store of constraints as an unordered multi-set.

3 ω_l and $\omega_l^{\otimes}$ sequent calculus

In this section, we first define two sequent calculi: the ω_l and the $\omega_l^{\otimes}$ sequent calculi. The first one keeps the multi-sets of the ω_t and ω_r semantics while the second uses a sequence. Then we prove that the ω_l system is sound and complete w.r.t. the ω_t semantics while the $\omega_l^{\otimes}$ system is sound (but not complete) w.r.t. the ω_t semantics. Finally we give a translation from the ω_l (and $\omega_l^{\otimes}$) system to the linear-logic sequent calculus and prove the soundness of this translation.

3.1 ω_l and $\omega_l^{\otimes}$ systems

We first define the notion of store for the ω_l and $\omega_l^{\otimes}$ systems.

▶ **Definition 2** (ω_l and $\omega_l^{\otimes}$ stores). An ω_l store is a multi-set of identified constrains. An $\omega_l^{\otimes}$ store is a sequence of identified constraints.

The ω_l and $\omega_l^{\otimes}$ systems are based on two kinds of sequents: the *focused* sequent is focused on a particular identified constraint, the current identified constraint, while the *non focused* sequent works on a sequence of identified constraints, the current goal.

We first define our sequents for the ω_l and $\omega_l^{\otimes}$ systems.

▶ **Definition 3** (non focused and focused ω_l and $\omega_l^{\otimes}$ sequents).

- A *non focused* sequent is a quadruple $(\Gamma \blacktriangleright \Omega_{\#} \blacktriangleleft S_{\uparrow} \vdash S_{\downarrow})$ where $S_{\downarrow}$, the *down store*, and $S_{\uparrow}$, the *up store*, are two stores of identified constraints, Γ is a sequence of CHR rules and $\Omega_{\#}$, the *goal*, is a sequence of identified constraints[3].
- A *focused* sequent is a quintuple $(\Gamma \, ! \, \Delta \rhd a \lhd S_{\uparrow} \vdash S_{\downarrow})$ where $S_{\downarrow}$, $S_{\uparrow}$ and Γ are defined as for the non focused sequent, Δ is an ending sequence of Γ and a is an identified constraint.

The intuitive meaning of a non focused sequent $(\Gamma \blacktriangleright \Omega_{\#} \blacktriangleleft S_{\uparrow} \vdash S_{\downarrow})$ is to try and consume the identified constraints $\Omega_{\#}$ [4] with the sequence of CHR rules Γ thanks to the store $S_{\uparrow}$. The elements of the store $S_{\downarrow}$ are the unconsumed identified constraints: the identified constraints of $S_{\uparrow}$ that have not been consumed and those produced by $\Omega_{\#}$ and not consumed during this production.

The intuitive meaning of a focused sequent $(\Gamma \, ! \, \Delta \rhd A\#i \lhd S_{\uparrow} \vdash S_{\downarrow})$ is the same as for a non focused sequent but restricted to a unique identified constraint $A\#i$ which may be consumed only by the sequence of CHR rules Δ [5].

In our sequent calculi, the final store of identified constraints is what we have to prove. Solve the problem represented by a CHR program and an initial goal is to prove *true*.

Now we define the $\omega_l^{\otimes}$ sequent calculus:

---

[3] Note that in the ω_t semantics the goal is a set of constraints.
[4] i.e. to solve the constraints of $const(\Omega_{\#})$
[5] the identified constraints produced by $A\#i$ may be consumed by the CHR rules of Γ

▶ **Definition 4** ($\omega_l^\otimes$ sequent calculus system). The $\omega_l^\otimes$ system is based on four types of $\omega_l^\otimes$ inference rules ($S_\downarrow$, $S_\downarrow^a$, $S_\uparrow$, $S_\uparrow^a$, $S_\uparrow^B$, $S_\downarrow^B$, $S_\downarrow^\Omega$, $S_\uparrow^\Omega$, S, S^K, S^D, $S^{\subseteq K}$, $S_\uparrow^{\subseteq K}$ some stores; $K_1, \ldots, K_m$, $D_1, \ldots, D_n$, $B_1, \ldots, B_p$ some constraints, B a sequence of constraints; a an identified constraint; $\Omega_\#$, the goal, a sequence of identified constraints; i, i' some integers).

- The *non focused* subsystem:
  - The *true* axiom:

$$\overline{\Gamma \blacktriangleright true \blacktriangleleft S \vdash S}\ true$$

  - The *Left-elimination-of-conjunction* inference rule:

$$\frac{\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow^a \vdash S_\downarrow^a \qquad \Gamma \blacktriangleright \Omega_\# \blacktriangleleft S_\downarrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega}{\Gamma \blacktriangleright a, \Omega_\# \blacktriangleleft S_\uparrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega}\ \otimes_L$$

  - The *Exchange* inference rule:

$$\frac{\Gamma \blacktriangleright \Omega_\# \blacktriangleleft A'\#i', A\#i, S_\uparrow \vdash S_\downarrow}{\Gamma \blacktriangleright \Omega_\# \blacktriangleleft A\#i, A'\#i', S_\uparrow \vdash S_\downarrow}\ X$$

    with the proviso that $A \neq A'$.

- The *focused* subsystem:
  - The *Inactivate* axiom:

$$\overline{\Gamma\ !\ \rhd a \lhd S \vdash a, S}\ \uparrow$$

  - The *Weakening* inference rule:

$$\frac{\Gamma\ !\ \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow}{\Gamma\ !\ (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow G), \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow}\ W$$

    with no j, ($1 \leq j \leq n$ such that $D_j = const(a)$ or $1 \leq j \leq m$ such that $K_j = const(a)$), $S^D \subseteq S_\uparrow$, $S^K \subseteq S_\uparrow$, $S^D \uplus S^K \uplus \{a\} = \{K_1\#i_1, \ldots, K_m\#i_m, D_1\#i_{m+1}, \ldots, D_n\#i_{m+n}\}$[6].

- The *Focusing* inference rule:

$$\frac{\Gamma\ !\ \Gamma \rhd a \lhd S_\uparrow \vdash S_\downarrow}{\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow \vdash S_\downarrow}\ F$$

- The *Apply* inference rule:

$$\frac{\Gamma \blacktriangleright B_1\#i', \ldots, B_p\#(i'+p) \blacktriangleleft S^K, S_\uparrow^B \vdash S^{\subseteq K}, S_\downarrow^B \qquad \Gamma \blacktriangleright S^{\subseteq K} \blacktriangleleft S_\downarrow^B, S_\uparrow^{\subseteq K} \vdash S_\downarrow}{\Gamma\ !\ (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B_1, \ldots, B_p), \Delta \rhd a \lhd S^D, S^K, S_\uparrow^B, S_\uparrow^{\subseteq K} \vdash S_\downarrow}\ \backslash \Leftrightarrow$$

  with either there exists j, $1 \leq j \leq n$ such that $D_j = const(a)$, $S^K = K_1\#i_1, \ldots, K_m\#i_m$, a inserted in S^D at place j is equal to $D_1\#i_{m+1}, \ldots, D_n\#i_{m+n}$, or there exists j, $1 \leq j \leq m$ such that $K_j = const(a)$, a inserted in S^K at place j is equal to $K_1\#i_1, \ldots, K_m\#i_m$, $S^D = D_1\#i_{m+1}, \ldots, D_n\#i_{m+n}$; i' a new integer, $S^{\subseteq K}$ is a subsequence of $K_1\#i_1, \ldots, K_m\#i_m$.

The ω_l sequent calculus system is less structurally constrained than the $\omega_l^\otimes$ system:

▶ **Definition 5** (ω_l sequent calculus system). The ω_l sequent calculus system is the $\omega_l^\otimes$ sequent calculus system where the store of identified constraints and the multiple heads of rules are multi-sets instead of sequences and the *Exchange* inference rule is omitted.

---

[6] When used with multi-set operations, sequences are considered as multi-sets

The *non focused* system splits the current goal and allocates the resources. If the current goal is the *true* goal then no identified constraint is consumed and the *true* axiom is applied. If the current goal is a sequence of identified constraints, the *Left-elimination-of-conjunction* inference rule is applied: The first identified constraint a of the sequence is isolated and a part of the resources $S_\uparrow^a$ are allocated to solve the constraint $const(a)$, the rest of the identified constraints, $S_\uparrow^\Omega$, and those produced by a but unconsumed, $S_\downarrow^a$, are allocated to the sequence of identified constraints $S^{\subseteq K}$[7]. This inference rule realizes in fact a hidden use of the cut-rule of the linear-logic sequent calculus: the $S_\downarrow^a$ is a lemma computed by the left subproof and used in the right subproof. Both operational semantics eliminate those instances of the cut-rule in order to linearize the derivation.

The *focused* system chooses, if any, a CHR rule to be applied on the focused identified constraint a. If no such CHR rule exists, the *Inactivate* axiom stores the identified constraint into the store. The *Weakening* inference eliminates, in the order of the sequence Δ, the first CHR rule $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)$ that cannot be applied since there are no subset S^K and S^D of $S_\uparrow$ such that $S^K \uplus S^D \uplus \{a\} = \{K_1 \# i_1, \ldots, K_m \# i_m, D_1 \# i_{m+1}, \ldots, D_n \# i_{m+n}\}$.

The *Focusing* inference rule flips from the non focused $\omega_l^\otimes$ system to the focused $\omega_l^\otimes$ system by focusing on an identified constraint.

The *Apply* inference rule flips from focused $\omega_l^\otimes$ system to non focused $\omega_l^\otimes$ system by applying a CHR rule $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)$ on the focused identified constraint a since there are two subsequences S^K and S^D of $S_\uparrow$ such that $S^K \uplus S^D \uplus \{a\} = \{K_1 \# i_1, \ldots, K_m \# i_m, D_1 \# i_{m+1}, \ldots, D_n \# i_{m+n}\}$. The solving of the constraint underlying the identified constraint a is reduced to the solving of the goal of the CHR rule $B = B_1, \ldots, B_p$ and eventually the solving of the constraints underlying $S^{\subseteq K}$ in the case that identified constraints from $S^{\subseteq K} \subseteq S^K$ were not consumed during the process of consumption/production of $B_1 \# i', \ldots, B_p \# i' + p$. As for the *Left-elimination-of-conjunction* inference rule a part of the resources $S^K \uplus S_\uparrow^B$ is allocated to solve the goal $B_1 \# i', \ldots, B_p \# (i' + p)$, the rest of the identified constraints $S^{\subseteq K}$ and those produced by $B_1 \# i', \ldots, B_p \# (i' + p)$ but unconsumed $S_\downarrow^B$ are allocated to a sequence $S^{\subseteq K}$. Since the $\omega_l^\otimes$ system only applies a CHR rule if one of the identified constraints of its head is focused on, the calculus of $(\Gamma \blacktriangleright S^{\subseteq K} \blacktriangleleft S_\downarrow^B, S_\uparrow^{\subseteq K} \vdash S_\downarrow)$ is necessary to the completeness. But $S^{\subseteq K}$ is not necessarily equal to $K_1 \# i_1, \ldots, K_m \# i_m$ since some identified constraints may have been consumed during the process of consumption/production of $B_1 \# i', \ldots, B_p \# (i' + p)$. Moreover, $S^{\subseteq K}$ may be empty if all the resources have been consumed.

In a classical implementation of CHR, $S_\uparrow^{\subseteq K}$ is captured by the flow $S_\uparrow^B / S_\downarrow^B$. In this configuration $S_\uparrow^B$ is not anymore the necessary resources to prove B and $S_\downarrow^B$ the resources produced but unconsumed by B but respectively the input store and the output store of the derivation of B.

Once again, this *Apply* inference rule realizes in fact a hidden use of the cut-rule of the linear-logic sequent calculus: a lemma is computed by the left subproof and used in the right subproof. Both operational semantics eliminate those instances of the cut-rule in order to linearize the derivation.

When the applied CHR rule is such that $S^{\subseteq K} = \emptyset$ the *Apply* inference rule is simplified to

$$\frac{\Gamma \blacktriangleright B_1 \# i', \ldots, B_p \# (i' + p) \blacktriangleleft S^K, S_\uparrow \vdash S_\downarrow}{\Gamma \, ! \, (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B_1, \ldots, B_p), \Delta \triangleright a \triangleleft S^D, S^K, S_\uparrow \vdash S_\downarrow} \, \backslash \Leftrightarrow$$

---

[7] In the case of the ω_l system, the elements of the multi-set $S^{\subseteq K}$ must be ordered in a sequence $\Omega_\#^{\subseteq K}$.

Moreover, when the applied CHR rule is a simplification rule ($S^K = \emptyset$ and $S^{\subseteq K} = \emptyset$) then the *Apply* inference rule is simplified to

$$\frac{\Gamma \blacktriangleright B_1\#i',\ldots,B_p\#(i'+p) \blacktriangleleft S_\uparrow \vdash S_\downarrow}{\Gamma \;!\; (D_1,\ldots,D_n \Leftrightarrow B_1,\ldots,B_p), \Delta \rhd a \lhd S^D, S_\uparrow \vdash S_\downarrow} \Leftrightarrow$$

▶ **Example 6.** What follows is a proof [8] in the ω_l system for the ω_l sequent

$$(\Gamma \blacktriangleright d\#1, a\#2 \blacktriangleleft \;\vdash\; S_\downarrow) = (r_1, r_2, r_3 \blacktriangleright d\#1, a\#2 \blacktriangleleft \;\vdash\; c\#6, f\#5, g\#4, d\#1).$$

with $S_\downarrow = \{c\#6, f\#5, g\#4, d\#1\}$, $\Gamma = r1@(d \Rightarrow e), r2@(a\backslash e \Leftrightarrow g), r3@(a \Leftrightarrow f, c)$

$$\cfrac{\nabla_1 \qquad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{\Gamma \blacktriangleright g\#4 \blacktriangleleft a\#2, d\#1 \vdash g\#4, a\#2, d\#1}}^{F\uparrow} \quad \cfrac{\nabla_2}{\Gamma \blacktriangleright a\#2 \blacktriangleleft g\#4, d\#1 \vdash S_\downarrow}^{F}}{\Gamma \;!\; r2, r3 \rhd a\#2 \lhd d\#1, e\#3 \vdash S_\downarrow}^{\backslash \Leftrightarrow}}{\Gamma \;!\; r1, r2, r3 \rhd a\#2 \lhd d\#1, e\#3 \vdash S_\downarrow}^{W}}{\Gamma \blacktriangleright a\#2 \blacktriangleleft d\#1, e\#3 \vdash S_\downarrow}^{F}}{\Gamma \blacktriangleright d\#1, a\#2 \blacktriangleleft \;\vdash\; S_\downarrow}}^{\otimes_L}$$

with ∇_1:

$$\cfrac{\cfrac{\overline{\Gamma \blacktriangleright e\#3 \blacktriangleleft d\#1 \vdash e\#3, d\#1}}^{F\uparrow} \quad \overline{\Gamma \blacktriangleright d\#1 \blacktriangleleft e\#3 \vdash d\#1, e\#3}^{F\uparrow}}{\cfrac{\Gamma \;!\; r1, r2, r3 \rhd d\#1 \lhd \;\vdash\; d\#1, e\#3}{\Gamma \blacktriangleright d\#1 \blacktriangleleft \;\vdash\; d\#1, e\#3}^{F}}^{\backslash \Leftrightarrow}$$

and ∇_2:

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{\Gamma \blacktriangleright f\#5 \blacktriangleleft g\#4, d\#1 \vdash f\#5, g\#4, d\#1}}^{F\uparrow} \quad \overline{\Gamma \blacktriangleright c\#6 \blacktriangleleft f\#5, g\#4, d\#1 \vdash S_\downarrow}^{F\uparrow}}{\Gamma \blacktriangleright f\#5, c\#6 \blacktriangleleft g\#4, d\#1 \vdash S_\downarrow}^{\otimes_L}}{\Gamma \;!\; r3 \rhd a\#2 \lhd g\#4, d\#1 \vdash S_\downarrow}^{\Leftrightarrow}}{\cfrac{\Gamma \;!\; r2, r3 \rhd a\#2 \lhd g\#4, d\#1 \vdash S_\downarrow}{\Gamma \;!\; r1, r2, r3 \rhd a\#2 \lhd g\#4, d\#1 \vdash S_\downarrow}^{W}}^{W}}$$

What follows is a proof in the $\omega_l^\otimes$ system:

$$\cfrac{\nabla_1 \qquad \cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\overline{\Gamma \blacktriangleright g\#4 \blacktriangleleft a\#2, d\#1 \vdash g\#4, a\#2, d\#1}}^{F\uparrow} \quad \cfrac{\nabla_2}{\Gamma \blacktriangleright a\#2 \blacktriangleleft g\#4, d\#1 \vdash S_\downarrow}^{F}}{\Gamma \;!\; r2, r3 \rhd a\#2 \lhd e\#3, d\#1 \vdash S_\downarrow}^{\backslash \Leftrightarrow}}{\Gamma \;!\; r1, r2, r3 \rhd a\#2 \lhd e\#3, d\#1 \vdash S_\downarrow}^{W}}{\Gamma \blacktriangleright a\#2 \blacktriangleleft e\#3, d\#1 \vdash S_\downarrow}^{F}}{\Gamma \blacktriangleright a\#2 \blacktriangleleft d\#1, e\#3 \vdash S_\downarrow}^{X}}{\Gamma \blacktriangleright d\#1, a\#2 \blacktriangleleft \;\vdash\; S_\downarrow}}^{\otimes_L}$$

Notice the use of the *Exchange* inference rule (X) in order to permute the identified constraints $d\#1$ and $e\#3$.

We give the first contribution of this article: the soundness and completeness theorem of the ω_l system w.r.t. the ω_t semantics:

---

[8] In this example, we define the $F \uparrow$ axiom: $\overline{r_1, r_2, \ldots, r_n \blacktriangleright a \blacktriangleleft S \vdash a, S}$ as a shorthand for an instance of a *Focusing* inference rule followed by many instances, as needed, of the *Weakening* inference rule and followed by an instance of the *Inactivate* axiom.

▶ **Theorem 7** (Soundness and completeness of the ω_l system w.r.t. the ω_t semantics). *Let Γ be a CHR program and $B_1, \ldots, B_p$ some constraints. The initial goal $B_1, \ldots, B_p$ is solved in the ω_t semantics by Γ with a final store (a multi-set) of identified constraints Σ if and only if there exists an ω_l proof of $(\Gamma \blacktriangleright B_1\#1, \ldots, B_p\#p \blacktriangleleft \;\vdash\; \Sigma)$.*

And as a corollary, we obtain the soundness of the $\omega_l^{\otimes}$ system w.r.t. the ω_t semantics:

▶ **Theorem 8** (Soundness of the $\omega_l^{\otimes}$ system w.r.t. the ω_t semantics). *Let Γ be a CHR program and $B_1, \ldots, B_p$ some constraints and Σ a store (a multi-set) of identified constraints. If there exists an $\omega_l^{\otimes}$ proof of $(\Gamma \blacktriangleright B_1\#1, \ldots, B_p\#p \blacktriangleleft \;\vdash\; S)$, where S is a sequence of Σ then the initial goal $B_1, \ldots, B_p$ is solved in the ω_t semantics by Γ with a final store Σ.*

The $\omega_l^{\otimes}$ system is not complete w.r.t. the ω_t semantics since the *Exchange* inference rule is limited to identified constraints that are based on different constraints.

▶ **Example 9** (Example of the introduction continued). We can prove with the $\omega_l^{\otimes}$ system the sequent $(r \blacktriangleright a(1)\#1, a(2)\#2, a(3)\#3, s\#4 \blacktriangleleft \;\vdash\; a(1)\#1)$:

$$
\cfrac{
\cfrac{}{r \blacktriangleright a(1)\#1 \blacktriangleleft \;\vdash\; a(1)\#1} {\scriptstyle F\uparrow}
\qquad
\cfrac{
\cfrac{\cfrac{}{r \blacktriangleright a(2)\#2 \blacktriangleleft a(1)\#1 \vdash\; a(2)\#2, a(1)\#1}{\scriptstyle F\uparrow} \qquad \nabla}
{r \blacktriangleright a(2)\#2, a(3)\#3, s\#4 \blacktriangleleft a(1)\#1 \vdash\; a(1)\#1} {\scriptstyle \otimes_L}
}{r \blacktriangleright a(1)\#1, a(2)\#2, a(3)\#3, s\#4 \blacktriangleleft \;\vdash\; a(1)\#1} {\scriptstyle \otimes_L}
$$

with ∇ ($S = a(3)\#3, a(2)\#2, a(1)\#1$):

$$
\cfrac{
\cfrac{}{r \blacktriangleright a(3)\#3 \blacktriangleleft a(2)\#2, a(1)\#1 \vdash S}{\scriptstyle F\uparrow}
\qquad
\cfrac{
\cfrac{\cfrac{}{r \blacktriangleright true \blacktriangleleft a(1)\#1 \vdash\; a(1)\#1}{\scriptstyle true}}{\cfrac{r \,!\, r \rhd s\#4 \lhd S \;\vdash\; a(1)\#1}{r \blacktriangleright s\#4 \blacktriangleleft S \vdash\; a(1)\#1}{\scriptstyle F}}{\scriptstyle \Leftrightarrow}
}{r \blacktriangleright a(3)\#3, s\#4 \blacktriangleleft a(2)\#2, a(1)\#1 \vdash\; a(1)\#1} {\scriptstyle \otimes_L}
$$

But not the sequent $(r \blacktriangleright a(3)\#3, a(2)\#2, a(1)\#1, s\#4 \blacktriangleleft \;\vdash\; a(2)\#2)$ of Example 1 nor the sequent $(r \blacktriangleright a(3)\#3, a(2)\#2, a(1)\#1, s\#4 \blacktriangleleft \;\vdash\; a(3)\#3)$ since the store S is a sequence (and not a multi-set) and the *Exchange* inference rule cannot be applied since the identified constraints $a(1)\#1$, $a(2)\#2$ and $a(3)\#3$ are based on the same constraint a.

3.2 Translation from ω_l and $\omega_l^{\otimes}$ systems into Linear Logic

We define a translation from the ω_l system into the linear-logic sequent calculus and prove that the result of the translation of a ω_l proof is a linear-logic proof in the sense of the definition of Section 2.1. We first give the translation of the CHR rules, then the translation for the ω_l sequents and finally the translation for the ω_l system. The translation from the $\omega_l^{\otimes}$ system into the linear-logic sequent calculus is directly obtained from previous translation by omitting the *Exchange* inference rule (and by considering sequences as multi-sets).

▶ **Definition 10** (Translation of the CHR rules and CHR programs into linear-logic formulas). The CHR rules are translated into linear-logic formulas as follows thanks to the function $(.)_\Gamma$:

- $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow true)_\Gamma =$
  $\forall x_1 \ldots \forall x_{m+n}$
  $((K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes D_{m+1}(x_{m+1}) \otimes \ldots \otimes D_{m+n}(x_{m+n}))$
  $\multimap (K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes \mathbf{1}))$

- $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow \textit{false})_\Gamma =$
  $\forall x_1 \ldots \forall x_{m+n}$
  $((K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes D_{m+1}(x_{m+1}) \otimes \ldots \otimes D_{m+n}(x_{m+n}))$
  $\multimap (K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes \mathbf{0}))$

- $(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B_1, \ldots, B_p)_\Gamma =$
  $\forall x_1 \ldots \forall x_{m+n} \exists y_1 \ldots \exists y_p$
  $((K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes D_{m+1}(x_{m+1}) \otimes \ldots \otimes D_{m+n}(x_{m+n}))$
  $\multimap (K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes B_1(y_1) \otimes \ldots \otimes B_p(y_p)))$

- $(r, \Delta)_\Gamma = (r)_\Gamma \mathbin{\&} (\Delta)_\Gamma$ with r a CHR rule and Δ a non empty sequence of CHR rules.

CHR constant *true* is interpreted to $\mathbf{1}$, the neutral of $\otimes$ the multiplicative conjunction. CHR constant *false* is interpreted to $\mathbf{0}$ which has no elimination rule. Introduction of new identities are interpreted to existential quantifications in order to generate a brand new one each time while transmission of identities of identified constraints are interpreted by universal quantifications. In a CHR rule, symbol "$\Leftrightarrow$" is interpreted to linear implication and symbol "," is interpreted to the multiplicative conjunction $\otimes$. Finally, in CHR program, symbol "," is interpreted to the additive conjunction $\&$, an (ordered) committed choice.

▶ **Example 11** (Example continued).

$(r_1)_\Gamma = (d \Rightarrow e)_\Gamma = (\forall x \ (\exists y \ (d(x) \multimap d(x) \otimes e(y))))$
$(r_2)_\Gamma = (a \backslash e \Leftrightarrow g)_\Gamma = (\forall x, x' \ (\exists y \ (a(x) \otimes e(x') \multimap a(x) \otimes g(y))))$
$(r_3)_\Gamma = (a \Leftrightarrow f, c)_\Gamma = (\forall x \ (\exists y, y' \ (a(x) \multimap f(y) \otimes c(y'))))$

▶ **Definition 12** (Translation of the ω_l sequents into linear sequent). The ω_l system language is translated into Linear Logic as follows thanks to three functions $(.)_\Omega$, $(.)_\uparrow$ and $(.)_\downarrow$ for translating respectively the goal, the up store and the down store of an ω_l sequent.

- $\begin{cases} (\textit{true})_\Omega = \mathbf{1}, (\textit{false})_\Omega = \mathbf{0}, \\ (A\#i)_\Omega = A(i) \text{ a token, with } A \text{ a constraint and } i \text{ an identity,} \\ ((a, \Omega^\#))_\Omega = ((a)_\Omega \otimes (\Omega^\#)_\Omega) \\ \quad \text{with } a \text{ an identified constraint and } \Omega^\# \text{ a sequence of identified constraints} \end{cases}$

- $\begin{cases} (A\#i)_\uparrow = A(i) \text{ a token, with } A \text{ a constraint and } i \text{ an identity,} \\ (S)_\uparrow = \{(a)_\uparrow \mid a \in S\} \\ \quad \text{with } a \text{ an identified constraint and } S \text{ a store.} \end{cases}$

- $\begin{cases} (A\#i)_\downarrow = A(i) \text{ a token, with } A \text{ a constraint and } i \text{ an identity,} \\ (S)_\downarrow = \bigotimes_{a \in S} (a)_\downarrow \\ \quad \text{with } a \text{ an identified constraint and } S \text{ a store.} \end{cases}$

For any ω_l sequent is translated into a linear sequent as follows thanks to the function $L(.)$:

- $L(\Gamma \blacktriangleright \Omega^\# \blacktriangleleft S_\uparrow \vdash S_\downarrow) = !(\Gamma)_\Gamma, (\Omega^\#)_\Omega, (S_\uparrow)_\uparrow \vdash (S_\downarrow)_\downarrow$
- $L(\Gamma \mathbin{!} \Delta \triangleright \Omega^\# \triangleleft S_\uparrow \vdash S_\downarrow) = !(\Gamma)_\Gamma, (\Delta)_\Gamma \& \mathbf{1}, (\Omega^\#)_\Omega, (S_\uparrow)_\uparrow \vdash (S_\downarrow)_\downarrow$
- $L(\Gamma \mathbin{!} \triangleright \Omega^\# \triangleleft S_\uparrow \vdash S_\downarrow) = !(\Gamma)_\Gamma, (\Omega^\#)_\Omega, (S_\uparrow)_\uparrow \vdash (S_\downarrow)_\downarrow$

The goal and the down store of identified constraints of the ω_l sequent are interpreted to multiplicative conjunctions of tokens while the up store of identified constraints is interpreted to a sequence of tokens. The multiplicative conjunction of the goal induces a sequence on the identified constraints of the goal. The multiplicative conjunction of the goal allows the introduction of the cut-rule of the *Left-elimination-of-conjunction* inference and *Apply* inference rules.

The CHR program is interpreted as a large additive conjunction of linear implications ended with the **1** constant in order to allow the move in the *Inactivate* inference rule of the identified constraint from the goal to the down store when no CHR rule is found to be applied.

▶ **Definition 13** (Translation of the ω_l system into the linear-logic sequent calculus).

- The *non focused* ω_l system:
  - *true* axiom

$$\frac{}{\Gamma \blacktriangleright true \blacktriangleleft S \vdash S} \; true$$

  is translated into [9]

$$\frac{\dfrac{}{(S)_\uparrow \vdash (S)_\downarrow} \; I\otimes}{\dfrac{\mathbf{1},(S)_\uparrow \vdash (S)_\downarrow}{L(\Gamma \blacktriangleright \mathbf{1} \blacktriangleleft S \vdash S)} \; !W} \; \mathbf{1}L$$

  - *Left-elimination-of-conjunction* inference rule:

$$\frac{\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow^a \vdash S_\downarrow^a \qquad \Gamma \blacktriangleright \Omega^\# \blacktriangleleft S_\downarrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega}{\Gamma \blacktriangleright a, \Omega^\# \blacktriangleleft S_\uparrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega} \; \otimes L$$

  is translated into

$$\frac{L(\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow^a \vdash S_\downarrow^a) \qquad \dfrac{\dfrac{L(\Gamma \blacktriangleright \Omega^\# \blacktriangleleft S_\downarrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega)}{!(\Gamma)_\Gamma, (\Omega^\#)_\Omega, (S_\downarrow^a)_\downarrow, (S_\uparrow^\Omega)_\uparrow \vdash (S_\downarrow^\Omega)_\downarrow} \; \otimes L*}{} }{\dfrac{\dfrac{!(\Gamma)_\Gamma, !(\Gamma)_\Gamma, (a)_\Omega, (\Omega^\#)_\Omega, (S_\uparrow^a)_\uparrow, (S_\uparrow^\Omega)_\uparrow \vdash (S_\downarrow^\Omega)_\downarrow}{!(\Gamma)_\Gamma, (a)_\Omega, (\Omega^\#)_\Omega, (S_\uparrow^a)_\uparrow, (S_\uparrow^\Omega)_\uparrow \vdash (S_\downarrow^\Omega)_\downarrow} \; !C}{L(\Gamma \blacktriangleright (a, \Omega^\#) \blacktriangleleft S_\uparrow^a, S_\uparrow^\Omega \vdash S_\downarrow^\Omega)} \; \otimes L} \; Cut$$

- The *focused* ω_l system:
  - Weakening rule:

$$\frac{\Gamma \, ! \, \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow}{\Gamma \, ! \, (K \backslash D \Leftrightarrow B), \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow} \; W$$

  is translated into

$$\frac{L(\Gamma \, ! \, \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow)}{L(\Gamma \, ! \, (K \backslash D \Leftrightarrow B), \Delta \rhd a \lhd S_\uparrow \vdash S_\downarrow)} \; \& L_2$$

  - Inactivate rule:

$$\frac{}{\Gamma \, ! \, \rhd A \# i \lhd S \vdash A \# i, S} \; \uparrow$$

  is translated into

---

[9] The following axiom $I\otimes$: $\dfrac{}{B_1, B_2, \ldots, B_n \vdash B_1 \otimes B_2 \otimes \ldots \otimes B_n}$ is a shorthand for the following linear proof

$$\frac{\dfrac{}{B_1 \vdash B_1} \; I \qquad \dfrac{\vdots}{B_2, \ldots, B_n \vdash B_2 \otimes \ldots \otimes B_n}}{B_1, B_2, \ldots, B_n \vdash B_1 \otimes B_2 \otimes \ldots \otimes B_n} \; \otimes R$$

where the top derivation is

$$\frac{\dfrac{}{B_{n-1} \vdash B_{n-1}} \; I \qquad \dfrac{}{B_n \vdash B_n} \; I}{B_{n-1}, B_n \vdash B_{n-1} \otimes B_n} \; \otimes R$$

$$\frac{\overline{A(i),(S)_\uparrow \ \vdash\ (A(i) \otimes (S)_\downarrow)}\ {\scriptstyle I\otimes}}{L(\Gamma\ !\ \rhd A\#i \lhd S \ \vdash\ A\#i, S)}\ {\scriptstyle !W}$$

▪ The *Focusing rule*:

$$\frac{\Gamma\ !\ \Gamma \rhd a \lhd S_\uparrow\ \vdash\ S_\downarrow}{\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow \vdash\ S_\downarrow}\ {\scriptstyle F}$$

is translated into

$$\frac{\dfrac{L(\Gamma\ !\ \Gamma \rhd a \lhd S_\uparrow\ \vdash\ S_\downarrow)}{!(\Gamma)_\Gamma, !(\Gamma)_\Gamma, (a)_\Omega, (S_\uparrow)_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle !D}}{L(\Gamma \blacktriangleright a \blacktriangleleft S_\uparrow \vdash\ S_\downarrow)}\ {\scriptstyle !C}$$

▪ The *Apply* rule with

$(K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)_\Gamma =$
$\forall x_1 \ldots \forall x_{m+n} \exists y_1 \ldots \exists y_p ((K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes D_{m+1}(x_{m+1}) \otimes \ldots \otimes D_{m+n}(x_{m+n}))$
$\multimap (K_1(x_1) \otimes \ldots \otimes K_m(x_m) \otimes B_1(y_1) \otimes \ldots \otimes B_p(y_p)))$

such that $K_\otimes = K_1(i) \otimes \ldots \otimes K_m(i+m)$, $D_\otimes = D_1(i+m+1) \otimes \ldots \otimes D_n(i+m+n)$,
$B = B_1, \ldots, B_p$, $B_\# = B_1 \# i', \ldots, B_p \#(i'+p)$, $i' = i+m+n+1$ and $p > 0$, and
$B_\otimes = B_1(i') \otimes \ldots \otimes B_p(i'+p) = (B_1 \# i', \ldots, B_p \#(i'+p))_\Omega$.
The *Apply* rule

$$\frac{\Gamma \blacktriangleright B_\# \blacktriangleleft S^K, S_\uparrow^B \vdash\ S^{\subseteq K}, S_\downarrow^B \qquad \Gamma \blacktriangleright \Omega_\#^{\subseteq K} \blacktriangleleft S_\downarrow^B, S_\uparrow^{\subseteq K} \vdash\ S_\downarrow}{\Gamma\ !\ (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B), \Delta \rhd a \lhd S^D, S^K, S_\uparrow^B, S_\uparrow^{\subseteq K}\ \vdash\ S_\downarrow}\ {\scriptstyle \backslash \Leftrightarrow}$$

is translated into

$$\frac{\dfrac{\dfrac{\overline{(a)_\Omega, (S^D)_\uparrow, (S^K)_\uparrow\ \vdash\ K_\otimes \otimes D_\otimes}\ {\scriptstyle I\otimes} \qquad \nabla}{!(\Gamma)_\Gamma, (K_\otimes \otimes D_\otimes \multimap K_\otimes \otimes B_\otimes), (a)_\Omega, (S^D)_\uparrow, (S^K)_\uparrow, (S_\uparrow^B)_\uparrow, (S_\uparrow^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle \multimap L}}{!(\Gamma)_\Gamma, (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B)_\Gamma, (a)_\Omega, (S^D)_\uparrow, (S^K)_\uparrow, (S_\uparrow^B)_\uparrow, (S_\uparrow^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle \exists L^* \forall L^*}}{L(\Gamma\ !\ (K_1, \ldots, K_m \backslash D_1, \ldots, D_n \Leftrightarrow B), \Delta \rhd a \lhd S^D, S^K, S_\uparrow^B, S_\uparrow^{\subseteq K}\ \vdash\ S_\downarrow)}\ {\scriptstyle \&L_1}$$

with $\nabla =$

$$\frac{\dfrac{L(\Gamma \blacktriangleright B_\# \blacktriangleleft S^K, S_\uparrow^B \vdash\ S^{\subseteq K}, S_\downarrow^B) \qquad \dfrac{\dfrac{L(\Gamma \blacktriangleright \Omega_\#^{\subseteq K} \blacktriangleleft S_\downarrow^B, S^{\subseteq K} \vdash\ S_\downarrow)}{!(\Gamma)_\Gamma, (\Omega_\#^{\subseteq K})_\Omega \otimes (S_\downarrow^B)_\downarrow, (S^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle \otimes L^*}}{}}{\dfrac{\dfrac{!(\Gamma)_\Gamma, !(\Gamma)_\Gamma, (S^K)_\uparrow, B_\otimes, (S_\uparrow^B)_\uparrow, (S_\uparrow^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}{!(\Gamma)_\Gamma, (S^K)_\uparrow, B_\otimes, (S_\uparrow^B)_\uparrow, (S_\uparrow^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle !C}}{!(\Gamma)_\Gamma, K_\otimes \otimes B_\otimes, (S_\uparrow^B)_\uparrow, (S_\uparrow^{\subseteq K})_\uparrow\ \vdash\ (S_\downarrow)_\downarrow}\ {\scriptstyle \otimes L^*}}\ {\scriptstyle Cut}$$

Note that since $\Omega_\#^{\subseteq K}$ is a sequence over $S^{\subseteq K}$, it may be chosen such that $(\Omega_\#^{\subseteq K})_\Omega = (S^{\subseteq K})_\downarrow$. If $S^{\subseteq K} = \emptyset$ or $B = true$ or $B = false$ the above translation is simplified in a straightforward manner.

The linear cut-rule is used in the translation of the *Left-elimination-of-conjunction* inference rule in order to transmit the down store of the left subproof to the right subproof. This down store which is a multiplicative conjunction is then split into a sequence of identified constraints thanks to linear-logic $\otimes$ left elimination $\otimes L$-rule.

Weakening inference rule tries the CHR rules in the order of the CHR program thanks to the linear-logic $\& L_2$ rule.

The linear cut-rule is also used in the translation of the *Apply* inference rule in order to transmit the down store of the left subproof to the the right subproof if S^K has not been completely consumed by the subproof (ie. $S^{\subseteq K} \neq \emptyset$). This down store which is a multiplicative conjunction is then split into a sequence of identified constraints thanks to the linear-logic $\otimes$ left elimination $\otimes L$-rule.

We now establish the second contribution of this article, the soundness of the translation from the ω_l system to the linear-logic sequent calculus:

▶ **Theorem 14.** *The result of the translation by Definitions 10, 12 and 13 of an ω_l proof is a linear proof.*

As a direct corollary, the soundness of the translation from the $\omega_l^{\otimes}$ system to the linear-logic sequent calculus with the same translation that for the ω_l system (instances of the *Exchange* inference rule are simply ignored):

▶ **Theorem 15.** *The result of the translation by Definitions 10, 12 and 13 of an $\omega_l^{\otimes}$ proof is a linear proof.*

4   Discussion

[3] proposes a normalization process of the Linear Logic proofs to a subclass of proofs, called the "focusing" proofs, which is complete (any derivable formula in Linear Logic has a focusing proof). Focusing proofs are expressed in a Triadic system, which respects the symmetry of Linear Logic. This process of normalization informally interleaves a *don't care* nondeterministic phase on *asynchronous* formulae and a phase applied on a *synchronous focused* formula. This last phase is a critical section and *don't know* nondeterminism can only appear during this phase. Since our $\omega_l^{\otimes}$ system is completely deterministic, the two phases of the $\omega_l^{\otimes}$ system are not based on the same principles as the two phases of the Triadic system. But, since the Triadic system is complete w.r.t. Linear Logic, it would be interesting to translate the ω_l and $\omega_l^{\otimes}$ proofs in focusing proofs to understand the semantics of CHR in terms of synchronous and asynchronous connectors.

5   Conclusion

We have proposed in this article two new proof-theoretical linear sequent systems for the semantics of CHR. The $\omega_l^{\otimes}$ system makes the semantics of the language completely deterministic. This semantics overcomes the hidden nondeterminism due to the management of the store of identified constraints and the multiple head of rules as multi-sets. But we can reintroduce the *don't care* nondeterminism of the committed choice principle if we allow the weakening inference rule even if the CHR rule is applicable (and of course also the *don't know* nondeterminism). Due to the lack of space, we cannot present a restricted version of the *Apply* inference rule (with $S^{\subseteq K}$ replaces only by K) which corresponds more faithfully to the ω_r semantics.

References

1 S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 252–266, 1997.

2 S. Abdennadher and H. Schütz. CHR$^\forall$: A Flexible Query Language. In *Proceedings of the 3rd International Conference on Flexible Query Answering Systems*, pages 1–14, 1998.

3 J.M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of logic and computation*, 2(3):297–347, 1992.

4 H. Betz. A linear-logic semantics for constraint handling rules With Disjunction. In *Proceedings of the 4th Workshop on Constraint Handling Rules (CP'07)*, pages 17–31, 2007.

5 H. Betz. A Unified Analytical Foundation for Constraint Handling Rules, PhD thesis, Ulm University, 2014.

6 H. Betz and T.W. Frühwirth. A Linear-Logic Semantics for Constraint Handling Rules. In *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, pages 137–151, 2005.

7 H. Betz and T.W. Frühwirth. Linear-Logic Based Analysis of Constraint Handling Rules with Disjunction. *ACM Transactions on Computational Logic*, 14(1), 2013.

8 G.J. Duck, P.J. Stuckey, M.G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In *Proceedings of the 20th International Conference on Logic Programming (ICLP'04)*, pages 90–104, 2004.

9 T.W. Frühwirth. Constraint Handling Rules. Technical report, ECRC, 1992.

10 T.W. Frühwirth. Constraint Handling Rules. In *Constraint Programming: Basics and Trends*, pages 90–107, 1994.

11 T.W. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1-3):95–138, 1998.

12 T.W. Frühwirth. *Constraint Handling Rules*. Cambridge University Press, 2009.

13 T.W. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer-Verlag, 2003.

14 T.W. Frühwirth and F. Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis*. Books on Demand, March 2011.

15 Jean-Yves Girard. Linear Logic. *Theoretical Computer Science*, 50:1–102, 1987.

16 P. Van Hentenryck. Constraint logic programming. *Knowledge Engineering Review*, 6(3):151–194, 1991.

17 J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

18 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

19 F. Raiser, H. Betz, and Thom Frühwirth. Equivalence of CHR states revisited. In *Proceedings of the 6th International Workshop on Constraint Handling Rules*, pages 34–48, 2009.

6 Appendix

In order to prove the soundness and completeness of the ω_l system w.r.t. the ω_t semantics, we first introduce the ω_t sequent calculus system that imitates faithfully the ω_t semantics. Hence we prove the soundness and completeness of this ω_t system w.r.t. the ω_t semantics and then prove the soundness and completeness of ω_l system w.r.t. ω_t system.

We first define what is a ω_t sequent.

▶ **Definition 16** (ω_t sequent). An ω_t sequent is a triplet ($\Gamma \blacktriangleright \Omega \blacktriangleleft S \vdash$) where S, the store of identified constraints, is a multi-set of identified constraints, Ω, the current goal, is a multi-set of constraints and Γ, the program, is a sequence of CHR rules.

Notice that in a ω_t sequent, compare to ω_l or $\omega_l^{\otimes}$ sequents, the final store is empty. It will be only known at the (unique) leaf of the ω_t proof.

Now we are able to define our ω_t system.

▶ **Definition 17** (ω_t system). The symbol Γ denotes a program, Ω a multi-set of constraints, S, S^K, S^D some sets of identified constraints, A, K_1, ..., K_m, D_1, ..., D_n some constraints, i, i_1 ..., i_{m+n} some distinct integers, B a sequence of constraints. The ω_t system is the set of the following ω_t *inference* rules:

- ω_t axiom:

$$\frac{}{\Gamma \blacktriangleright \ \blacktriangleleft S \vdash} \ \omega_t$$

with no simpagation rule $(K_1,\ldots,K_m \backslash D_1,\ldots,D_n \Leftrightarrow B) \in \Gamma$ such that $S^K = \{K_1 \# i_1,\ldots,K_m \# i_m\}$ and $S^D = \{D_1 \# i_{m+1},\ldots,D_n \# i_{m+n}\}$ and $S^K \cup S^D \subseteq S$.

- ω_t-*Tokenize* inference rule:

$$\frac{\Gamma \blacktriangleright \Omega \blacktriangleleft A\#i, S \vdash}{\Gamma \blacktriangleright A, \Omega \blacktriangleleft S \vdash} \ \#$$

A usual proviso for quantifier elimination is assumed: i must be a brand new integer.
- ω_t-*Apply* inference rule[10]:

$$\frac{\Gamma \blacktriangleright B, \Omega \blacktriangleleft S^K, S \vdash}{\Gamma \blacktriangleright \Omega \blacktriangleleft S^K, S^D, S \vdash} \ \backslash \Leftrightarrow$$

with $(K_1,\ldots,K_m \backslash D_1,\ldots,D_n \Leftrightarrow B)$ in Γ and $S^K = \{K_1 \# i_1,\ldots,K_m \# i_m\}$ and $S^D = \{D_1 \# i_{m+1},\ldots,D_n \# i_{m+n}\}$.

- ω_t-*true Apply* inference rule:

$$\frac{\Gamma \blacktriangleright \Omega \blacktriangleleft S^K, S \vdash}{\Gamma \blacktriangleright \Omega \blacktriangleleft S^K, S^D, S \vdash} \ \backslash \Leftrightarrow$$

with $(K_1,\ldots,K_m \backslash D_1,\ldots,D_n \Leftrightarrow true)$ in Γ and $S^K = \{K_1 \# i_1,\ldots,K_m \# i_m\}$ and $S^D = \{D_1 \# i_{m+1},\ldots,D_n \# i_{m+n}\}$.

We define also what are a ω_t proof tree and an ω_t proof.

▶ **Definition 18** (ω_t proof tree and ω_t proof). The set of ω_t *proof trees* is the least set of trees containing all one-node trees labeled with an ω_t sequent, and closed under the rules of Definition 17 in the following sense: For any ω_t proof tree ∇ whose root is labeled with sequent ω_t, s (and whose unique leaf is labeled with sequent s'') and for any instance of an inference rule $\frac{s}{s'}$ of Definition 17, the tree $\frac{\nabla}{s'}$ is an ω_t proof tree whose root is labeled with s' (and whose unique leaf is labeled with s'').

An ω_t *proof* of a sequent s is any ω_t proof tree whose root is labeled with s and whose unique leaf is labeled with an ω_t axiom.

---

[10] If B is the sequence $B_1,\ldots,B_p$, $p > 0$ then B, Ω means $\{B_1,\ldots,B_p\} \uplus \Omega$.

The following lemma expressing the completeness of the ω_t system w.r.t. the ω_t semantics is straightforward.

▶ **Lemma 19** (Completeness of the ω_t system w.r.t. ω_t semantics). *Let Γ be a program, Ω and Ω' two goals, S and S' two stores, c and c' integers such that $c \leq c'$, H and H' two propagation histories such that $H \subseteq H'$.*

If $\langle \Omega, S, H \rangle_c \rightsquigarrow_t^* \langle \Omega', S', H' \rangle_{c'}$ is an ω_t derivation then there exists an ω_t proof tree whose root is $(\Gamma \blacktriangleright \Omega \blacktriangleleft S \vdash)$ and such that there is only one sequent leaf $(\Gamma \blacktriangleright \Omega' \blacktriangleleft S' \vdash)$.

The following lemma expressing the soundness of the ω_t system w.r.t. the ω_t semantics is a little more difficult since the policy applied to avoid trivial loops has to be maintained.

▶ **Lemma 20** (Soundness of ω_t system w.r.t. ω_t semantics). *Let Γ be a program, Ω and Ω' two multi-sets of constraints, $\Omega_{\#}$ and $\Omega'_{\#}$ two multi-sets of identified constraints and H a set of identities of instantiated rules.*

If $(\Gamma \blacktriangleright \Omega_{\#} \blacktriangleleft S \vdash)$ admits an ω_t proof tree such that there is only one sequent leaf $(\Gamma \blacktriangleright \Omega'_{\#} \blacktriangleleft S' \vdash)$ with no identity of an instantiated rule in the ω_t proof tree appearing twice nor in H, then there exists an ω_t derivation $\langle \Omega, S, H \rangle_i \rightsquigarrow_t^* \langle \Omega', S', H' \rangle_{i'+1}$ with i (resp. i') the integer introduced by the first (last) instance of the ω_t-Tokenize inference rule in the ω_t proof tree and H' is the union of H and all the identities of the instantiated rules of the ω_t proof tree.

The following theorem of completeness and soundness of the ω_t system w.r.t. the ω_t semantics is a direct corollary of the two previous lemmas.

▶ **Theorem 21** (Soundess and completeness of ω_t system w.r.t. ω_t semantics). *Let Γ be a program and Ω an initial goal.*

$\langle \Omega, \emptyset, \emptyset \rangle_1$ admits a successful ω_t derivation if and only if $(\Gamma \blacktriangleright \Omega \blacktriangleleft \vdash)$ admits an ω_t proof with no identity of instantiated rule appearing twice.

▶ **Lemma 22** (Completeness of ω_l system w.r.t. ω_t system). *Let Γ be a CHR program and $B_1, \ldots, B_p$ some constraints.*

If the ω_t sequent $(\Gamma \blacktriangleright B_1, \ldots, B_p \blacktriangleleft \vdash)$ admits an ω_t proof with a last sequent $(\Gamma \blacktriangleright \blacktriangleleft S \vdash)$ then the ω_l sequent $(\Gamma \blacktriangleright B_1 \#1, \ldots, B_p \#p \blacktriangleleft \vdash S)$ admits an ω_l proof.

▶ **Lemma 23** (Soundness of ω_l system w.r.t. ω_t system). *Let Γ be a CHR program and $B_1, \ldots, B_p$ some constraints.*

If the ω_l sequent $(\Gamma \blacktriangleright B_1 \#1, \ldots, B_p \#p \blacktriangleleft \vdash S)$ admits an ω_l proof then the ω_t sequent $(\Gamma \blacktriangleright B_1, \ldots, B_p \blacktriangleleft \vdash)$ admits an ω_t proof with a last sequent $(\Gamma \blacktriangleright \blacktriangleleft S \vdash)$.

▶ **Theorem 24** (Soundness and completeness of ω_l system w.r.t. ω_t system). *Let Γ be a CHR program and $B_1, \ldots, B_p$ some constraints.*

The ω_t sequent $(\Gamma \blacktriangleright B_1, \ldots, B_p \blacktriangleleft \vdash)$ admits an ω_t proof with a last sequent $(\Gamma \blacktriangleright \blacktriangleleft S \vdash)$ if and only if the ω_l sequent $(\Gamma \blacktriangleright B_1 \#1, \ldots, B_p \#p \blacktriangleleft \vdash S)$ admits an ω_l proof.

Proof of Theorem 24. Direct consequence of Lemmas 22 and 23.   ◀

Proof of Theorem 7. Direct consequence of Theorems 21 and 24.   ◀

Proof of Theorem 8. The soundness is a direct consequence of Theorem 7.   ◀

CHR^{vis}: Syntax and Semantics

Nada Sharaf
The German University in Cairo, Egypt
nada.hamed@guc.edu.eg

Slim Abdennadher
The German University in Cairo, Egypt
slim.abdennadher@guc.edu.eg

Thom Frühwirth
Ulm University, Germany
thom.fruehwirth@uni-ulm.de

Abstract

The work in the paper presents an animation extension (CHR^{vis}) to Constraint Handling Rules (CHR). Visualizations have always helped programmers understand data and debug programs. A picture is worth a thousand words. It can help identify where a problem is or show how something works. It can even illustrate a relation that was not clear otherwise. CHR^{vis} aims at embedding animation and visualization features into CHR programs. It thus enables users, while executing programs, to have such executions animated. The paper aims at providing the operational semantics for CHR^{vis}. The correctness of CHR^{vis} programs is also discussed.

1 Introduction

Animation tools are considered as a basic construct of programming languages. They are used to visualize the execution of a program. They provide users with a simple and intuitive method to debug and trace programs. This paper presents an extension to Constraint Handling Rules (CHR). The extension adds new visual features to CHR. It enables users to animate executions of CHR programs.

CHR [9, 8] has evolved over the years into a general purpose language. Originally, it was proposed for writing constraint solvers. Due to its declarativity, it has, however, been used with different algorithms such as sorting algorithms, graph algorithms, ... etc. CHR lacked tracing and debugging tools. Users were only able to use the textual trace facility of SWI-Prolog as shown in Figure 1 which is hard to follow especially with big programs.

Two types of visual facilities are important for a CHR programmer/beginner. Firstly, the programmer would like to get a visual trace showing which CHR rule gets applied at every step and its effect. Secondly, since CHR has developed into a general purpose language, it has been used with different types of algorithms such as sorting and graph algorithms. It is thus important to have a visual facility to animate the execution of the algorithms rather than just seeing the rules being executed. CHR lacked such a tool. The tool should be able to adapt with the execution nature of CHR programs where constraints are added and removed continuously from the constraint store.

```
[trace] 1 ?- min(4),min(1),min(10).
   Call: (7) min(4) ? creep
^  Call: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
235), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
CHR:   (0) Insert: min(4) # <71>
^  Exit: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
235), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
   Exit: (7) min(4) ? creep
   Call: (7) min(1) ? creep
^  Call: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
430), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
lush|...])) ? creep
CHR:   (0) Insert: min(1) # <72>
^  Exit: (18) notrace(print_message_lines_guarded(current_output, [begin(trace, _G31
430), prefix('~N'), 'CHR: ', '~t(~D)~10| '-[0], 'Insert: ', '~w # <~w>'-[...|...], f
```

(a) Using the normal trace option.

```
2 ?- min(4),min(1),min(10).
CHR:   (0) Insert: min(4) # <71>
CHR:   (1) Call: min(4) # <71> ? [creep]
CHR:   (1) Exit: min(4) # <71> ? [creep]
CHR:   (0) Insert: min(1) # <72>
CHR:   (1) Call: min(1) # <72> ? [creep]
CHR:   (1) Try: min(1) # <72> \ min(4) # <71> <=> 1<_G31917 | true.
CHR:   (1) Apply: min(1) # <72> \ min(4) # <71> <=> 1<_G31917 | true. ? [creep]
CHR:   (1) Remove: min(4) # <71>
CHR:   (1) Exit: min(1) # <72> ? [creep]
CHR:   (0) Insert: min(10) # <73>
CHR:   (1) Call: min(10) # <73> ? [creep]
CHR:   (1) Try: min(1) # <72> \ min(10) # <73> <=> 1<10 | true.
CHR:   (1) Apply: min(1) # <72> \ min(10) # <73> <=> 1<10 | true. ? [creep]
CHR:   (1) Remove: min(10) # <73>
CHR:   (1) Exit: min(10) # <73> ?
```

(b) Using the chr_trace option.

Figure 1 Current Tracing Facilities in SWI-Prolog.

Several approaches have been devised for visualizing CHR programs and their executions. In [2], a tool called *VisualCHR* was proposed. VisualCHR allows its users to visually debug constraint solving. The compiler of JCHR [13] (on which VisualCHR is based) was modified. The visualization feature was thus not available for Prolog versions, the more prominent implementation of CHR. [3] introduced a tool for visualizing the execution of CHR programs. It was able to show at every step the constraint store and the effect of applying each CHR rule in a step-by-step manner. The tool was based on the SWI-Prolog implementation of CHR. Source-to-source transformation was used in order to eliminate the need of doing any changes to the compiler. The tool could thus be deployed directly by any user.

Despite of the availability of such visualization tools, CHR was still missing a system for animating algorithms. The available tools were able to show at each point in time the executed rule and the status of the constraint store [3, 1]. However, the algorithm implemented had no effect on the produced visualization. Existing algorithm animation tools could not be adopted with CHR. For example, one of the available tools is XTANGO [16] which is a general purpose animating system. However, the algorithm should be implemented in C or another language such that it produces a trace file to be read by a C program driver making it difficult to use with CHR. Due to the wide range of algorithms implemented through CHR, an algorithm-based animation was needed. Such animation should show at each step in time the changes to the data structure affected by the algorithm.

The paper presents a different direction for animating CHR programs. It allows users to animate any kind of algorithm implemented in CHR. This direction thus augments CHR with an animation extension. As a result, it allows a CHR programmer to trace the program from an algorithmic point of view independent of the details of the execution of its rules. The formal analysis of the new extension is presented in the paper. The paper thus presents a new operational semantics of CHR that embeds visualization into its execution. The formalism is able to capture not only the behavior of the CHR rules, it is also able to represent the graphical objects associated with the animation. It is used to prove the correctness of the programs extended with animation features. To eliminate the need of users learning the new syntax for using the extension, a transformation approach is also provided.

The paper is organized as follows: Section 2 introduces CHR. Section 3 introduces the new extension. In Section 3.2, the formalization is given by introducing ω_{vis}, a new operational semantics for CHR that accounts for annotation rules. Conclusions and directions for future work are presented at the end of the paper.

2   Constraint Handling Rules

CHR was initially developed for writing constraint solvers [9, 8, 10]. The rules of a CHR program keeps on rewriting the constraints in the constraint store until a fixed point is reached. At that point no CHR rules could be applied. The constraint store is initialized by the constraints in the query of ths user. CHR has implementations in different languages such as Java, C and Haskell. The most prominent implementation is the Prolog one. A CHR program has two types of constraints: user-defined/CHR constraints and built-in constraints. CHR constraints are defined by the user at the beginning of a program. Built-in constraints, on the other hand, are handled by the constraint theory ($\mathcal{CT}$) of the host language. A CHR program consists of a set of "simpagation rules". A simpagation rule has the following format:

$$optional\_rule\_name @ H_k \setminus H_r \Leftrightarrow G \mid B.$$

H_k and H_r represent the head of the rule. The body of the rule is B. The guard G represents a precondition for applying the rule. A rule is only applied if the constraint store contains constraints that match the head of the rule and if the guard is satisfied. As seen from the previous rule, the head has two parts: H_k and H_r. The head of a rule could only contain CHR constraints. The guard should consist of built-in constraints. The body, on the other hand, can contain CHR and built-in constraints. On applying the rule, the constraints in H_k are kept in the constraint store. The constraints in H_r are removed from the constraint store. The body constraints are added to the constraint store.

There are two special kinds of CHR rules: propagation rules and simplification rules. A propagation rule has an empty H_r. A propagation rule does not remove any constraint from the constraint store. It has the following format:

$$optional\_rule\_name @ H_k \Rightarrow G \mid B.$$

A simplification rule on the other hand has an empty H_k. A simplification rule removes all the head constraints from the constraint store. A simplification rule has the following format:

$$optional\_rule\_name @ H_r \Leftrightarrow G \mid B.$$

The following program aims at sorting numbers in an array/list. Each number is represented by the constraint `cell(I,V)`. I represents the index and V represents the value of the element. The program contains one rule: `sort_rule`. It is applied whenever the constraint store contains two `cell` constraints representing two unsorted elements. The guard makes sure that the two elements are not sorted with respect to each other. The element at index `I1` has a value (`V1`) that is greater than the value (`V2`) of the element at index `I2`. `I1` is less than `I2`. Thus, `V1` precedes `V2` in the array despite of the fact that it is greater than it. Since `sort_rule` is a simplification rule, the two constraints representing the unsorted elements are removed from the constraint store. Two `cell` constraints are added through the body of the rule to represent the performed swap to sort the two elements. Successive applications of the rule makes sure that any two elements that are not sorted with respect to each other are swapped. The fixed point is reached whenever `sort_rule` is no longer applicable. At this point, the array is sorted. The program is shown below:

```
:-chr_constraint cell/2.
sort_rule @ cell(I1,V1), cell(I2,V2) <=> I1<I2,V1>V2 |
                                   cell(I2,V1), cell(I1,V2).
```

2.1   Refined Operational Semantics ω_r

In the theoretical semantics of CHR (ω_t), a state is represented by the tuple $\langle G, S, B, T \rangle_n^V$ [9, 4]. G represents the goal store. It initially contains the query of the user. S is the CHR constraint store containing the currently available CHR constraints. B, on the other hand, is the built-in store with the built-ins handled by the host language (Prolog in this case). The propagation history, T, holds the names of the applied CHR rules along with the identifiers of the CHR constraints that activated the rules. T is used to eliminate the trivial nontermination problem. Each CHR constraint is associated with an identifier. n represents the next available identifier. V represents the set of global variables. Such variables are the ones that exist in the initial query of the user. V does not change during execution, it is thus omitted throughout the rest of the paper. A variable $v \notin V$ is called a local variable [12].

▶ **Definition 1.** The function `chr` is defined such that `chr(c#n) = c`. It is extended into sequences and sets of CHR constraints. Likewise, the function `id` is defined such that `id(c#n) = n`. It is also extended into sequences and sets of CHR constraints.

The refined operational semantics [7, 9] is adapted in most implementations of CHR. It removes some of the sources of the non-determinism that exists in the theoretical operational semantics (w_t). In w_t the order in which constraints are processed and the order of rule application is non-deterministic. However, in w_r, rules are executed in a top-down manner. Thus, in the case where there are two matching rules, w_r ensures that the rule that appears on top is executed. Each atomic head constraint is associated with a number (occurrence). Numbering starts from 1. It follows a top-down approach as well. For example, a CHR program to find the minimum value is numbered as follows:

```
remove_dup @ min(X)_2 \ min(X)_1 <=> true.
remove_min @ min(X)_4 \ min(Y)_3 <=> X<Y | true.
```

▶ **Definition 2.** The active/occurrenced constraint $c\#i : j$ refers to a numbered constraint that should only match with occurrence j of the constraint c inside the program. i is the identifier of the constraint [7].

A state in w_r is the tuple $< A, S, B, T >_n$. Unlike w_t, the goal A is a stack instead of a multi-set. S, B, T and n have the same interpretation as an w_t state. In the refined operational semantics, constraints are executed similar to procedure calls. Each constraint added to the store is activated. An active constraint searches for an applicable rule. The rule search is done in a top-down approach. If a rule matches, the newly added constraints (from the body of the applied rule) could in turn fire new rules. Once all rules are fired, execution resumes from the same point. Constraints in the constraint store are reconsidered/woken if a newly added built-in constraint could affect them (according to the wakeup policy). An active constraint thus tries to match with all the rules in the program. Table 1 shows the transitions of w_r.

- Solve+Wake: This transition introduces a built-in constraint c to the built-in store. In addition, all constraints that could be affected by c ($S1$ in this case) are woken up by adding them on top of the stack. These constraints are thus re-activated. A constraint where all its terms have become ground will not be thus woken up by the implemented wake-up policy since it is never affected by a new built-in constraint. $vars(S_0) \subseteq fixed(B)$ where $fixed(B)$ represents the variables fixed by B.
- Activate: This transition introduces a CHR constraint into the constraint store and activates it. The introduced constraint has the occurrence value 1 as a start.

Table 1 Transitions of ω_r.

---

1. **Solve+wakeup** : $\langle [c|A], S_0 \cup S_1, B, T \rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, B', T \rangle_n$
   given that c is a built-in constraint and $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
   and $wakeup(S_0 \cup S_1, c, B) = S_1$

2. **Activate** $\langle [c|A], S, B, T \rangle_n \mapsto_{activate} \langle [c\#n : 1|A], c\#n \cup S, B, T \rangle_{n+1}$ given that c is a CHR constraint.

3. **Reactivate** $\langle [c\#i|A], S, B, T \rangle_n \mapsto_{reactivate} \langle [c\#i : 1|A], S, B, T \rangle_n$ given that c is a CHR constraint.

4. **Apply** $\langle [c\#i : j|A], H_1 \cup H_2 \cup S, B, T \rangle_n \mapsto_{apply\ r}$
   $\langle C + H + A, H_1 \cup S, chr(H_1) = (H'_1) \wedge chr(H_2) = (H'_2) \wedge g \wedge B, T \cup \{(r, id(H_1) + id(H_2))\}\rangle_n$
   given that the jth occurrence of c is part of the head of the re-named apart rule with variables x':
   $r\ @\ H'_1 \setminus H'_2 \Leftrightarrow g \mid C$.
   where $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'\ ((chr(H_1) = (H'_1) \wedge chr(H_2) = (H'_2) \wedge g)))$
   and $(r, id(H_1) + id(H_2)) \notin T$.
   If c occurs in H'_1 then $H = [c\#i : j]$ otherwise $H = []$.

5. **Drop** $\langle [c\#i : j|A], S, B, T \rangle_n \mapsto_{drop} \langle A, S, B, T \rangle_n$
   given that $c\#i : j$ is an occurrenced active constraint and c has no occurrence j in the program.
   That could thus imply that all existing occurrences were tried before.

6. **Default** $\langle [c\#i : j|A], S, B, T \rangle_n \mapsto_{default} \langle [c\#i : j+1|A], S, B, T \rangle_n$
   in case there is no other applicable transition.

- Reactivate: The reactivate transition considers a constraint that was already added to the store before. It became re-activated and was added to the stack. The transition activates the constraint by associating it with an occurrence value starting with 1.

- Apply: This transition applies a CHR rule r if an active constraint matched a constraint in the head of r with the same occurrence number. If the matched constraint is part of the constraints to be removed, it is also removed from the stack. Otherwise, it is kept in the constraint store and the stack.

- Drop: This transition removes the active constraint $c\#i : j$ from the stack when there no more occurrences to check. This occurs when the occurrence number of the active constraint does not appear in the program. In other words, the existing ones were tried.

- Default: This transition proceeds to the next occurrence of the constraint if the currently active one could not be matched with the associated rule. This transition ensures that all occurrences are tried.

3 CHR^{vis}: An Animation Extension for CHR

The proposed extension aims at embedding visualization and animation features into CHR programs. The basic idea is that some constraints, the interesting ones, are annotated by visual objects. Thus on adding/removing such constraints to/from the constraint store, the corresponding graphical object is added/removed to/from the graphical store. These constraints are thus treated as interesting events. Interesting constraints are those constraints that directly represent/affect the basic data structure used along the program. Visualizing such constraints thus leads to a visualization of the execution of the corresponding program. In addition, changes in the constraint store affects the data structure and its visualization. This results in an animation of the execution. For example, in a program to encode the "Sudoku" game, the interesting constraints would be those representing the different cells in the board and their values [15, 14].

The approach aims at introducing a generic animation platform independent of the implemented algorithm. This is achieved through two features. First, annotation rules are

■ **Figure 2** Annotating the cell/2 constraint.

used. The idea of using interesting events for animating programs was introduced before in Balsa [6] and Zeus [5]. Both systems use the notion of interesting events. However, users need to know many details to be able to use them. CHR^{vis} eliminated the need for the user to know any details about the animation. The second feature is outsourcing the animation into an existing visual tool. For proof of concept, Jawaa [11], was used. Jawaa provides its users with a wide range of basic structures such as circle, rectangle, line, textual node , ... etc. Users can also apply actions on Jawaa objects such as movement, changing a color , ... etc. In order to define interesting events and their annotations, users are able to write their own CHR^{vis} programs with the syntax discussed later in this section. However, users are also provided with an interface (as shown in Figure 2) that allows them to specify every interesting event/constraint. In that case, the programs are automatically generated. They are then able to choose the visual object/action (from the list of Jawaa objects/actions) to link the constraint to. Once they make a choice, the panel is populated with the corresponding parameters. Parameters represent the visual properties of the object such as: color, x-coordinate, ... etc. Users have to specify a value for each parameter. A value could be one of/combinations of:

1. a constant value e.g. `100`, `blue`, ... etc.
2. the function `valueOf/1`. valueOf(X) outputs the value of the argument `X` such that `X` is one of the arguments of the interesting constraint.
3. the function `prologValue/1`. prologValue(Exp) outputs the value of the argument "`X`" computed through the mathematical expression `Exp`.
4. The keyword `random` that generates a random number.

3.1   Extended Programs

This section introduces the syntax of the CHR programs that are able to produce animations on execution. In addition to the basic constructs of a CHR program, the extended version needs to specify the graphical objects to be used throughout the programs. In addition, the interesting constraints and their associations with graphical objects should be described.

3.1.1   Syntax of CHR^{vis}

The annotation rules that associate CHR constraint(s) with visual objects have the following format:
$g\ opt\_rule\_name @ H_{vis} \Rightarrow Condition \mid graphical\_obj\_name\,(par_1, par_2, \ldots, par_n)\,.$
H_{vis} contains either one interesting constraint or a group of interesting constraints that are associated with a graphical object. Similar to normal CHR rules, graphical annotation rules

could have a pre-condition that has to be satisfied for the rule to be applied. The literal g is added at the beginning of the rule to differentiate between CHR rules and annotation rules. A CHR^{vis} program thus has two types of rules. There are the normal CHR rules and the annotation rules responsible for associating CHR constraint(s) with graphical object(s). Moreover, there are meta-annotation rules that associate CHR rules with graphical object(s). In this case, instead of associating CHR constraint(s) with visual object(s), the association is for a CHR rule. In other words, once such rule is executed the associated visual objects are produced. The association is thus done with the execution of the rule rather than the generation of a new CHR constraint. The rule annotation is done through associating a rule with an auxiliary constraint. The auxiliary constraint has a normal constraint annotation rule with the required visual object. Such meta-annotation rule has the following format:

$$g\; opt\_rule\_name\; @\; chr\_rule\_name\; \Rightarrow\; Condition\; |\; aux\_constraint\left(par_{1_{aux}}, \ldots, par_{m_{aux}}\right).$$

$$g\; aux\_constraint\left(par_{1_{aux}}, \ldots, par_{m_{aux}}\right)\; \Rightarrow\; graphical\_obj\_name\left(par_1, par_2, \ldots, par_n\right).$$

The CHR^{vis} program has to determine whether head constraints affect the visualization. If this is the case, the removed head constraints would result in removing the associated objects. In this case, head constraints should be comminicated to the tracer, Thus, a rule for `comm_head/1` has to be added to the CHR^{vis} program.
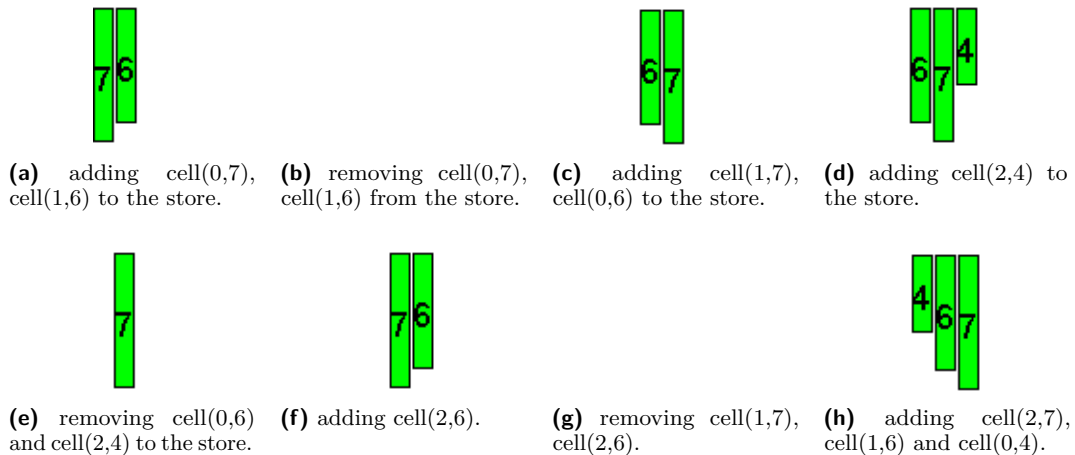The rule (`comm_head(T) ==> T=true.`) means that head constraints are to be communicated to the tracer. On the other hand, the rule (`comm_head(T) ==> T=false.`) means that the removed head constraints should not affect the visualization.

The program provided in Section 2 aims at sorting a list of numbers. In order to animate the execution, the elements of the list should be visualized. Changes of the elements lead to a change in the visualization and thus animating the algorithm. The interesting constraint in this case is the `cell` constraint. As shown in Figure 2, every `cell(Index, Value)` constraint was associated with a rectangular node whose height is a factor of the value of the element. The x-coordinate is a factor of the index. That way, the location and size of a node represent an element of the array. The new CHR^{vis} program is:

```
:-chr_constraint cell/2.
:-chr_constraint comm_head/1.
comm_head(T) ==> T=true.
sort_rule @ cell(I1,V1), cell(I2,V2) <=> I1<I2,V1>V2 |
                                    cell(I2,V1), cell(I1,V2).
g ann_rule_cell @ cell(Index,Value) ==> node(valueOf(Value),
                    valueOf(Index)*12+2,
                    50,10,valueOf(Value)*5 ,1,valueOf(Value),
                    black, green, black, RECT).
```

Figure 3 shows the result of running the query `cell(0,7),cell(1,6),cell(2,4)`. As shown from the taken steps, each number added to the list and thus to the constraint store adds a corresponding rectangular node. Once `cell(0,7)` and `cell(1,6)` are added to the constraint store, the rule `sort_rule` is applicable. Thus, the two constraints are removed from the store. The rule adds `cell(1,7)` and `cell(0,6)` to the constraint store.[1]

---

[1] More examples are available through `met.guc.edu.eg/chrvis/index.aspx`.

(a) adding cell(0,7), cell(1,6) to the store.

(b) removing cell(0,7), cell(1,6) from the store.

(c) adding cell(1,7), cell(0,6) to the store.

(d) adding cell(2,4) to the store.

(e) removing cell(0,6) and cell(2,4) to the store.

(f) adding cell(2,6).

(g) removing cell(1,7), cell(2,6).

(h) adding cell(2,7), cell(1,6) and cell(0,4).
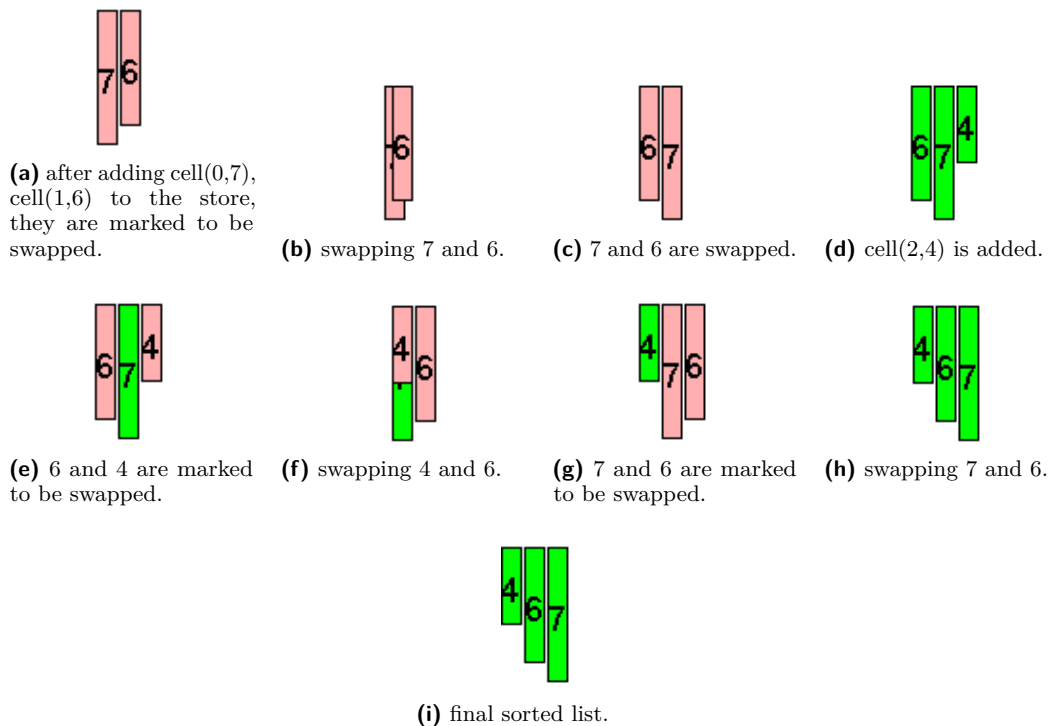
Figure 3 Sorting an array of numbers.

Afterwards, `cell(2,4)` is added to the store. At this point `cell(0,6)` and `cell(2,4)` activate `sort_rule` and are removed from the constraint store. The rule first adds `cell(2,6)` to the store. At this point `cell(1,7)` and `cell(2,6)` activate `sort_rule` again. Thus they are both removed from the store. The constraints `cell(2,7)`, `cell(1,6)` are added. Afterwards, the last constraint `cell(0,4)` is added to the store. As seen from Figure 3, using annotations for constraints has helped animate the execution of the sorting algorithm. However, in some of the steps, it might not have been clear which two numbers are being swapped. In that case it would be useful to use an annotation for the rule `sort_rule` instead of only annotating the constraint `cell`. The resulting program looks as follows:

```
:-chr_constraint cell/2.
:-chr_constraint comm_head/1.

comm_head(T) ==> T=false.
sort_rule @ cell(I1,V1), cell(I2,V2) <=> I1<I2,V1>V2 |
                  cell(I2,V1), cell(I1,V2).
g ann_rule_cell @ cell(Index,Value) ==> node(nodevalueOf(Value),
                        valueOf(Index)*12+2,50,10,
                        valueOf(Value)*5 , 1, valueOf(Value), black,
                        green, black, RECT).
g swap(I1,V1,I2,V2) ==> changeParam(nodevalueOf(V1),bkgrd,pink)
g swap(I1,V1,I2,V2) ==> changeParam(nodevalueOf(V2),bkgrd,pink)
g swap(I1,V1,I2,V2) ==> moveRelative(nodevalueOf(V1),
                        (valueOf(I2)-valueOf(I1))*12,0)
g swap(I1,V1,I2,V2) ==> moveRelative(nodevalueOf(V2),
                        (valueOf(I2)-valueOf(I1))*(-12),0)
g swap(I1,V1,I2,V2) ==> changeParam(nodevalueOf(V1),bkgrd,green)
g swap(I1,V1,I2,V2) ==> changeParam(nodevalueOf(V2),bkgrd,green)

g        sort_rule ==>        swap(I1,V1,I2,V2).
```

The annotations make sure that once two numbers are swapped, they are first marked with a different color (pink in this case). The two rectangular bars are then moved. The bar on the

(a) after adding cell(0,7), cell(1,6) to the store, they are marked to be swapped.

(b) swapping 7 and 6.

(c) 7 and 6 are swapped.

(d) cell(2,4) is added.

(e) 6 and 4 are marked to be swapped.

(f) swapping 4 and 6.

(g) 7 and 6 are marked to be swapped.

(h) swapping 7 and 6.

(i) final sorted list.

Figure 4 Sorting an array of numbers through a rule annotation.

left is moved to the right. The bar on the right is moved to the left (negative displacement). The space between the start of one node and the start of the next node is 12 pixels.

Thus the displacement is calculated as the difference between the two indeces multiplied by 12. After the swap is done, the two bars are colored back into green. The result of executing the query: `cell(0,7),cell(1,6),cell(2,4)` is shown in Figure 4.

3.2 Animation Formalization

The rest of the section offers a formalization of the animation to be able to run CHR^{vis} programs and reason about their correctness. The basic idea is introducing a new "graphical" store. CHR^{vis} adds, besides the classical constraint store of CHR, a new store called the graphical store. As implied by the name, the graphical store contains graphical/visual objects. Such objects are the visual mappings of the interesting constraints. Over the course of the program execution, and as a result of applying the different rules, the constraint store and the graphical store would change. As introduced before, the change of the visual objects leads to an animation of the program. The rest of the section introduces some needed definitions. It then proceeds to show the transitions of the new operational semantics.

▶ **Definition 3.** In CHR^{vis}, a state is represented by a tuple $\langle G, S, Gr, B, T, H\_ann \rangle_n$. G, S, B, T, and n have the same meanings as in a normal CHR state (goal store, CHR constraint store, built-in store, propagation history and the next available identification number) introduced in Section 2.1. Gr is a store of graphical objects. H_ann is the history of the applications of the visual annotation rules. Each element in H_ann has the following format: $\langle rule\_name, Head\_ids, Object\_ids \rangle$ where

- $rule\_name$ represents the name of the fired annotation rule.

- $Head\_ids$ contain the ids of the head constraints that fired the annotation rule.
- $Object\_ids$ are the ids of the graphical objects added to the graphical store through firing $rule\_name$ using $Head\_ids$.

▶ **Definition 4.** For a sequence $Sq = (c_1\#id_1, \ldots, c_n\#id_n)$, the function $get\_constraints\,(Sq) = (c_1 \ldots, c_n)$.

▶ **Definition 5.** Two sequences A and B are equivalent: $A \doteq B$ if
1. For every X, if X exists N times in A such that $N > 0$, then X exists N times in B.
2. For every Y, if Y exists N times in B such that $N > 0$, then Y exists N times in A.

▶ **Definition 6.**

The function $output\_graphical\_object(c(Arg_0, \ldots, Arg_n), \{Arg_0', \ldots, Arg_n'\},$
$output(Object, OArg_0, \ldots, OArg_k)) = graphical\_object(Actual_0, \ldots, Actual_k)$ such that:
- $graphical\_object = Object$.
- Each parameter $Actual_n = get\_actual\,(OArg_n)$ such that
    - if $OArg_n$ is a constant value then $get\_actual\,(OArg_n) = OArg_n$.
    - if $OArg_n = valueOf\,(Arg_m)$ then $get\_actual\,(OArg_n) = (Arg_n')$.
    - if $OArg_n = prologValue\,(Expr)$ then $get\_actual\,(OArg_n) = X$ where $Expr$ is evaluated in SWI-Prolog and binds the variable X to a value.
    - if $OArg_n = random$, then $get\_actual\,(OArg_n)$ is a randomly computed number.

▶ **Definition 7.**

The function
$generate\_new\_ann\_history\,(Graph\_obj, Obj\_id, rule\_name, Head\_id, H\_ann) =$
H'_ann such that: in the case where $\langle rule\_name, Head\_id, Objects\_ids \rangle \in H\_ann$,
$H'\_ann = H\_ann - \langle rule\_name, Head\_id, Objects\_ids \rangle$
$\cup \langle rule\_name, Head\_id, Objects\_ids \cup \{Obj\_id\} \rangle$,

▶ **Definition 8.**

The function $remove\_gr\_obj\,(G\_store, Rem\_head\_id, H\_ann) = G'\_store$ such that: in the case where there is some Tuple $T : \langle rule\_name, Head\_ids, Objects\_ids \rangle$ such that
$T \in H\_ann \wedge Rem\_head\_id \subseteq Head\_ids$.
In this case, $G'\_store = G\_store - \cup_i (Obj_i \; where \; Obj_i \in Objects\_ids)$.

▶ **Definition 9.**

The function $contains\,(H\_ann, \langle rule, Head_{ids} \rangle)$ is:
- $true$ in the case where H_ann contains a tuple of the form $\langle rule, Head_{ids}, Objects \rangle$.
- $false$ in the case where H_ann does not contain a tuple of the form $\langle rule, Head_{ids}, Objects \rangle$.

Table 2 shows the basic transitions of ω_{vis}. To make the transitions easier to follow, table 2 shows the transitions needed to run CHR programs with constraint annotation rules. Annotations of CHR rules are thus discarded from the set of transitions. ω_{vis} allows for running programs that contain constraint annotations. The three transitions *apply_annotation*, *draw* and *updatestore* are responsible for dealing with the graphical store and its constituents. The transition, *apply_annotation*, applies a constraint annotation rule. The rest of the transitions, such as solve, introduce and apply, have the same behavior as in ω_r. These transitions do not affect the graphical store or the application history of the annotation rules. The transitions affecting the graphical store are:

■ **Table 2** Transitions of ω_{vis}.

1. **Solve+wakeup**:
   $\langle[c|A], S_0 \cup S_1, Gr, B, T, H\_ann\rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, Gr, B', T, H\_ann\rangle_n$
   given that c is a built-in constraint and $\mathcal{CT} \models \forall\left(\left(c \wedge B \leftrightarrow B'\right)\right)$
   and $wakeup\left(S_0 \cup S_1, c, B\right) = S_1$

2. **Activate**:
   $\langle[c|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{activate} \langle[c\#n : 1|A], \{c\#n\} \cup S, Gr, B, T, H\_ann\rangle_{n+1}$
   given that c is a CHR constraint.

3. **Reactivate**:
   $\langle[c\#i|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{reactivate} \langle[c\#i : 1|A], S, Gr, B, T, H\_ann\rangle_n$
   given that c is a CHR constraint.

4. **Draw**:
   $\langle[\langle Obj\#\langle r, id\,(H)\rangle\rangle|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{draw}$
   $\langle A, S, Gr \cup \{Obj\#n\}, B, T, H\_ann'\rangle_{n+1}$
   given that Obj is a graphical object: $graphical\_object\,(Actual_0, \ldots, Actual_k)$.
   and $H\_ann' = generate\_new\_ann\_history\,(Obj, n, r, id\,(H), H\_ann)$
   The actual parameters of $graphical\_object$ are used to visually render the object.

5. **Update Store**:
   $\langle[\langle Obj\#\langle r, id\,(H)\rangle\rangle|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{updatestore} \langle A, S, Gr', B, T, H\_ann\rangle_n$
   given that Obj is a graphical action: $graphical\_action\,(Actual_0, \ldots, Actual_k)$.
   $Gr' = update\_graphical\_store\,(Gr, graphical\_action\,(Actual_0, \ldots, Actual_k))$
   The function $update\_graphical\_store$ uses the actual parameters of $graphical\_action$ to update
   the attributes of the graphical objects available in the graphical store tht are affected by the action.

6. **Apply_Annotation**:
   $\langle[c\#i : j|A], H \cup S, Gr, B, T, H\_ann\rangle_n \mapsto_{apply\_annotation}$
   $\langle[Obj\#\langle r, id\,(H)\rangle, c\#i : j|A], H \cup S, Gr, B, T, H\_ann \cup \{\langle r, id\,(H), \{\}\rangle\}\rangle_n$
   where there is: a renamed, constraint annotation rule with variables y' of the form:
   $g\ r\ @\ H'\ ==> Condition \mid Obj'$
   where c is part of H' and
   $(CT) \models \exists(B)$
   $\wedge \forall(B \Rightarrow \exists y'(chr(H) = (H') \wedge Condition \wedge output\_graphical\_object(H', y', Obj') = Obj))$
   and $\neg(contains(H\_ann, (r, id(H))))^{\dagger}$

7. **Apply**:
   $\langle[c\#i : j|A], H_k \cup H_r \cup S, Gr, B, T, H\_ann\rangle_n \mapsto_{apply}$
   $\langle C + H + A, H_k \cup S, Gr, chr\,(H_k) = (H'_k) \wedge chr\,(H_r) = (H'_r) \wedge G \wedge B$
   $T \cup \{\langle r, id\,(H_k) + id\,(H_r)\rangle\}, H\_ann\rangle_n$ where:
   - there is no applicable constraint annotation rule for c (or part of it).
     (i.e. every applicable rule has already been applied).
     In other words, for renamed-apart every annotation rule with variables y':
     $g\ r\ @\ H'\ ==> Cond \mid Obj'$ where,
     c is part of $H' \wedge (CT) \models \exists(B) \wedge \forall\left(B \implies \exists y'(chr(H) = (H') \wedge Condition)\right)$
     , it is already the case that: $(contains\,(H\_ann, (r, id\,(H)))) = true$
   - There is a renamed rule in P_{vis} with the form $r\ @\ H'_k \setminus H'_r \Leftrightarrow G \mid C$.
     with variables x' and the jth occurrence of c is part of the head of the renamed rule,
     where $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'\left((chr\,(H_k) = (H'_k) \wedge chr\,(H_r) = (H'_r) \wedge G)\right)$
     and $\langle r, id\,(H_k) + id\,(H_r)\rangle \notin T$.
     If c occurs in H'_k then $H = [c\#i : j]$ otherwise $H = []$.
     If the program communicates the head constraints (i.e. contains `comm_head(T) ==> T=true`)
     then $Gr' = remove\_gr\_obj\,(G, id\,(H_r), H\_ann)$

8. **Drop**:
   $\langle[c\#i : j|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{drop} \langle A, S, Gr, B, T, H\_ann\rangle_n$
   given that $c\#i : j$ is an occurrenced active constraint
   and c has no occurrence j in the program
   and that there is no applicable constraint annotation rule for the constraint c.
   That could thus imply that all existing ones were tried before.

9. **Default**:
   $\langle[c\#i : j|A], S, Gr, B, T, H\_ann\rangle_n \mapsto_{default} \langle[c\#i : j + 1|A], S, Gr, B, T, H\_ann\rangle_n$
   in case there is no other applicable transition.

---

† For simplicity, the annotation rule is considered to contain one graphical output object. In general, the rule could associate constraint(s) with multiple objects.

1. *Draw:* The new transition *draw* adds a graphical object (*Obj*) to the graphical store. Since multiple copies of a graphical object are allowed, each object is associated with a unique identifier.
2. *Update Store:* This transition applies a graphical action to the objects in the graphical store. This could thus change some of the aspects of the drawn graphical object(s).
3. *Apply_Annotation:* The *Apply_Annotation* transition applies a constraint annotation rule (ann_rule). An annotation rule is applicable if the CHR constraint store contains matching constraints. The condition of the rule has to be implied by the built in store under the matching. The built in constraint store B is also first checked for satisfiability. For the rule to be applied, it should not have appeared in the history of applied annotation rules with the same constraints i.e. it should be the first time the constraint(s) fire this annotation rule. Executing the rule adds to the goal the graphical object in the body of the executed annotation rule. The history of annotation rules is updated accordingly with the name of the rule in addition to the id(s) of the CHR constraint(s) in the head. In fact, this transition has a higher precedence than the transition *apply*. Thus in the case where an annotation rule and a CHR rule are applicable, the annotation rule is triggered first. The precedence makes sure that graphical objects are added in the intended order to ensure producing correct animations.

▶ **Definition 10** (Built-In Store Equivalence). Two built-in constraint stores B_1 and B_2 are considered equivalent iff:
$(CT) \models \forall(\exists_{y1}(B_1) \leftrightarrow \exists_{y2}(B_2))$ where y_1 and y_2 are the local variables inside B_1 and B_2 respectively. The equivalence thus basically ensures that there are no contradictions in the substitutions since local variables are renamed apart in every CHR program. The equivalence check thus ensures the logical equivalence rather than the syntactical equivalence.

▶ **Definition 11.** A CHR^{vis} state $St_{vis} = \langle G_{vis}, S_{vis}, Gr_{vis}, B_{vis}, T_{vis}, T_{visAnn} \rangle_{n_{vis}}$ is equivalent to a CHR state $St = \langle G, S, B, T \rangle_n$ if and only if
1. $get\_constraints(G_{vis}) \doteq get\_constraints(G)$ according to Definition 5.
2. $get\_constraints(St_{vis}) \doteq get\_constraints(S) = C$ according to Definition 5.
3. B_{vis} and B are equivalent according to Definition 10.
4. $T_{vis} = T$
5. $n_{vis} \geq n$

The idea is that a CHR^{vis} state basically has an extra graphical store. The correspondence check is effectively done through the CHR constraints since they are the most distinguishing constituents of a state. Thus, the constraint store and the stack should contain the same constraints. The propagation history should be also the same indicating that the same CHR rules have been applied. n_{vis} could, however, have a value higher than n. This is due to the fact that graphical objects have identifiers. The definition of state equivalence described here follows the properties introduced in [12]. However, it is stricter.

▶ **Theorem 12** (Soundness). *Given a CHR program P (running under ω_r) along with its user defined annotations and its corresponding $P_{CHR^{vis}}$ program (running under ω_{vis}), for the same query Q, every derived state $S_{chr_{vis}}$: $Q \mapsto^*_{\omega_{vis}} S_{chr_{vis}}$ has en equivalent state S_{chr}: $Q \mapsto^*_{\omega_r} S_{chr}$*

Proof.
Base Case:
For the initial query the two states $Q, S_{chr_{vis}} = \langle Q, \{\}, \{\} \rangle$ and $S_{chr} = \langle Q, \{\} \rangle$ are equivalent according to Definition 11.

Induction Hypothesis: Suppose that there are two equivalent derived states
$S_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann \rangle_m$ and $S_{chr} = \langle A, S, B, T \rangle_n$ such that $Q \mapsto^i_{\omega_{vis}} S_{chr_{vis}}$ and
$Q \mapsto^j_{\omega_r} S_{chr}$.
Induction Step:
The proof shows that any transition applicable to $S_{chr_{vis}}$ under ω_{vis} produces a state S'_{chr}
such that under ω_r applying a transition to S_{chr} (which is equivalent to $S_{chr_{vis}}$) produces a
state S'_{chr} that is equivalent to S_{chr}.
The different cases are enumerated below:

1. **Applying solve+wakeup to $\mathbf{S_{chr_{vis}}}$:**
   Under ω_{vis}, solve+wakeup is applicable in the case where the stack has the form $[c|A]$
   such that c is a built-in constraint and $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
   and $wakeup(S_0 \cup S_1, c, B) = S_1$ such that
   $S_{chr_{vis}} \mapsto_{solve+wake} S'_{chr_{vis}} : \langle S_1 + A, S_0 \cup S_1, Gr, B', T, H\_ann \rangle_m$. Since $S_{chr_{vis}}$ and S_{chr}
   are equivalent, S_{chr} has an equivalent stack and built-in store according to Definition 11.
   Thus the corresponding transition *solve+wakeup* is applicable to S_{chr} under ω_r producing
   a state S'_{chr} such that: $S'_{chr} = \langle S_1 + A, S_0 \cup S_1, B', T \rangle_n$. According to Definition 11, the
   two states $S'_{chr_{vis}}$ and S'_{chr} are equivalent.

2. **Applying Activate:**
   Such a transition is applicable to $S_{chr_{vis}}$ under ω_{vis} in the case where the top of the stack
   of $S_{chr_{vis}}$ contains a CHR constraint c. In this case:
   $S_{chr_{vis}} : \langle [c|A], S, Gr, B, T, H\_ann \rangle_m \mapsto_{activate}$
   $$S'_{chr_{vis}} : \langle [c\#m:1|A], \{c\#m\} \cup S, Gr, B, T, H\_ann \rangle_{m+1}$$
   given that c is a CHR constraint.
   The equivalent state S_{chr} has the same stack triggering the transition *Activate* under
   ω_r producing a state $S'_{chr} : \langle [c\#n:1|A], \{c\#n\} \cup S, Gr, B, T, H\_ann \rangle_{n+1}$ which is also
   equivalent to $S'_{chr_{vis}}$

3. **Applying Reactivate:**
   In this case, $S_{chr_{vis}} \mapsto_{reactivate} S'_{chr_{vis}} \langle [c\#i:1|A], S, Gr, B, T, H\_ann \rangle_m$
   such that $S_{chr_{vis}} = \langle [c\#i|A], S, Gr, B, T, H\_ann \rangle_m$ and c is a CHR constraint.
   The equivalent state S_{chr} has an equivalent stack triggering the transition *reactivate*
   under ω_r. The transition application produces $S'_{chr} : \langle [c\#i:1|A], S, B, T \rangle_n$ which is also
   equivalent to $S'_{chr_{vis}}$.

4. According to Definition 11 and since $S_{chr_{vis}}$ is equivalent to S_{chr}, they both have the
   same stack. The transition **Draw** is only applicable if the top of the stack contains a
   graphical object. Since the stack of S_{chr} never contains graphical objects and since it is
   equivalent to $S_{chr_{vis}}$, the stack of $S_{chr_{vis}}$ at this point does not contain graphical objects
   as well. Thus, in this case, the transition *draw* would not be applicable to $S_{chr_{vis}}$ under
   ω_{vis}.

5. Similarly, according to Definition 11 and since $S_{chr_{vis}}$ is equivalent to S_{chr}, the stack of
   $S_{chr_{vis}}$ at this point does not contain graphical actions since both states should have the
   same stack. The transition *update store* is only applicable if the top of the stack contains
   a graphical action. Thus, similarly, at this point, the transition *update store* could not be
   applied to $S_{chr_{vis}}$ under ω_{vis}.

6. ***Apply Annotation Rule* Transition:**
   The transition *Apply Annotation* is triggered when the stack has on top a constraint
   associated with an annotation rule. The constraint store should contain constraints
   matching the head of the annotation rule such that this rule was not fired with those
   constraint(s) before and the pre-condition of the annotation rule is satisfied. Thus, the

rule could be associated with more than one constraint including the one on top of the stack. The constraint store should however, contain matching constraints for the rest of the constraints in the head of the annotation rule.

$S_{chr_{vis}} \mapsto_{apply\_annotation} S'_{chr_{vis}}$:

$\langle [Obj \# \langle r, id(H) \rangle | A], H \cup S, Gr, B, T, H\_ann \cup \{ \langle r, id(H), \{ \} \rangle \} \rangle_m$

such that $\neg contains(H\_ann, \langle r, id(H) \rangle)$. The renamed annotation rule with variables x' is :

$g \ r \ @ \ H' \ ==> \ Condition \ | \ Obj'$

$(CT) \models \exists(B) \wedge \forall(B \implies \exists x'((chr(H) =$

$H' \wedge Cond \wedge output\_graphical\_object(H', x', Obj') = Obj)))$

Either the transition *draw* or *update store* is applicable to $S'_{chr_{vis}}$. The output is $S''_{chr_{vis}} : \langle A, S, Gr', T, H''\_ann \rangle_{m'}$. In case, Obj is a graphical object, then $H'\_ann = generate\_new\_ann\_history(Obj, m, r, id(H), H\_ann \cup \{ \langle r, id(H), \{\} \rangle \}) \wedge Gr' = Gr \cup \{ Obj \# m \} \wedge m' = m + 1$. In case, Obj is a graphical action, then $Gr' = update\_graphical\_store(Gr, Obj) \wedge Gr' = Gr \wedge m' = m$. Any transition applicable to $S''_{chr_{vis}}$ at this stage is covered through the rest of the cases. Thus the application of the transition *apply_annotation* is considered as not to affect the equivalence of the output state with S_{chr}.

7. **The Apply transition:**
   In the case where a CHR rule is applicable to $S_{chr_{vis}}$, the transition *Apply* is triggered under ω_{vis}. A CHR rule r is applicable when there is a renamed version of the rule r with variables x': $(r @ H'_k \setminus H'_r \Leftrightarrow g \mid C.)$ where $\langle r, id(H_k) + id(H_r) \rangle \notin T$ and $\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'(chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g))$. In this case, $S_{chr_{vis}}$ has the form: $\langle [c \# i : j | G], H_k \cup H_r \cup S, Gr, B, T, H\_ann \rangle_m$. The output state $S'_{chr_{vis}}$ has the form
   $\langle C + H + G, H_k \cup S, Gr, B \wedge chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g, T \cup \{ \langle r, id(H_k) + id(H_r) \rangle \}, H\_ann \rangle_m$. Due to the fact that S_{chr} is equivalent to $S_{chr_{vis}}$, it has the following form: $\langle [c \# i : j | G], H_k \cup H_r \cup S, B, T \rangle_n$. For the same program, the CHR rule r is applicable producing S'_{chr}:
   $\langle C + +H + +G, H_k \cup S, chr(H_k) = (H'_k) \wedge chr(H_r) = (H'_r) \wedge g \wedge B, T \cup \{ \langle r, id(H_k) + id(H_r) \rangle_n \}$

   $$H = \begin{cases} [c \# i : j] & \text{if } c \text{ occurs in } H'_k \\ [\,] & \text{otherwise} \end{cases}$$

   We assume, without loss of generality, that the same renaming variables are used in both cases. Due to the fact that the same CHR rule is applied for both states, the new built-in stores are equivalent according to Definition 10. This is due to the fact that since the original states have equivalent constraint stores, we assume without loss of generality that the matchings in both cases are the same since the same rule was applied. Thus, the rule in the two programs P_{chr} and $P_{chr_{vis}}$ are renamed similarly. Since no annotation rule could be applied to a non-occurred constraint and according to Definition 11, the two states are equivalent.

8. **Applying Drop:**
   In the case where $S_{chr_{vis}} = \langle [c \# i : j | A], S, Gr, B, T, H\_ann \rangle_m$ such that c has no occurrence j in the program and case 5 is not applicable, the transition *Drop* is triggered. *Drop* produces the state $S'_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann \rangle_m$
   Since S_{chr} is equivalent to $S_{chr_{vis}}$, they both have the same stack $[c \# i : j | A]$. Thus under

ω_{vis}, the same transition *drop* is triggered producing $S'_{chr} : \langle A, S, B, T\rangle_n$. According to Definition 11, $S'_{chr_{vis}}$ and S'_{chr} are equivalent as well.

9. Applying Default:

In the case where none of the above cases hold, the transition *Default* transforms $S_{chr_{vis}}$ to

$S'_{chr_{vis}} : \langle [c\#i : j + 1|A], S, Gr, B, T, H\_ann\rangle_m$. Similarly the equivalent state S_{chr} triggers the same transition *Default* in this case. The output state $S'_{chr} : [c\#i : j + 1|A], S, B, T\rangle_n$ is still equivalent to $S'_{chr_{vis}}$

Thus in all cases an equivalent state is produced under ω_r ◀

▶ **Theorem 13** (Completeness). *Given a CHR program P (running under ω_r) along with its user defined annotations and its corresponding $P_{CHR^{vis}}$ (running under ω_{vis}) program, for the same query Q, every derived state S_{chr}: $Q \mapsto^*_{\omega_r} S_{chr}$ has an equivalent state $S_{chr_{vis}}$: $Q \mapsto^*_{\omega_{vis}} S_{chr_{vis}}$.*

For space limitations, the proof is given in B.

4   Conclusions

In conclusion, the paper presented a formalization for embedding animation features into CHR programs. The new extension, CHR^{vis} is able to allow for dynamic associations of constraints and rules with visual objects. The annotation rules are thus activated on the execution of the program to produce algorithm animations. Although the idea of using interesting events was introduced in earlier work, it was (to the best knowledge of the authors) never formalized before. In fact, no operational semantics for animation was proposed before. The paper offered operational semantics for CHR^{vis}. It thus provides a foundation for formalizing the animation process in general and for CHR programs in particular. In the future, with the availability of formal foundations through ω_{vis}, the possibility of using CHR^{vis} as the base of a pure a visual representation for CHR should be investigated. A group of students in the German University in Cairo were exposed to the classic textual tracer and the new visual racing facility in a focus group. Most of the students stated that for them it was hard to use the textual trace to understand how a program works. They preferred to see the visual tracer which according to a conducted survey helped them understand what the presented CHR programs do.

─── **References** ───

1   Slim Abdennadher, Ekkerhard Krämer, Matthias Saft, and Matthias Schmauss. JACK: A java constraint kit. *Electr. Notes Theor. Comput. Sci.*, 64:1–17, 2002.

2   Slim Abdennadher and Matthias Saft. A Visualization Tool for Constraint Handling Rules. In *In Proceedings of 11th Workshop on Logic Programming Environments*, 2001.

3   Slim Abdennadher and Nada Sharaf. Visualization of CHR through Source-to-Source Transformation. In Agostino Dovier and Vítor Santos Costa, editors, *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 109–118. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012. `doi:10.4230/LIPIcs.ICLP.2012.109`.

4   Hariolf Betz, Frank Raiser, and Thom Frühwirth. A Complete and Terminating Execution Model for Constraint Handling Rules. *Theory and Practice of Logic Programming*, 10:597–610, July 2010.

5   Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *VL*, pages 4–9, 1991.

6 Marc H. Brown and Robert Sedgewick. A System for Algorithm Animation. In *Proceedings of*, SIGGRAPH '84, pages 177–186, New York, NY, USA, 1984. ACM.

7 Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference, ICLP 2004, Saint-Malo, France, September 6-10, 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2004.

8 Thom Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, 37(1-3):95–138, October 1998.

9 Thom Frühwirth. *Constraint Handling Rules*. Cambridge University Press, August 2009.

10 Thom W. Frühwirth. Constraint Handling Rules - What Else? In Nick Bassiliades, Georg Gottlob, Fariba Sadri, Adrian Paschke, and Dumitru Roman, editors, *Rule Technologies: Foundations, Tools, and Applications - 9th International Symposium, RuleML 2015, Berlin, Germany, August 2-5, 2015, Proceedings*, volume 9202 of *Lecture Notes in Computer Science*, pages 13–34. Springer, 2015.

11 Willard C. Pierson and Susan H. Rodger. Web-based Animation of Data structures using JAWAA. In John Lewis, Jane Prey, Daniel Joyce, and John Impagliazzo, editors, *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, 1998, Atlanta, Georgia, USA, February 26 - March 1, 1998*, pages 267–271. ACM, 1998.

12 GFrank Raiser, Hariolf Betz, and Thom Frühwirth. Equivalence of CHR states revisited. In *6th International Workshop on Constraint Handling Rules (CHR)*, pages 34–48, 2009.

13 Matthias Schmauss. An Implementation of CHR in Java. Master's thesis, Master Thesis, Institute of Computer Science, LMU, Munich, Germany, November 1999.

14 Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. CHRAnimation: An Animation Tool for Constraint Handling Rules. In Maurizio Proietti and Hirohisa Seki, editors, *Logic-Based Program Synthesis and Transformation - 24th International Symposium, LOPSTR 2014, Canterbury, UK, September 9-11, 2014. Revised Selected Papers*, volume 8981 of *Lecture Notes in Computer Science*, pages 92–110. Springer, 2014.

15 Nada Sharaf, Slim Abdennadher, and Thom W. Frühwirth. Visualization of Constraint Handling Rules. *CoRR*, abs/1405.3793, 2014.

16 John Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, May 1992.

A CHR^{vis} to CHR^r Transformation Approach

The aim of the transformation is to eliminate the need of doing any compiler modifications in order to animate CHR programs. A CHR^{vis} program P^{vis} is thus transformed to a corresponding CHR^r program P with the same behavior. P is thus able to produce the same states in terms of CHR constraints and visual objects as well. A similar transformation was introduced in [14].

As a first step, the transformation adds for every constraint `constraint/n` a rule of the form:

$$comm\_cons\_constraint @ constraint\,(X_1, X_2, ..., X_n) \Rightarrow check\,(status, false)\mid$$
$$communicate\_constraint\,(constraint\,(X_1, X_2, ..., X_n))\,.$$

The extra rule ensures that every time a `constraint` is added to the store, the tracer (*external module*) is notified. If `constraint` was annotated as an interesting constraint, its corresponding annotation rule is activated producing the corresponding visual object(s). The new rules communicate any `constraint` added to the constraint store.

The user can also choose to communicate to the tracer the head constraints since they could affect the animation. A removed head constraint could affect the visualization in case it is an interesting constraint. In this case, if the user chose to communicate head constraints, the associated visual object, produced before, should be removed from the visual trace.[2].

As a second step, the transformer adds for every compound constraint-annotation of the form:

$cons_1, \ldots, cons_n ==> annotation\_constraint_{cons_1,\ldots,cons_n} (Arg1, \ldots, Arg_m)$, a new rule of the form:

$compound_{cons_1,\ldots,cons_n} @ cons_1 \left(Arg_{cons_{1_1}}, \ldots, Arg_{cons_{1_{1x}}}\right), \ldots, cons_n \left(Arg_{cons_{n_1}}, \ldots, Arg_{cons_{n_{ny}}}\right)$
$\Rightarrow check\,(status, false) \mid annotation\_constraint_{cons_1,\ldots,cons_n} (Arg1, \ldots, Arg_m).$

By default, a propagation rule is produced to keep $cons_1, \ldots, cons_n$ in the constraint store. However, the transformer could be instructed to produce a simplification rule instead. The annotation is triggered whenever $cons_1, \ldots, cons_n$ exist in the constraint store. Whenever this is the case, the rule $compound_{cons_1,\ldots,cons_n}$ is triggered producing the annotation constraint. Since the annotation constraint is a normal CHR constraint, it is automatically communicated to the tracer using the previous step.

As a third step, the CHR rules annotated by the user as interesting rules should be transformed. The idea is that the CHR constraints produced by such rules should be ignored. In other words, even if the rule produces an interesting CHR constraint, it should not trigger the corresponding constraint annotation. Instead, the rule annotation is triggered.

Hence, to avoid having problems with this case, a generic *status* is used throughout the transformed program P_{Trans}. Any rule annotated by the user as an interesting rule changes the `status` to *true* at execution. However, the rules added in the previous two steps check that the status is set to *false*. In other words, if the interesting rule is triggered, no constraint is communicated to the tracer since the guard of the corresponding *communicate_constraint* rule fails. Any rule $rule_i @ H_K \setminus H_R \Leftrightarrow G \mid B$ with the corresponding annotation $rule_i ==> annotation\_constraint_{rule_i}$ is transformed to:
$rule_i @ H_K \setminus H_R \Leftrightarrow G \mid set\,(status, true)\,, B, annotation\_constraint_{rule_i},$
$set\,(status, false)\,.$ In addition, the transformer adds the following rule to P_{Trans}:
$comm\_cons_{annotation\_constraint_{rule_i}} @ annotation\_constraint_{rule_i} \Leftrightarrow$
$$communicate\_constraint\,(annotation\_constraint_{rule_i}).$$
The new rule thus ensures that the events associated with the rule annotation are considered and that all annotations associated with the constraints in the body of the rule are ignored.

The aim of the transformation process is to produce a CHR^r program (P_{trans}) that is able to perform the same behavior of the corresponding CHR^{vis} program (P_{vis}) which basically contains the original CHR program P along with the constraint(s) and rule annotations. This section shows that the transformed program, using the steps shown previously, is a correct one. In other words, for the same query Q, P_{trans} produces an equivalent state to the one produced by P. As seen from the previous section ω_{vis} was proven to be sound and complete. This implies that any state reachable by ω_r is also reachable by ω_{vis}. In addition, any state reachable by ω_{vis} is also reachable by ω_r. The focus of this section is the initial CHR program provided by the user. The aim is to make sure that P_{trans} produces the same CHR constraints that P produces to make sure that the transformation did not change the behavior that was initially intended by the programmer. The focus is thus to compare how P and P_{trans} perform over ω_r.

---

[2] The tracer is able to handle the problem of having multiple Jawaa objects with the same name by removing the old object having the same name before adding the new one. This is possible even if the removed head constraint was not communicated.

B   Completeness Proof

Proof.
Base Case: For a given query Q, the initial state in ω_r is $S_{chr} = \langle Q, \{\}, \{\}, \{\} \rangle_1$. The initial state in ω_{vis} is $S_{chr_{vis}} = \langle Q, \{\}, \{\}, \{\}, \{\}, \{\} \rangle_1$.[3] According to Definition 11 S_{chr} and $S_{chr_{vis}}$ are equivalent.

Induction Hypothesis:
Suppose that there are two equivalent derived states $S_{chr} = \langle A, S, B, T \rangle_n$ and $S_{chr_{vis}} = \langle A, S, Gr, B, T, H\_ann \rangle_m$ such that $Q \mapsto^i_{\omega_r} S_{chr}$ and $Q \mapsto^j_{\omega_{vis}} S_{chr_{vis}}$.

Induction Step:
According to the induction hypothesis, S_{chr} and $S_{chr_{vis}}$ are equivalent. The rest of the proof shows that any transition applicable to S_{chr} in ω_r produces a state that has an equivalent state produced by applying a transition to $S_{chr_{vis}}$ in ω_{vis}. Thus, no matter how many times the step is repeated, the output states are equivalent.

- **Applying solve+wakeup:**
  In this case, $S_{chr} \mapsto S'_{chr}$ such that:
  $S_{chr} : \langle [c|A], S_0 \cup S_1, B, T \rangle_n \mapsto_{solve+wake} \langle S_1 + A, S_0 \cup S_1, B', T \rangle_n$
  Transition *solve+wakeup* is applicable if:
  1. c is a built-in constraint
  2. $\mathcal{CT} \models \forall((c \wedge B \leftrightarrow B'))$
  3. $wakeup(S_0 \cup S_1, c, B) = S_1$
  $S_{chr_{vis}}(\langle Stack, S_{chr_{vis}}, Gr, B_{vis}, T_{vis}, T_{ann} \rangle_m)$ is equivalent to $S_{chr}(\langle [c|A], S_0 \cup S_1, B, T \rangle_n)$. Thus according to Definition 11, $Stack = [c|A] \wedge S_{chr_{vis}} = S_0 \cup S_1 \wedge B_{vis} = B \wedge T_{vis} = T \wedge m \geq n$. Thus accordingly, the transition $solve + wakeup$ is applicable to $S_{chr_{vis}}$ under ω_{vis} producing $S'_{chr_{vis}} : \langle S_1 + A, S_0 \cup S_1, Gr, B \wedge c, T, H\_ann \rangle_m$. According to Definition 11, S'_{vis} is equivalent to S'_{chr}

- **Applying Activate:**
  In this case, $S_{chr} = \langle [c|A], S, B, T \rangle_n$ where c is a CHR constraint. Thus $S_{chr} \mapsto_{activate} S'_{chr} : \langle [c\#n : 1|A], c\#n \cup S, B, T \rangle_{n+1}$.
  Since $S_{chr_{vis}}(\langle Stack, S_{chr_{vis}}, Gr, B_{vis}, T_{vis}, T_{ann} \rangle_m)$ is equivalent to $S_{chr}(\langle [c|A], S_0 \cup S_1, B, T \rangle_n)$. Thus according to Definition 11: $Stack = [c|A] \wedge S_{chr_{vis}} = S \wedge B_{vis} = B \wedge T_{vis} = T \wedge m \geq n$
  Accordingly, $S_{chr_{vis}} \mapsto_{activate} S'_{chr_{vis}} : \langle [c\#m : 1|A], \{c\#m\} \cup S, Gr, B, T, T_{ann} \rangle_{m+1}$ which is equivalent to S'_{chr}. (Since $m \geq n$, then $m + 1 \geq n + 1$).

- **Applying Reactivate:**
  The transition *reactivate* is applicable if the stack has on top of it an element of the form $c\#i$ where c is a CHR constraint. In this case $S_{chr} = \langle [c\#i|A], S, B, T \rangle_n$. Accordingly, $S_{chr} \mapsto_{reactivate} S'_{chr} : \langle [c\#i : 1|A], S, B, T \rangle_n$. Since $S_{chr_{vis}}$ and S_{chr} are equivalent, then $S_{chr_{vis}}$ has the same stack. $S_{chr_{vis}} = \langle [c\#i|A], S, Gr, B, T, T_{ann} \rangle_m$ triggers the transition reactivate producing $S'_{chr_{vis}} : \langle [c\#i : 1|A], S, Gr, B, T, T_{ann} \rangle_m$ which is also equivalent to S'_{chr}. Since c is not associated with an occurrence yet, no annotation rule is applicable at this point.

---

[3] Throughout the different proofs, identifiers are omitted for brevity.

■ **Applying the transition Apply**

The transition *Apply* is triggered under ω_r in the case where $S_{chr} = \langle [c\#i : j|A], H_1 \cup H_2 \cup S, B, T \rangle_n$ such that the jth occurrence of c is part of the head of the re-named apart rule with variables x': $r @ H_1' \setminus H_2' \Leftrightarrow g \mid C$.

such that:

$\mathcal{CT} \models \exists(B) \wedge \forall(B \implies \exists x'(chr(H_1) = (H_1') \wedge chr(H_2) = (H_2') \wedge g)))$ and $\langle r, id(H_1) + id(H_2) \rangle \notin T$.

Thus in such a case $S_{chr} \mapsto_{apply\ r} S_{chr}' : \langle C + H + A, H_1 \cup S, chr(H_1) = (H_1') \wedge chr(H_2) = (H_2') \wedge g \wedge B, T \cup \{\langle r, id(H_1) + id(H_2) \rangle\} \rangle_n$

$$H = \begin{cases} [c\#i : j] & \text{if } c \text{ occurs in } H_1' \\ [\,] & \text{otherwise} \end{cases}$$

Due to the fact that S_{chr} and $S_{chr_{vis}}$ are equivalent, in the case where S_{chr} triggers the transition *Apply* under ω_r, the same rule is also applicable under ω_{vis} to $S_{chr_{vis}}$. However for $S_{chr_{vis}}$, one of two possibilities could happen:

1. There is no applicable constraint annotation rule:

   This could be due to the fact that any applicable annotation rule was already executed or that there are no applicable annotation rules at this point. In this case, the transition *apply* is triggered right away under ω_{vis} producing a state $(S_{chr_{vis}}' : \langle C + H + A, H_1 \cup S, Gr, chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge g \wedge B, T \cup \{\langle r, id(H_1) + id(H_2) \rangle, H\_ann\} \rangle_m)$ equivalent to (S_{chr}'). The original states are equivalent and the same rule is applied in both cases. We can assume that, without loss of generality , in the chr_{vis} program, the rule is renamed using the same variables x' resulting in the same matching. This is because the same matching should happen to be able to apply the same rule using the given constraint stores.

2. There is an applicable annotation rule:

   In this case an annotation rule (r_{ann}) for c is applicable such that:
   $S_{chr_{vis}} \langle [c\#i : j|A], H_1 \cup H_2 \cup S, Gr, B, T, H\_ann \rangle_m \mapsto_{apply\_annotation}$
   $S_{chr_{vis}}' : \langle [Obj\#\langle r, id(H) \rangle, c\#i : j|A], H_1 \cup H_2 \cup S, Gr, B, T, H\_ann \cup \{\langle r_{ann}, id(H), \{\ \}\rangle\} \rangle_m$
   according to the previously mentioned conditions.

   At this point either the transition *draw* or *update store* is applicable such that:
   $S_{chr_{vis}}' \mapsto_{draw / updatestore} S_{chr_{vis}}'' : \langle [c\#i : j|A], H_1 \cup H_2 \cup S, Gr', B, T, H'\_ann \rangle_{m'}$
   In case Obj is a graphical object, the transition *draw* is applied such that: $Gr' = Gr \cup \{Obj\#m\} \wedge m' = m+1 \wedge H'\_ann = generate\_new\_ann\_history(Obj, m, r, id(H), H\_ann \cup \{\langle r_{ann}, id(H), \{\ \}\rangle\})$.

   In case, Obj is a graphical action, the transition *update store* is applied such that:
   $Gr' = update\_graphical\_store(Gr, Obj) \wedge m' = m \wedge H'\_ann = H\_ann \cup \{\langle r_{ann}, id(H), \{\ \}\rangle\}$
   Since the two transitions, could only change the graphical stores, annotation history and the next available identifier, the equivalence of the states is not affected.

   At this point ω_{vis} fires the transition *Apply* for the same CHR rule that triggered the same transition under ω_r earlier. The produced state $S_{chr_{vis}}'''$ has the format:
   $\langle C + H + A, H_1 \cup S, Gr', chr(H_1) = H_1' \wedge chr(H_2) = H_2' \wedge B, T \cup \{\langle r, id(H_1) + id(H_2) \rangle\}, H'\_ann \rangle_{m'}$. Similarly the same matching (local variable renaming x') has to be applied for the rule to fire.

   Consequently, according to Definition 11, the state $S_{chr_{vis}}'''$ is still equivalent to S_{chr}'

- **Applying the transition drop:**

  In the case where the top of the stack has an occurrenced active constraint $c\#i : j$ such that c has no occurrence j in the program, the transition drop is applied. Thus,

  $S_{chr} : \langle [c\#i : j|A], S, B, T \rangle_n \mapsto_{drop} S'_{chr} : \langle A, S, B, T \rangle_n$

  Since $S_{chr_{vis}}$ and S_{chr} are equivalent, the stack of both states have to be equivalent. Thus $S_{chr_{vis}} = \langle [c\#i : j|A], S, Gr, B, T, H\_ann \rangle_m$. For ω_{vis} one of two possibilities is applicable:

  1. No annotation rule is applicable. This could be either because c is not associated with any visual annotation rules or because all such rules have been already applied. In this case

     $S_{chr_{vis}} : \langle [c\#i : j|A], S, Gr, B, T, H\_ann \rangle_m \mapsto_{drop} S'_{CHR_{vis}} : \langle A, S, Gr, B, T, H\_ann \rangle_m$

     which is equivalent to S'_{chr}.

  2. The second possibility is the existence of an applicable annotation rule: transforming $S_{chr_{vis}}$ to $S'_{chr_{vis}} : \langle [Obj\#\langle r, id(H) \rangle, c\#i : j|A], S, Gr, B, T, H'\_ann \rangle_m$. At that point either *draw* or *update store* are to be applied transforming $S'_{chr_{vis}}$ to $S''_{chr_{vis}} : \langle [c\#i : j|A], S, Gr', B, T, H''\_ann \rangle_{m'}$. At that point, the transition drop is applicable converting $S''_{chr_{vis}}$ to $S'''_{chr_{vis}} : \langle A, S, Gr', B, T, H''\_ann \rangle_{m'}$. $S'''_{chr_{vis}}$ is equivalent to S'_{chr}.

- **Applying the default transition**

  If none of the previous cases is applicable, $S_{chr} : \langle [c\#i : j|A], S, B, T \rangle_n \mapsto_{default}$
  $S'_{chr} : \langle [c\#i : j + 1|A], S, B, T \rangle_n$.

  For the equivalent $S_{chr_{vis}}$, one of two possible cases could happen:

  1. **Apply annotation is not applicable:**

     In that case, the *Default* transition is directly applied transforming $S_{chr_{vis}}$ to $S'_{chr_{vis}}$ such that

     $\langle [c\#i : j|A], S, Gr, B, T, H\_ann \rangle_m \mapsto_{default} \langle [c\#i : j + 1|A], S, Gr, B, T, H\_ann \rangle_m$.

     The produced state $(S'_{chr_{vis}})$ is equivalent to S'_{chr} as well.

  2. **Apply annotation is applicable:**

     In this case an annotation rule for one of the existing constraints is applicable such that:

     $S_{chr_{vis}} \langle [c\#i : j|A], S, Gr, B, T, H\_ann \rangle_m \mapsto_{apply\_annotation}$
     $S'_{chr_{vis}} : \langle [Obj\#\langle r, id(H) \rangle, c\#i : j|A], S, Gr, B, T, H'\_ann \rangle_m$ according to the previously mentioned conditions.

     At this point, either the transition *draw* or the transition *update store* is applicable such that:

     $S'_{chr_{vis}} \mapsto_{draw} S''_{chr_{vis}} : \langle [c\#i : j|A], S, Gr', B, T, H''\_ann \rangle_{m'}$
     $S''_{chr_{vis}}$ is still equivalent to S_{chr}.

     At the point where the transition *apply_annotation* is no longer applicable, the only applicable transition is *Default* transforming $S''_{chr_{vis}}$ to $S'''_{chr_{vis}}$ such that $S'''_{chr_{vis}} = \langle [c\#i : j + 1|A], S, Gr', B, T, H''\_ann \rangle_{m'}$. According to Definition 11, $S'''_{chr_{vis}}$ is equivalent to S'_{chr}

  Thus in all cases an equivalent state is produced under ω_{vis}. ◄

  ◄

Improving Candidate Quality of Probabilistic Logic Models

Joana Côrte-Real[1]

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
jcr@dcc.fc.up.pt
 https://orcid.org/0000-0002-1085-3264

Anton Dries

KU Leuven, Department of Computer Science
Celestijnenlaan 200A bus 2402, 3001 Leuven, Belgium
anton.dries@cs.kuleuven.be
 https://orcid.org/0000-0003-2944-2067

Inês Dutra

CINTESIS and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021, 4169-007 Porto, Portugal
ines@dcc.fc.up.pt
 https://orcid.org/0000-0002-3578-7769

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
 https://orcid.org/0000-0003-4502-8835

Abstract

Many real-world phenomena exhibit both relational structure and uncertainty. Probabilistic Inductive Logic Programming (PILP) uses Inductive Logic Programming (ILP) extended with probabilistic facts to produce meaningful and interpretable models for real-world phenomena. This merge between First Order Logic (FOL) theories and uncertainty makes PILP a very adequate tool for knowledge representation and extraction. However, this flexibility is coupled with a problem (inherited from ILP) of exponential search space growth and so, often, only a subset of all possible models is explored due to limited resources. Furthermore, the probabilistic evaluation of FOL theories, coming from the underlying probabilistic logic language and its solver, is also computationally demanding. This work introduces a *prediction-based pruning strategy*, which can reduce the search space based on the probabilistic evaluation of models, and a *safe pruning criterion*, which guarantees that the optimal model is not pruned away, as well as two alternative more aggressive criteria that do not provide this guarantee. Experiments performed using three benchmarks from different areas show that prediction pruning is effective in (i) maintaining predictive accuracy for all criteria and experimental settings; (ii) reducing the execution time when using some of the more aggressive criteria, compared to using no pruning; and (iii) selecting better candidate models in limited resource settings, also when compared to using no pruning.

---

1 Introduction

The ability to take uncertainty into account when building a declarative model of a real-world phenomena can result in a closer representation of reality. The Probabilistic Logic Programming (PLP) paradigm addresses this issue by encoding knowledge as facts or rules, which are believed to be true to some degree or with a given frequency, instead of using crisp true or false statements. There are several Prolog-based probabilistic logic languages in the literature that can represent and manipulate uncertainty, such as SLP [15], ICL [17], Prism [20], BLP [12], CLP($\mathcal{BN}$) [19], MLN [18], ProbLog [13], among others. Please see the work by [8] for a recent survey of PLP.

Performing structure learning over PLP produces models which are understandable by humans whilst still taking uncertainty into account. Probabilistic Inductive Logic Programming (PILP) is a subset of Statistical Relational Learning (SRL) that uses a probabilistic First Order Logic (FOL) language to represent data and their induced models. PILP differs from traditional Inductive Logic Programming (ILP) in that facts and rules have success probabilities ranging between 0 and 1, as opposed to being either 0 or 1 (false or true, respectively). In this setting, there are no longer positive and negative examples, but only *target probabilities* for each example. The aim of a PILP model is to predict probability values which are as close as possible to the target probabilities of each example. PILP algorithms use (i) a set of Probabilistic Examples (PE), and (ii) logical information pertaining complex relations expressed as logic facts and rules, the Probabilistic Background Knowledge (or PBK), to find a FOL model that explains the PE. PILP focuses on structure learning – the logic rules compose a theory that models the structure of the PE w.r.t PBK – but parameter learning can also be incorporated by tuning the probabilistic output of the rules which are learned [7].

A number of PILP systems exist in the literature: ProbFOIL [9, 7], SLIPCOVER [1, 2], and SkILL [5, 4]. Additionally, there are other ILP-based structure learning methods such as CLP($\mathcal{BN}$) [19] and MLN [14]. One of the limitations of the available PILP systems is that they inherit the exponential search space from ILP, and must in addition evaluate the fitness of each candidate model by computing, for each example, the likelihood of that example given the model. This can be very time consuming, since the evaluation process must consider all possible worlds where the theory in the model may be true. For a small number of facts and rules in the PBK this is not a problem, but computation grows exponentially as the size of the PBK is increased [10].

To address this problem, this work introduces *prediction pruning*. Prediction pruning prunes the PILP search space based on previously evaluated theories by taking into account the logical operation (conjunction or disjunction) that will be performed next. Prediction pruning can be effective in reducing the execution time, compared to using no pruning. Additionally, the quality of the explored candidate models is improved when prediction pruning is used in conjunction with beam search. Unlike other pruning approaches, such as

beam search as used in [7, 2, 5], or estimation pruning as used in [4], prediction pruning can guarantee *safety* such that when the safe criterion is used the optimal model is never pruned away. This work thus also investigates three possible criteria for prediction pruning: a *safe criterion* and two other more aggressive pruning criteria. Experiments using three benchmarks and two PILP systems show that all three criteria are effective in maintaining (or increasing) predictive accuracy for all experimental settings. Furthermore, the more aggressive criteria reduce execution time compared to using no pruning, without loss of predictive accuracy. Finally, in limited resource settings, better candidate models are generated when compared to using no pruning.

This paper is organized as follows. Section 2 briefly introduces the main concepts of PILP. Next, Section 3 presents the proposed pruning strategy and the proposed pruning criteria. Section 4 evaluates the proposed approach and discusses the results. Finally, conclusions and perspectives of future work are put forward in Section 5.

2 Background

Traditional ILP generates sets of FOL rules (or theories) trying to describe a problem, given as a *target predicate*, in terms of the clauses contained in a given background knowledge. The theory's fitness to describe the problem is assessed according to a *loss function*. The aim of ILP is to find a theory that explains all given positive examples and does not explain any of the given negative examples, but in practice it is common to relax these criteria and allow for some noise (misclassified examples). It is also common to define a declarative *language bias* using mode declarations in order to specify which rules are valid within the search space.

PILP extends the ILP setting by introducing a Probabilistic Background Knowledge (or PBK), where FOL data descriptions can be annotated with a probability value ranging from 0 to 1, and by introducing a set of Probabilistic Examples (PE), no longer positive or negative, also with a value ranging between 0 and 1. Facts and rules in the PBK and PE can represent either statistical information or the degree of belief in a statement, using type I or type II probability structures, respectively [11]. Non-annotated data is assumed to have a probabilistic value of 1. Because PILP theories are still generated based on the logical information of the data, the ILP language bias translates directly to PILP. The process of generating theories also mimics ILP, since they are based on the logical clauses in the PBK. Good theories are the ones which most closely predict the values of the PE or rather that minimize the error between predictions and the PE values.

In this work, probabilities are annotated according to ProbLog's syntax, using *possible world semantics* [8]. In ProbLog, each fact $p_j :: c_j$ in the PBK represents an independent binary random variable, meaning that it can either be true with probability p_j or false with probability $1 - p_j$. This means that each probabilistic fact introduces a probabilistic choice in the model. Each set of possible choices over all facts of the PBK represents a possible world ω_i, where ω_i^+ is the set of facts that are true in that particular world, and $\omega_i^- = \omega_i \setminus \omega_i^+$ is the set of facts that are false. Since these facts have a probabilistic value, a ProbLog program defining a probabilistic distribution over the possible worlds can be formalized as shown in Eq. 1.

$$P(\omega_i) = \prod_{c_j \in \omega_i^+} p_j \prod_{c_j \in \omega_i^-} (1 - p_j) \tag{1}$$

A ProbLog *query* q is said to be true in all worlds w^q where $w^q \models q$, and false in all other worlds. As such, the *success probability* of a query is given by the sum of the probabilities of

all worlds where it is found to be true, as denoted in Eq. 2.

$$P(q) = \sum_{\omega_i \models q} P(w_i) \tag{2}$$

Even though the prediction (success probability) of a rule changes according to the literals contained in its body, the probabilistic model generated from the PBK is not altered throughout the execution of the program. The search for the best model in PILP thus consists of finding the theory whose success probabilities (for all examples) have the best fitness w.r.t. the PE values (according to some loss function), given a PBK. This allows for defining standard scoring metrics such as *probabilistic accuracy (or PAcc)*, as introduced by De Raedt *et al.* in [9]. PAcc can also be represented in terms of the mean absolute error (MAE) between predictions and example values as used by Chen *et al.* in [3]. These two formulations are equivalent.

3   Prediction Pruning

The PILP search space can be split in two separate dimensions w.r.t. the operation that is being used to traverse it, i.e., there is a dimension for rules (or theories of length one), which uses the AND operation to generate new rules, and a dimension for theories (of length greater than one), which in turn uses the OR operation to generate new theories. Fully exploring the PILP search space is equivalent to evaluating each theory in the theory lattice in order to determine the best theory according to a given metric.

The theories used to explain examples in PILP are built from the literals that are present in the program's PBK. The rule (AND) search space is composed by all rules whose body contains one or more of those literals. Rules can be combined using logical conjunction to form longer, more *specific* rules. The theory (OR) search space can be defined in a similar way. Theories are formed by combining a set of distinct rules using logical disjunction. In the same way that literals are the building blocks of rules, rules are the building blocks of theories. Adding a rule to a theory makes it more *general*.

The procedure to explore the PILP search space can thus be done in two steps: (i) explore the AND search space, and (ii) explore the OR search space. An exhaustive search strategy would be very time-consuming leading to a scenario where good theories might never have a chance to be evaluated due to the complexity of the probabilistic evaluation. When resources are limited, it is thus preferable to focus on good candidate theories and avoid candidate theories which are below a threshold of quality to transition to the next iteration. Prediction pruning is thus applied over previously evaluated theories which are *determined* to be useless for further combination. Prediction pruning excludes theories whose predictions suggest that the theory is already too specific, for the AND operation, or too general, for the OR operation. Algorithm 1 presents this procedure.

Algorithm 1 starts by exploring the AND search space in a direction of increasing specificity. It starts out by generating rules containing only one literal (line 3) and then uses these rules to generate combinations for the next iteration (lines 5–8). In order to prevent rules which are determined to be too specific from being considered for combination in the next iteration, prediction pruning is applied according to a given $CriterionAND$ (procedure $AND\_pred\_pruning$ on line 7). Rules that are pruned by this criterion are still included in R_{all} but they are not further specialized in R_{new} (line 8). The combination process is repeated until it yields no new rules. The set of initial theories T_1 is then populated with all rules in R_{all} (line 9). Similarly to the AND search space, T_1 is used to generate new

---

Algorithm 1 $PILP\_algorithm(PBK, PE, CriterionAND, CriterionOR)$.

 1: $T_{all} = \emptyset$
 2: $R_{all} = \emptyset$
 3: $R_1 = generate\_rules\_one\_literal(PBK, PE)$
 4: $R_{new} = R_1$
 5: **while** $R_{new} \neq \emptyset$ **do**
 6:     $R_{all} = R_{all} \cup R_{new}$
 7:     $R_{pru} = \textbf{AND\_pred\_pruning}(R_{new}, CriterionAND)$
 8:     $R_{new} = \{r_1 \wedge r_{pru} \mid (r_1, r_{pru}) \in R_1 \times R_{pru}\}$
 9: $T_1 = R_{all}$
10: $T_{new} = T_1$
11: **while** $T_{new} \neq \emptyset$ **do**
12:     $T_{all} = T_{all} \cup T_{new}$
13:     $T_{pru} = \textbf{OR\_pred\_pruning}(T_{new}, CriterionOR)$
14:     $T_{new} = \{t_1 \vee t_{pru} \mid (t_1, t_{pru}) \in T_1 \times T_{pru}\}$
15: **return** T_{all}

---

theories T_{new} through combination using logical disjunction (lines 11-14). This process is analogous to the exploration of the AND search space, except that the pruning criterion $CriterionOR$, used in procedure $OR\_pred\_pruning$ (line 13), is based on generality as opposed to specificity.

The decision on whether a candidate theory should be further explored is made based on the theory's individual prediction values for each example. Depending on which search space is being explored, the criterion to exclude theories will differ. When two rules r^a and r^b are combined using logical conjunction, a more *specific* rule $r^{a,b} = r^a \wedge r^b$ will result. This is due to the fact that more literals in the body of the rule must succeed simultaneously so that the rule can be verified.

In the probabilistic setting, a rule r is composed of a logical part $l(r)$ and a prediction value $p(r)$ ranging from 0 to 1. The prediction value of rule r for a given example i, $p_i(r)$ is equal to the sum of the probabilities $P(\omega_n)$ of each world ω_n in the program in which $\omega_n \models l_i(r)$ for that same example i. This means that for the more specific rule $r^{a,b}$ to be true, both r^a and r^b must be true simultaneously, i.e. only the worlds where both r^a and r^b are true can be considered. This is equivalent to the intersection of the set of worlds which entail $l(r^a)$ and $l(r^b)$, taking also into account the variable groundings for r^a and r^b. Therefore, the prediction value of a specific rule for an example i can be defined in terms of the prediction values of less specific rules which compose it.

$$p_i(r^{a,b}) \quad = \sum_{\omega_n \models l_i(r^{a,b})} P(\omega_n) \quad = \sum_{\substack{\omega_n \models l_i(r^a) \cap \\ \omega_n \models l_i(r^b)}} P(\omega_n) \tag{3}$$

From Eq. 3, it follows that, for an example i, the prediction value of a more specific rule $p_i(r^{a,b})$ will always be less than or equal to the prediction value of $p_i(r^a)$ and $p_i(r^b)$. Therefore, the prediction value of rule $p_i(r)$ will be monotonically decreasing with the application of the AND operation, since in each iteration the rules become more specific.

Having established this ordering allows prediction pruning to be applied over previously evaluated rules to determine whether they are useless for further combination, given some criterion. For a given example i, if the prediction value of a rule $p_i(r)$ is less than the example value e_i, then continuing to apply the AND operation can only result in distancing $p_i(r)$

▇ **Table 1** Expressions for the soft, hard and safe criteria.

Criterion	Search Space	
	AND	*OR*
Soft	$\sum_i \big(p_i(t) - e_i\big) < 0$	$\sum_i \big(p_i(t) - e_i\big) > 0$
Hard	$\exists i : p_i(t) < e_i$	$\exists i : p_i(t) > e_i$
Safe	$\forall i : p_i(t) < e_i$	$\forall i : p_i(t) > e_i$

from e_i further, since $p_i(r)$ can only decrease from the application of the AND operation. As such, prediction pruning excludes rules whose prediction values for all examples suggest that the theory is already too specific when compared to the example values. A similar argument can be made for the OR operation and the *generality* of theories.

To determine whether theories will be pruned away or not, several criteria are possible. This work proposes three criteria for deciding if a theory is too specify/general: a *soft criterion*, a *hard criterion* and a *safe criterion*. These three criteria take into account the predictions of a theory $p_i(t)$ for the given examples, as well as the example values e_i themselves. Table 1 presents the expressions for the pruning criteria when applied to the AND and OR search spaces. The soft pruning criterion takes into account the theory's predictions for every example, and only prunes the theory away if it is *overall* more specific (for the AND operation) or more general (for the OR operation) than the values of the examples. The hard pruning criterion prunes a theory away if, in *any* example, the theory made a prediction that was more specific (for the AND operation) or more general (for the OR operation) than the annotated value for that example. The soft criterion differs from the hard criterion in that it takes into account the *aggregate* value of all examples, whilst the hard pruning criterion can discard theories based on one example value only. On the other hand, the safe pruning criterion excludes theories only when *all* of their predictions are found to be too specific (for the AND operation) or too general (for the OR operation), and no prediction can be improved by continuing with the search in that search space. Therefore, it is *safe* to prune away these candidate theories, since they can never perform better with more specialisation/generalisation, respectively.

Figure 1 illustrates these concepts for a PILP setting with three examples and three theories. For each example i, the example value e_i (squares in black) and three predictions of theories $p_i(t^1)$, $p_i(t^2)$ and $p_i(t^3)$ are plotted. The ground truth model would predict exactly e_i for every example. If a prediction value $p_i(t)$ is plotted *below* the example value e_i, then that theory is too specific for that example. Conversely, if $p_i(t)$ is plotted *above* e_i, the theory is more general for that example.

In Fig. 1, for the AND operation, the safe pruning criterion would prune away theory t^1 because, for every example, its prediction values are lower than the example values. The soft pruning criterion would prune away theories t^1 and t^2 because their prediction values are overall lower than the example values. Finally, the hard pruning criterion would prune away all theories. For example, theory t^3 is pruned away because its prediction for $e = 1$ is lower than the example value. An analogous reasoning can be made for the OR operation and higher prediction values. In summary, the theories pruned away by the safe criterion are a subset of the theories pruned away by the soft criterion, and similarly the theories pruned away by the soft criterion are a subset of those pruned away by the hard criterion.
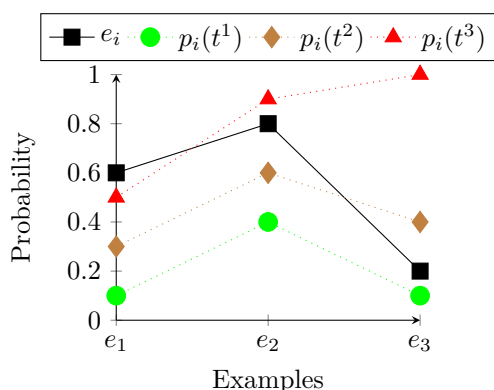
Figure 1 PILP setting with three examples and three theories. For each example, the example values (squares in black) and three predictions of theories (green circles for $p_i(t^1)$, brown diamonds for $p_i(t^2)$ and red triangles for $p_i(t^3)$) are plotted.

4 Experiments

The experiments presented in this section are aimed at answering the following three questions: (i) how much does prediction pruning reduce the exhaustive PILP search space? (ii) can prediction pruning maintain predictive quality of models? (iii) how does prediction pruning impact the quality of the candidate models explored in a limited resource setting?

Prediction pruning was implemented and evaluated in two state-of-the-art PILP systems: SkILL [5] and ProbFOIL+ [7]. SkILL runs on top of the Yap Prolog system [6], uses TopLog [16] as the basis for rule generation and the ProbLog Yap library as its probabilistic inference engine. The experiments using the SkILL system were run on a machine containing 4 AMD Opteron 6300 processors with 16 cores each and a total of 250GB of RAM. ProbFOIL+ is based on Python and it uses the Yap Prolog system for logical inference of theories. In these experiments, ProbFOIL+ uses only the examples provided in the training data (without generation of additional negative examples as used in the original paper) and it uses negated literals in the theories. The experiments using ProbFOIL+ were run on a machine containing an Intel Core i7 processor with 4 cores and a total of 16GB of RAM. All experiments use five-fold stratified cross validation and results presented are the average values for all folds. The evaluation was performed using three different datasets: **metabolism**, **athletes** and **breast cancer**.

The metabolism dataset consists of an adaptation of the dataset originally from the 2001 KDD Cup Challenge[2]. It is composed of 230 examples (half positive and half negative) and approximately 7000 BK facts. To obtain probabilistic facts for the PBK, the predicate `interaction(gene1,gene2,type,strength)` was adapted from the original metabolism dataset. The fourth argument of this predicate indicates the strength of the interaction between a pair of genes. This fact was converted to the probabilistic fact `p_strength::interaction(gene1,gene2,type)`, where `p_strength` was calculated from strength interactions as follows:

$$p\_strength = \frac{strength - min_{strength}}{max_{strength} - min_{strength}}$$

---

[2] `http://www.cs.wisc.edu/~dpage/kddcup2001`

This resulted in about 3200 probabilistic facts in the PBK. 5 folds were generated from this dataset, and each one of them is composed of 46 test examples selected randomly from the main dataset (but keeping the same positive/negative ratio) and, for each fold, the 184 remaining examples are used for training.

The athletes dataset consists of a subset of facts regarding athletes and the sports they play collected by the never-ending language learner NELL[3]. NELL iteratively reads the web, gathering knowledge, and for each fact that it comes across it assigns a weight that can be used as a probability. As NELL iterates, the weights of the facts in its database are updated, and the dataset used for this experiment contains the facts and weights from iteration 850. The dataset is composed of 720 probabilistic examples of athletes that play for a team, and 4294 probabilistic facts in the PBK pertaining to the origin of the player, his/her gender, the city where a team plays, and so on. 5 folds were generated from this dataset, and each one of them is composed of 144 test examples selected randomly from the main dataset and the 576 remaining examples are used for training. Because in this case examples do not clearly belong to one of two classes, the test examples were randomly selected from the dataset without taking their expected value into account.

The breast cancer dataset contains data from 130 biopsies dating from January 2006 to December 2011, which were prospectively given a non-definitive diagnosis at radiologic-histologic correlation conferences. Twenty-one cases were determined to be malignant after surgery, and the remaining 109 proved to be benign. The probabilities assigned to the examples represent the chance of malignancy for each patient. A high probability indicates the team of physicians thinks the case is most likely malignant, and conversely a low probability indicates the case is most likely benign. Five folds were generated from this dataset, and each one of them is composed of 26 test examples selected randomly from the main dataset (but keeping the same positive/negative ratio) and the 104 remaining examples are used for training.

4.1 Probabilistic Accuracy and Search Space Reduction

Baseline. Because exploring the search space exhaustively is computationally taxing, the quality of candidate theories was assessed in a limited resource setting. Resources can be limited in two ways: either a timeout is imposed or a maximum number of evaluations is defined, which corresponds to using beam search (or the fitness pruning setting in the case of the SkILL system). To this effect, the impact of prediction pruning was assessed by comparing the AND and OR search spaces that are evaluated without pruning with those which are evaluated in a pruning setting, given the same limitation of resources. In these experiments, the default fitness pruning / beam search settings of both systems are used (that is, for SkILL, primary and secondary population sizes of 25/20 for both AND and OR space, and for ProbFOIL+, a beam size of 5 for the AND space and greedy search in the OR space, as ProbFOIL+ only supports greedy search there).

Prediction Pruning. The use of prediction pruning enables PILP systems to focus their (limited) resources on more promising candidates, when traversing the search space. Table 2 presents the results of applying prediction pruning in the AND search space in combination with fitness pruning / beam search. It shows the execution time (in seconds), the number of theories evaluated probabilistically and the probabilistic accuracy of the best theory found for different pruning criteria (Safe, Soft and Hard), using the SkILL and ProbFOIL+ systems. Please note that execution times between systems are not comparable.

---

[3] http://rtw.ml.cmu.edu

■ **Table 2** Execution time in seconds, number of probabilistic evaluations performed and probabilistic accuracy for datasets metabolism, athletes and breast cancer using the SkILL and ProbFOIL+ systems with prediction pruning for the AND search space. Standard deviation is presented in brackets. Execution times between systems are not comparable.
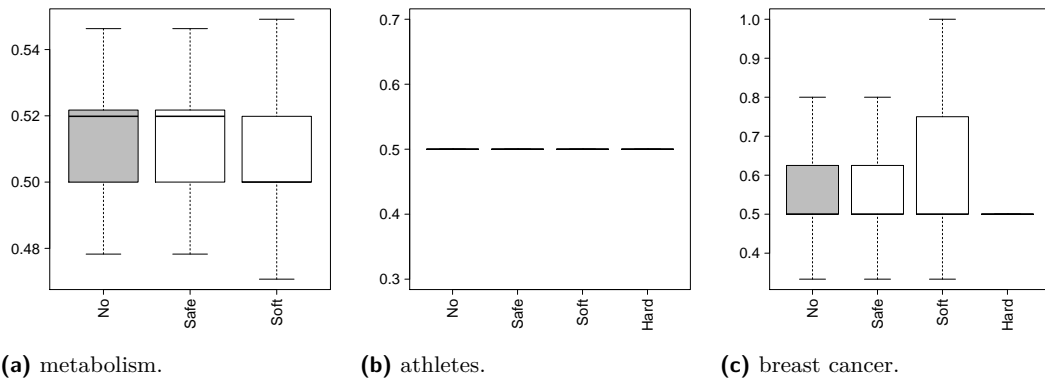
(a) SkILL.

	Baseline	Safe	Soft	Hard
	Execution Time (s)			
metabolism	3353 (204)	2286 (185)	3216 (472)	1791 (37)
athletes	4610 (79)	4230 (582)	2322 (164)	2358 (73)
breast cancer	1449 (63)	616 (50)	636 (26)	353 (42)
	No. Evaluations			
metabolism	2151 (44)	2150 (44)	3234 (90)	2103 (37)
athletes	1852 (25)	1896 (18)	994 (3)	994 (3)
breast cancer	1235 (68)	1234 (67)	1306 (43)	941 (70)
	Probabilistic Accuracy			
metabolism	0.67 (0.05)	0.67 (0.05)	0.67 (0.05)	0.67 (0.05)
athletes	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)	0.95 (0.01)
breast cancer	0.86 (0.04)	0.86 (0.04)	0.84 (0.08)	0.86 (0.03)

(b) ProbFOIL+.

	Baseline	Safe	Soft	Hard
	Execution Time (s)			
metabolism	2008 (2016)	1999 (2019)	752 (215)	464 (71)
athletes	57 (5)	57 (5)	55 (4)	14 (0)
breast cancer	3890 (339)	3828 (302)	8093 (2101)	725 (38)
	No. Evaluations			
metabolism	3734 (2328)	4549 (3734)	4518 (1493)	2452 (492)
athletes	201 (43)	201 (43)	171 (21)	0 (0)
breast cancer	24290 (851)	24267 (828)	26495 (3542)	3532 (231)
	Probabilistic Accuracy			
metabolism	0.51 (0.04)	0.51 (0.03)	0.63 (0.11)	0.58 (0.07)
athletes	0.80 (0.01)	0.80 (0.01)	0.80 (0.01)	0.80 (0.01)
breast cancer	0.85 (0.01)	0.85 (0.01)	0.85 (0.03)	0.87 (0.01)

Probabilistic Accuracy.   Prediction pruning results in Table 2 show that applying the Soft or Hard strategies leads to clear improvements in probabilistic accuracy for ProbFOIL+ and does not lead to degradation in SkILL. The effect of prediction pruning is more evident for ProbFOIL+ because it selects fewer candidates in each iteration, when compared to the SkILL's primary and secondary populations. It is therefore more important that bad candidates are pruned such that the limited beam is filled with better candidates. The prediction pruning strategy is thus particularly useful when traversing the search space with a narrow beam, so that the candidates selected to populate it are of greater predictive value when compared to using no prediction pruning. Safe pruning has no effect on these datasets because its pruning power is too limited.

(a) metabolism.   **(b)** athletes.   **(c)** breast cancer.

Figure 2 Distribution of theories' AUCs for the AND search space for datasets metabolism, athletes and breast cancer using different prediction pruning settings in the SkILL system.

Search Space Reduction. Table 2 also shows that applying prediction pruning does not necessarily reduce the search space. It can actually increase the number of rules evaluated during the execution, and even the execution time in some cases. This happens because prediction pruning provides a type of lookahead, that is, it makes an assessment of the predictive power of a rule in future iterations. When no prediction pruning is used, the algorithms have a strong bias toward rules that show good performance early on and the best rule (in the limited search space) is found after a few iterations. Prediction pruning counteracts this bias, and also allows candidates that only reach their full predictive accuracy after a higher number of iterations to be explored. However, since the algorithm may take more iterations, this can lead to more evaluations and longer rules that are harder to evaluate.

4.2 Search Space Quality

Each theory in the PILP search space can be thought of as a predictor, and for this reason its predictive quality can be assessed using the area under the ROC curve (AUC). Since prediction pruning removes theories from the search space based upon the operation that is being performed (AND or OR), the distribution of the remaining candidate theories can change (there may be cases where no candidate theories are left for the next iteration). As such, comparing the two search spaces using the AUCs of the theories they contain shows how the predictive quality of their candidates compares.

For the SkILL experiments, the AUC of all rules containing more than one literal (AND search space) and all theories (OR search space) was calculated. The AUC of rules composed of only one literal was not considered because prediction pruning has no effect on these rules, which must always be evaluated. Analysing the distribution of the AUC values is relevant because if the upper quartiles of the distribution are improved, this shows that there are better candidate members selected to be explored given limited resources. Lower quartiles will naturally be discarded by the PILP algorithm's metric to select the best final theory. The distribution of these values for each setting and search space are presented in Figures 2 and 3 for the AND and OR search spaces, respectively. Each box depicts percentiles 0 and 100 (the lower and upper whiskers, respectively), percentiles 25 and 75 (lower and upper box boundaries, respectively), and the percentile 50 (median) using a bold line.

In Figs. 2–3, the higher the AUC value (y-axis), the greater the predictive power of the theory. Each boxplot corresponds to a setting. In Fig. 2 (AND search space only), the first boxplot corresponds to the rules generated using no prediction pruning, the second

(a) metabolism.　　　　**(b)** athletes.　　　　**(c)** breast cancer.
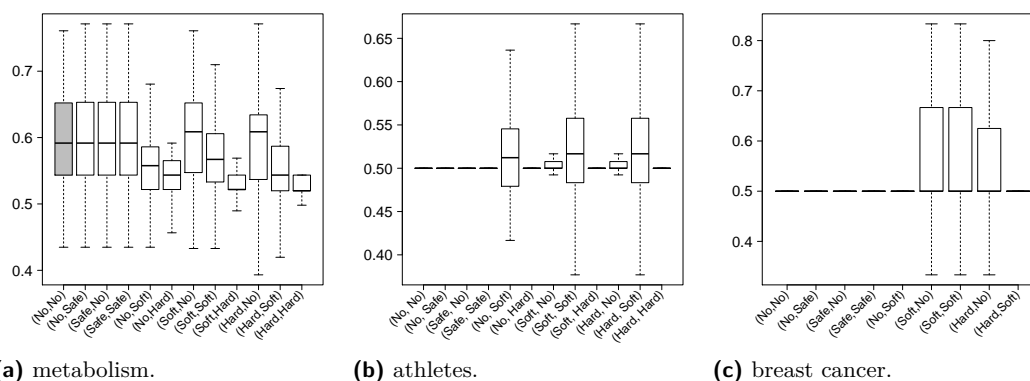
■ **Figure 3** Distribution of theories' AUCs for the OR search space, for datasets metabolism, athletes and breast cancer using different prediction pruning settings in the SkILL system.

■ **Table 3** Number of significant differences (left) for the number of tested folds (right) in the AND and OR AUC distributions for datasets metabolism, athletes and breast cancer using different prediction pruning settings in the SkILL system.

Setting (AND,OR)	metabolism AND	metabolism OR	athletes AND	athletes OR	breast cancer AND	breast cancer OR
(No, No)	0/4	0/5	0/5	0/5	0/5	0/5
(No, Safe)		0/5		2/5		0/5
(Safe, No)	0/4	0/5	0/5	2/5	0/5	0/5
(Safe, Safe)		0/5		2/5		0/5
(No, Soft)	0/4	4/4	0/5	4/5	0/5	0/5
(No, Hard)		4/4		4/5		–
(Soft, No)		2/5		3/5		2/5
(Soft, Soft)	4/4	5/5	0/5	3/5	3/5	3/5
(Soft, Hard)		5/5		4/5		–
(Hard, No)		5/5		3/5		5/5
(Hard, Soft)	–	3/4	0/5	4/5	1/4	4/4
(Hard, Hard)		1/1		4/5		–

boxplot to the rules generated using safe prediction pruning, and so on. In Fig. 3 (AND and OR search spaces), the pruning settings are reported as a tuple where the first value is the AND prediction pruning option and the second is the OR prediction pruning option. For example, the tuple (Soft,Hard) stands for soft AND prediction pruning and hard OR prediction pruning, whilst the tuple (No,Safe) stands for no AND pruning and safe OR prediction pruning.

For the AUC distributions, statistical significance is also calculated (using non-paired two-tailed t-test) by comparing the distribution of AUCs fold to fold (e.g. fold 1 using soft OR prediction pruning against fold 1 without pruning). Table 3 reports the number of folds where the results were statistically significant for both the AND and the OR search spaces. In some cases, some folds do not produce an AND or OR search space because all theories are pruned away, and this is the cause for not always reporting five folds in comparison.

In Fig. 3, it is visible that prediction pruning can improve the general quality of the evaluated theories, particularly in the case of the athletes and breast cancer datasets. In the breast cancer dataset, the two upper quartiles of the AUC distribution are clearly improved

in three settings. This trend is also clear in the athletes dataset, where again prediction pruning significantly increases the predictive quality of the evaluated theories in three cases (and slightly in two other settings). On the metabolism dataset, the improvements due to prediction pruning are not as evident, but it is noteworthy that there is in fact a slight increase in the maximum AUC value for the case of hard AND pruning and no OR pruning, as well as in all safe pruning settings. The boxplots with range zero indicate that in those settings the candidates that populate the beam do not have any predictive power in the test set. However, this does not imply a loss in predictive accuracy of the optimal model since rules of only one literal are not included in these boxplots because they are not affected by prediction pruning.

Regarding the quality of the AND search space (Fig. 2), it is only significantly improved in the breast cancer dataset, using soft prediction pruning. However, the candidate rules that are selected for the AND search space impact the OR search space, since candidate theories will be selected from the rules that were previously explored in the AND search space. As such, even though the AND search space only shows direct impact from using prediction pruning in the breast cancer dataset, it indirectly impacts the candidate theories available for the OR search space in all datasets. This is particularly relevant for the athletes dataset, where the quality of the OR search space is affected by soft and hard AND pruning. For instance, setting (Soft, Soft) performs significantly better when compared to setting (No, Soft), and setting (Hard, No)'s 50 and 100 percentiles are higher than its counterpart setting (No, No). This effect is also visible in the breast cancer dataset, where the settings using soft or hard AND prediction pruning present the greatest improvement. In most cases where the quality of the OR search space increased, AND prediction pruning had previously been applied to the AND search space.

Table 3 shows that the safe pruning criterion causes no significant difference in candidate theory predictive quality, both for the AND and the OR operation (lines 2–4). This is due to the fact that the safe pruning criterion is the least aggressive criterion and therefore the proportion of candidates that are pruned in this setting is limited. On the other hand, both soft and hard pruning criteria cause a significant difference in the AUC distribution of candidates, in particular for the OR operation, where most folds present a significant difference (lines 5–12 and columns 2, 4 and 6 in Table 3). However, for the AND operation, aggressive criteria do not cause such a significant difference in the distribution, in particular for the athletes dataset. This happens because the predictive power of rules in this dataset is similar among candidates, and so even though different rules can be selected, this is not reflected in the distribution of AUC values. In cases where aggressive pruning causes the search space to be empty for all folds, there is no boxplot in Figs. 2–3, and no value reported in Table 3.

Prediction pruning thus impacts the quality of the search space positively, allowing for limited resources to be targeted towards better candidate theories. Furthermore, even though in some cases the quality of the search space decreases (for instance the quality of the AND search space using hard prediction pruning in the breast cancer dataset), the accuracy of the best final theory found never decreases significantly, thus showing that prediction pruning can be applied to better select candidate theories without risk of impacting the final test accuracy.

5 Conclusion

This work proposes a novel prediction pruning methodology whose aim is to improve the quality of the explored candidate models in a PILP search space. Unlike previously proposed pruning approaches, such as beam search and estimation pruning, prediction pruning focuses on improving the quality of the search space. In doing so, it can direct the search towards more promising candidates which can lead to a reduction in execution time or an increase in predictive accuracy.

This work also introduces three pruning criteria, with increasing pruning power, which can be used to decide which models should be pruned away during the prediction pruning stage in the PILP algorithm. All pruning criteria are based on the probabilistic information of candidate models and depend on which operation is being performed in the PILP algorithm: logic conjunction (AND search space) or disjunction (OR search space). The safe pruning criterion guarantees the safeness of the prediction pruning strategy, meaning that the optimal model is never pruned away during the search, but experiments show that this criterion is not very successful in pruning the search space significantly. The soft and hard pruning criteria, however, do exhibit pruning power while not suffering from a reduction in predictive performance.

Results also show that prediction pruning maintains the predictive quality of the generated models. Prediction pruning impacts the distribution of the predictive quality of theories and the use of prediction pruning can shift the maximum value and upper quartile of the distribution upwards, thus indicating improved candidate theory quality. Deeper analysis of the AUC of theories shows that all three criteria improve the quality of the OR search space. AND prediction pruning, while not presenting a significant difference in all datasets, can influence the OR search space quality, and so using prediction pruning for both operations can increase the quality of the candidate theories while not sacrificing the final predictive accuracy.

An interesting direction for future work is to study how to automatically adjust the pruning criterion based on data characteristics of the dataset. Further work also includes developing a search space traversal strategy combining several pruning strategies and, in particular, study how prediction pruning interacts with beam search and estimation pruning.

───── **References** ─────

1   E. Bellodi and F. Riguzzi. Learning the structure of probabilistic logic programs. In *Inductive Logic Programming*, pages 61–75. Springer, 2012.

2   E. Bellodi and F. Riguzzi. Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming*, 15(02):169–212, 2015.

3   J. Chen, S. Muggleton, and J. Santos. Learning Probabilistic Logic Models from Probabilistic Examples. *Machine Learning*, 73(1):55–85, October 2008. `doi:10.1007/s10994-008-5076-4`.

4   J. Côrte-Real, I. Dutra, and R. Rocha. Estimation-Based Search Space Traversal in PILP Environments. In A. Russo and J. Cussens, editors, *Proceedings of the 26th International Conference on Inductive Logic Programming (ILP 2016)*, LNAI, pages –, London, UK, September 2016. Springer. Published in 2017.

5   J. Côrte-Real, T. Mantadelis, I. Dutra, R. Rocha, and E. Burnside. SkILL - a Stochastic Inductive Logic Learner. In *International Conference on Machine Learning and Applications*, pages –, Miami, Florida, USA, December 2015.

6   V. Santos Costa, R. Rocha, and L. Damas. The YAP Prolog System. *Journal of Theory and Practice of Logic Programming*, 12(1 & 2):5–34, 2012.

7 L. De Raedt, A. Dries, I. Thon, G. Van den Broeck, and M. Verbeke. Inducing Probabilistic Relational Rules from Probabilistic Examples. In *International Joint Conference on Artificial Intelligence*, pages 1835–1843. AAAI Press, 2015.

8 L. De Raedt and A. Kimmig. Probabilistic (logic) programming concepts. *Machine Learning*, 100(1):5–47, 2015. `doi:10.1007/s10994-015-5494-z`.

9 L. De Raedt and I. Thon. Probabilistic Rule Learning. In *Inductive Logic Programming*, pages 47–58. Springer, 2011.

10 Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and Learning in Probabilistic Logic Programs using Weighted Boolean Formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

11 J. Halpern. An Analysis of First-Order Logics of Probability. *Artificial intelligence*, 46(3):311–350, 1990.

12 K. Kersting, L. De Raedt, and S. Kramer. Interpreting Bayesian Logic Programs. In *AAAI Workshop on Learning Statistical Models from Relational Data*, pages 29–35, 2000.

13 A. Kimmig, B. Demoen, L. De Raedt, V. Santos Costa, and R. Rocha. On the Implementation of the Probabilistic Logic Programming Language ProbLog. *Theory and Practice of Logic Programming*, 11(2 & 3):235–262, 2011.

14 S. Kok and P. Domingos. Learning the Structure of Markov Logic Networks. In *International Conference on Machine learning*, pages 441–448. ACM, 2005.

15 S. Muggleton. Stochastic Logic Programs. *Advances in inductive logic programming*, 32:254–264, 1996.

16 S. Muggleton, J. Santos, C. Almeida, and A. Tamaddoni-Nezhad. TopLog: ILP Using a Logic Program Declarative Bias. In *International Conference on Logic Programming*, pages 687–692. Springer, 2008.

17 D. Poole. The independent choice logic for modelling multiple agents under uncertainty. *Artificial intelligence*, 94(1):7–56, 1997.

18 M. Richardson and P. Domingos. Markov Logic Networks. *Machine learning*, 62(1-2):107–136, 2006.

19 V. Santos Costa, D. Page, M. Qazi, and J. Cussens. CLP(BN): Constraint Logic Programming for Probabilistic Knowledge. In *Conference on Uncertainty in Artificial Intelligence*, pages 517–524, 2002.

20 T. Sato and Y. Kameya. PRISM: A language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, volume 97, pages 1330–1339. Morgan Kaufmann, 1997. URL: `http://ijcai.org/Proceedings/97-2/Papers/078.pdf`.

Towards Incremental and Modular Context-Sensitive Analysis

Isabel Garcia-Contreras

IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain and
Universidad Politécnica de Madrid (UPM)
isabel.garcia@imdea.org
🆔 https://orcid.org/0000-0001-6098-3895

José F. Morales

IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain
josef.morales@imdea.org
🆔 https://orcid.org/0000-0001-9782-8135

Manuel V. Hermenegildo

IMDEA Software Institute, Pozuelo de Alarcón, Madrid, Spain and
Universidad Politécnica de Madrid (UPM)
manuel.hermenegildo@imdea.org
🆔 https://orcid.org/0000-0002-7583-323X

── Abstract ──

This is an extended abstract of [1].

Static program analysis (generally based on computing fixpoints using the technique of abstract interpretation) is widely used for automatically inferring program properties such as correctness, robustness, safety, cost, etc. Performing such analysis interactively during software development allows early detection and reporting of bugs, such as, e.g., assertion violations, back to the programmer. This can be done as the program is being edited by (re-)running the analysis in the background each time a set of changes is made, e.g., when a file is saved, or a commit made in the version control system. However, real-life programs are large, and, typically, have a complex structure combining a good number of modules with other modules in system libraries. Global analysis of such large code bases can be very expensive, and more so if context-sensitivity is supported for precision. This renders triggering a complete reanalysis for each set of changes too costly. A key observation, however, is that in practice each development or transformation iteration is normally formed by relatively small modifications, which in turn are isolated inside a small number of modules. This property can be taken advantage of in order to reduce the cost of re-analysis by reusing

as much information as possible from previous analyses. Such cost reductions have been achieved to date at two different levels, using relatively different techniques:

- *Modular* context-sensitive analyses obtain global information on the whole program by performing local analyses one module at a time. They are typically aimed at reducing memory consumption (working set size) but can also localize the (re)computation of the analysis to the modules affected by changes, achieving some coarse-grained incrementality.
- Context-sensitive (non-modular) *incremental* analyses identify, invalidate, and recompute only those parts of the analysis results that are affected by fine-grain program changes. These analyses have been shown to achieve very high levels of incrementality, at fine levels of granularity (e.g., program line level).

The problem that we address is that while, as mentioned before, large programs are typically highly modular, the context-sensitive, fine-grained incremental analysis techniques presented to date are not easily applicable to the modular setting: The flow of analysis information through the module interfaces requires iterations, since the analysis of a module depends on the analysis of other modules in complex ways, through several paths to different versions of the procedures.

In order to bridge the gap we propose a framework that analyzes separately the modules of a modular program, using context-sensitive fixpoint analysis while achieving both inter-modular (coarse-grain) and intra-modular (fine-grain) incrementality. The proposed analysis algorithm assumes a setting in which we analyze successive "snapshots" of modular programs, i.e., at each analysis iteration, a snapshot of the sources is taken and used to perform the next analysis. Each time an analysis is started, the modules will be analyzed independently and incrementally (possibly several times) until a global fixpoint is reached. The algorithm is designed to work with any partition of the sources. The essential point of the algorithm is that analysis results are represented in a way that allows to partially invalidate the results that are no longer valid, correct, or accurate, while keeping the information that does not need recomputation. The information of source changes is used to invalidate (if necessary), and then decide which parts of the program (modules or predicates) need to be reanalyzed. We solve the problems related to the propagation of the fine-grain change information across module boundaries. We also work out the actions that need to be performed in order to recompute the analysis fixpoint incrementally after multiple additions and deletions across modules in the program. Finally, we prove that the analysis result is always correct and it is the best (most accurate) over-approximation of the actual behavior of the program.

We have implemented the proposed approach within the Ciao/CiaoPP system [2]. Our preliminary results show promising speedups for medium and large programs. The added finer granularity (which allows reusing analysis information both at the intra- and inter-modular levels) reduces significantly the cost with respect to modular analysis alone. The advantages of fine-grain incremental analysis –making the cost ideally proportional to the size of the changes– thus seem to carry over with our algorithm to the modular analysis case. Furthermore, the fine-grained propagation of analysis information of our algorithm improves performance with respect to traditional modular analysis even when analyzing from scratch.

References
---

1 I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. An Approach to Incremental and Modular Context-sensitive Analysis of Logic Programs. Technical Report CLIP-2/2018.0, The CLIP Lab, April 2018. URL: `https://arxiv.org/abs/1804.01839`.

2 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. *TPLP*, 12(1–2):219–252, 2012.

MASP-Reduce: A Proposal for Distributed Computation of Stable Models

Federico Igne
University of Udine, Udine, Italy and New Mexico State University, NM, USA
ignefederico@gmail.com

Agostino Dovier
University of Udine, Udine, Italy
agostino.dovier@uniud.it

Enrico Pontelli
New Mexico State University, NM, USA
epontell@cs.nmsu.edu

── Abstract ──

There has been an increasing interest in recent years towards the development of efficient solvers for Answer Set Programming (ASP) and towards the application of ASP to solve increasing more challenging problems. In particular, several recent efforts have explored the issue of scalability of ASP solvers when addressing the challenges caused by the need to ground the program before resolution. This paper offers an alternative solution to this challenge, focused on the use of distributed programming techniques to reason about ASP programs whose grounding would be prohibitive for mainstream ASP solvers. The work builds on a proposal of a characterization of answer set solving as a form of non-standard graph coloring. The paper expands this characterization to include syntactic extensions used in modern ASP (e.g., choice rules, weight constraints). We present an implementation of the solver using a distributed programming framework specifically designed to manipulate very large graphs, as provided by Apache Spark, which in turn builds on the MapReduce programming framework. Finally, we provide a few preliminary results obtained from the first prototype implementation of this approach.

The availability of efficient answer set solvers (e.g., CLASP and its descendants [8, 2]) gave Answer set programming (ASP) a leading role in languages for knowledge representation and reasoning. The simple syntax is surely one of the main strengths of the paradigm; moreover the stable models semantics intuitively resembles the human reasoning process in a clean and *logical* way. ASP is regarded as the computational embodiment of non-monotonic reasoning because of its simple syntax and elegant non-monotonic semantics. The popularity of ASP is demonstrated by the increasing number of authors publishing ASP-based research work in artificial intelligence as well as non-logic programming venues, and its use as a natural alternative to other paradigms (e.g., SAT solving). Most of the answer set solvers

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 8; pp. 8:1–8:4
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

are currently developed as two-phases procedures (save some exceptions – e.g., [3, 11]) . The first stage is called *grounding* and computes the equivalent propositional logic program of an input logic program, instantiating each rule over the domain of its variables. Modern solvers also apply some simplifications and heuristics to the program, in order to ease the computation during the second stage. The computation of the answer sets of a logic program is carried out by the *solving* stage, which also deals with the non-deterministic reasoning involved in the model.

ASP encoding of sophisticated applications in real-world domains (e.g., planning, phylogenetic inference) highlighted the strengths and weaknesses of this paradigm. Most of the times, the technology underlying the ASP solvers, lacks the ability to keep up with the demand of complex applications. This has been, for example, highlighted in a study on the use of ASP to address complex planning problems [13, 5, 6]. With respect to these studies, it is clear that one of the main limitations of this paradigm resides in the grounding process and the ability to compute the stable models of large ground programs. This limitation is even more obvious when the whole computation is performed in-memory.

This work tries to partially solve the problem of processing large ground programs that can exceed capabilities for in-memory computation – using parallelism and distributed computing. We aim to study, analyse and develop a fully distributed answer set solver and use a distributed environment to efficiently represent and reason over large programs whose grounding would be prohibitive for a single general-purpose machine. We popose a solver that uses *MapReduce*, a distributed programming paradigm, mainly used to work with huge volumes of data on structured networks of computers (*workers*) [4]. Implementations of the MapReduce model (e.g., [4]) are usually executed on clusters to take full advantage of the parallel nature of the architecture. The paradigm provides a basic interface consisting of two methods: map($\cdot$) that maps a function over a collection of objects and outputs a collection of "key-value" tuples; reduce($\cdot$) that takes as input a collection of key-value pairs and merges the values of all entries with the same key; the merging operation is user-defined.

An inspiration for the work proposed here comes from the proposal by Konczak et al. [9, 10], which addresses the problem of finding the answer sets of a ground normal logic program by means of computing the admissible colorings of the relative Rule Dependency Graph (RDG). This is done by defining a set of operators on the RDG of a program. These operators deal with the non-deterministic coloring of nodes and the deterministic propagation of colors. [1] used this technique in the development of the NoMoRe (Non-monotonic Reasoning with Logic Programs) solver. This implementation is purely sequential and in-memory.

In this research we investigate the above-mentioned graph coloring approach and extended it so as to include weight constraint rules. We investigate its mapping to MapReduce and other distributed programming paradigms that build over MapReduce. The solver we are developing, called MASP-Reduce, is written in Scala [12, 7], it uses Apache Spark [14] as a library for distributed computation, and it natively works on the Hadoop Distributed File System (HDFS). The library gives access to a complete set of primitives for the *MapReduce* programming paradigm, and on top of this, it implements GraphX, a distributed direct multigraph with a complete and easy-to-use interface [14].

The development of *MASP-Reduce* is heavily based on the concept of rule dependency graph of ASP programs. Graphs turn out to be a good data structure for distributed programming, since they can directly exploit the underlying network configuration. Up to now, the software is comprehensive of a solver and of a graph generator that converts a ground program in a rule dependency graph (see Figure below). As a future work, we plan

to implement a distributed grounder taking full advantage of the MapReduce paradigm, so that the Grounding block is incorporated into the Parallelization block.
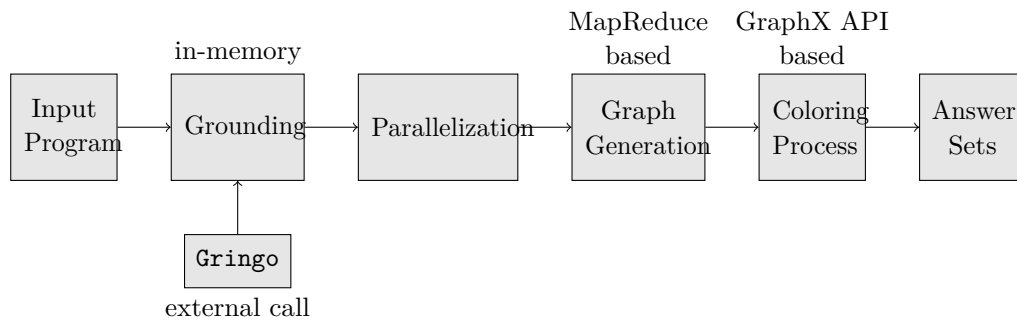
We tested the solver both in a local environment (a notebook) and in a distributed environment, namely four nodes of a cluster, where each node is a 12-core Intel CPUs, with each core dual hyperthreaded for a total of 48 OS-visible processors per node; each node has 256GB of RAM, ∼3TB of hard disk local storage and ∼512GB solid state local storage. The implementation works on simple examples. However, during the development we encountered a few challenges that prevented us from providing a full testing phase report. Spark is presented as an easy and ready-to-use tool for distributed programming; this might be true in a few cases, but most of the times one needs to fine-tune the system in order to reach an optimal configuration; this tuning process takes into account a vast number of parameters, and is mostly program-specific – and it is work in progress for our project.

As far as we know, this is the first work addressing the implementation of a distributed answer set solver using MapReduce paradigm and non-standard graph coloring as answer set characterization. This deeply influenced own roadmap, which couldn't take advices from previous works, leading to an incremental approach to development.

The system is still far from complete; we are planning on working on the development of a distributed grounder in the next few months. We are also considering the implementation of a few coloring heuristics and learning techniques to improve the performances of the solver.

References

1   C. Anger, M. Gebser, T. Linke, A. Neumann, and T. Schaub. The NoMoRe++ System. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*, volume 3662 of *LNCS*, pages 422–426. Springer, 2005.

2   Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. Clingcon: The next generation. *TPLP*, 17(4):408–461, 2017. `doi:10.1017/S1471068417000138`.

3   A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. GASP: Answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009.

4   J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*, volume 51, pages 107–113. ACM, January 2008.

5   Agostino Dovier, Andrea Formisano, and Enrico Pontelli. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *J. Exp. Theor. Artif. Intell.*, 21(2):79–121, 2009. `doi:10.1080/09528130701538174`.

6   Agostino Dovier, Andrea Formisano, and Enrico Pontelli. Perspectives on Logic-Based Approaches for Reasoning about Actions and Change. In Marcello Balduccini and Tran Cao

Son, editors, *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 259–279. Springer, 2011.

7 École Polytechnique Fédérale. Scala — Object-Oriented Meets Functional (website), 2018. [last accessed Feb. 2018] `http://www.scala-lang.org/`.

8 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694, 2014. `arXiv:1405.3694`.

9 K. Konczak, T. Linke, and T. Schaub. Graphs and Colorings for Answer Set Programming: Abridged Report. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Logic Programming and Nonmonotonic Reasoning: 7th International Conference, LPNMR 2004 Fort Lauderdale, FL, USA, January 6-8, 2004 Proceedings*, volume 2923 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2004.

10 Kathrin Konczak, Thomas Linke, and Torsten Schaub. Graphs and colorings for answer set programming. *TPLP*, 6(1-2):61–106, 2006. `doi:10.1017/S1471068405002528`.

11 Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. ASPeRiX, a first-order forward chaining approach for answer set computing. *TPLP*, 17(3):266–310, 2017.

12 Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: Updated for Scala 2.12*. Artima Incorporation, USA, 3rd edition, 2016.

13 T. Son and E. Pontelli. Planning for biochemical pathways: A case study of answer set planning in large planning problem instances. In Marina De Vos and Torsten Schaub, editors, *Proceedings of the First International SEA'07 Workshop, Tempe, Arizona, USA*, volume 281 of *CEUR Workshop Proceedings*, pages 116–130, January 2007.

14 The Apache Software Foundation. Apache Hadoop, Spark, and Graphx (websites), 2018. [last accessed Feb. 2018] `http://hadoop.apache.org/`, `https://spark.apache.org/`, `https://spark.apache.org/docs/latest/graphx-programming-guide.html`.

Declarative Algorithms in Datalog with Extrema: Their Formal Semantics Simplified

Carlo Zaniolo
University of California, Los Angeles, USA
zaniolo@cs.ucla.edu

Mohan Yang
Google, USA
yang@cs.ucla.edu

Matteo Interlandi
Microsoft Corporation, USA
matteo.interlandi@microsoft.com

Ariyam Das
University of California, Los Angeles, USA
ariyamo@cs.ucla.edu

Alexander Shkapsky
University of California, Los Angeles, USA
shkapsky@gmail.com

Tyson Condie
University of California, Los Angeles, USA
tcondie@cs.ucla.edu

Abstract

Recent advances are making possible the use of aggregates in recursive queries thus enabling the declarative expression classic algorithms and their efficient and scalable implementation. These advances rely the notion of Pre-Mappability (*PreM*) of constraints that, along with the seminaive-fixpoint operational semantics, guarantees formal non-monotonic semantics for recursive programs with min and max constraints. In this *extended abstract*, we introduce basic templates to simplify and automate task of proving *PreM*.

1 Pre-Mappable Extrema constraints in Recursive Rules

Pre-mappable (*PreM*) extrema constraints in recursive Datalog programs enable concise declarative formulations for classical algorithms [3]. The programs expressing these algorithms have formal non-monotonic semantics [1, 2]. For instance, a classical recursive application for traditional databases is Bill of Materials (BOM), where we have a Directed Acyclic Graph (DAG) of parts-subparts, `assbl(Part, Subpart, Qty)` describing how a given part is assembled using various subparts, each in a given quantity. Not all subparts are assembled, since basic parts are instead supplied by external suppliers in a given number of days, as per the facts `basic(Part, Days)`. Simple assemblies, such as bicycles, can be put together the very same day in which the last basic part arrives. Thus, the time needed to produce the

assembly is the maximum number of days required by the basic parts it uses. This can be computed by the following stratified program:

▶ **Example 1** (How many days till all the required parts arrive).

$$\text{deliv}(\text{Part},\text{Days}) \leftarrow \qquad \text{basic}(\text{Part},\text{Days}), \text{is\_max}(\text{Part},\text{Days}).$$
$$\text{deliv}(\text{Part},\text{Days}) \leftarrow \qquad \text{deliv}(\text{Sub},\text{Days}), \text{assbl}(\text{Part},\text{Sub}).$$
$$\text{actualDays}(\text{Part},\text{Days}) \leftarrow \; \text{deliv}(\text{Part},\text{Days}), \text{is\_max}(\text{Part},\text{Days}).$$

But the iterated fixpoint computation of the perfect model of this program can be very inefficient. This problem is solved by transferring `is_max((Part), Days)`, to the rules defining `deliv`, whereby the rule defining `actualDays` now becomes a redundant copy rule. Thus from the previous exo-max version of our program we obtain its endo-max version as follows:

$$\text{deliv}(\text{Part},\text{Days}) \leftarrow \qquad \text{basic}(\text{Part},\text{Days}), \text{is\_max}((\text{Part}),\text{Days}).$$
$$\text{deliv}(\text{Part},\text{Days}) \leftarrow \qquad \text{deliv}(\text{Sub},\text{Days}), \text{assbl}(\text{Part},\text{Sub}), \text{is\_max}((\text{Part}),\text{Days}).$$
$$\text{actualDays}(\text{Part},\text{Days}) \leftarrow \; \text{deliv}(\text{Part},\text{Days}).$$

The questions we need to answer about our `exo2endo` transformation are the following two:
  (i) is this a valid optimization inasmuch as the fixpoint computation of the endo-max program delivers the same model as the iterated fixpoint of the original exo-max program (and still allows recursive optimizations such as seminaive-fixpoint and magic-sets)?
  (ii) once we re-express `is_max` using negation, does the transformed program has a unique stable model semantics, efficiently computed as described jn (i).
The notion of *PreM* [3] provides provides a formal answer to both these questions?

▶ **Definition 2** (The *PreM* Property). In a given Datalog program, let P be the rules defining a (set of mutually) recursive predicate(s). Also let T be the ICO defined by P. Then, the constraint γ will be said to be *PreM* to T (and to P) when, for every interpretation I of P, we have that: $\gamma(T(I)) = \gamma(T(\gamma(I)))$.

The importance of this property follows from the fact that if $I = T(I)$ is a fixpoint for T, then we also have that $\gamma(I) = \gamma(T(I))$, and when γ is *PreM* to T then: $\gamma(I) = \gamma(T(I)) = \gamma(T(\gamma(I)))$. Now, let T_γ denote the application of T followed by γ, i.e., $T_\gamma(I) = \gamma(T(I))$. If I is a fixpoint for T and $I' = \gamma(I)$, then the above equality can be rewritten as: $I' = \gamma(I) = \gamma(T(\gamma(I))) = T_\gamma(I')$. Thus, when γ is *PreM*, the fact that I is a fixpoint for T implies that $I' = \gamma(I)$ is a fixpoint for $T_\gamma(I)$. In many programs of practical interest, the transfer of constraints under *PreM* produces optimized programs for the naive fixpoint computation that are safe and terminating even when the original programs were not. Thus we focus on programs where, for some integer n, $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, i.e., the fixpoint iteration converges after a finite number of steps n. As proven in [3], the fixpoint $T_\gamma^{\uparrow n}(\emptyset)$ so obtained is in fact a minimal fixpoint for T_γ, where γ denotes a min or max constraint:

▶ **Theorem 3.** *If γ is PreM to a positive program P with ICO T and, for some integer n, $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$, then:*
  (i) $T_\gamma^{\uparrow n}(\emptyset) = T_\gamma^{\uparrow n+1}(\emptyset)$ *is a minimal fixpoint for T_γ, and*
  (ii) $T_\gamma^{\uparrow n}(\emptyset) = \gamma(T^{\uparrow \omega}(\emptyset))$.
Therefore, when the *PreM* holds, declarative exo-min (or exo-max) programs are transformed into endo-min (or endo-max) programs having highly optimized operational semantics that computes the perfect model of the former and the unique stable modelof the latter.

2   Proving Premappability

The application of a min or max constraint to the ICO of a rule r can be expressed by the addition of a min or max goal to r, whereby *PreM* holds if this insertion of a new goal does not change the mapping defined by the rule. For the example at hand, we have:

$$\texttt{deliv}(\texttt{Part}, \texttt{Days}) \leftarrow$$
$$\texttt{deliv}(\texttt{Sub}, \texttt{Days})\backslash^{\texttt{is\_max((Sub),Days)}}/\texttt{assbl}(\texttt{Part}, \texttt{Sub}), \texttt{is\_max}((\texttt{Part}), \texttt{Days}).$$

Thus, we must prove that the insertion $\backslash^{\texttt{is\_max((Sub),Days)}}/$ does not change the mapping defined by our rule–a property that is guaranteed to hold if can prove that the original mapping already satisfies this constraint. We next define the concept of min- and max-constraints for individual tuples:

▶ **Definition 4.** We will say that a tuple $t \in R$ satisfies the min-constraint `is_min((X), A)` and write $X \xrightarrow{min} A$ when R contains no tuple having the same X-value and a smaller A-value. Symmetrically, we say that the tuple $t \in R$ satisfies the max-constraint `is_max((X), A)` and write $X \xrightarrow{max} A$ when R contains no tuple with the same X-value and a larger A-value.

Thus in our example we have $\texttt{Part} \xrightarrow{max} \texttt{Days}$ and must prove that $\texttt{Sub} \xrightarrow{max} \texttt{Days}$. Toward that goal, we observe that $X \xrightarrow{min} A$ and $X \xrightarrow{max} A$ can be informally viewed as "half functional dependencies (FDs)", since both must hold before we can conclude that $X \to A$. In fact, although min- and max-constraints on single tuples are much weaker than regular FDs, they preserve some of their important formal properties including those involving multivalued dependencies (MVDs) that result from the natural joins in the recursive rules – e.g. $\texttt{Sub} \longrightarrow\!\!\!\!\rightarrow \texttt{Days}$ and $\texttt{Sub} \longrightarrow\!\!\!\!\rightarrow \texttt{Part}$, in our example.

Therefore, the following properties, proven in [4], hold for tuple for min-constraints, max-constraints, and MVDs, and also illustrate the appeal of the arrow-based notation:

Min/Max Augmentation: If $X \xrightarrow{min} A$ and $Z \subseteq \Omega$, then $X \cup Z \xrightarrow{min} A$.
                                          If $X \xrightarrow{max} A$ and $Z \subseteq \Omega$, then $X \cup Z \xrightarrow{max} A$.
MVD Augmentation: If $X \longrightarrow\!\!\!\!\rightarrow Y$, $Z \subseteq \Omega$ and $Z \subseteq W$, then $X \cup W \longrightarrow\!\!\!\!\rightarrow Y \cup Z$.
Mixed Transitivity: If $Y \longrightarrow\!\!\!\!\rightarrow Z$ and $Z \xrightarrow{min} A$, with $A \notin Z$, then $Y \xrightarrow{min} A$.
                                          If $Y \longrightarrow\!\!\!\!\rightarrow Z$ and $Z \xrightarrow{max} A$, with $A \notin Z$, then $Y \xrightarrow{max} A$.

For the example at hand, $\texttt{Sub} \longrightarrow\!\!\!\!\rightarrow \texttt{Part}$ and $\texttt{Part} \xrightarrow{max} \texttt{Days}$ implies $\texttt{Sub} \xrightarrow{max} \texttt{Days}$, by mixed transitivity. Since this constraint holds, the additional goal $\backslash^{is\_max((Sub),Days)}/$ enforcing thus max constraint does not change it. Q.E.D.

────── **References** ──────

1   Alexander Shkapsky, Mohan Yang, Matteo Interlandi, Hsuan Chiu, Tyson Condie, and Carlo Zaniolo. Big Data Analytics with Datalog Queries on Spark. In *SIGMOD*, pages 1135–1149. ACM, 2016.

2   Mohan Yang, Alexander Shkapsky, and Carlo Zaniolo. Scaling up the performance of more powerful Datalog systems on multicore machines. *The VLDB Journal*, 26(2):229–248, 2017.

3   Carlo Zaniolo, Mohan Yang, Ariyam Das, Alexander Shkapsky, Tyson Condie, and Matteo Interlandi. Fixpoint semantics and optimization of recursive Datalog programs with aggregates. *TPLP*, 17(5-6):1048–1065, 2017.

4   Carlo Zaniolo, Mohan Yang, Matteo Interlandi, Ariyam Das, Alexander Shkapsky, and Tyson Condie. Declarative Algorithms by Aggregates in Recursive Queries: their Formal Semantics Simplified. Report no. 180001, Computer Science Department, UCLA, April, 2018.

Towards Static Performance Guarantees for Programs with Run-Time Checks

Maximiliano Klemen
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Spain
maximiliano.klemen@imdea.org
https://orcid.org/0000-0002-8503-8379

Nataliia Stulova
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Spain
nataliia.stulova@imdea.org
https://orcid.org/0000-0002-6804-2253

Pedro Lopez-Garcia
IMDEA Software Institute and Spanish Council for Scientific Research (CSIC), Spain
pedro.lopez@imdea.org
https://orcid.org/0000-0002-1092-2071

José F. Morales
IMDEA Software Institute, Spain
josef.morales@imdea.org
https://orcid.org/0000-0001-9782-8135

Manuel V. Hermenegildo
IMDEA Software Institute and Universidad Politécnica de Madrid (UPM), Spain
manuel.hermenegildo@imdea.org
https://orcid.org/0000-0002-7583-323X

—— **Abstract** ——
This document is an extended abstract of the Technical Report CLIP-1/2018.0.

Dynamic programming languages, such as Prolog, are a popular programming tool for many applications (e.g., web programming, prototyping, and scripting) due to their flexibility. The lack of inherent mechanisms for ensuring program data manipulation correctness (e.g., via full static typing or other forms of full static built-in verification) has sparked the evolution of flexible solutions, including assertion-based approaches in (constraint) logic languages, *soft-*

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 10; pp. 10:1–10:2
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and *gradual-* typing in functional languages, and contract-based approaches in imperative languages. A trait that many of these approaches share is that some parts of the specifications may be the subject of *run-time checking* (e.g., those that cannot be discharged statically in systems that support this functionality). However, such run-time checking comes at the price of overhead during program execution, that can affect execution time, memory use, energy consumption, etc., often in a significant way.

Reducing run-time checking overhead is a challenging problem. Proposed approaches include discharging as many checks as possible via static analysis, optimizing the dynamic checks themselves, or limiting run-time checking points. Nevertheless, there are cases in which a number of checks cannot be optimized away and must remain in place, because of software architecture choices (e.g., the case of the external interfaces of reusable libraries or servers), the need to ensure a high level of safety (e.g., in safety-critical systems), etc.

At the same time, low program performance may not always be due to the run-time checks. A technique that can help in this context is *profiling*, often used to detect performance "hot spots" and guide program optimization. Prior work on using profiling in the context of optimizing the performance of programs with run-time checks clearly demonstrates the benefits of this approach. Still, profiling infers information that is valid only for some particular input data values, and thus the results obtained may not be valid for other inputs, and thus detecting the worst cases can take a long time, and is impossible in general.

We propose a method that uses *static cost analysis* (instead of – or as a complement to – dynamic profiling) to infer *upper and lower bounds* (guarantees) on the costs introduced by the run-time checks in a program (i.e., on the run-time checking overhead). Such bounds are safe, in the sense that are guaranteed to never be violated in actual executions. Since such costs are data-dependent, these bounds take the form of functions that depend on certain characteristics (generally, data sizes) of the program inputs. Our method provides the programmer with feedback and guarantees at compile-time regarding the overhead that run-time checking will introduce. Unlike profiling, the bounds provided hold for all possible execution traces, and allow assessing how such overhead varies with the size of the input. We also propose an assertion-based mechanism (as an extension to the Ciao assertion verification framework [1]) that allows programmers to specify bounds on the admissible overhead introduced by run-time checking. Our method then statically and automatically compares the inferred run-time checking overhead against the admissible levels and provides guarantees on whether the instrumented program conforms with the specifications.

We formalize and implement the method in the context of the Ciao assertion language and the CiaoPP verification framework, and present results from its experimental evaluation. Such results suggest that our method is feasible and also promising in providing bounds that help the programmer understand at the algorithmic level the overheads introduced by the run-time checking required for the assertions in the program, in different scenarios, such as performing full run-time checking or checking only the module interfaces.

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. *TPLP*, 12(1–2):219–252, 2012.

2   M. Klemen, N. Stulova, P. Lopez-Garcia, J. F. Morales, and M. V. Hermenegildo. An Approach to Static Performance Guarantees for Programs with Run-time Checks. Technical Report CLIP-1/2018.0, The CLIP Lab, IMDEA Software Institute and T.U. Madrid, April 2018. `arXiv:1804.02380`.

SMT-Based Answer Set Solver CMODELS(DIFF) (System Description)

Da Shen

Department of Computer Science, University of Nebraska at Omaha
South 67th Street, Omaha, NE 68182, USA
dashen@unomaha.edu

Yuliya Lierler

Department of Computer Science, University of Nebraska at Omaha
South 67th Street, Omaha, NE 68182, USA
ylierler@unomaha.edu
🆔 https://orcid.org/0000-0002-6146-623X

―――― **Abstract** ――――

Many answer set solvers utilize Satisfiability solvers for search. Satisfiability Modulo Theory solvers extend Satisfiability solvers. This paper presents the CMODELS(DIFF) system that uses Satisfiability Modulo Theory solvers to find answer sets of a logic program. Its theoretical foundation is based on Niemala's characterization of answer sets of a logic program via so called level rankings. The comparative experimental analysis demonstrates that CMODELS(DIFF) is a viable answer set solver.

1 Introduction

This paper describes a new answer set solver CMODELS(DIFF). Its theoretical foundation lies on the generalizations of Niemela's ideas. Niemela [19] characterized answer sets of a normal logic program as models of a propositional formula called program's completion that satisfy "level ranking" requirements. In this sense, this system is a close relative of an earlier answer set solver LP2DIFF developed by Janhunen et al. [10]. Yet, LP2DIFF only accepts programs of a very restricted form. For example, neither choice rules nor aggregate expressions are allowed. Solver CMODELS(DIFF) permits such important modeling constructs in its input. Also, unlike LP2DIFF, the CMODELS(DIFF) system is able to generate multiple solutions.

The CMODELS(DIFF) system follows the tradition of answer set solvers such as ASSAT [16] and CMODELS [11]. In place of designing specialized search procedures targeting logic programs, these tools compute a program's completion and utilize Satisfiability solvers [9] – systems for finding satisfying assignments for propositional formulas – for search. Since

not all models of a program's completion are answer sets of a program, both ASSAT and CMODELS implement specialized procedures (based on loop formulas [16]) to weed out such models. Satisfiability Modulo Theory (SMT) solvers [2] extend Satisfiability solvers. They process formulas that go beyond propositional logic and may contain, for example, integer linear expressions. The CMODELS(DIFF) system utilizes this fact and translates a logic program into an SMT formula so that any model of this formula corresponds to an answer set of the program. It then uses SMT solvers for search. Unlike CMODELS or ASSAT, the CMODELS(DIFF) system does not need an additional step to weed out unwanted models. Also, it utilizes SMT-LIB – a standard input language of SMT solvers [1] – to interface with these systems. This makes its architecture open towards new developments in the realm of SMT solving. There is practically no effort involved in incorporating a new SMT system into the CMODELS(DIFF) implementation.

Creation of the CMODELS(DIFF) system was inspired by the development of recent constraint answer set programming solver EZSMT [21] that utilizes SMT solvers for finding solutions for "tight" constraint answer set programs. On the one hand, CMODELS(DIFF) restricts its attention to pure answer set programs. On the other hand, it goes beyond tight programs. In the future, we will extend CMODELS(DIFF) to accept non-tight constraint answer set programs. The theory developed in this work paves the way for such an extension.

Lierler and Susman [13] demonstrate that SMT formulas are strongly related to constraint programs [17]. Many efficient constraint solvers[1] exist. Majority of these systems focus on finite-domain constraint problems. The theoretical contributions of this work provide a foundation for developing a novel constraint-solver-based method in processing logic programs. Currently, CMODELS(DIFF) utilizes SMT-LIB to interface with SMT solvers. By producing output in MINIZINC – a standard input language of constraint solvers [18] – in place of SMT-LIB, CMODELS(DIFF) will become a constraint-based answer set solver. This is another direction of future work.

The outline of the paper is as follows. We start by reviewing the concepts of a logic program, a completion, tightness and an SMT logic SMT(IL). We then present a key concept of this work, namely, a level ranking; and state theoretical results. Section 4 presents transformations from logic programs to SMT(IL) by means of variants of level rankings. After that, we introduce the architecture of the CMODELS(DIFF) system and conclude with comparative experimental analysis.

2   Preliminaries

A *vocabulary* is a finite set of propositional symbols also called atoms. As customary, a *literal* is an atom a or its negation, denoted $\neg a$. A *(propositional) logic program*, denoted by Π, over vocabulary σ is a finite set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m,\ \ not\ \ not\ b_{m+1}, \ldots,\ not\ \ not\ b_n \tag{1}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq n$, is an atom or symbol $\top$ and $\bot$ in σ. Sometimes we use the abbreviated form of rule (1)

$$a \leftarrow B \tag{2}$$

---

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

Π_1	Π_2
$\{c\}.$	$\{c\}.$
$a \leftarrow c.$	$a \leftarrow c.$
	$a \leftarrow b.$
	$b \leftarrow a.$

$Comp(\Pi_1)$	$Comp(\Pi_2)$
$\neg\neg c \rightarrow c.$	$\neg\neg c \rightarrow c.$
$c \rightarrow a.$	$c \rightarrow a.$
$c \rightarrow \neg\neg c.$	$b \rightarrow a.$
$a \rightarrow c.$	$a \rightarrow b.$
	$c \rightarrow \neg\neg c.$
	$a \rightarrow c \vee b.$

Figure 1 Sample programs and their completions.

where B stands for the right hand side of an arrow in (1) and is also called a *body*. We identify rule (1) with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n \rightarrow a \qquad (3)$$

and B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \qquad (4)$$

Note that (i) the order of terms in (4) is immaterial, (ii) *not* is replaced with classical negation ($\neg$), and (iii) comma is replaced with conjunction ($\wedge$). When the body is empty it corresponds to the empty conjunction or $\top$. Expression $b_1 \wedge \ldots \wedge b_\ell$ in formula (4) is referred to as the *positive* part of the body and the remainder of (4) as the *negative* part of the body.

The expression a is the *head* of the rule. When a is $\bot$, we often omit it and say that the head is empty. We denote the set of nonempty heads of rules in Π by $hd(\Pi)$. We call a rule whose body is empty a *fact*. In such cases, we drop the arrow. We sometimes may identify a set X of atoms with the set of facts $\{a. \mid a \in X\}$.

We say that a set X of atoms *satisfies* a rule (1) if X satisfies a formula (3). The reduct Π^X of a program Π relative to a set X of atoms is obtained by first removing all rules (1) such that X does not satisfy its negative part $\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n$ and replacing all of its remaining rules with $a \leftarrow b_1, \ldots, b_\ell$. A set X of atoms is an *answer set*, if it is a minimal set that satisfies all rules of Π^X [15].

Ferraris and Lifschitz [6] show that a choice rule $\{a\} \leftarrow B$ can be seen as an abbreviation for a rule $a \leftarrow$ *not not* a, B. We adopt this abbreviation here. Choice rules were introduced in [20] and are commonly used in answer set programming languages.

It is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value *true* to every atom in X and *false* to every atom in $\sigma \setminus X$.

Consider sample programs listed in Figure 1. Program Π_1 has two answer sets, namely, $\{a, c\}$ and an empty set. Program Π_2 has two answer sets: $\{a, b, c\}$ and an empty set.

Completion and Tightness

Let σ be a vocabulary and Π be a program over σ. For every atom a in Π, by $Bodies(\Pi, a)$ we denote the set composed of the bodies B appearing in the rules of the form $a \leftarrow B$ in Π. The *completion* of Π [3], denoted by $Comp(\Pi)$, is the set of classical formulas that consists of the rules (1) in Π (recall that we identify rule (1) with implication (3)) and the implications

$$a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B \qquad (5)$$

for all atoms a in σ. When set $Bodies(\Pi, a)$ is empty, the implication (5) has the form $a \rightarrow \bot$. When a rule (2) is a fact $a.$, then we identify this rule with the unit clause a.

For example, completions of programs Π_1 and Π_2 are presented in Figure 1.

For the large class of logic programs, called *tight*, their answer sets coincide with models of their completion [5, 4]. This is the case for program Π_1 (we illustrate that Π_1 is tight, shortly). Yet, for non-tight programs, every answer set is a model of completion but not necessarily the other way around. For instance, set $\{a, b\}$ is a model of $Comp(\Pi_2)$, but not an answer set of Π_2. It turns out that Π_2 is not tight.

Tightness is a syntactic condition on a program that can be verified by means of program's dependency graph. The *dependency graph* of Π is the directed graph G such that

- the nodes of G are the atoms occurring in Π, and
- for every rule (1) in Π whose head is an atom, G has an edge from atom a to each atom $b_1, \ldots, b_\ell$.

A program is called *tight* if its dependency graph is acyclic.

For example, the dependency graph of program Π_1 consists of two nodes, namely, a and c, and a single edge from a to c . This graph is acyclic and hence Π_1 is tight. On the other hand, it is easy to see that the graph of Π_2 is not acyclic.

Logic SMT(IL)

We now introduce the notion of Satisfiability Modulo Theory (SMT) [2] for the case when Linear Integer Arithmetic is a considered theory. We denote this SMT instance by SMT(IL).

Let σ be a vocabulary and χ be a finite set of integer variables. The set of *atomic formulas* of SMT(IL) consists of propositions in σ and linear constraints of the form

$$a_1 x_1 \pm \cdots \pm a_n x_n \bowtie a_{n+1} \tag{6}$$

where $a_1, \ldots, a_{n+1}$ are integers and $x_1, \ldots, x_n$ are variables in χ, $\pm$ stands for $+$ or $-$, and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. The set of SMT(IL) formulas is the smallest set that contains the atomic formulas and is closed under $\neg$ and conjunction $\wedge$. Other connectives such as $\top, \bot, \vee, \rightarrow$, and $\leftrightarrow$ can be defined in terms of $\neg$ and $\wedge$ as customary.

A valuation τ consists of a pair of functions

- $\tau_\sigma : \sigma \rightarrow \{true, false\}$ and
- $\tau_\chi : \chi \rightarrow \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers.

A valuation interprets all SMT(IL) formulas by defining

- $\tau(p) = \tau_\sigma(p)$ when $p \in \sigma$,
- $\tau(a_1 x_1 \pm \cdots \pm a_n x_n \bowtie a_{n+1}) = true$ iff $a_1 \tau_\chi(x_1) \pm \cdots \pm a_n \tau_\chi(x_n) \bowtie a_{n+1}$ holds,

and applying the usual rules for the Boolean connectives.

We say that an SMT(IL) formula Φ is satisfied by a valuation τ when $\tau(\Phi) = true$. A set of SMT(IL) formulas is satisfied by a valuation when every formula in the set is satisfied by this valuation. We call a valuation that satisfies an SMT(IL) formula a *model*.

3 Level Rankings

Niemela [19] characterized answer sets of "normal" logic programs in terms of "level rankings". *Normal* programs consist of rules of the form (1), where $n = m$ and a is an atom. Lierler and Susman [13] generalized the concept of level ranking to programs considered in this paper that include choice rules and denials (rules with empty head).

By $\mathbb{N}$ we denote the set of natural numbers. For a rule (2), by B^+ we denote its positive part and sometimes identify it with the set of atoms that occur in it, i.e., $\{b_1, \ldots, b_l\}$. For a program Π, by $At(\Pi)$ we denote the set of atoms occurring in it.

▶ **Definition 1.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \to \mathbb{N}$ is a *level ranking* of X for Π when for each $a \in X$, there is B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+$ it holds that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$.

Niemela [19] observed that for a normal logic program, a model X of its completion is also its answer set when there is a level ranking of X for the program. Lierler and Susman [13] generalized this result to programs with double negation *not not*:

▶ **Theorem 2** (Theorem 1 [13]). *For a program Π and a set X of atoms that is a model of its completion $Comp(\Pi)$, X is an answer set of Π if and only if there is a level ranking of X for Π.*

The nature of a level ranking is such that there is an infinite number of level rankings for the same answer set of a program. Theorem below illustrates that we can add a *single* linear constraint to limit the number of level rankings by utilizing the size of a program.

▶ **Theorem 3.** *For a logic program Π and its answer set X, we can always construct a level ranking of X for Π such that, for every $a \in X$, $lr(a) \leq |At(\Pi)|$.*

Proof. Since there is an answer set X, by Theorem 2 there exists some level ranking lr' of X for Π. Then, we can always use the level ranking lr' to construct a level ranking lr of X for Π such that, for every $a \in X$, $lr(a) \leq |At(\Pi)|$. Below we describe the method.

For an integer y, by $s(y)$ we denote the following set of atoms

$$\{a \mid a \in X, lr'(a) = y\}.$$

Let Y be the set of integers so that

$$\{y \mid a \in X, lr'(a) = y\}.$$

Let Y^s denote the sorted list $[y_1, \ldots, y_k]$ constructed from all integers of Y, such that $y_1 < y_2 < \ldots < y_k$. Note that $y_i > y_j$ if and only if $i > j$. Obviously, $|Y| \leq |At(\Pi)|$. Thus, $k \leq |At(\Pi)|$. For every element y_i in Y^s and every atom $a \in s(y_i)$, we assign $lr(a) = i$. Consequently, $lr(a) \leq |At(\Pi)|$.

Now we prove that lr is indeed a level ranking. According to the definition of lr', for each atom $a \in X$, there exists B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+$ it holds that $\mathrm{lr}'(a) - 1 \geq \mathrm{lr}'(b)$. We show that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$ also holds for each b in this B^+. Atoms a, b belong to some sets $s(y_{k_a})$ and $s(y_{k_b})$ respectively, where $k_a, k_b \leq k$. By the definition of $s(\cdot)$, $y_{k_a} = lr'(a)$ and $y_{k_b} = lr'(b)$. Since $lr'(a) > lr'(b)$, $y_{k_a} > y_{k_b}$. Since for any i and j, $y_i > y_j$ if and only if $i > j$, we derive that $k_a > k_b$. By the construction of lr, $lr(a) = k_a$ and $lr(b) = k_b$. Consequently, $lr(a) - 1 \geq lr(b)$ also holds. Thus, lr is a level ranking by definition. ◀

Strong level ranking

Niemela [19] introduced the concept of a strong level ranking so that only one strong level ranking exists for an answer set. It is obviously stricter than the condition captured in Theorem 3. Yet, the number of linear constraints in formulating the conditions of strong level ranking is substantially greater. We now generalize the concept of a strong level ranking to the case of logic programs considered here and then state the formal result on the relation of answer sets and strong level rankings.

▶ **Definition 4.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \to \mathbb{N}$ is a *strong level ranking* of X for Π when lr is a level ranking and for each $a \in X$ the following conditions hold:

1. If there is B in $Bodies(\Pi, a)$ such that X satisfies B and B^+ is empty, then $\mathrm{lr}(a) = 1$.
2. For every B in $Bodies(\Pi, a)$ such that X satisfies B and B^+ is not empty, there is at least one $b \in B^+$ such that $\mathrm{lr}(b) + 1 \geq \mathrm{lr}(a)$.

▶ **Theorem 5.** *For a program Π and a set X of atoms that is a model of its completion $Comp(\Pi)$, X is an answer set of Π if and only if there is a strong level ranking of X for Π.*

Proof. This proof follows the argument provided for Theorem 2 in [19], but respects the terminology used here. We start by defining an operator $T_\Pi(I)$ for a program Π and a set I over $At(\Pi) \cup \bot$ as follows:

$$T_\Pi(I) = \{a \mid a \leftarrow B \in \Pi, I \text{ satisfies } B\}.$$

For this operator we define

$$T_\Pi \uparrow 0 = \emptyset,$$

and for $i = 0,1,2, \dots$

$$T_\Pi \uparrow (i + 1) = T_\Pi(T_\Pi \uparrow i).$$

Left-to-right: Assume X is an answer set of Π. We can construct a strong level ranking lr of X for Π using the $T_{\Pi^X}(\cdot)$ operator. As X is an answer set of Π, we know that $X = T_{\Pi^X} \uparrow \omega$ and for each $a \in X$ there is a unique i such that $a \in T_{\Pi^X} \uparrow i$, but $a \notin T_{\Pi^X} \uparrow (i - 1)$. Let $lr(a) = i$. We now illustrate that lr is indeed a strong level ranking.

First, we illustrate that lr is a level ranking. For $a \in X$ there is a rule $a \leftarrow B$ of the form (1) such that $a \leftarrow b_1, \dots, b_l \in \Pi^X$ and $T_{\Pi^X} \uparrow (i - 1)$ satisfies $b_1 \wedge \cdots \wedge b_l$. Consequently, for every b_j in $\{b_1, \dots, b_l\}$, $lr(b_j) \leq i - 1$. Thus, $lr(a) - 1 \geq lr(b_j)$. Also, from the way the reduct is constructed, it follows that X satisfies body B of rule $a \leftarrow B$.

Second, we show that Condition 1 of the definition of strong level ranking holds for lr. If there is $a \leftarrow B \in \Pi$ such that X satisfies B and B^+ is empty, then $a \leftarrow \top$ is in Π^X. By definition of the $T_{\Pi^X}(\cdot)$ operator, $a \in T_{\Pi^X} \uparrow 1$. Consequently, $lr(a) = 1$ holds.

Third, we demonstrate that Condition 2 holds for lr. For $a \in X$, by the construction of lr using the $T_{\Pi^X}(\cdot)$ operator we know that there is a unique i such that $lr(a) = i$, $a \in T_{\Pi^X} \uparrow i$, but $a \notin T_{\Pi^X} \uparrow (i - 1)$. Proof by contradiction. Assume that there is a rule $a \leftarrow B \in \Pi$ such that X satisfies B and B^+ is not empty, but for all $b \in B^+$, $lr(b) + 1 < lr(a)$ holds. Then for all $b \in B^+$, $lr(b) < lr(a) - 1$. Thus, $lr(b) < i - 1$. It follows that all $b \in B^+$ belong to $T_{\Pi^X} \uparrow (i - 2)$. Hence, by the definition of $T_{\Pi^X}(\cdot)$ operator, $a \in T_{\Pi^X} \uparrow (i - 1)$, which contradicts that $a \notin T_{\Pi^X} \uparrow (i - 1)$. Thus, there is at least one $b \in B^+$ such that $\mathrm{lr}(b) + 1 \geq \mathrm{lr}(a)$.

Right-to-left: Assume that there is a strong level ranking of X for Π. By the definition, it is also a level ranking. Recall that X is a model of $Comp(\Pi)$. By Theorem 2, X is an answer set of Π. ◀

SCC level ranking

Niemela [19] illustrated how one can utilize the structure of the dependency graph corresponding to a normal program to reduce the number of linear constraints in capturing conditions similar to these of level ranking. We now generalize these results to logic programs with doubly negated atoms and denials.

Recall that a strongly connected component of a directed graph is a maximal set V of nodes such that each pair of nodes in V is reachable from each other. We call a set of atoms in a program Π a *strongly connected component* (SCC) of Π when it is a strongly connected component in the dependency graph of Π. The SCC including an atom a is denoted by $SCC(a)$. A *non-trivial* SCC is an SCC that consists of at least two atoms. We denote the set of atoms in all non-trivial SCCs of Π by $NT(\Pi)$.

▶ **Definition 6.** For a logic program Π and a set X of atoms over $At(\Pi)$, a function lr: $X \cap NT(\Pi) \to \mathbb{N}$ is a *SCC level ranking* of X for Π when for each $a \in X \cap NT(\Pi)$, there is B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+ \cap SCC(a)$ it holds that $\text{lr}(a) - 1 \geq \text{lr}(b)$.

The byproduct of the definition of SCC level rankings is that for tight programs SCC level ranking trivially exists since it is a function whose domain is empty. Thus no linear constraints are produced.

▶ **Theorem 7.** *For a program Π and a set X of atoms that is a model of its completion $Comp(\Pi)$, X is an answer set of Π if and only if there is an SCC level ranking of X for Π.*

This is a generalization of Theorem 4 in [19]. Its proof follows the lines of the proof presented there with the reference to Theorem 2.

▶ **Theorem 8.** *For a satisfiable logic program Π and its answer set X, we can always construct an SCC level ranking of X for Π such that, for every $a \in X$, $lr(a) \leq |SCC(a)|$.*

This theorem can be proved by applying the similar argument as in the proof of Theorem 3 to each SCC. This result allows us to set minimal upper bounds for $lr(a)$ in order to reduce search space.

Further, Niemela [19] introduces the concept of strong SCC level ranking and states a similar result to Theorem 7 for that concept. It is straightforward to generalize these results to logic programs considered here.

4 From Logic Programs to SMT(IL)

In this section we present a mapping from a logic program to SMT(IL) such that the models of a constructed SMT(IL) theory are in one-to-one correspondence with answer sets of the program. Thus, any SMT solver capable of processing SMT(IL) expressions can be used to find answer sets of logic programs. The developed mappings generalize the ones presented by Niemela [19].

For a rule $a \leftarrow B$ of the form (1), the auxiliary atom β_B, equivalent to its body, is defined as

$$\beta_B \leftrightarrow b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge b_{m+1} \wedge \ldots \wedge b_n \tag{7}$$

When the body of a rule consist of a single element, no auxiliary atom is introduced (the single element itself serves the role of an auxiliary atom).

Let Π be a program. We say that an atom a is a *head atom* in Π if it is the head of some rule. Any atom a in Π such that

- it is a head atom, or
- it occurs in some positive part of the body of some rule whose head is an atom,

we associate with an integer variable denoted by lr_a. We call such variables level ranking variables. For each head atom a in Π, we construct an SMT(IL) formula

$$a \to \bigvee_{a \leftarrow B \in \Pi} (\beta_B \wedge \bigwedge_{b \in B^+} lr_a - 1 \geq lr_b). \tag{8}$$

We call the conjunction of formulas (8) for the head atoms in program Π a *level ranking formula* of Π.

For example, the level ranking formula of program Π_2 in Figure 1 follows

$$\big(c \to \neg\neg c\big) \wedge \big(a \to (c \wedge lr_a - 1 \geq lr_c) \vee (b \wedge lr_a - 1 \geq lr_b)\big) \wedge \big(b \to a \wedge lr_b - 1 \geq lr_a\big). \tag{9}$$

▶ **Theorem 9.** *For a logic program Π and the set F of SMT(IL) formulas composed of $Comp(\Pi)$ and a level ranking formula of Π*
1. *If a set X of atoms is an answer set of Π, then there is a satisfying valuation τ for F such that $X = \{a \mid a \in At(\Pi)$ and $\tau(a) = true\}$.*
2. *If valuation τ is satisfying for F, then the set $\{a \mid a \in At(\Pi)$ and $\tau(a) = true\}$ is an answer set for Π.*

This is a generalization of Theorem 6 in [19]. Its proof follows the lines of the proof presented there with the reference to Theorem 2.

SCC level ranking

For each atom a in the set $NT(\Pi)$, we introduce an auxiliary atom ext_a. If there exists some rule $a \leftarrow B$ in Π such that $B^+ \cap SCC(a) = \emptyset$, then we construct an SMT(IL) formula

$$ext_a \leftrightarrow \bigvee_{a \leftarrow B \in \Pi \text{ and } B^+ \cap SCC(a) = \emptyset} \beta_B; \tag{10}$$

otherwise, we construct a formula

$$\neg ext_a. \tag{11}$$

We also introduce an SMT(IL) formula:

$$a \to ext_a \vee \bigvee_{a \leftarrow B \in \Pi \text{ and } B^+ \cap SCC(a) \neq \emptyset} (\beta_B \wedge \bigwedge_{b \in B^+ \cap SCC(a)} lr_a - 1 \geq lr_b). \tag{12}$$

We call the conjunction of formulas (10), (11) and (12) a *SCC level ranking formula* of Π.

For instance, $NT(\Pi_1)$ is empty, so we introduce no SCC level ranking formula for program Π_1. The SCC level ranking formula of program Π_2 follows

$$\big(ext_a \leftrightarrow c\big) \wedge \neg ext_b \wedge \big(a \to ext_a \vee (b \wedge lr_a - 1 \geq lr_b)\big) \wedge \big(b \to ext_b \vee (a \wedge lr_b - 1 \geq lr_a)\big). \tag{13}$$

The claim of Theorem 9 holds also when we replace *a level ranking formula of* Π with *an SCC level ranking formula of* Π in its statement.

Strong level ranking

For each rule $a \leftarrow B$ in program Π we construct an SMT(IL) formula

$$\begin{aligned} a \wedge \beta_B &\to lr_a = 1 &&\text{when } B^+ = \emptyset, \\ a \wedge \beta_B &\to \bigvee_{b \in B^+} lr_b + 1 \geq lr_a &&\text{otherwise.} \end{aligned} \tag{14}$$

We call the conjunction of formulas (8) and (14) a *strong level ranking formula* of Π.

For example, the strong level ranking formula of program Π_2 is a conjunction of formula (9) and formula

$$
\begin{aligned}
&\left(c \wedge \neg\neg c \rightarrow lr_c = 1\right) \;\wedge\; \left(a \wedge c \rightarrow lr_c + 1 \geq lr_a\right) \;\wedge\; \\
&\left(a \wedge b \rightarrow lr_b + 1 \geq lr_a\right) \;\wedge\; \left(b \wedge a \rightarrow lr_a + 1 \geq lr_b\right).
\end{aligned}
$$

We now state a similar result to Theorem 9 that makes an additional claim on one-to-one correspondence between the models of a constructed SMT(IL) formula with the use of strong level ranking formula and answer sets of a program.

▶ **Theorem 10.** *For a logic program Π and the set F of* SMT(IL) *formulas composed of* $Comp(\Pi)$ *and a strong level ranking formula of Π*
1. *If a set X of atoms is an answer set of Π, then there is a satisfying valuation τ for F such that $X = \{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$.*
2. *If valuation τ is satisfying for F, then the set $\{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\}$ is an answer set for Π.*
3. *If valuations τ and τ' satisfy F and are distinct, then*

$$
\{a \mid a \in At(\Pi) \text{ and } \tau(a) = true\} \neq \{a \mid a \in At(\Pi) \text{ and } \tau'(a) = true\}.
$$

Strong SCC level ranking

For each atom $a \in NT(\Pi)$, we construct a formula

$$
ext_a \rightarrow lr_a = 1, \tag{15}
$$

and for each rule $a \leftarrow B$ such that $B^+ \cap SCC(a) \neq \emptyset$, we introduce a formula

$$
a \wedge \beta_B \rightarrow \bigvee_{b \in B^+ \cap SCC(a)} lr_b + 1 \geq lr_a. \tag{16}
$$

We call the conjunction of formulas (10), (11), (12), (15) and (16) a *strong SCC level ranking formulas* of Π.

For instance, $NT(\Pi_1)$ is empty, so we introduce no strong SCC level ranking formula for program Π_1. The strong SCC level ranking formula of program Π_2 is a conjunction of formula (13) and formula

$$
\left(ext_a \rightarrow lr_a = 1\right) \wedge \left(ext_b \rightarrow lr_b = 1\right) \wedge \left(a \wedge b \rightarrow lr_b + 1 \geq lr_a\right) \wedge \left(b \wedge a \rightarrow lr_a + 1 \geq lr_b\right).
$$

The claim of Theorem 10 holds also when we replace *a strong level ranking formula of* Π with *a strong SCC level ranking formula of* Π in its statement.

5 The CMODELS(DIFF) system

We are now ready to describe the the CMODELS(DIFF)[2] system in detail. It is an extension of the CMODELS [11] system. Figure 2 illustrates the pipeline architecture of CMODELS(DIFF). This system takes an arbitrary (tight or non-tight) logic program in the language supported

---

[2] CMODELS(DIFF) is posted at `https://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/cmodels-diff.php`
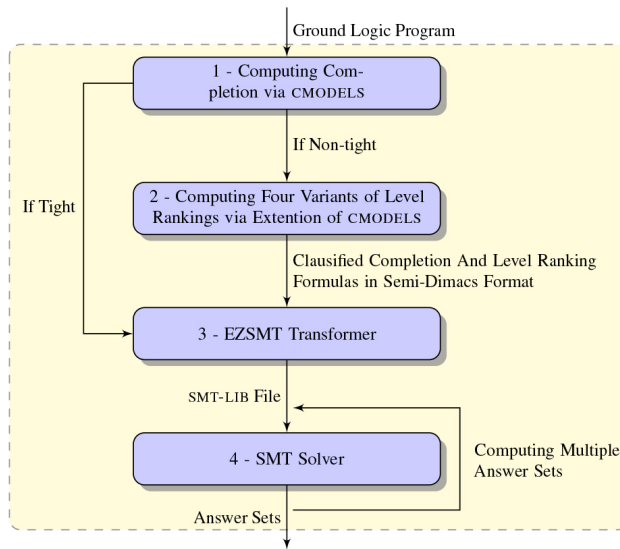
Figure 2 CMODELS(DIFF) Pipeline.

by CMODELS as an input. These logic programs may contain such features as choice rules and aggregate expressions. The rules with these features are translated by CMODELS [11] into rules considered here. The CMODELS(DIFF) system translates a logic program into SMT(IL) formulas, after which an SMT solver is called to find models of these formulas (that correspond to answer sets).

(1, 2) Computing Completion and Level Ranking Formulas

The CMODELS(DIFF) system utilizes the original algorithm of CMODELS to compute completion, during which CMODELS determines whether the program is tight or not. If the program is not tight, the corresponding level ranking formula is added.

Flags `-levelRanking`, `-levelRankingStrong`, `-SCClevelRanking`, and `-SCClevelRankingStrong` instruct CMODELS(DIFF) to construct a level ranking formula, a strong level ranking formula, a SCC level ranking formula, and a strong SCC level ranking formula, respectively. And, `-SCClevelRanking` is chosen by default. Finally, completion and the level ranking formula are clausified using the same technique as in original CMODELS. The CMODELS(DIFF) system outputs the resulting clauses into a text file in semi-Dimacs format [21].

(3, 4) Transformation and Solving

The transformer is taken from EZSMT v1.1. It converts the semi- Dimacs output from step (2) into SMT-LIB syntax (SMT-LIB is a standard input language for SMT solvers [1]). By default, the SMT-LIB output contains an instruction that sets the logic of SMT solvers to Linear Integer Arithmetic. If the transformer is invoked with the parameter `difference-logic`, then the SMT-LIB output sets the logic of SMT solvers to Difference Logic instead.

Finally, one of the SMT solvers CVC4, Z3, or YICES is called to compute models by using flags `-cvc4`, `-z3`, or `-yices`. (In fact, any other SMT solver supporting SMT-LIB can be utilized.) The CMODELS(DIFF) system post-processes the output of the SMT solvers

mentioned above to produce answer sets in a typical format disregarding any auxiliary atoms or integer variables that are created during the system's execution.

The CMODELS(DIFF) system allows us to compute multiple answer sets. Currently, SMT solvers typically find only a single model. We design a process to enumerate all models. For a logic program Π, after an SMT solver finds a model and exits, the CMODELS(DIFF) system constructs a clause that consists of (i) atoms in $At(\Pi)$ that are assigned *false* by the model and (i) negations of atoms in $At(\Pi)$ that are assigned *true* by the model. This clause is added into the SMT-LIB formula previously computed. Then, the SMT solver is called again taking the new input. The process is performed repeatedly, until the SMT-LIB formula becomes unsatisfiable.

In summary, CMODELS(DIFF) has eight possible configurations. We can choose one from the four variants of level ranking formulas, and choose a logic from either Linear Integer Arithmetic or Difference Logic for the invoked SMT solver.

6 Experiments

We benchmark CMODELS(DIFF) on seven problems, to compare its performance with that of other ASP solvers, namely CMODELS and CLASP [7]. All considered benchmarks are *non-tight* programs. The first two benchmarks are Labyrinth and Connected Still Life, which are obtained from the Fifth Answer Set Programming Competition[3]. We note that the original encoding of Still Life is an optimization problem, and we turn it into a decision one. The next three benchmarks originate from Asparagus[4]. The selected problems are RandomNonTight, Hamiltonian Cycle and Wire Routing. We also consider five instances of Wire Routing from RST Construction[5]. Then, we use Bounded Models as the sixth benchmark[6]. Our last benchmark, Mutual Exclusion, comes from Synthesis Benchmarks[7]. We rewrite the seven encodings to fit the syntax of GRINGO 4, and call GRINGO v. 4.5.3[8] to produce ground programs serving as input to all benchmarked systems. All benchmarks are posted at the CMODELS(DIFF) website provided at Footnote 2.

All benchmarks are run on an Ubuntu 16.04.1 LTS (64-bit) system with an Intel core i5-4250U processor. The resource allocated for each benchmark is limited to one cpu core and 4GB RAM. We set a timeout of 1800 seconds. No problems are solved simultaneously.

Numbers of instances are shown in parentheses after names of benchmarks. We present cumulative time of all instances for each benchmark with numbers of unsolved instances due to timeout or insufficient memory inside parentheses. All the steps involved, including grounding and transformation, are reported as parts of solving time.

Five distinct solvers are benchmarked: (1) CMODELS(DIFF) invoking SMT solver CVC4 v. 1.4; (2) CMODELS(DIFF) invoking SMT solver Z3 v. 4.5.1; (3) CMODELS(DIFF) invoking SMT solver YICES v. 2.5.4; (4) CLASP v. 3.1.3; (5) CMODELS v. 3.86.1 with Satisfiability solver Minisat v. 2.0 beta. We use DIFF-CVC4, DIFF-Z3, and DIFF-YICES to denote three variants of CMODELS(DIFF) used in the experiments.

---

[3] `https://www.mat.unical.it/aspcomp2014/`

[4] `https://asp.haiti.cs.uni-potsdam.de/`

[5] `http://people.sabanciuniv.edu/~esraerdem/ASP-benchmarks/rst-basic.html`

[6] `http://users.ics.aalto.fi/~kepa/experiments/boundsmodels/`

[7] `http://www2.informatik.uni-stuttgart.de/fmi/szs/research/projects/synthesis/benchmarks030923.html`

[8] `http://potassco.sourceforge.net/`

Table 1 Experimental Summary.

Benchmark	DIFF-CVC4 LIA	DIFF-Z3 LIA	DIFF-YICES LIA	DIFF-Z3 DL	DIFF-YICES DL	CMODELS	CLASP
Still Life (26)	731	5423(1)	203	899	194	<u>647</u>	**10.8**
Ham. Cycl. (50)	15.39	9.78	4.54	6.61	3.57	<u>1.19</u>	**0.53**
Wire Rout. (10)	1378	562.36	1562	2983(1)	2089(1)	<u>409</u>	**12.5**
Bound. Mod. (8)	6.08	4.30	2.34	2.93	2.20	<u>1.59</u>	**1.38**
Labyrinth (30)	19543(8)	27794(12)	20425(10)	22023(9)	21836(9)	<u>16408</u>(7)	**5826**(2)
Rand. Nont. (20)	27.8	8.65	6.84	7.72	6.47	**<u>1.39</u>**	3.52
Mut. Excl. (5)	5.26	2.72	1.70	2.28	1.50	<u>0.30</u>	**0.13**

Table 1 summarizes main results. Under the name of variants of the CMODELS(DIFF) systems, we state the configuration used for this solver. Namely, "LIA" and "DL" denote that the logic of SMT solvers is set to Linear Integer Arithmetic and Difference Logic, respectively. All DIFF systems in the table are invoked with flag `-SCClevelRanking`. Systems CLASP and CMODELS are run with default settings. We benchmarked CMODELS(DIFF) with all eight possible configurations. Yet, we do not present all of the data here. CMODELS(DIFF) invoked with `-levelRanking` and `-levelRankingStrong` flags shows worse performance than settings `-SCClevelRanking` and `-SCClevelRankingStrong`, respectively. That is why we avoid presenting the results on configurations `-levelRanking` and `-levelRankingStrong`. Also, adding constraints for strong level ranking typically slightly degrades the performance so we do not present the results for the `-SCClevelRankingStrong` configuration. We note that SMT solver CVC4 implements the same procedure for processing Difference Logic statement and Linear Integer Arithmetic statements.

Observations

We observe that system CLASP almost always displays the best results. This is not surprising as this is one of the best *native* answer set solvers currently available. Its search method is attuned towards processing logic programs. Given that SMT solvers are agnostic towards specifics of logic programs it is remarkable how good the performance of CMODELS(DIFF) is. In some cases it is comparable to that of CLASP.

It is the case that many Satisfiability solvers and answer set solvers share a lot in common [12]. For example, answer set solver CLASP starts by computing clausified programs completion and then later applies to it *Unit* propagator search technique stemming from Satisfiability solving. That is reminiscent of the process that system CMODELS(DIFF) undertakes. It also computes program's completion so that *Unit* propagator of SMT solvers is applicable to it.

We conjecture that the greatest difference between CMODELS(DIFF) and CLASP lies in the fact that

- in CMODELS(DIFF) integer linear constraints encode the conditions to weed out unwanted models of completion; SMT solvers implement search techniques/propagators to target these integer linear constraint;
- in CLASP the structure of the program is taken into account by the so called *Unfounded* propagator for this task.

In case of Still Life, Hamiltonian Cycle, Wire Routing, and Bounded Models benchmarks (marked in bold in Table 1) there is *one more substantial difference.* These encodings contain aggregates. CLASP implements specialized search techniques to benefit from the compact

representations that aggregates provide. System CMODELS(DIFF) translates aggregates away, which often results in a bigger problem encoding that the system has to deal with. System CMODELS also translates aggregates away. This is why we underline the solving times of CMODELS, as it is insightful to compare the performance of CMODELS to that of CMODELS(DIFF) alone. Indeed, CMODELS(DIFF) utilizes the routines of CMODELS for eliminating aggregates and computing the completion of the resulting program. Thus, the only difference between these systems is in how they eliminate models of completion that are not answer sets. System CMODELS(DIFF) utilizes level rankings for that. System CMODELS implements a propagator in spirit of *Unfounded* propagator of CLASP, but the propagator of CMODELS is only used when a model of completion is found; CLASP utilizes this propagator as frequently as it utilizes *Unit* propagator [14, Section 5]. We believe that when we observe a big difference in performance of CMODELS(DIFF) and CLASP, this attributes to the benefits gained by the utilization of specialized *Unfounded* and "aggregate" propagators by CLASP. Yet, level ranking formulas seem to provide a viable alternative to *Unfounded* propagator and open a door for utilization of SMT solvers for dealing with non-tight programs. This gives us grounds to believe that the future work on extending constraint answer set solver EZSMT to accept non-tight programs is a viable direction.

As we noted earlier SCC level rankings yield best performance among the four variants of level rankings. Furthermore, Table 1 illustrates the following. The logic of SMT solvers does not make an essential difference. Overall, CMODELS(DIFF)-YICES with Linear Integer Arithmetic logic performs best within the presented CMODELS(DIFF) configurations. Obviously, utilizing better SMT solvers can improve the performance of CMODELS(DIFF) in the future. Notably, this does not require modifications to CMODELS(DIFF), since SMT-LIB used by CMODELS(DIFF) is a standard input language of SMT solvers.

7 Conclusion

In this paper we presents the CMODELS(DIFF) system that takes a logic program and translates it into an SMT-LIB formula which is then solved by an SMT solver to find answer sets of the given program. Our work parallels the efforts of an earlier answer set solver LP2DIFF [10]. The CMODELS(DIFF) system allows richer syntax such as choice rules and aggregate expressions, and enables computation of multiple solutions. (In this work we extended the theory of level rankings to the case of programs with choice rules and denials.) We note that the LP2NORMAL[9] tool can be used as a preprocessor for LP2DIFF in order to enable this system to process logic programs with richer syntax. In the future, we will compare performance of CMODELS(DIFF) and LP2DIFF experimentally. Yet, we do not expect to see great difference in their performance when the same SMT solver is used as a backend. Also, we would like to conduct more extensive experimental analysis to support our conjecture on the benefits of specialized "aggregate" propagator and *Unfounded* propagator employed by CLASP.

The technique implemented by CMODELS(DIFF) for enumerating multiple answer sets of a program is basic. In the future we would like to adopt the nontrivial methods for model enumeration discussed in [8] to our settings. The theory developed in this paper provides a foundation to extend the recent constraint answer set programming solver EZSMT [21] to accept non-tight constraint answer set programs. The contributions of this work also open a door to the development of a novel constraint-based method in processing logic programs

---

[9] `https://research.ics.aalto.fi/software/asp/lp2normal/`

by producing intermediate output in MINIZINC [18] in place of SMT-LIB. We believe our work will boost the cross-fertilization between the three areas: SMT, constraint answer set programming, and constraint programming.

References

1   Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

2   Clark Barrett and Cesare Tinelli. Satisfiability Modulo Theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014.

3   Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

4   Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 242–254, 2001.

5   François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

6   Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

7   Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in CLASP Series 3. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 2015.

8   Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven Answer Set Enumeration. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR'07, pages 136–148, Berlin, Heidelberg, 2007. Springer-Verlag. URL: `http://dl.acm.org/citation.cfm?id=1758481.1758496`.

9   Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability Solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

10   Tomi Janhunen, Ilkka Niemelä, and Mark Sevalnev. Computing Stable Models via Reductions to Difference Logic. In *Logic Programming and Nonmonotonic Reasoning*, pages 142–154. Springer Berlin Heidelberg, 2009.

11   Yuliya Lierler. *SAT-based Answer Set Programming*. PhD thesis, University of Texas at Austin, 2010.

12   Yuliya Lierler. What is answer set programming to propositional satisfiability. *Constraints*, pages 1–31, 2016. `doi:10.1007/s10601-016-9257-7`.

13   Yuliya Lierler and Benjamin Susman. On relation between constraint answer set programming and satisfiability modulo theories. *Theory and Practice of Logic Programming*, 17(4):559–590, 2017.

14   Yuliya Lierler and Miroslaw Truszczyński. Transition Systems for Model Generators — A Unifying Approach. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP) Special Issue*, 11(4-5):629–646, 2011.

15   Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

16   Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157:115–137, 2004.

17   Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

18 N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, , and G. Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, page 529–543, 2007.

19 Ilkka Niemela. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53:313–329, 2008.

20 Ilkka Niemelä and Patrik Simons. Extending the Smodels System with Cardinality and Weight Constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

21 Benjamin Susman and Yuliya Lierler. SMT-Based Constraint Answer Set Solver EZSMT (System Description). In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52, pages 1:1–1:15, 2016.

Learning Commonsense Knowledge Through Interactive Dialogue

Benjamin Wu
Imperial College London
Department of Computing, Imperial College London, SW7 2AZ, UK
benjamin.wu16@imperial.ac.uk

Alessandra Russo
Imperial College London
Department of Computing, Imperial College London, SW7 2AZ, UK
a.russo@imperial.ac.uk

Mark Law
Imperial College London
Department of Computing, Imperial College London, SW7 2AZ, UK
mark.law09@imperial.ac.uk

Katsumi Inoue
National Institute of Informatics
2 Chome-1-2 Hitotsubashi, Chiyoda, Tokyo
inoue@nii.ac.jp

---- **Abstract** ----

One of the most difficult problems in Artificial Intelligence is related to acquiring commonsense knowledge – to create a collection of facts and information that an ordinary person should know. In this work, we present a system that, from a limited background knowledge, is able to learn to form simple concepts through interactive dialogue with a user. We approach the problem using a syntactic parser, along with a mechanism to check for synonymy, to translate sentences into logical formulas represented in Event Calculus using Answer Set Programming (ASP). Reasoning and learning tasks are then automatically generated for the translated text, with learning being initiated through question and answering. The system is capable of learning with no contextual knowledge prior to the dialogue. The system has been evaluated on stories inspired by the Facebook's bAbI's question-answering tasks, and through appropriate question and answering is able to respond accurately to these dialogues.

1 Introduction

Learning commonsense knowledge is one of the major long-term goals in the research of Artificial Intelligence [6]. In recent years, there have been major developments in the area of Natural Language Processing, particularly in the automation of linguistic structure analysis [10], however the challenge of disambiguation and learning commonsense still remains [26]. Consider the sentence, "I pulled the pin out of the apple and there was a hole in it." Immediately we understand that the "it" in the sentence is referring to the apple and not

the pin. Consider another example, "I saw a bird flying with binoculars." This could be interpreted in two different ways: either I used the binoculars to see the bird; or the bird was in possession of the binoculars while flying. These are examples of some of the current difficulties and challenges faced by computers in text comprehension and is what motivates our research.

Research into the problem of Commonsense reasoning is divided mainly into knowledge-based approaches and statistical machine learning approaches that use large corpora of data. Some of these approaches include: Word Sense Disambiguation using Topic Models [4]; Learning Common Sense through Visual Abstraction [28]; Representation Learning for Predicting Commonsense Ontologies [19]; and Commonsense Knowledge Base Completion [18] amongst others. For humans, learning commonsense through dialogue is a very natural thing to do. In fact, students who are placed in a learning environment may very well need to interact with a teacher or some learning facilitator in order to receive feedback and guidance [5]. Just as it is natural for humans to learn through dialogue, this paper presents a system that is able to automatically acquire common-sense knowledge through dialogic interaction.

There are two main aspects to the system: the knowledge representation, reasoning and inductive learning; and understanding and conversion from text to logic. For the first aspect, we use Answer Set Programming (ASP) in combination with a suitable form of Event Calculus [14] to represent the knowledge, along with ILASP [15] [16] [17] and clingo [8] as our main systems for reasoning and learning commonsense knowledge; and for the second, we use spaCy [11] as the syntactic parser along with WordNet [23] to help with checking synonymy for the translation from text to logic. The project uses stories as inspired by those from the Facebook's bAbI dataset [29] to see if the system can understand simple sentences with little ambiguity and develop the system's ability to gradually learn from and form valid hypotheses about such stories.

In Section 2 of this paper we will first discuss some background knowledge required to understand the tools used in our system. Following this in Section 3 we overview and explain the approach taken towards solving the problem at hand. Section 4 then discusses how we have evaluated the system and its capabilities and Section 5 touches briefly upon other related works that have been done on machine learning from dialogues. The paper is then concluded in Section 6 with remarks about potential areas for future work followed by the Appendix which contains examples of interactive dialogues between a user and our system.

2   Background

2.1   Answer Set Programs

Answer Set Programming (ASP) is a form of declarative programming directed at complex search problems [20]. ASP is based on stable model semantics of logic programming [9] and the search problems in ASP are reduced to computing such stable models using solvers that perform such search tasks. For the purposes of this paper, we will assume the following subset of the ASP language:

A *literal* can be either an atom p or its *default negation*, $not\,p$. A *normal rule* is of the form $h \leftarrow b_1, ..., b_n, not\,c_1, ..., not\,c_m$ where h is the *head* and $b_1, ..., b_n, not\,c_1, ..., not\,c_m$ collectively is the *body* of the rule, where b_i and c_j are *atoms*. A *constraint* is a rule with an empty head and is of the form $\leftarrow b_1, ..., b_n, not\,c_1, ..., not\,c_m$. An expression of the form $l\{h_1, ..., h_k\}u$ is called an *aggregate*, where h_i are atoms for $1 \leq i \leq k$, and l and u are integers such that $0 \leq l \leq u \leq k$. A variable V that occurs in a rule R is considered *safe* if it occurs in at least one positive body literal of R.

An Answer Set Program P is a finite set of normal rules and constraints. Given an ASP program P, the Herbrand Base of P, denoted as HB_P, is the set of all ground atoms that can be formed from the predicates and constants that appear in P. When P includes only normal rules, a set $A \subseteq HB_P$ is an Answer Set of P iff it is the minimal model of the reduct P^A. The reduct P^A is constructed from the grounding of P by removing any rule whose body contains a literal $not\, c_i$ where $c_i \in A$, and removing any negative literals in the remaining rules. Given an Answer Set Program P, we denote the set of all Answer Sets of P as $AS(P)$. A *partial interpretation* e is a pair $e = \langle e^{inc}, e^{exc} \rangle$ of sets of ground atoms, called the *inclusions* and *exclusions* respectively. An Answer Set A *extends* $e = \langle e^{inc}, e^{exc} \rangle$ if and only if $(e^{inc} \subseteq A) \wedge (e^{exc} \cap A = \emptyset)$.

2.2 ILASP

ILASP [15] is an ILP algorithm targeted at learning answer set programs. In this paper, we consider a simplification of ILASP's full learning framework presented in [17], called *Context-Dependent Learning from Answer Sets*. In this framework, examples are context-dependent partial interpretations, which each consist of a partial interpretation and an ASP program called the *context* of the example. Contexts allow the expression of background concepts that only apply to specific examples. They can also be used to further structure the background knowledge in an example-specific manner, thus bringing about improvements in the performance of the learning algorithm [17].

▶ **Definition 1.** A context-dependent partial interpretation is a tuple $\langle e, C \rangle$, where $e = \langle e^{inc}, e^{exc} \rangle$ is a partial interpretation and C is an ASP program called the *context*.

The hypothesis space is defined by $M = \langle M_h, M_b \rangle$ called the *language bias* of the task which is made up of a set of *head mode declarations* (M_h) and a set of *body mode declarations* (M_b). A rule $h \leftarrow b_1, ..., b_n, not\, c_1, ..., not\, c_m$ is contained within the search space S_M if and only if it satisfies the following:

1. The head is empty; or h is an atom compatible with a mode declaration in M_h.
2. The atoms b_i and c_j are all compatible with mode declarations in M_b, $\forall i \in [1, n]$ and $\forall j \in [1, m]$.
3. All variables in the rule are safe.

Each rule R in S_M is given a unique identifier R_{id}.

▶ **Example 2.** Consider the mode declarations $M = \langle M_h, M_b \rangle$ with $M_h = \{is\_in(v, v), is\_holding(v, v)\}$ and $M_b = \{went\_to(v, v), picked\_up(v, v)\}$, and where v denotes that the arguments of the predicates are variables. Some of the possible rules that are contained in the hypothesis space S_M include:

$$is\_in(V0, V1) \leftarrow went\_to(V0, V1)$$
$$is\_holding(V0, V1) \leftarrow picked\_up(V0, V1)$$
$$\leftarrow went\_to(V0, V0)$$

Examples of rules that are not in S_M are:

$$is\_in(V0, V1) \leftarrow is\_holding(V0, V1)$$
$$went\_to(V0, V1) \leftarrow picked\_up(V0, V1)$$

▶ **Definition 3.** A *Context-Dependent Learning from Answer Sets* task is a tuple $T = \langle B, S_M, E^+, E^- \rangle$ where B is the background knowledge, S_M is the hypothesis space defined by a language bias M, E^+ is a set of context-dependent partial interpretations, called the *positive examples* and E^- is a set of context-dependent partial interpretations called the *negative examples*. An hypothesis H is an *inductive solution* of T if and only if:

1. $H \subseteq S_M$
2. $\forall \langle e, C \rangle \in E^+ \quad \exists A \in AS(B \cup H \cup C)$ *such that* A *extends* e
3. $\forall \langle e, C \rangle \in E^- \quad \nexists A \in AS(B \cup H \cup C)$ *such that* A *extends* e

Such a solution is written as $H \in ILP_{LAS}^{context}(T)$.

▶ **Example 4.** An example of a Context-Dependent Learning from Answer Sets ($ILP_{LAS}^{context}$) task can be represented in the following manner:

```
% Background Knowledge
picked_up(john,football).

% Context dependent partial interpretations
% positive examples have the form:
% #pos(id, inclusions, exclusions, Context).
#pos(p1, { is_in(mary,garden), is_holding(john,football) },
         { is_in(john,garden) },
         { went_to(mary,garden). } ).

#pos(p2, { is_in(john,garden), is_holding(john,football) },
         { is_holding(mary,football) },
         { went_to(john,garden). } ).

% Mode declarations
#modeh(is_in(var(person),var(location))).
#modeh(is_holding(var(person),var(object))).
#modeb(1, went_to(var(person),var(location))).
#modeb(1, picked_up(var(person),var(object))).
```

In this example, the Context-Dependent Learning from Answer Sets task includes a single fact in the Background Knowledge and two positive examples with no negative examples. The mode head (#modeh) and mode body (#modeb) declarations here generate the same hypothesis space as discussed in Example 2. Here both the positive examples share the background knowledge that John picked up the football, however the contexts of each example differs in that Mary went to the garden in the first positive example and John in the second. The above learning task would produce as optimal solution the following hypothesis, H:

```
is_in(V0,V1) :- went_to(V0,V1).
is_holding(V0,V1) :- picked_up(V0,V1).
```

The Answer Set produced by $AS(B \cup H \cup C_1)$ extends the first example and $AS(B \cup H \cup C_2)$ extends the second positive example. This illustrates how Context-Dependent Learning by Answer Sets allows for consistent hypotheses to be learned even in the presence of conflicting facts from different contexts. If both contexts were added directly to the background knowledge, the learning task would have no solution for the given examples.

2.3 Event Calculus

With the comprehension of stories involving agents and their actions, there needs to be some logic-based formalism to represent actions and effects of actions. Such formalisms include *event calculus* [14] and *situation calculus* [22] among others. Even among the formalisms of Event Calculus there exists multiple variants but for the purposes of this paper, we will look at a particular variant developed for the use in Inductive Logic Programming from [13].

For our use of Event Calculus in this paper, we introduce the following predicates.

| initiatedAt(F,T) | terminatedAt(F,T) | holdsAt(F,T) | happensAt(E,T) |

The variable F represents a fluent, E represents an event and T represents a time point. Along with these predicates we add the following axioms.

```
holdsAt(F,T+1) :- initiatedAt(F,T).
holdsAt(F,T+1) :- holdsAt(F,T), not terminatedAt(F,T).
```

These axioms basically mean that if a fluent is initiated at a time point, then the fluent will hold at the next time point, and if a fluent holds at a particular time point and is not terminated at that time, it will continue to hold.

3 Learning Commonsense Knowledge

In this section, we present our system for learning commonsense rules by interacting with a user through simple dialogue. The system starts with a very limited background knowledge that includes only the domain-independent axioms of Event Calculus given in the previous section. The user inputs a series of sentences and responses through the keyboard to the system in a simulated conversation. Through a mixture of story telling and question-answering, the system remembers facts about the narrative being told by the user and learns to form rules and relations that are consistent with the responses given by the user about the questions that have been asked.

An illustration of the overall structure and pipeline for the system can be seen in Figure 1. There are two main components to this system: the translation module and the reasoning module, which are described in detail in the following sections. The system was coded in python 2.7 using spaCy [11] as the syntactic parser, WordNet [23] for the synonym database, clingo5 [8] as the ASP solver, and ILASP [17] for the learning tasks.

3.1 Translation

Each sentence that passes through the translation module is parsed and its dependency tree is generated. Each word is tokenised and tagged with their part-of-speech (POS) tag, their dependency tag and their parent node in the tree.

After a sentence is parsed we need to generate the predicate symbol for the sentence. Firstly, the "ROOT" of the sentence is taken and put through WordNet to find the closest general synonym. We then append any adpositions (ADP), adjectives (ADJ) or auxiliary verbs (aux / VERB) to the root verb to complete the predicate symbol. Then the sentence is scanned through for nouns (NOUN), pronouns (PRON) and proper nouns (PROPN). The lemmas of the n-number of nouns are then added to the predicate symbol to form an n-ary predicate. This is illustrated in Figure 2. For the rest of this paper we will refer to this translated predicate as the logical representation of the sentence.
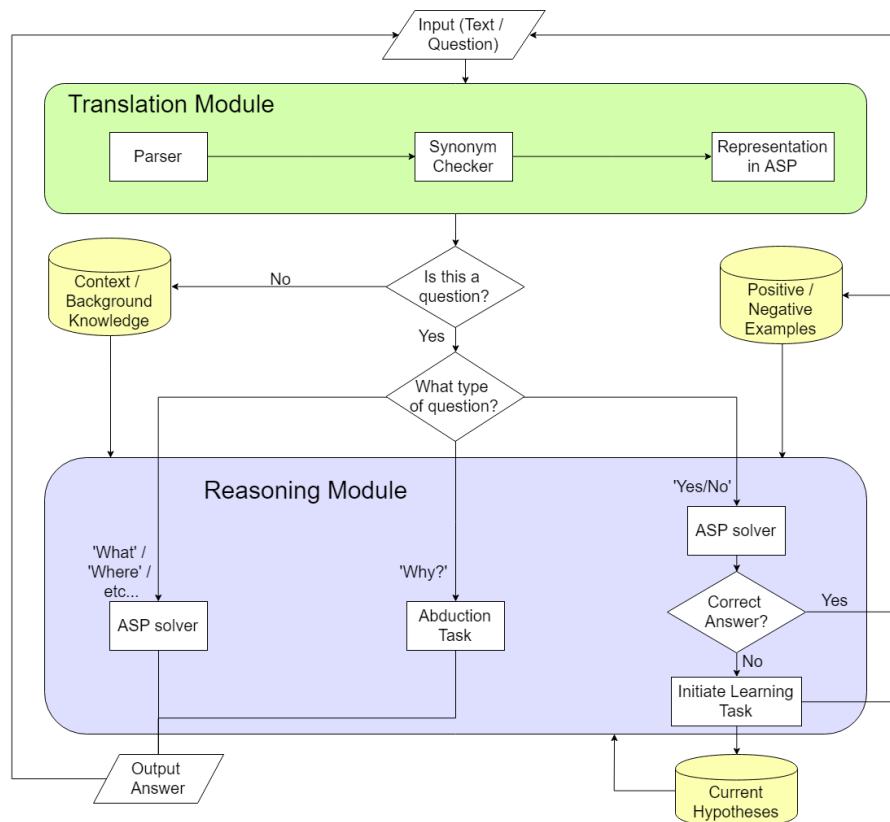
■ **Figure 1** The general work pipeline of the system.

To complete the conversion of the sentence into its representation in ASP for the system, the logical representation of the sentence is wrapped with the appropriate predicate of Event Calculus. The system has an internal counter that begins at "1" for each story and increments after each sentence/question that is input by the user. This counter is used to determine the time stamp for each predicate of Event Calculus while the logical representations form the events/fluents. With the exception of sentences that contain the root verb translating to "be" and sentences that contain negation, all facts are wrapped with the "happensAt" predicate. These sentences are all treated as events that happen at that particular time stamp. For sentences with the root verb "be", these are wrapped with the "holdsAt" predicate as they are concerned with the state something is in rather than a particular event.

Some additional rules for translation have been added to deal with sentences that include negation, conjunctions, disjunctions and coreferencing. These rules have been constructed to be general in nature however they are only applicable to sentences of relatively simple structures. If a negation modifier (neg) is found linked directly to the root verb, then the resulting logical representation is wrapped in the "terminatedAt" predicate for Event Calculus. This stops the fluent from holding from that time point onwards, regardless of if it were true or not beforehand. For conjunctions and disjunctions, the noun that is tagged as the conjunct (conj) is stored and a predicate is formed whilst disregarding the conjunct. A second predicate is then formed by substituting in the conjunct in the place of the noun it is linked to. In the case of conjunctions (sentences involving "and") the extra predicate is added on as an additional fact. In the case of disjunctions (sentences involving "or") the two predicates are combined into an aggregate where only one may be chosen, thus creating an exclusive or.
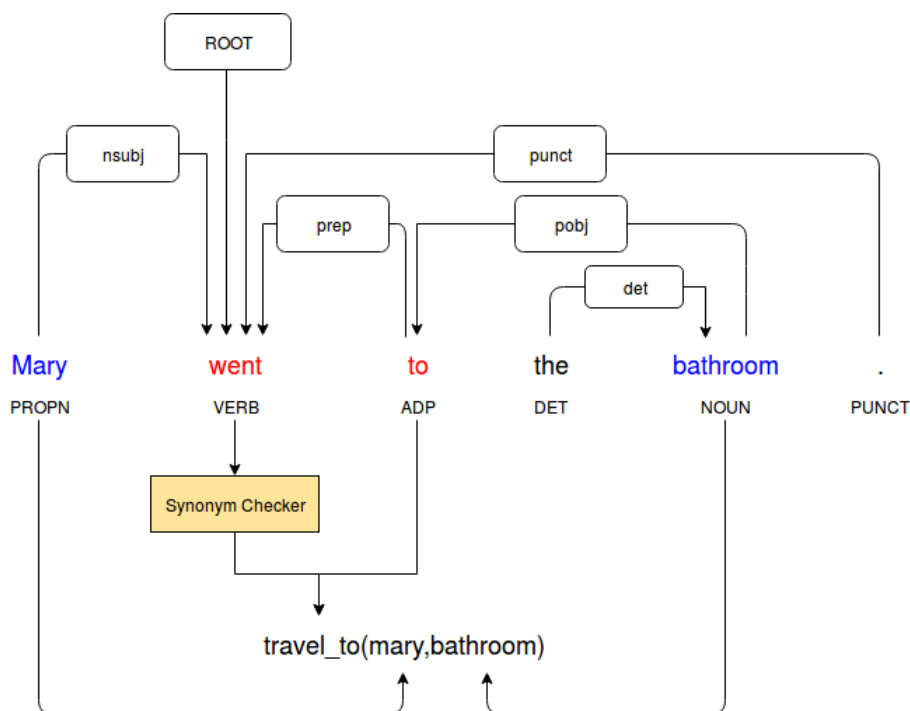
Figure 2 Illustration of how a sentence is generally translated into its logical representation.

For all inputs that are not questions, the sentences are translated, stored into the context of the story, then the system waits for the next input. When the translation module notices that the input sentence is a question, the reasoning module will be called.

3.2 Reasoning

The reasoning module deals with a variety of types of questions, however the most important type of question for this system are "yes/no" questions. Questions that ask "who", "what", "where" and "why" will only generate a response that the system can give using its current knowledge, whereas asking "yes/no" questions may initiate learning tasks depending on the user's feedback.

"Yes/No" Questions

All of the learning that happens in the system is through the interactions that result from "yes/no" questions. All our reasoning tasks are run using clingo5 with the files created dynamically during the conversations. The system begins with only the axioms of Event Calculus as described from Section 2.3 except with an added predicate for time.

```
holdsAt(F,T+1) :- initiatesAt(F,T), time(T).
holdsAt(F,T+1) :- holdsAt(F,T), not terminatedAt(F,T), time(T).
```

The type predicate for time is used here to limit the relevant grounding of the reasoning tasks. Without specifying this type predicate, our ASP solver will continue to generate "holdsAt" predicates indefinitely.

All the sentences that are not questions pass through the translation module and are stored directly into the *context* of the story without interacting with the reasoning module. When a "yes/no" question is asked, the background knowldge axioms, the context up to that time and the current hypothesis is written into a ".lp" file along with the translated query as the goal. The ASP solver will try to satisfy the goal using the given context with the current hypothesis. If the result of the reasoning task comes back as "UNSATISFIABLE" then the system will respond "No" to the question; otherwise it will respond "Yes" or "Maybe" in some exceptions. The user will then tell the system if its conclusion was correct or not. After the user responds, the query is stored as a positive example if it is supposed to true, a negative example if it is supposed to be false, and if the system is told that it was incorrect a learning task will be initiated.

In the case where a disjunction was present in the context, a second reasoning task is run alongside the original except with the negation of the query as the goal. If the results of running both reasoning tasks returns "SATISFIABLE" (i.e. the query can be considered both true or false with respect to the context), then the system responds with "Maybe". If the response of "Maybe" is deemed correct by the user, then the query is not stored in the examples, however if "Maybe" is incorrect, then the query is added to the negative examples. For example, if in the story we have "John went to either the hallway or the garden." then we ask "Is John in the hallway?", two reasoning tasks will be run trying to satisfy John being in the hallway and John not being in the hallway. In this case, both cases are satisfiable and therefore the system would respond with "Maybe".

Learning

When a learning task is triggered, the system will construct an $ILP_{LAS}^{context}$ task that contains the background knowledge, bias constraints, the context-dependent partial interpretations and the mode declarations. We use ILASP here to run the learning task from the generated file. The background knowledge added to the task is the axioms of Event Calculus as it is for the reasoning tasks. The context-dependent partial interpretations are generated by using the positive and negative examples that were stored from the "yes/no" questions in the dialogue, along with the context of the story thus far.

The mode declarations are automatically generated for each learning task. To generate the mode declarations, for each logical representation that is seen throughout the story, the original nouns in the arguments are replaced with variable types. With the current implementation, variable types are given as inputs by the user; when the system encounters a noun that it has not yet seen, it will prompt the user to enter the noun's variable type. For each type of predicate that occurs in the context, a corresponding mode body declaration is created. For each type of logical representation that occurs as an example, additional mode head declarations are made using both "initiatedAt" and "terminatedAt" wrappers, along with a mode body declaration using the "holdsAt" wrapper if it is not already present.

Additional bias constraints are used to help decrease the size of the hypothesis search space by restricting ILASP to only searching for hypotheses about a single time point rather than multiple. As all of the hypotheses that we are currently aiming to learn consist of rules that are each triggered by events that happen at one specific time point, this constraint does not negatively impact our system's ability to learn.

The $ILP_{LAS}^{context}$ task is solved with the maximum number of variables of possible hypotheses set to three. The resulting hypothesis is stored and added to subsequent reasoning tasks as the current hypothesis. The current hypothesis is continually overwritten with each

Table 1 List of special inputs and their functions.

Input	Description
end	ends the current session and exits the program
new story	starts a new story with empty context
save hypothesis	stores the current hypothesis into the Background Knowledge
check hypothesis	prints the stored and current hypothesis

learning task. If ILASP is unable to find a suitable hypothesis, the maximum number of variables is incremented and the task is run again. The system halts if it still fails to find an inductive solution with the maximum number of variables set to five.

Other Questions

With questions starting with "What", "Where" or "Who", a slightly different reasoning task to the one generated for "yes/no" questions is created. We create a file for the reasoning task with the background knowledge axioms, the context of the story and the current hypothesis, however we do not add in a goal to this ASP program. Rather than add the translation of the query as the goal, we generate a pattern from the translation and see what positive atoms match this pattern in the resulting answer set. From those atoms we extract the variables that have been matched and output them to the screen. For questions that start with 'How many', the same task as what has just been described is generated but rather than return the strings that result from pattern matching, it returns the number of items matched.

For questions that ask "Why", a different type of reasoning task is generated. The background knowledge and learned hypothesis are written into the ".lp" file along with the translated query as the goal, and the facts from the context of the story are written into an aggregate. For each fact in the aggregate, a weight is applied so that when the ASP program is then run, only the minimum number of facts from the story will be chosen to make the goal "SATISFIABLE". This in essence is an abductive task where we look for the causal relationship between what events have happened and how the goal has been reached. This differs to the type of "why" questions asked in Facebook bAbI's task 20 where it asks for the motivation of the agent in question, which is outside the scope of the story.

3.3 Special inputs

This system runs in a constant loop where it will keep waiting for the user's next input. Whenever a new sentence is expected by the system, there are a few specific inputs that are recognised by the system as different function calls. These special inputs are described in Table 1.

One thing to note is that when "new story" is called, the context and examples up until that point all get stored as a context-dependent partial interpretation which is still used for subsequent learning tasks so that what has been learned previously is not forgotten, however the reasoning tasks will not be affected by facts from previous stories. By checking and saving the hypothesis, the user can also choose to keep rules that they consider to be desirable. The "save hypothesis" function also clears all mode head declarations up to that point; this aids the systems scalability and helps when learning more difficult concepts.

4   Evaluation

The system was tested with various stories that draw inspiration from the themes of those seen in Facebook bAbI's question-answering data set [29]. Many of these stories are to do with a number of people moving to different locations and then asking about their whereabouts or about objects that they are or were carrying. By the nature of our system, our results are more qualitative than quantitative, and so it is easier for us to demonstrate the capabilities of the system via an example.

▶ **Example 5.** For this example, consider Listings 1, 2 and 3 (the Listings are found in the Appendix). This story is inspired by tasks 1, 2, 6, 7, 8, 11 and 12 of Facebook bAbI's QA dataset.

In Listing 1 it can be seen how initially when the system is asked about the location of "Mary" from the story, it cannot answer correctly. It then forms a concept to initiate the state of Mary being in the location due to travelling there. However with what it had first learned, it did not understand that if Mary moved to a different location, she would no longer be in the previous location. This is then corrected after making its second mistake, which results in learning the correct hypothesis, which is displayed after the input 'check hypothesis'.

Following the story from Listing 1 we look at the dialogue from Listing 2. Here we can see that initially when asked about the items that Mary is carrying, the system responds incorrectly as it has not learned anything about the concept of "carrying" yet. So then the concept of carrying is then taught to the system as a result of picking up or dropping objects through additional questioning and the system is then able to answer the questions correctly. By further interactions with the system through dialogue, the system is able to learn more interesting and complex concepts, such as what is displayed in Listing 3. Here the system is able to answer questions that require two supporting facts, this being equivalent to questions from task 2 of the bAbI dataset. More specifically, in Listing 3, it has learned that objects will be in the locations they are picked up in or will move to new locations with the person who is carrying them, and that they will no longer be in previous locations if the person carrying them has moved.

Using Facebook bAbI's QA dataset as a means for comparison, the presented system is able to learn concepts that are able to deal with dialogues that are equivalent to eleven of the bAbI tasks. The tasks that can be solved are shown in Table 2. However due to some limitations in our system with translation, some of these tasks need to be slightly adapted for our system. For questions such as "Where is the apple?", our system needs the question to be phrased using the same language as what was used to teach it. Since the concept determining the location of an object or person would have been translated as something "being in" a location, we would need to change the question to "Where is the apple in?" for the system to be able to correctly understand and answer the question. Another translation limitation that has been found during various tests is that some names are not always recognised as proper nouns by the spaCy parser. Names such as Emily and Will are sometimes tagged as adverbs or verbs by the parser. To deal with this problem, it is sufficient to replace these names in the tasks with other less ambiguous names such as Emma and Brian. With these minor changes to the eleven bAbI tasks, after appropriate questioning and answering, this system is able to learn to solve them completely.

The number of questions required to learn each concept correctly is also difficult to quantify as it can vary greatly depending on the complexity of the concept, the context of the story, the types of questions you ask and the order in which you ask them. Some of these

■ **Table 2** The Facebook bAbI question-answering tasks that are able to be solved by the system.

Task	Description	Examples of sentences that feature in the task
1	single supporting fact	Where is Mary? (see Listing 1)
2	two supporting facts	Where is the apple? (see Listing 3)
6	yes/no questions	Is Mary in the kitchen?
7	counting	How many objects is Mary carrying?
8	lists / sets	What is Mary carrying?
9	simple negation	Sandra is no longer in the bedroom.
10	indefinite knowledge	John is either in the office or the bathroom.
11	basic coreference	Then he moved to the hallway.
12	conjunctions	Mary and Sandra journeyed to the garden.
13	compound coreference	After that they went back to the kitchen.
15	basic deduction	What is gertrude afraid of? (see Listing 4)

concepts can be learned in as few as two or three questions as can be seen from Listings 1 and 2, however the same concepts can also take much longer to learn if not questioned appropriately. To minimise the number of questions required to learn a concept, questions should be directed towards the gaps of any incomplete learned hypothesis. Since learning is only initiated when a "yes/no" question is answered incorrectly, actively trying to increase the number of mistakes the system makes will allow the learning to progress much faster.

We have yet to implement a way for dialogue with the system to be automated and allow for quantitative analyses to be generated. With the current interactive approach, to do a quantitative analysis over a large dataset is impractical. Due to the size of these data sets, it is likely that many examples will be covered by the same hypotheses. Since ILASP2i [16] iteratively computes a subset of the examples which are relevant to the search, the size of which generally being much smaller than the entire set of examples, we may be able to take advantage of this capability when automating the process to scale on entire bAbI datasets. This could potentially scale better compared to other batch learning systems, however this has yet to be tested.

Limitations

For the majority of the other tasks in the bAbI dataset, the reason why the tasks are unable to be completed is because of the limitations of the translation module. Some of these troubles are to do with inconsistent parsing of sentences with similar structure and some are to do with the challenges in representing sentences with multiple arguments. For instance, take the sentences "Fred gave John the football.", "Who gave the football to John?" and "Who did Fred give the football to?". By using the current method of translation, the logical representation of the first sentence would be "give(Fred, John, football)", and the two questions would ideally translate to "give(?, John, football)" and "give(Fred, ?, football)". However, rules that determine the order in which the multiple arguments are put into the predicate, and also the position of the missing argument that needs to be found, are very hard to generalise without programming it for a specific sentence structure. Currently the system is unable to translate these types of sentences well enough.

Another challenge this system faces is a problem with scalability. As stories get more involved and new types of predicates get introduced, the hypothesis space gets exponentially larger and can expand to the point where learning takes too long to be considered reasonable

for dialogue. Although this problem arises from our choice of automating the generation of mode declarations, we believe that this does not outweigh the benefits we gain from being able to generalise with no prior knowledge of the contexts. An example of encountering scalability problems can be seen when trying to solve task 2 (two supporting facts) of the bAbI dataset. This task is currently solvable by using the "save hypothesis" functionality that can be used to clear the old mode head declarations. By being a bit more strategic with the order in which you ask questions and help the system learn, concepts that otherwise would be too difficult for it to learn in one go can be broken down into manageable steps for incremental learning.

5   Related Work

To our knowledge, the problem of learning commonsense knowledge through a dialogic interaction is a novel task. However there has been significant research done on solving Facebook bAbI's question-answering tasks.

Mitra and Baral [24] developed a system to solve the toy tasks from Facebook's bAbI dataset. In their work, they describe an agent architecture that works with a formal reasoning model together with a statistical inference based model in tandem, to face the task of question answering. There are three layers to their implementation: the Statistical Inference Layer, the Formal Reasoning Layer and the Translation Layer. The Statistical Inference Layer contains the statistical NLP models which uses an Abstract Meaning Representation (AMR) Parser [1] [7]. The Formal Reasoning Layer uses a modified version of the ILP algorithm XHAIL [25] to learn the knowledge for reasoning in ASP. The Translation Layer encodes the sentences from the text into the syntax of Event Calculus with the help of the AMR Parser. This layer enables the communication between the two aforementioned layers and allows information to be passed from one to the other. Their system achieved a mean accuracy of 99.68% over the entire bAbI dataset and shows that with the addition of a formal reasoning layer, the reasoning capability of an agent increases significantly. With this approach, mode declarations had to be manually defined for each task, and some tasks had hypotheses learned from previous tasks added to the background knowledge of other more complex tasks. This differs to our approach as mode declarations are automatically defined during the dialogue and we do not augment our background knowledge.

An approach to the problem of machine comprehension of text has recently been developed by Chabierski [3]. The approach here utilises Combinatory Categorial Grammar [27] and Montague-style semantics [12] to perform a semantic analysis of text to derive Answer Set Program representations expressed in the form of λ-ASP calculus [2]. These representations are used to automatically derive the mode declarations for the generation of ILP tasks to be computed by the ILASP algorithm. To evaluate the performance of this approach, the system was tested using a subset of the bAbI question-answering tasks. Using only 25 training examples for each task, the system is able to fully solve six of the twenty QA tasks from the bAbI dataset, namely tasks 1, 6, 8, 9, 12 and 15. This approach also automatically derives its mode declarations from the task context and background knowledge, however the background knowledge is manually added for each task to increase the capability of the learner.

6 Conclusion and future work

What distinguishes our presented work from others is the interactive nature of our system and its ability to be able to learn from dialogues without having any prior knowledge about the contexts of their stories. More specifically, our system has the ability to automatically define the mode declarations for each task without the need to manually augment the system's background knowledge. This allows our system to be more flexible and able to understand contexts outside the scope of stories in the bAbI tasks. Although interactions with the system are limited to relatively simple sentences, these sentences are not confined to a preset type of contexts. Another strength of our system is that it is able to learn concepts with relatively few questions; it does not require large datasets for it to be trained on.

To expand the scope of what the system is able to understand and learn from, the translation module needs to be improved. It is very hard to define general rules that would allow for accurate translations of all the complexities in the English language, therefore it would probably be better to use statistical methods of machine learning to train the translation module instead. Many measures have already been taken to improve the scalability of the system but further improvements should be looked into as well. Allowing the user to define the variable types is one such measure taken to improve the scalability. This is not a very elegant implementation and ideally the system would be able to accurately define the variable types itself in a way which is not too specific to the point where the system is unable to generalise, yet not too general so that the scalability is affected. It may be possible to introduce some way of using ontologies or by using other relational databases such as ConceptNet [21] to map these variables to similar types.

References

1   Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Herm-jakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. Abstract Meaning Representation for Sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria, August 2013. Association for Computational Linguistics. URL: `http://www.aclweb.org/anthology/W13-2322`.

2   Chita Baral, Juraj Dzifcak, and Tran Cao Son. Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI'08, pages 818–823. AAAI Press, 2008. URL: `http://dl.acm.org/citation.cfm?id=1620163.1620199`.

3   Piotr Chabierski, Alessandra Russo, Mark Law, and Krysia Broda. Machine Comprehension of Text Using Combinatory Categorial Grammar and Answer Set Programs. In *COMMONSENSE*, 2017.

4   Devendra Singh Chaplot and Ruslan Salakhutdinov. Knowledge-based Word Sense Disambiguation using Topic Models. *CoRR*, abs/1801.01900, 2018. `arXiv:1801.01900`.

5   John Cook. Dialogue in Learning: Implications for the Design of Computer-based Educational Systems. In *ICCE*, pages 987–991, 2002. `doi:10.1109/CIE.2002.1186131`.

6   Ernest Davis and Gary Marcus. Commonsense reasoning and commonsense knowledge in artificial intelligence. *Communications of the ACM*, 58(9):92–103, September 2015. `doi:10.1145/2701413`.

7   Jeffrey Flanigan, Sam Thomson, Jaime Carbonell, Chris Dyer, and Noah A. Smith. A Discriminative Graph-Based Parser for the Abstract Meaning Representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1:*

Long Papers), pages 1426–1436, Baltimore, Maryland, June 2014. Association for Computational Linguistics. URL: `http://www.aclweb.org/anthology/P14-1134`.

8   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 14 (4-5), 2014. Theory and Practice of Logic Programming, Online Supplement. `arXiv:1405.3694v1`.

9   M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP '88*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.

10 Julia Hirschberg and Christopher D. Manning. Advances in natural language processing. *Science*, 349(6245):261–266, 2015. `doi:10.1126/science.aaa8685`.

11 Matthew Honnibal and Mark Johnson. An Improved Non-monotonic Transition System for Dependency Parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1373–1378, Lisbon, Portugal, September 2015. Association for Computational Linguistics. URL: `https://aclweb.org/anthology/D/D15/D15-1162`.

12 Theo M. V. Janssen. Montague Semantics. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2017 edition, 2017.

13 Nikos Katzouris, Alexander Artikis, and Georgios Paliouras. Incremental learning of event definitions with Inductive Logic Programming. *Machine Learning*, 100(2-3):555–585, 2015. `doi:10.1007/s10994-015-5512-1`.

14 Robert Kowalski and Marek Sergot. A Logic-Based Calculus of Events. *New Generation Computing*, 4:67–95, 1986.

15 Mark Law, Alessandra Russo, and Krysia Broda. Inductive Learning of Answer Set Programs. In Eduardo Fermé and João Leite, editors, *JELIA*, volume 8761 of *Lecture Notes in Computer Science*, pages 311–325. Springer, 2014. `doi:10.1007/978-3-319-11558-0_22`.

16 Mark Law, Alessandra Russo, and Krysia Broda. Learning weak constraints in answer set programming. *TPLP*, 15(4-5):511–525, 2015. `doi:10.1017/S1471068415000198`.

17 Mark Law, Alessandra Russo, and Krysia Broda. Iterative Learning of Answer Set Programs from Context Dependent Examples. *TPLP*, 16(5-6):834–848, 2016. `doi:10.1017/S1471068416000351`.

18 Xiang Li, Aynaz Taheri, Lifu Tu, and Kevin Gimpel. Commonsense Knowledge Base Completion. In *ACL (1)*. The Association for Computer Linguistics, 2016. URL: `http://aclweb.org/anthology/P/P16/P16-1137.pdf`.

19 Xiang Li, Luke Vilnis, and Andrew McCallum. Improved Representation Learning for Predicting Commonsense Ontologies. *CoRR*, abs/1708.00549, 2017. `arXiv:1708.00549`.

20 Vladimir Lifschitz. What is answer set programming? In *23rd AAAI Press Conference on Artificial Intelligence (AAAI 2008)*, pages 1594–1597, Menlo Park, CA, USA, 2008. AAAI Press. URL: `http://www.cs.utexas.edu/~vl/papers/wiasp.pdf`.

21 H. Liu and P. Singh. ConceptNet &Mdash; A Practical Commonsense Reasoning Tool-Kit. *BT Technology Journal*, 22(4):211–226, October 2004. `doi:10.1023/B:BTTJ.0000047600.45421.6d`.

22 J. McCarthy and P. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. *Machine Intelligence*, 4:463–502, 1969.

23 George Miller. WordNet A Lexical Database for English. *Communications of ACM*, 38(11):39–41, 1995.

24 Arindam Mitra and Chitta Baral. Addressing a Question Answering Challenge by Combining Statistical Methods with Inductive Rule Learning and Reasoning. In Dale Schuurmans and Michael P. Wellman, editors, *AAAI*, pages 2779–2785. AAAI Press, 2016. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12345`.

25 Oliver Ray. Nonmonotonic abductive inductive learning. *Journal of Applied Logic*, 7(3), September 2009. Other identifier: 2001069. `doi:10.1016/j.jal.2008.10.007`.

26 D. Singh Chaplot and R. Salakhutdinov. Knowledge-based Word Sense Disambiguation using Topic Models. *ArXiv e-prints*, January 2018. `arXiv:1801.01900`.

27 Mark Steedman. *The Syntactic Process*. MIT Press, Cambridge, MA, USA, 2000.

28 Ramakrishna Vedantam, Xiao Lin, Tanmay Batra, C. Lawrence Zitnick, and Devi Parikh. Learning Common Sense through Visual Abstraction. In *ICCV*, pages 2542–2550. IEEE Computer Society, 2015. `doi:10.1109/ICCV.2015.292`.

29 Jason Weston, Antoine Bordes, Sumit Chopra, and Tomas Mikolov. Towards AI-Complete Question Answering: A Set of Prerequisite Toy Tasks. *CoRR*, abs/1502.05698, 2015. `arXiv:1502.05698`.

A   Demonstrations

The following listings here in the Appendix show examples of dialogue between a user and the system during a session. For clarity, the dialogues have been annotated with "U:" for what the user has input and "S:" to signify the system's responses.

Listing 1 An example of a dialogue between the system and a user showing how the concept of people being or not being in locations as a result of travelling is formed.

```
System: Tell me a story:
User: Mary went to the bathroom.
System: What is "mary" a type of? User: person
S: What is "bathroom" a type of? U: location

U: Is Mary in the bathroom?
S: No.
S: Am I right? U: No.
S: Alright, please continue.

U: Mary moved to the kitchen.
S: What is "kitchen" a type of? U: location

U: John journeyed to the kitchen.
S: What is "john" a type of? U: person

U: Is Mary in the kitchen?
S: Yes.
S: Am I right? U: Yes.
S: Alright, please continue.

U: Is Mary in the bathroom?
S: Yes.
S: Am I right? U: No.
S: Alright, please continue.

U: check hypothesis

S: initiatedAt(be_in(V0,V1),V2) :- happensAt(travel_to(V0,V1),V2).
S: terminatedAt(be_in(V0,V1),V2) :- holdsAt(be_in(V0,V1),V2),
       happensAt(travel_to(V0,V3),V2).

S: Please continue with the story:
```

```
U: Mary and John travelled to the garden.
S: What is "garden" a type of? U: location

U: Where is John in?
S: garden

U: Who is in the garden?
S: john
S: mary
```

■ **Listing 2** A continuation of the dialogue from Listing 1, this time showing the concept of carrying objects being learned.

```
User: Mary picked up the apple and the milk.
System: What is "apple" a type of? U: object
S: What is "milk" a type of? U: object

U: How many objects is Mary carrying?
S: 0

U: Is Mary carrying the apple?
S: No.
S: Am I right? U: No.
S: Alright, please continue.

U: How many objects is Mary carrying?
S: 2

U: What is Mary carrying?
S: milk
S: apple

U: Mary dropped the milk.

U: Is Mary carrying the milk?
S: Yes.
S: Am I right? U: No.
S: Alright, please continue.

U: What is Mary carrying?
S: apple

U: check hypothesis

S: initiatedAt(be_in(V0,V1),V2) :- happensAt(travel_to(V0,V1),V2).
S: terminatedAt(be_in(V0,V1),V2) :- holdsAt(be_in(V0,V1),V2),
       happensAt(travel_to(V0,V3),V2).

S: initiatedAt(be_transport(V0,V1),V2) :- happensAt(pick(V0,V1),V2).
S: terminatedAt(be_transport(V0,V1),V2) :- happensAt(drop(V0,V1),V2).

S: Please continue with the story:
```

■ **Listing 3** A demonstration of further concepts that can be learned through more questioning and answering (equivalent to task 2 - two-supporting facts from the bAbI dataset).

```
S: initiatedAt(be_in(V0,V1),V2) :- happensAt(travel_to(V0,V1),V2).
S: terminatedAt(be_in(V0,V1),V2) :- holdsAt(be_in(V0,V1),V2),
        happensAt(travel_to(V0,V3),V2).

S: initiatedAt(be_transport(V0,V1),V2) :- happensAt(pick(V0,V1),V2).
S: terminatedAt(be_transport(V0,V1),V2) :- happensAt(drop(V0,V1),V2).

S: initiatedAt(be_in(V1,V3),V2) :- holdsAt(be_transport(V0,V1),V2),
        happensAt(travel_to(V0,V3),V2).
S: initiatedAt(be_in(V1,V3),V2) :- happensAt(pick(V0,V1),V2),
        holdsAt(be_in(V0,V3),V2).
S: terminatedAt(be_in(V1,V3),V2) :- holdsAt(be_transport(V0,V1),V2),
        holdsAt(be_in(V0,V3),V2), happensAt(travel_to(V0,V4),V2).

S: Please continue with the story:
U: new story
S: Tell me a new story:
U: Mary went to the kitchen.

U: She picked up the apple and the milk.

U: John moved to the hallway.

U: He picked up the football.

U: What is in the kitchen?
S: milk
S: mary
S: apple

U: Mary dropped the apple.

U: Then she travelled to the garden.

U: Where is the apple in?
S: kitchen

U: Where is the milk in?
S: garden

U: Where is the football in?
S: hallway
```

■ **Listing 4** Running example of a dialogue inspired by bAbI task 15 that learns an extended concept.

```
System: Tell me a story:
User: Tom is a cat.
S: What is "tom" a type of? U: name
S: What is "cat" a type of? U: animal

U: Jerry is a mouse.
S: What is "jerry" a type of? U: name
S: What is "mouse" a type of? U: animal

U: Mice are afraid of cats.

U: Is Jerry afraid of cats?
S: No.
S: Am I right? U: No.
S: Alright, please continue.

U: Is Jerry afraid of Tom?
S: No.
S: Am I right? U: No.
S: Alright, please continue.

U: Is Tom afraid of mice?
S: No.
S: Am I right? U: Yes.
S: Alright, please continue.

U: Is Tom afraid of Jerry?
S: No.
S: Am I right? U: Yes.
S: Alright, please continue.

U: check hypothesis

S: initiatedAt(be_afraid_of(V3,V0),V2) :- holdsAt(be(V0,V1),V2),
        holdsAt(be_afraid_of(V3,V1),V2).
S: initiatedAt(be_afraid_of(V0,V3),V2) :- holdsAt(be(V0,V1),V2),
        holdsAt(be_afraid_of(V1,V3),V2).
```

■ **Listing 5** Continuation of the dialogue seen from Listing 4 that tests the learned hypothesis.

```
System: Please continue with the story:
User: Gertrude is a wolf.
S: What is "gertrude" a type of? U: name
S: What is "wolf" a type of? U: animal

U: Cats are afraid of wolves.

U: Is Tom afraid of wolves?
S: Yes.
S: Am I right? U: Yes.
S: Alright, please continue.

U: Is Tom afraid of Gertrude?
S: Yes.
S: Am I right? U: Yes.
S: Alright, please continue.

U: What is Gertrude afraid of?
S: Nothing

U: Mary is a sheep.
S: What is "mary" a type of? U: name
S: What is "sheep" a type of? U: animal

U: Sheep are afraid of mice.

U: Sheep are afraid of wolves.

U: What is Mary afraid of?
S: wolf
S: mouse
S: jerry
S: gertrude
```

Application of Logic-Based Methods to Machine Component Design

Bram Aerts
EAVISE

Technology Campus De Nayer, KU Leuven, Belgium

b.aerts@kuleuven.be

Joost Vennekens
EAVISE

Technology Campus De Nayer, KU Leuven, Belgium

joost.vennekens@kuleuven.be

Abstract

This paper describes an application worked out in collaboration with a company that produces made-to-order machine components. The goal of the project is to develop a system that can support the company's engineers by automating parts of their component design process. We propose a knowledge extraction methodology based on the recent DMN (Decision Model and Notation) standard and compare a rule-based and a constraint-based method for representing the resulting knowledge. We study the advantages and disadvantages of both approaches in the context of the company's real-life application.

1 Introduction

This research is conducted in collaboration with a company that has engineering and manufacturing offices all over the world. To protect its trade secrets, the company wishes to remain anonymous and they have also requested that we avoid providing too much information about its products. In certain branches of its activities, the company specialises in producing made-to-order components, designed specifically to meet a customer's particular requirements. Like many such companies, it has significantly automated its manufacturing activities, but the design activities of its engineers are still performed "manually". That is to say, the engineers of course make use of computers to perform calculations or create 3D models of the components they design, but there is no software support for the crux of their activity, namely the actual design process itself. To perform this task, the engineers follow an *ad hoc* process, based on past experience, talks with their colleagues, their own preferences, etc.

This way of working is still common in industry. However, it has several downsides. First, the lack of standardisation means that different engineers at different locations may come up with different designs for the same set of requirements, some of which may be worse than others. Second, the company also depends to a large extent on the expertise of some of its key senior engineers. If these should suddenly leave the company, a great deal of the knowledge they have built up over the years would leave with them, significantly reducing

the efficacy of the engineering department. Finally, the lack of software support also means that – in particular, for less challenging design tasks – the engineers often have to spend time carrying out the same routine tasks, reducing their efficiency.

The goal of this research is to develop a system to assist the engineers in their design process. We focus specifically on the design of one particular type of component. This type of component consists of a number of different subparts, each of which exists in a number of different variants and sizes, and which can be produced from different kinds of materials. Customers request components for a specific set of requirements, including a temperature range under which the component should function, pressures the component should be able to withstand, the size that the component should have, etc. The engineers then decide which combination of subparts should be used, which variants of these subparts should be chosen, how big each subpart should be and out of which material it should be made. It is with this task that we want to assist them.

We follow a knowledge-based approach, in we represent the engineers' knowledge in a suitable formal language, and then apply logical inference to this representation in order to provide suggestions to the engineers. This approach starts with a knowledge extraction step in which a knowledge engineer works together with a number of domain experts, in this case the company's design engineers, in order to construct the formal model of their knowledge. Typically, this knowledge extraction is a challenging task, because the knowledge engineers are not familiar with the problem domain, while the domain experts are not familiar with the idea knowledge representation. Good communication between the parties is therefore very important.

In addition to providing automated support, the knowledge extraction process also has the benefit of producing a standardised formal description of the company's design knowledge, thereby eliminating personal preferences of each engineer, regional differences, out-dated habits, and of course human mistakes. For this process to be successful, we believe that it is crucial that the formal specification is not only executable, but that it is also understandable by the engineers. This helps to avoid misunderstandings and errors in the knowledge extraction process. Moreover, it will also allow the engineers to get a better understanding of what is going on inside the decision support system, it will help them to adopt and evaluate the standardised procedure, and it will allow the knowledge base to be maintained after completion of the project.

In [24], the ability to extract knowledge in a format readable by domain expert was identified as a weakness of current product configuration methods. In order to achieve our stated goals, we therefore propose a novel method, consisting of a two-step knowledge extraction methodology. First, we focus on representing the *decision process* that the engineers follow when making a new design. For this, we make use of the recent *Decision Model and Notation (DMN)* [13] standard, which has been developed with the specific aim of being usable by domain experts, without help from a knowledge engineer or software developer. Using an off-the-shelf implementation of the standard, such as that provided by the OpenRules system [14], this DMN model is already fully executable, which allows it to be used by the engineers and validated w.r.t. a batch of test cases.

As we will discuss below, the DMN model by itself is not expressive enough to achieve all of the project's goals. We therefore propose a second knowledge extraction step, in which the DMN model is further analysed together with the design engineers. Having the DMN model already available in this step provides a way of focusing the discussion, ensuring that all the relevant questions end up being discussed, and avoiding misunderstandings. The result of

this second step is a logical specification, written in classical first-order logic – which can be used by an automated reasoning system – in our case the IDP system [3]. This specification can then be validated by comparing its conclusions to those of the original DMN models.

In the following sections, we first provide some more details on the context and goals of the project. We then discuss the first step of the knowledge extraction methodology, using DMN, together with its implementation and limitations. We then present the second step, using the IDP system, again also discussing implementation and limitations. We discuss the validation efforts that were made and finally also related work.

2 Problem Description

The company designs and produces components based on specific customer requests. These customers typically are engineerings from other production companies, who want a specific part to be manufactured according to a detailed set of requirements. In contrast to typical configuration problems, understanding and explicitating the customers' needs is therefore not an issue in this application.

Incoming requests are initially handled by the sales staff. If the customer's requirements can be met by one of the companies standard solutions, the sales staff autonomously handles the request. They are supported in this by a Visual Basic tool that inspects a Microsoft Access database to select the appropriate standard design for a particular request. Requests that fall outside the scope of this tool are forwarded to the engineering department. Here, one of the engineers analyses the requirements and proposes a suitable component design. A distinction is made between requests that fall within known application areas and those that do not. Handling the first kind of requests is a routine job for the engineers and they always follow roughly the same procedure when doing so. The second kind of requests are more challenging and may require a significant amount of creativity from the engineers.

Our project has three main goals. First, the company has noted that is quite difficult and time-consuming to extend the scope of the tool that is used by the sales staff and they are looking for a more maintainable solution. Second, the "routine" work done by the engineers for known applications should be standardised and automated as much as possible. Third, the company also wishes to develop a decision support system that the engineers can use when handling the more challenging requests.

3 Knowledge extraction of the design process

The engineers have a "standard" decision process that they use to handle routine requests. However, this process is not explicitly standardised and different engineers at different locations may do certain things somewhat differently. To fully standardize this process and to be able to automate it, the engineers' detailed technical knowledge needs to be represented in a formal and structured manner. This section describes the knowledge extraction methodology that we have followed.

Because the design process had not yet been internally standardised, we chose to start from a series of brainstorming workshops with all of the involved parties. Each workshop takes a couple of days and results in an initial representation of the design process for a specifically delineated application area. The involved parties are a number of design engineers (representing each of the locations worldwide that are involved in the particular application area), a manager and one external knowledge engineer to guide the workshop. This approach offers a number of advantages.

- Since multiple participants are involved, we do not blindly adopt the approach of one engineer or one particular location.
- The face-to-face time allows intensive discussion about why certain decisions are taken, which is often necessary when different engineers are used to follow different approaches.
- During one multiple-day workshop, all parties focus solely on one specific application, which helps to keep the discussion focused.
- The knowledge engineer not only helps with technical issues concerning the representation, but he also assists the engineers in clarifying their design process: as a non-expert in the domain, he is able to ask "trivial" questions that help to ensure that all the engineers are on the same page and that nothing is being overlooked.

Such a workshop results in a formal representation of the engineers' relevant knowledge, which is then used to build an initial prototype of a decision support system for that particular application area. This prototype is then presented to the design engineers for evaluation. The evaluation can be done briefly by e-mail or in another workshop, depending on how close to reality the preliminary model is. Based on the feedback, the model is refined. This process is repeated until all parties agree that the model is correct.

To support this knowledge extraction process, we need a notation that allows all aspects of the decision process to be expressed. In addition, the notation should not only be readable by the knowledge engineer, but also by the domain experts, who have no background in computer science or logic. This will allow the notation to be used as an effective communication tool throughout the brainstorming workshops and will also give the domain experts confidence in the correctness of the automated system. After surveying the different possibilities, we have decided to use the DMN standard that is explained in the following section.

3.1   The Decision Model and Notation (DMN)

The Decision Model and Notation (DMN) is a relative new standard [13], which is best known for also being responsible for the widely used UML standard. This standard was developed specifically for describing and modeling repeatable decision processes. In addition, it is especially designed to be usable by "business users", without involvement of IT personnel. These two properties make it uniquely well-suited for our purposes. In addition, as an open standard from a well-known organisation, it enjoys tool-support from multiple vendors, which means that it can be adopted without running the risk of vendor lock-in.

In general, a DMN model consists of two components. The first is a Decision Requirement Diagram (DRD). This is a tree-like graph which specifies dependencies between different (sub-) decisions. Figure 1 displays a fragment of the complete DRD representing the decision procedure used in our application.

The other part of a DMN model consists a number of in-depth decision tables, one for each decision in the DRD. An example can be found in Table 1. The purpose of this table is to decide whether the chosen design should contain a *wiper*, a bent piece of plastic that protects the component from environmental factors, such as dirt or reverse pressure (i.e., pressure from the outside to the inside, instead of the other way around). Each column of such a table corresponds to either an input variable (*Dirty Environment* and *Reverse Pressure*, in this case) or an output variable (*Wiper*). In this example, all variables are boolean, but in general DMN also allows other data types. A row in a decision table specifies that if the row is applicable (i.e., all of the input variables satisfy the conditions given by this row) then all of the output variables must have the values given by this row. For instance, the first row of Table 1 states that a wiper must be used whenever the environment is dirty (regardless of
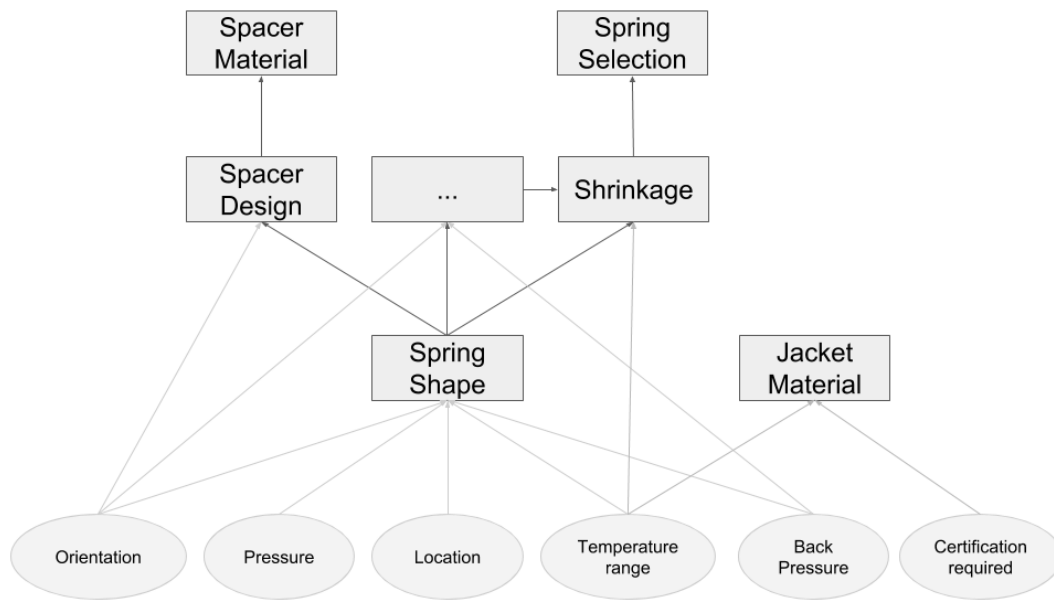
Figure 1 Fragment of the Decision Requirement Diagram.

Table 1 Decision table describing whether or not to use a wiper.

Any	Input		Output
	Dirty Environment	Reverse Pressure	Wiper
1	True	-	True
2	-	True	True
3	False	False	False

whether there is reverse pressure); the second row states that if there is reverse pressure, a wiper must also be used; finally, the third row states that if the environment is not dirty and there is no reverse pressure, a wiper should not be used.

The entries in the table are written in a syntax called the Friendly Enough Expression Language (FEEL), which is also part of the DMN standard. In addition to simple values (as used in Table 1), FEEL also allows numerical comparisons, ranges of values and calculations to be expressed.

If multiple rows in a table might be applicable for some combination of input values, then the table's so-called *hit policy* determines how this should be handled. Table 1 has the hit policy *Any*, as can be seen in its upper left cell. This means that different rows may be applicable for a given input (e.g., the first two rows are applicable in a dirty environment with reverse pressure), but that all applicable rows have the same output, so that it does not matter which row is applied. Other hit policies are *Unique* (only one row may be applicable) and *First* (when multiple rows are applicable, only the top one is considered). In addition, there are also *multiple hit policies* that allow, e.g., the output of all applicable rows to be gathered into a list.

Another, more advanced example is the following. In the design of a component, a spring is used to keep it in place. The type of this spring is determined by two decision nodes in the DRD. First, the general shape of spring is determined (whether to use a stiffer closed spring or a weaker open spring). This influences the overall form of the design. Later, the

■ **Table 2** Decision table for *Spring Shape.*

First	Input					Output
	Orientation	Reverse Pressure	Location	Pressure	Temperature	Spring Shape
1	Radial	True	Pressure Accumulating	-	-	Open
2			Bi-directional	-	-	Open
3			-	≤ 100	-	Open
4	-	-	-	-	-	Closed

■ **Table 3** Decision table for *Use Of Spacer.*

Unique	Input	Output
	Spring Shape	Use Spacer
1	Closed	False
2	Open	True

specific spring is selected, based on how much the component would shrink in the given circumstances. Table 2 shows how the general shape of the spring is decided, based on the reverse pressure and various other inputs.

Another part of the design is a spacer, whose purpose is to keep the component in place, even when there is a high pressure from the backside of the seal. Based on the spring shape, the need for a spacer is decided in Table 3.

3.2   Results

Following the methodology outlined above, we have extracted the knowledge of the routine design process in six different application fields. A total of 75 decision tables were constructed. In each of the applications, one or two tables were pure data tables, consisting of all numerical data for dimensioning the component. Since the discussed applications are more or less similar, some of the already constructed data and decision tables from one application could be reused in another. The extracted tables had an average size of approximately 5 rows and 3 input conditions.

Each workshop started with a brief introduction to DMN, after which the knowledge engineer started to guide the domain experts through the modelling process. We typically started by constructing a DRD to get a general overview of the structure of the design process, and then proceeded to construct detailed decision tables for each of the decisions. The company's engineers found the DMN format quite intuitive and after some initial questions, they were typically able to easily interpret and reason about the knowledge in the tables. Our experiences therefore indeed confirm that DMN's readability for domain experts is a big advantage of this standard.

A small exception to our normal way of working occurred when representing the design process for applications that fall within the scope of the Visual Basic tool that had already been developed for the sales staff. Here, we simply started from the existing VB code and transformed this into a DMN model, which proved to be significantly shorter (360 lines of VB code were reduced to 80 table rows) and easier to maintain.

Overall, the DMN representation seemed to fit well with the engineers' own way of thinking about their design process. However, there were some exceptions. In a few limited cases, the engineers themselves do not follow a strict bottom-up decision procedure when making their design. For instance, in certain circumstances, it is necessary to ensure that the component stays in place. This can be done by using a stiffer spring than usual to prevent

the component from sliding in the wrong direction. Adding a spacer and keeping the normal shape of spring is the preferred approach, but this is not always feasible. In particular, in cold circumstances, the component may shrink to such an extent that the normal spring would fail. However, to know whether this is the case, the shrinkage of the component has to be computed. Because this depends on the materials being used and the precise layout of the different parts of the component, this computation can only be done at the very end of the design process. Therefore, what the engineers currently do is they assume that the spacer option will work, completely design the component based on this assumption, compute the shrinkage and then backtrack over their initial choice if it turns out that the shrinkage is too big. Such a "guess and check" procedure cannot be elegantly represented in DMN. In Section 4 we discuss the work-around that we have used for this.

In general, we perceived the use of a formal representation in the workshop as a significant added value. The precision of the notation allowed us to quickly detect inconsistencies and missing cases in the information that the domain experts were providing. In addition, once they had gotten used to the notation, also the design engineers themselves started to notice flaws in the decision tables, such as implementation mistakes from our side or previously unnoticed exceptions in their own design process. Towards the end of a workshop, the design engineers were comfortable enough with the notation that we could leave certain decision tables to be constructed as "homework" after the end of the workshop.

Based on our experiences, we are confident that the design engineers will be able to maintain the existing decision tables and, with a bit more experience, would be able to construct additional DMN models for new application areas.

4 Direct implementation of the design process

DMN is designed to be a fully executable specification and is currently supported by a number of different tools, both commercial and open source. By providing it with the constructed DMN tables, we have implemented an automated design system in the OpenRules [14] system, currently for two of the six application areas for which the DMN knowledge extraction has been performed.

This direct encoding of the design engineers' design process has the advantage that it is easy to implement, and that is easy to understand for the engineers what is going on. However, there are also downsides to this approach.

First, as mentioned in Section 3.2, a few aspects of the design process do not fit readily into the DMN model. Currently, we have worked around this problem by an "err on the side of safety" approach: for the example given Section 3.2, the engineers have determined a set of parameters within which it is always safe to use the preferred solution of adding a spacer; whenever the input falls outside of this safe range, the alternative option of using a stiffer spring is always chosen. While this solution is suboptimal (in the sense that sometimes a stiffer spring is used when the combination of a weaker spring and a spacer would have sufficed), it avoids the risk of suggesting faulty designs in a way that does not introduce complicated decision structures, which would reduce the legibility of the DMN model.

Second, the DMN representation forces one to mix different kinds of knowledge within a single table, which reduces the maintainability. For instance, Table 2 is based on both physical constraints and preferences of the company. However, the actual constraints and preferences cannot be deduced from this table alone. For instance, the decisions could be explained in any of the following three ways:

- A closed spring is always preferred, but it is unusable in situations 1, 2 and 3;
- An open spring is always preferred, but it can only be used in situations 1, 2 and 3;
- An open spring is preferred if there is reverse pressure, while closed springs are preferred in all situations when there is no reverse pressure.

Now, suppose that a supplier changes the price of the closed spring. This will have an impact on which shape of spring is preferred, but it is impossible to judge the impact of this change on Table 2, without knowing the underlying reason for why this table is as it is. A representation that separates preferences from constraints would not have this problem.

Third, all of the currently available DMN rule engines support only a single inference task, namely that of computing the "output" decision variables given values for all the input variables. In a system that is used interactively by a design engineer, however, we may also envisage other useful inference tasks. For instance, after filling out only a subset of the input variables, the engineer may be interested in knowing whether a design with a closed spring is still possible. Or, in discussions with a customer, he may interested in knowing which values of the input variables would have allowed such a spring to be used if one cannot be used now.

Fourth, DMN keeps the complexity of the decision process manageable by splitting it into different decision tables. A downside of this approach is that it is not possible to talk about global properties of the design. For instance, we may be interested in selecting the cheapest possible design. The cost of a design depends on which parts are included in the design and on which materials are used to make these parts. Both of these decisions influence each other: certain parts can only be made out of certain materials, while the use of a better material might eliminate the need for a particular additional part. This interdependency means that we cannot hope to always find the cheapest global design by making a sequential series of local decisions.

Finally, the entire DMN approach of course assumes that there *is* a decision procedure to model. If we want to develop a system that could provide some assistance to engineers in those challenging new application areas where they themselves do no yet know how precisely a new design should be made, then there is no decision procedure and the DMN approach will be of no use at all.

5   A Constraint-Based Approach

As discussed in the previous section, we cannot hope to achieve all of our stated goals by an approach in which we simply use a direct implementation of the design procedure as the engineers follow it. We will need to take into account also the underlying physical constraints that have led the engineers to adopt this procedure in the first place.

In general, the design process followed by the engineers is governed by a number of physical constraints (e.g., a material $M1$ can only be used in temperatures $< 100°C$) and preferences (e.g., material $M2$ is preferred over material $M1$, perhaps because it is cheaper or more durable). In order to develop a decision support system that can also provide useful information for challenging new application areas, we need to make direct use of these underlying constraints and preferences, rather than of the engineers' existing design process. These constraints provide more information than is explicitly present in the design procedure, because they also explain *why* certain designs are impossible. Therefore, it is not possible to automatically deduce these constraints from the design procedure. Instead, coming up with them requires additional discussions with the design engineers.

To illustrate the constraint-based approach, we return to the running examples of Section 3.1. First, we consider Table 1. The engineers explain the contents of this table as follows: they prefer not to include a wiper unless one is necessary, and a wiper is required

to cope with either reverse pressure or a dirty environment. In other words, this table can be explained as the combination of a preference for not having a wiper, together with two constraints: $ReversePressure \Rightarrow Wiper$ and $DirtyEnvironment \Rightarrow Wiper$.

The underlying reasons for Tables 2 and 3 are more complex. Discussions with the engineers have revealed that these tables can be explained as follows:

1. Only open springs are able to release reverse pressure.
$SpringShape = "Open" \Leftrightarrow AbleToReleaseBP$.

2. It is impossible to use a spacer in combination with a closed spring.
$SpringShape = "Closed" \Rightarrow SpacerDesign = "null"$.

3. When the component should be placed in a pressure accumulating location, it should be able to release reverse pressure.
$Location = "PressureAccumulating" \Rightarrow AbleToReleaseBP$.

4. A spacer is needed (in radial applications) if the reverse pressure is bigger than 100 bar.
$ReversePressure \land Pclass > 100 \Rightarrow SpacerDesign \neq "null"$.

5. In the bi-directional location, the component tends to move back and forth excessively, so in order to avoid damage, a spacer is always needed.
$Location = "Bi - directional" \Rightarrow SpacerDesign \neq "null"$.

6. Lastly, closed springs tend to be cheaper and outperform open springs, so they are the preferred type of spring.

Notice that 1–5 are constraints, while 6 is a preference.

The first line in Table 2 is a result of combining constraint 1 and 3. The component should be able to release reverse pressure and since closed spring designs cannot do that, an open spring design is the only option. The second row is a combination of constraint 2 and constraint 5. In the "Bi-directional" location a spacer is always needed, and since it is impossible to have a spacer in closed spring designs, the only remaining possibility is to go for an open spring design. Analogously, the third line in the decision procedure can be obtained from combining constraint 4 and 2. In all other situations, both closed and open spring designs are possible, but closed designs are preferred, which explains the last row in the decision procedure.

5.1 Knowledge extraction of the physical constraints and preferences

In order to use the physical constraints, we must of course again first elicitate them from the design engineers. In our experience, it was difficult to do this directly. The engineers often did not know quite where to start and discussions tended to be chaotic and unstructured. For this reason, we have chosen to base the knowledge extraction of the constraints on the DMN models. We again organise a discussion with the engineers who were originally involved in the construction of these models and then go over each row of each table and ask them *why* this row produces that particular output. Unlike the workshops in which the DMN models are initially constructed, here it is less crucial to involve different engineers: even though different engineers may disagree on the best solution for a given problem, they tend to all agree on the reason why certain solutions might or might not work.

This use of the DMN tables provides a structured way of working, in which different topics are addressed in a meaningful order and we can be sure that all of the relevant constraints will eventually be mentioned. Moreover, because the engineers know and understand the DMN model, there is never any confusion about which particular question is being discussed at any particular point in time.

To reduce the time investment required from the engineers, it is useful to carefully prepare these discussions in advance. Often, the form in which a particular table has been written down already suggests a certain underlying reason (e.g., the "default" row at the bottom of Table 2 suggests that the closed spring is the preferred choice, with the other rows describing circumstances in which this preferred choice is not possible). In addition, general knowledge about how the components function or considerations that were mentioned during the workshops that constructed the DMN models may provide further clues. In practice, we have found that we can construct most of the constraints without help of the engineers and only need them to verify and help us revise our initial guesses.

Most of the decision tables can be discussed independently. However, certain constraints influence multiple tables. Section 5 handles a detailed example of this.

The preferences we have encountered so far have been quite simple: when a particular part exists in a number of different variants or can be made from a number of different materials, the engineers have been able to rank the variants/materials in an absolute order of preference, typically based on cost and reliability. There has been no need to handle more complex issues such as conditional preferences.

6   Implementation of a constraint-based approach

We have used the knowledge based IDP system [3] to implement a prototype of a constraint-based design system. IDP allows constraints to be expressed in a rich extension of classical first-order logic. Some examples of constraints used, are:

$$\forall s[Subpart] : SubpartUsed(s) \Rightarrow \exists\, 1\; m[Material] : Material(s, m).$$

This IDP formula states that for each subpart it holds that if the subpart is used, there exists exactly one material for that subpart.

$$sum\{s[Subpart] : SubpartUsed(s) \land Length(s, l) : l\} < AvailableSpace.$$

This formula states that the length of the component, computed as the sum of the lengths of all its subparts, must fit in the available space.

The IDP system offers a number of different algorithms, implementing a number of logical inference tasks, based on Answer Set Programming (ASP), Logic Programming (LP) and SAT solving technology. In recent editions of the ASP Competition [1], it was shown to be competitive with other state-of-the-art ASP systems, though typically somewhat slower than systems such as Clasp.

Our main reason for using IDP is its use of classical logic as an input language. This allows individual constraints to be represented in a modular way, which can typically be reasonably well explained to the company's design engineers without requiring much additional background. While the engineers would probably not be able to write down constraints correctly, they are able to read them pretty well. We suspect that for instance ASP specifications would have been harder for the engineers to read, due to the presence of non-classical connectives such as negation-as-failure. A second advantage of IDP is that it provides support for different logic inference tasks. Our current prototype only offers the functionality of generating design proposals, but IDP's different logic inference methods may prove useful if we would want to extend this to other functionalities in the future. This is one potential advantage that IDP offers over the use of constraint-programming languages such as MiniZinc [12].

Our input for IDP consists of six theories: one theory expresses the constraints about the general design of the component; another describes the material choice of each of the parts; the third defines how the component shrinks in low temperatures; a fourth theory describes whether the component will remain in place also in cold environments; the fifth defines whether the complete component fits in the available space; the final theory expresses the preferences by assigning a cost to the design, based on price, durability, availability, etc.

In order to use these theories to compute a design, we can apply the logical inference task of Model Expansion [11]. This takes as input a theory T and a structure S_{in} for part of the vocabulary of T, and the goal is to produce a structure S_{out} for the remaining part of the vocabulary such that $S_{in} \cup S_{out} \models T$. In our case, the structure S_{in} describes the problem specification, by providing an interpretation for predicates such as $Temperature$, $Pressure$ and $Location$ (giving the temperature and pressure ranges and the location in which the component should function); the structure S_{out} then describes a design, by providing an interpretation for predicates such as $SpringShape$ and functions such as $Material$, which maps each component used in the design to the material it should be made from.

However, rather than just computing any model expansion, we make use of IDP's optimisation functionality. This allows us to specify a numerical term t for a model expansion problem (T, S_{in}). IDP will then compute not just any solution to the model expansion problem, but the solution S_{out} that, in addition to being such that $S_{in} \cup S_{out} \models T$ also minimizes the value $t^{S_{in} \cup S_{out}}$ of this term. In our case, the term t is of the form $sum\{p[Penalty] : Violation(p) : p\}$, i.e., we associate to each violation of a preference a certain penalty and the goal is to compute the design for which the sum of all incurred penalties is minimal. IDP implements this inference task by an optimisation loop, which iteratively produces better solutions by each time adding as a new constraint that the next solution must have a lower score than the previous solution. This is the same method as is typically used in, e.g., ASP solvers.

As an implementation of the knowledge base paradigm [5], IDP allows different inference tasks to be performed on the same knowledge base in order to provide different functionalities. Currently, our focus lies on generating designs using the inference task of model expansion. However, in the future, other inference tasks may prove useful for offering additional functionalities, such as explaining why a certain design is not feasible.

6.1 Limitations

Even though using the constraint representation has a lot of interesting advantages, there are also a few downsides to it. The main disadvantage is that it is harder for the domain experts to understand. On the one hand, the syntax for writing down individual constraints is more complex. While we have used IDP because we believe it is quite understandable for untrained experts, it is still much more complex that the simple table-based DMN format. On the other hand, also the constraint-based approach itself seems inherently more difficult for the domain experts. In a DMN decision model, there is always a clear link between input and output, which makes the model easy to interpret and inspect by a domain expert. When using constraints to express design knowledge, a single decision may be affected by numerous constraints. For example in Section 5, the spring design is influenced by a multitude of constraints. Finding out which constraints influence a particular aspect of the design and determining their joint outcome is not a straightforward task and we find this often confuses the domain experts.

A second downside is tied to the particular technology used in the IDP system. IDP's model expansion algorithm follows a ground-and-solve strategy (similar to, e.g., ASP solvers), in which all variables are first translated away, by replacing them with all of their possible

values. However, this requires that each variable must have a finite domain, such that the grounding phase can enumerate all of its possible values. Moreover, in order for the grounding to be computed in reasonable time, these domains should be relatively small. Because our application requires some calculations with floating point numbers (e.g., when calculating the shrinkage in cold circumstances), we have had to implement a work-around to perform these calculations outside of the normal ground-and-solve workflow.

7 Validation and Experimental Results

The DMN model. Starting from a direct formalisation of the engineers' design process proved noticeably useful. Not only did the engineers appreciate the intuitive way of reasoning in the DMN standard, it made them think about how they come to a design in a given situation and about why certain design decisions are made. Moreover, when transforming the Visual Basic tool developed for the sales staff into a DMN model, a number of irregularities surfaced. Without a formal representation of this knowledge, it would have been a far more difficult and time consuming task to detect these faults.

To ensure correctness of the DMN model, the engineers not only inspected the decision tables in detail, but also provided us with ten test cases that represent both normal sets of requirements and a number of edge cases. Our OpenRules implementation using the DMN model generates the correct design in all of the test cases. Computing a design takes about 0.3 seconds single core on an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz.

The IDP model. While it proved relatively easy to construct the DMN model in collaboration with the engineers, constructing the more expressive IDP constraint-based model was more significantly more challenging. We therefore want to use to former to validate the latter. In particular, we want to check two correspondences between the output $D(I)$ of the DMN model D for a given input I and the solutions S_{out} of the model expansion problem (T, S_I) for the IDP constraint theory T. The vocabulary of the theory T was chosen such that the DMN input I and output $D(I)$ can be easily translated into structures S_I and $S_{D(I)}$.

The first property to check is that the constraints should not be too strict: for each possible set of inputs I, the design $D(I)$ that would be constructed by the DMN model $D(I)$ should satisfy the constraints in theory T, i.e., $S_I \cup S_{D(I)} \models T$ or in other words, $S_{D(I)}$ is a solution the model expansion problem (T, S_I).

Second, to verify that the constraints are not too weak, we also check that the design $D(I)$ proposed by the DMN model D is among the optimal solutions of this model expansion problem, i.e., that $t^{S_I \cup S_{D(I)}} \leq t^{S_I \cup S'}$ for any other solution S' to the model expansion problem (T, S_I), where t is the optimisation term that should be minimised. This both checks that the constraints do not fail to rule out designs with a higher score that are in fact impossible and that the weights used in the optimisation criterion are assigned correctly.

We implement both of these checks using IDP. We first transform the DMN model to IDP syntax as described in [4]. We can then use IDP to perform the required checks on relation between the IDP theory derived directly from the DMN model and the IDP theory that represents the constraints.

The first check initially revealed a small number of errors in the constraint-based representation. After minor fixes to the constraints, the first check was concluded successfully. The second check then revealed that, in a number of cases, the constraint-based model produced more optimal designs than the DMN model. While we initially thought that this was due to more errors in the constraints, an analysis together with the design engineers revealed that the outcome of the constraint-based model was in fact correct and that their own design

process was in these cases non-optimal. This non-optimality turned out to be caused by the difficulty of making the decisions in a fixed order. When using the constraint-based method, no fixed decision order is needed, so a better scoring global optimum can be found.

The IDP system typically finds the optimal design in about 3.15 seconds on one core of an Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz.

8   Related Work

A large body of research has been conducted on the topic of automatic product configuration, typically defined as the task of automatically constructing a design from a set of pre-defined components, considering several constraints and some optimisation criteria [2]. Research shows that product configurators have a positive impact on lead time [8, 6] and quotation time [10]. Other comparison studies [18] investigate the effect of configuration systems on product quality, also showing promising results.

A thorough literature review on product configuration was performed by [24]. Their findings reveal that, despite the wide range of existing research, several topics still require further exploration. First, although knowledge acquisition from historical data has been extensively studied, less research has been done on extracting knowledge from domain experts. Moreover, knowledge representation research typically focusses on methods that are intended to be used by knowledge engineers. Little attention has been paid to representations that are usable by domain experts. Our work examines the use of DMN to address these issues in the context of one concrete application domain.

A second aspect which according to [24] has not yet received much attention is the ability to suggest new designs. The majority of existing product configuration approaches focus on selecting the most appropriate option among a fixed range of possibilities. By contrast, our constraint-based approach is also able to provide useful information to the engineers in cases that fall outside the scope of existing solutions.

Third, [24] also identifies several ways in which additional forms of inference might be useful to provide functionality other than suggesting a design. For example, she identifies such tasks as explaining which conflicting constraints have led to a rejected design or reconfiguring an existing design to cope with changed requirements. The IDP system has been developed according to the knowledge base paradigm [5], in which different logical inference methods can be applied to the same knowledge base in order to implement different functionalities. Both of the tasks of explaining conflicts and of reconfiguration have already been considered in the context of this system [19, 22]. The IDP system therefore provides a suitable formalism to express the design knowledge.

9   Conclusions and future work

In this paper, we have presented an approach to develop a decision support system for the design of mechanical components. This research was conducted in collaboration with a multinational company that wants to standardise and partially automate its design process, both for "routine" applications and challenging new application areas.

This project's main challenge is that there are two potentially contradictory requirements. On the one hand, a flexible and powerful knowledge representation is needed that will allow useful conclusions to be provided to the engineers even in circumstances that fall outside of their designs' usual scope. On the other hand, the engineers need to be closely involved in the formal specification since they are expected to agree on and understand the model, and to help maintain it.

To cope with these two requirements, we propose a two-step methodology. First, we use the new DMN standard to extract the "routine" design process into an executable formal model, which can already by automatically validated. We then use this DMN model as a basis to perform a second knowledge extraction step, which results in a first-order logic representation that can be given to the state-of-the-art IDP knowledge base system in order to also perform useful inferences in circumstances that fall outside the scope of the routine design process. This IDP model can then be automatically validated w.r.t. the DMN model.

In future work, we plan to examine the possibility of extending the expressivity of DMN to reduce the gap between DMN and IDP, without sacrificing the ease of understanding for the domain experts. Moreover, we also plan to examine the use of IDP's different inference algorithms to address some of the issues highlighting by [24]. Finally, we also wish to develop a method that would allow the more general knowledge expressed in the IDP model to automatically derive DMN design procedures for new application areas.

References

1   Mario Alviano, Francesco Calimeri, Günther Charwat, Minh Dao-Tran, Carmine Dodaro, Giovambattista Ianni, Thomas Krennwallner, Martin Kronegger, Johannes Oetsch, Andreas Pfandler, et al. The fourth answer set programming competition: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning*, pages 42–53. Springer, 2013.

2   David C Brown. Defining configuring. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 12(4):301–305, 1998. `doi:10.1017/S0890060498124034`.

3   M. Bruynooghe, H. Blockeel, B. Bogaerts, B. De Cat, S. De Pooter, J. Jansen, A. Labarre, J. Ramon, M. Denecker, and S. Verwer. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *Theory and Practice of Logic Programming*, 15(6):783–817, 2015. `doi:10.1017/S147106841400009X`.

4   Ingmar Dasseville, Laurent Janssens, Gerda Janssens, Jan Vanthienen, and Marc Denecker. Combining DMN and the knowledge base paradigm for flexible decision enactment. In Tara Athan, Adrian Giurca, Rolf Grütter, Marc Proctor, Kia Teymourian, and William Van Woensel, editors, *Supplementary Proceedings of the RuleML 2016 Challenge, RuleML, Stony Brook, 6-9 July 2016*. CEUR-WS.org, 2016. URL: `https://lirias.kuleuven.be/handle/123456789/546123`.

5   Marc Denecker and Joost Vennekens. Building a knowledge base system for an integration of logic programming and classical logic. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Lecture Notes in Computer Science, International Conference on Logic Programming, ICLP, Udine, Italy, 9-13 December 2008*, pages 71–76. Springer, 2008. `doi:10.1007/978-3-540-89982-2_12`.

6   C. Forza and F. Salvador. Product configuration and inter-firm co-ordination: An innovative solution from a small manufacturing enterprise. *Computers in Industry*, 49(1):37–46, 2002. `doi:10.1016/S0166-3615(02)00057-X`.

7   Anders Haug, Lars Hvam, and Niels Henrik Mortensen. A layout technique for class diagrams to be used in product configuration projects. *Computers in Industry*, 61(5):409–418, 2010. `doi:10.1016/j.compind.2009.10.002`.

8   Anders Haug, Lars Hvam, and Niels Henrik Mortensen. The impact of product configurators on lead times in engineering-oriented companies. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing: AIEDAM*, 25(2):197–206, 2011. `doi:10.1017/S0890060410000636`.

9   Lars Hvam, Anders Haug, Niels Henrik Mortensen, Christian Thuesen, Relationship Management, and Product Architecture Group. observed benefits from product configuration systems (Hvam) 2011. *International Journal of Industrial Engineering*, 2013.

10 Lars Hvam, Martin Malis, Benjamin Hansen, and Jesper Riis. Reengineering of the quotation process: application of knowledge based systems. *Business Process Management Journal*, 10(2):200–213, 2004. `doi:10.1108/14637150410530262`.

11 David Mitchell, Eugenia Ternovska, Faraz Hach, and Raheleh Mohebali. Model expansion as a framework for modelling and solving search problems. Technical report, Technical Report TR 2006-24, School of Computing Science, Simon Fraser University, 2006.

12 N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, 2007.

13 Object Modelling Group. Decision Model and Notation, 2016. Version 1.1. URL: `http://www.omg.org/spec/DMN/`.

14 OpenRules, Inc. OpenRules, 2017. Version 6.4.2. URL: `http://openrules.com`.

15 F. Salvador and C. Forza. Configuring products to address the customization-responsiveness squeeze: A survey of management issues and opportunities. *International Journal of Production Economics*, 91(3):273–291, 2004. `doi:10.1016/j.ijpe.2003.09.003`.

16 Fabrizio Salvador, Aravind Chandrasekaran, and Tashfeen Sohail. Product configuration, ambidexterity and firm performance in the context of industrial equipment manufacturing. *Journal of Operations Management*, 32(4):138–153, 2014. `doi:10.1016/j.jom.2014.02.001`.

17 Alessio Trentin, Elisa Perin, and Cipriano Forza. Overcoming the customization-responsiveness squeeze by using product configurators : Beyond anecdotal evidence. *Computers in Industry*, 62(3):260–268, 2011. `doi:10.1016/j.compind.2010.09.002`.

18 Alessio Trentin, Elisa Perin, and Cipriano Forza. Product configurator impact on product quality. *International Journal of Production Economics*, 135(2):850–859, 2012. `doi:10.1016/j.ijpe.2011.10.023`.

19 Hanne Vlaeminck, Joost Vennekens, and Marc Denecker. A logical framework for configuration software. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming PPDP '09, Principles and Practice of Declarative Programming, Coimbra, Portugal, 7-9 September, 2009*. ACM, September 2009. `doi:10.1145/1599410.1599428`.

20 Bob Wielinga and Guus Schreiber. Configuration-design problem solving. *IEEE Expert-Intelligent Systems and their Applications*, 12(2):49–56, 1997. `doi:10.1109/64.585104`.

21 Olga Willner, Jonathan Gosling, and Paul Schönsleben. Establishing a maturity model for design automation in sales-delivery processes of ETO products. *Computers in Industry*, 82:57–68, 2016. `doi:10.1016/j.compind.2016.05.003`.

22 Johan Wittocx, Broes De Cat, and Marc Denecker. Towards computing revised models for FO theories. In Salvador Abreu and Dietmar Seipel, editors, *Proceedings of the International Conference on Applications of Declarative Programming and Knowledge Management 2009, International Conference on Applications of Declarative Programming and Knowledge Management, Evora, Portugal, 5-7 November 2009*, pages 199–211, November 2009. `doi:10.1007/978-3-642-20589-7_6`.

23 Dong Yang, Ming Dong, and Rui Miao. Development of a product configuration system with an ontology-based approach. *CAD Computer Aided Design*, 40(8):863–878, 2008. `doi:10.1016/j.cad.2008.05.004`.

24 Linda L. Zhang. Product configuration: A review of the state-of-the-art and future research. *International Journal of Production Research*, 52(21):6381–6398, 2014. `doi:10.1080/00207543.2014.942012`.

25 Linda L. Zhang, Petri T. Helo, Arun Kumar, and Xiao You. Implications of product configurator applications: An empirical study. *IEEE International Conference on Industrial Engineering and Engineering Management*, 2016-Janua(i):57–61, 2016. `doi:10.1109/IEEM.2015.7385608`.

Explanations Generation For Web Service Workflow

Van Duc Nguyen
Computer Science Department
New Mexico State University, USA
vnguyen@cs.nmsu.edu

Son Cao Tran
Computer Science Department
New Mexico State University, USA
tson@cs.nmsu.edu

Enrico Pontelli
Computer Science Department
New Mexico State University, USA
epontell@cs.nmsu.edu

── Abstract ──────────────

The motivation for the work is the challenge of providing textual explanations of automatically generated scientific workflows (e.g., paragraphs that scientists can include in their publications). The extended abstract presents a system which generates explanations for a web service workflow from sets of atoms derived from a collection of ontologies. The system, called `nlgPhylogeny`, demonstrates the feasibility of the task in the *Phylotastic* project, that aims at providing evolutionary biologists with a platform for automatic generation of phylogenetic trees given some suitable inputs.

1 Introduction

The *Phylotastic*[1] project is an attempt for making use of phylogeny trees in education or for researching in biology. To perform a phylogeny tree extraction, the project involves in a series of tasks which is invisible to users. From the need of verification that the phylogeny tree is correctly extracted, some method of describing how the phylogeny tree is delivered to user are provided. One popular way is to describe the progress by natural language. The problem of generating natural language explanations has been explored in several research efforts. For example, the problem has been studied in the context of question-answering systems[2], providing recommendations[3], etc.

In this paper, we describe a system called `nlgPhylogeny`for generating natural language explanations for *Phylotastic* project. The system is powered by Grammatical Framework.

---

[1] http://phylotastic.org
[2] http://coherentknowledge.com
[3] http://gem.med.yale.edu/ergo/default.htm

■ **Figure 1** Overview of `nlgPhylogeny`.

2   Methodology

In this section, we describe the `nlgPhylogeny` system. Figure 1 shows the overall architecture of `nlgPhylogeny`. The main component of the system is the *GF generator* whose inputs are the ontology and the elements necessary for the NLG task (i.e., the set of linearizations, the set of pre-define conjunctives, the set of vocabularies, and the set of sentence models) and whose output is a GF program, i.e., a pair of GF abstract and concrete syntax. This GF program is used for generating the descriptions of workflows via the GF runtime API. The adapter provides the GF generator with the information from the ontology, such as the classes, instances, and relations.

2.1   Web Service Ontology (WSO)

Phylotastic uses web service composition to generate workflows for the extraction/construction of phylogenetic trees. It makes use of two ontologies: WSO and PO. WSO encodes information about registered web services and their abstract classes. In the following discussion, we refer to a simplified version of the ASP encoding of the ontologies used in [2], to facilitate readability. In WSO, a service has a name and is associated with a list of inputs and a list of outputs.

2.2   GF generator

Each Phylotastic workflow is an acyclic directed graph, where the nodes are web services, each consumes some resources (inputs) and produces some resources (outputs). The GF generator produces a portable grammar format (**pgf**) file [1]. This file is able to encode and generate sentences by using GF Runtime API. The GF generator (see Figure 1) accepts two flows of input data:

- The flow of data from the ontology which is maintained by an adapter. The *adapter* is the glue code that connects the ontology to the GF generator. Its main function is to extract classes and properties from the ontology.
- The flow of data from predefined resources that cannot be automatically obtained from the ontology – instead they require manual effort from both ontology experts and linguistic developers;

- A list of *linearizations*; these are essentially the translations of names of ontology entities into linguistic terms. This translation is performed by experts who have knowledge of the ontology domain. An important reason for the existence of this component is that some classes or terms used in the ontology might not be directly understandable by the end user. This may be the result of very specialized strings used in the encoding of the ontology by the ontology engineer (e.g., abbreviations), or the use of URIs for the representation of certain concepts.
- Some *model sentences* which are principally Grammatical Framework syntax trees with meta-information. The meta-information denotes which part of syntax tree can be replaced by some *vocabulary* or *linearization.*
- A list of *pre-defined vocabularies* which are domain-specific for the ontology. A *pre-defined vocabulary* is different from linearizations, in the sense that some lexicon may not be present in the ontology but might be needed in the sentence construction; the predefined vocabulary is also useful to bring variety in word choices when parts of a *model sentence* are replaced by the GF generator.
- A configuration of *pre-defined conjunctives* which depend on the document planning result. Basically, this configuration defines which sentences accept a conjunctive adverb in order to provide generated text transition and smoothness.

Based on the number of inputs and outputs of a service, the GF generator determines how many parameters will be included in the GF abstraction function corresponding to the service. Furthermore, for each input or output of a service, the GF generator includes an *Input* or *Output* in the GF abstract function.

Next, the GF generator looks up in the *sentence models* a model syntax tree whose structure is suitable for the number of inputs and outputs of the service. If such syntax tree exists, the GF generator will replace parts of the syntax tree with the GF service input and output functions, to create a new GF syntax tree which can be appended in the GF concrete function.

From the abstract and concrete syntax built by GF generator, it is possible to generate the sentence

> *The input of service phylotastic_FindScientificNamesFromWeb_GET is a web link and its outputs are a set of species names and a set of scientific names.*

for the atom `occur_concrete(phylotastic_FindScientificNamesFromWeb_GET,1)`. We use the same technique to encode the other types of sentences to describe a full workflow.

References

1 Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-time Format for Type-theoretical Grammars. *Journal of Logic, Language and Information*, 19:201–228, 2010.

2 Thanh H. Nguyen, Tran Cao Son, and Enrico Pontelli. Automatic Web Services Composition for Phylotastic. In *Practical Aspects of Declarative Languages - 20th International Symposium*, pages 186–202, 2018. `doi:10.1007/978-3-319-73305-0_13`.

Probabilistic Action Language $p\mathcal{BC}+$

Yi Wang

Arizona State University
School of Computing, Informatics, and Decision Systems Engineering
Fulton Schools of Engineering, Arizona State University
P.O. Box 878809, Tempe, AZ 85287-8809, United States
ywang485@asu.edu

─── Abstract ───

We present an ongoing research on a probabilistic extension of action language $\mathcal{BC}+$. Just like $\mathcal{BC}+$ is defined as a high-level notation of answer set programs for describing transition systems, the proposed language, which we call $p\mathcal{BC}+$, is defined as a high-level notation of LP^{MLN} programs – a probabilistic extension of answer set programs.

As preliminary results accomplished, we illustrate how probabilistic reasoning about transition systems, such as prediction, postdiction, and planning problems, as well as probabilistic diagnosis for dynamic domains, can be modeled in $p\mathcal{BC}+$ and computed using an implementation of LP^{MLN}.

For future work, we plan to develop a compiler that automatically translates $p\mathcal{BC}+$ description into LP^{MLN} programs, as well as parameter learning in probabilistic action domains through LP^{MLN} weight learning. We will work on defining useful extensions of $p\mathcal{BC}+$ to facilitate hypothetical/counterfactual reasoning. We will also find real-world applications, possibly in robotic domains, to empirically study the performance of this approach to probabilistic reasoning in action domains.

1 Introduction and Problem Description

Action languages, such as $\mathcal{A}$ [9], $\mathcal{B}$ [10], $\mathcal{C}$ [12], $\mathcal{C}+$ [11], and $\mathcal{BC}$ [15], are formalisms for describing actions and their effects. Many of these languages can be viewed as high-level notations of answer set programs structured to represent transition systems. The expressive possibility of action languages, such as indirect effects, triggered actions, and additive fluents, has been one of the main research topics. Most of the extensions accounting for that are logic-oriented, and less attention has been paid to probabilistic reasoning, with a few exceptions such as [6, 8], let alone automating such probabilistic reasoning and learning parameters of an action description.

Action language $\mathcal{BC}+$ [2], one of the most recent additions to the family of action languages, is no exception. While the language is highly expressive to embed other action languages, such as $\mathcal{C}+$ [11] and $\mathcal{BC}$ [14], it does not have a natural way to express the likelihood of histories (i.e., a sequence of transitions).

▶ **Example 1.** Consider an extension of the robot example from [13]: A robot and a book that can be picked up are located in a building with 2 rooms `r1` and `r2`. The robot can move to rooms, pick up the book and put down the book. There is 0.1 chance that it fails when it tries to enter a room, a 0.2 chance that the robot drops the book when it has the book, and 0.3 chance that the robot fails when it tries to pick up the book. The robot, as well as the book, was initially at `r1`. It executed the following actions to deliver the book from `r1` to `r2`: pick up the book; go to `r2`; put down the book. However, after the execution, it observes that the book is not at `r2`. What was the problem?

To answer the above query, an action language needs the capabilities of not only probabilistic reasoning, but also abductive reasoning in a probabilistic setting. In my research, we are working on a probabilistic extension of $\mathcal{BC}+$, which we call $p\mathcal{BC}+$, with the expressivity to answer queries such as the one in Example 1. Just like $\mathcal{BC}+$ is defined as a high-level notation of answer set programs for describing transition systems, $p\mathcal{BC}+$ is defined as a high-level notation of LP$^{\text{MLN}}$ programs – a probabilistic extension of answer set programs. Language $p\mathcal{BC}+$ inherits expressive logical modeling capabilities of $\mathcal{BC}+$ but also allows us to assign a probability to a sequence of transitions so that we may distinguish more probable histories.

In this paper, as preliminary results accomplished, we will show how probabilistic reasoning about transition systems, such as prediction, postdiction, and planning problems, can be modeled in $p\mathcal{BC}+$ and computed using an implementation of LP$^{\text{MLN}}$[16]. Further, we will show that it can be used for probabilistic abductive reasoning about dynamic domains, where the likelihood of the abductive explanation is derived from the parameters manually specified or automatically learned from the data.

For future work, we plan to develop a compiler that automatically translates $p\mathcal{BC}+$ description into LP$^{\text{MLN}}$ programs, as well as parameter learning in probabilistic action domains through LP$^{\text{MLN}}$ weight learning. We will work on defining useful extensions of $p\mathcal{BC}+$ to facilitate hypothetical/counterfactual reasoning. We will also find real-world applications, possibly in robotic domains, to empirically study the performance of this approach to probabilistic reasoning in action domains.

This paper will give a summary of my research on $p\mathcal{BC}+$, including the background and some review of existing literature (Section 2), goal of the research (Section 3), the current status of the research (Section 4), preliminary results accomplished (Section 5) as well as issues and expected achievements (Section 6).

2 Background and Overview of Existing Literature

2.1 Probabilistic Reasoning and Diagnosis in the Context of Action Languages

There are various formalisms for reasoning in probabilistic action domains. $P\mathcal{C}+$ [8] is a generalization of the action language $\mathcal{C}+$ that allows for expressing probabilistic information. $P\mathcal{C}+$ expresses probabilistic transition of states through so-called *context variables*, which are exogenous variables associated with predefined probability distributions. $P\mathcal{C}+$ allows for expressing qualitative and quantitative uncertainty about actions by referring to the sequence of "belief" states – possible sets of states together with probabilistic information. On the other hand, the semantics is highly complex and there is no implementation of $P\mathcal{C}+$ as far as we know.

[20] defined a probabilistic action language called $\mathcal{NB}$, which is an extension of the (deterministic) action language $\mathcal{B}$. $\mathcal{NB}$ can be translated into P-log [4] and since there exists a system for computing P-log, reasoning in $\mathcal{NB}$ action descriptions can be automated. Like $\mathcal{PC}+$, probabilistic transitions are expressed through dynamic causal laws with random variables associated with predefined probability distribution. In $\mathcal{NB}$, however, these random variables are hidden from the action description and are only visible in the translated P-log representation. In order to translate $\mathcal{NB}$ into executable low-level logic programming languages, some semantical assumptions have to be made in $\mathcal{NB}$, such as all actions have to be always executable and nondeterminism can only be caused by random variables.

Probabilistic action domains, especially in terms of probabilistic effects of actions, can be formalized as Markov Decision Process (MDP). The language proposed in [6] aims at facilitating elaboration tolerant representations of MDPs. The syntax is similar to $\mathcal{NB}$ and $\mathcal{PC}+$. The semantics is more complex as it allows preconditions of actions and imposes less semantical assumption. The concept of *unknown variables* associated with probability distributions is similar to random variables in $\mathcal{NB}$. There is, as far as we know, no implementation of the language. There is no discussion about probabilistic diagnosis in the context of the language. PPDDL [19] is a probabilistic extension of the planning definition language PDDL. Like $\mathcal{NB}$, the nondeterminism that PPDDL considers is only the probabilistic effect of actions. The semantics of PDDL is defined in terms of MDP. There are also probabilistic extensions of the Event Calculus such as [7] and [18].

In the above formalisms, the problem of probabilistic diagnosis is only discussed in [20]. [3] and [5] studied the problem of diagnosis. However, they are focused on diagnosis in deterministic and static domains. [13] has proposed a method for diagnosis in action domains with situation calculus. Again, the diagnosis considered there does not involve any probabilistic measure.

2.2 Review: Language LP$^{\mathrm{MLN}}$

We review the definition of LP$^{\mathrm{MLN}}$ from [17]. An LP$^{\mathrm{MLN}}$ program is a finite set of weighted rules $w : R$ where R is a rule and w is a real number (in which case, the weighted rule is called *soft*) or α for denoting the infinite weight (in which case, the weighted rule is called *hard*). An LP$^{\mathrm{MLN}}$ program is called *ground* if its rules contain no variables. We assume a finite Herbrand Universe so that the ground program is finite. Each ground instance of a non-ground rule receives the same weight as the original non-ground formula.

For any ground LP$^{\mathrm{MLN}}$ program Π and any interpretation I, $\overline{\Pi}$ denotes the usual (unweighted) ASP program obtained from Π by dropping the weights, Π_I denotes the set of $w : R$ in Π such that $I \models R$, and SM$[\Pi]$ denotes the set $\{I \mid I$ is a stable model of $\overline{\Pi_I}\}$. The *unnormalized weight* of an interpretation I under Π is defined as

$$
W_\Pi(I) = \begin{cases} exp\left(\displaystyle\sum_{w:R \,\in\, \Pi_I} w \right) & \text{if } I \in \mathrm{SM}[\Pi]; \\ 0 & \text{otherwise.} \end{cases}
$$

The *normalized weight* (a.k.a. *probability*) of an interpretation I under Π is defined as

$$
P_\Pi(I) = \lim_{\alpha \to \infty} \frac{W_\Pi(I)}{\displaystyle\sum_{J \in \mathrm{SM}[\Pi]} W_\Pi(J)}.
$$

Interpretation I is called a *(probabilistic) stable model* of Π if $P_\Pi(I) \neq 0$. The most probable stable models of Π are the stable models with the highest probability.

2.3   Review: Multi-Valued Probabilistic Programs

Multi-valued probabilistic programs [17] are a simple fragment of $\mathrm{LP}^{\mathrm{MLN}}$ that allows us to represent probability more naturally.

We assume that the propositional signature σ is constructed from "constants" and their "values." A *constant c* is a symbol that is associated with a finite set $Dom(c)$, called the *domain*. The signature σ is constructed from a finite set of constants, consisting of atoms $c{=}v$ [1] for every constant c and every element v in $Dom(c)$. If the domain of c is $\{\mathbf{f}, \mathbf{t}\}$ then we say that c is *Boolean*, and abbreviate $c{=}\mathbf{t}$ as c and $c{=}\mathbf{f}$ as $\sim c$.

We assume that constants are divided into *probabilistic* constants and *non-probabilistic* constants. A multi-valued probabilistic program $\mathbf{\Pi}$ is a tuple $\langle PF, \Pi \rangle$, where

- *PF* contains *probabilistic constant declarations* of the following form:

$$p_1 :: \ c{=}v_1 \mid \cdots \mid p_n :: \ c{=}v_n \tag{1}$$

  one for each probabilistic constant c, where $\{v_1, \ldots, v_n\} = Dom(c)$, $v_i \neq v_j$, $0 \leq p_1, \ldots, p_n \leq 1$ and $\sum_{i=1}^{n} p_i = 1$. We use $M_{\mathbf{\Pi}}(c = v_i)$ to denote p_i. In other words, *PF* describes the probability distribution over each "random variable" c.
- Π is a set of rules such that the head contains no probabilistic constants.

The semantics of such a program $\mathbf{\Pi}$ is defined as a shorthand for $\mathrm{LP}^{\mathrm{MLN}}$ program $T(\mathbf{\Pi})$ of the same signature as follows.

- For each probabilistic constant declaration (1), $T(\mathbf{\Pi})$ contains, for each $i = 1, \ldots, n$, (i) $ln(p_i) : c{=}v_i$ if $0 < p_i < 1$; (ii) $\alpha : \ c{=}v_i$ if $p_i = 1$; (iii) $\alpha : \ \bot \leftarrow c{=}v_i$ if $p_i = 0$.
- For each rule *Head $\leftarrow$ Body* in Π, $T(\mathbf{\Pi})$ contains $\alpha : \ \ Head \leftarrow Body$.
- For each constant c, $T(\mathbf{\Pi})$ contains the uniqueness of value constraints

$$\alpha : \ \ \bot \leftarrow c{=}v_1 \wedge c = v_2 \tag{2}$$

  for all $v_1, v_2 \in Dom(c)$ such that $v_1 \neq v_2$, and the existence of value constraint

$$\alpha : \ \ \bot \leftarrow \neg \bigvee_{v \in Dom(c)} c{=}v \ . \tag{3}$$

In the presence of the constraints (2) and (3), assuming $T(\mathbf{\Pi})$ has at least one (probabilistic) stable model that satisfies all the hard rules, a (probabilistic) stable model I satisfies $c = v$ for exactly one value v, so we may identify I with the value assignment that assigns v to c.

3   Goal of the Research

The following are our research objectives.

- **Designing Probabilistic Action Language on the Foundation of $\mathrm{LP}^{\mathrm{MLN}}$.** We design the syntax and semantics of the language $p\mathcal{BC}+$ to allow for commonsense reasoning, probabilistic inference and statistical learning. Furthermore, we study the theoretical properties of the action language to establish its relation with probabilistic transition systems.

---

[1] Note that here "=" is just a part of the symbol for propositional atoms, and is not equality in first-order logic.

- **Defining the Extension of the Action Language to Explain the Reason of Failure in Dynamic Domains.** We extend the probabilistic action language to account for diagnostic reasoning when the observation conflicts with the way the system is supposed to behave. This will be in contrast with diagnostic reasoning in other action languages, which is logical and does not distinguish which diagnosis is more probable.
- **Extending the Action Language For Hypothetical/Counterfactual Reasoning.** We extend the probabilistic action language to answer queries involving hypothetical/-counterfactual reasoning, where the diagnosis or observation is given, we are interested in how the outcome would have been affected if some action happened instead.
- **Implementing a Compiler that Automatically Translates $p\mathcal{BC}+$ Descriptions to LP$^{\mathrm{MLN}}$ Programs.** Since $p\mathcal{BC}+$ can be executable through translation to LP$^{\mathrm{MLN}}$, it is desirable to have a compiler that automates this translation. We plan to develop such a compiler.
- **Empirically Studying the Performance of $p\mathcal{BC}+$ with Real-World Applications.** After we have the implementation for inference and learning on $p\mathcal{BC}+$ action descriptions, we will apply $p\mathcal{BC}+$ on reasoning and learning tasks in real-world applications, possibly robotic domains.

4 Current Status of the Research

This research is at its starting phase. In our recent paper accepted by ICLP 2018, we have defined the syntax and semantics of $p\mathcal{BC}+$, and experimented with several examples through manual translation to LP$^{\mathrm{MLN}}$. We have also defined the extension that allows diagnostic reasoning in probabilistic action domains.

Currently we are investigating on parameter learning of $p\mathcal{BC}+$ through LP$^{\mathrm{MLN}}$ weight learning. We are developing a prototype system for LP$^{\mathrm{MLN}}$ weight learning, and several examples of parameter learning of $p\mathcal{BC}+$ descriptions are part of the benchmarks we use for the prototype system.

5 Preliminary Results Accomplished

In this section, we will present the syntax and semantics of $p\mathcal{BC}+$, and illustrate how various reasoning tasks involving probabilistic inference can be automated in this language, through translation to LP$^{\mathrm{MLN}}$.

5.1 Syntax of $p\mathcal{BC}+$

We assume a propositional signature σ as defined in Section 2.3. We further assume that the signature of an action description is divided into four groups: *fluent constants*, *action constants*, *pf (probability fact) constants* and *initpf (initial probability fact) constants*. Fluent constants are further divided into *regular* and *statically determined*. The domain of every action constant is Boolean. A *fluent formula* is a formula such that all constants occurring in it are fluent constants.

The following definition of $p\mathcal{BC}+$ is based on the definition of $\mathcal{BC}+$ language.

A *static law* is an expression of the form

caused F **if** G $\hspace{4cm}$ (4)

where F and G are fluent formulas.

A *fluent dynamic law* is an expression of the form

caused F **if** G **after** H $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (5)

where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants and H does not contain initpf constants.

A *pf constant declaration* is an expression of the form

caused $pf = \{v_1 : p_1, \ldots, v_n : p_n\}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (6)

where pf is a pf constant with domain $\{v_1, \ldots, v_n\}$, $0 < p_i < 1$ for each $i \in \{1, \ldots, n\}^2$, and $p_1 + \cdots + p_n = 1$. In other words, (6) describes the probability distribution of pf.

An *initpf constant declaration* is an expression of the form (6) where pf is an initpf constant.

An *initial static law* is an expression of the form

initially F **if** G $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ (7)

where F is a fluent formula and G is a formula that contains neither action constant nor pf constant.

A *causal law* is a static law, a fluent dynamic law, a pf constant declaration, an initpf constant declaration, or an initial static law. An *action description* is a finite set of causal laws.

We use σ^{fl} to denote the set of fluent constants, σ^{act} to denote the set of action constants, σ^{pf} to denote the set of pf constants, and σ^{initpf} to denote the set of initpf constants in D. For any signature σ' and any $i \in \{0, \ldots, m\}$, we use $i : \sigma'$ to denote the set $\{i : a \mid a \in \sigma'\}$.

By $i : F$ we denote the result of inserting $i :$ in front of every occurrence of every constant in formula F. This notation is straightforwardly extended when F is a set of formulas.

▶ **Example 2.** The following is an action description in $p\mathcal{BC}+$ for the transition system shown in Figure 1, P is a Boolean regular fluent constant, and A is an action constant. Action A toggles the value of P with probability 0.8. Initially, P is true with probability 0.6 and false with probability 0.4. We call this action description *PSD*. (x is a schematic variable that ranges over $\{\mathbf{t}, \mathbf{f}\}$.)

$\qquad$ **caused** P **if** $\top$ **after** $\sim P \land A \land Pf$, $\qquad$ **caused** $Pf = \{\mathbf{t} : 0.8, \mathbf{f} : 0.2\}$,
$\qquad$ **caused** $\sim P$ **if** $\top$ **after** $P \land A \land Pf$, $\qquad$ **caused** $Init\_P = \{\mathbf{t} : 0.6, \mathbf{f} : 0.4\}$,
$\qquad$ **caused** $\{P\}^{\mathrm{ch}}$ **if** $\top$ **after** P, $\qquad\qquad\quad$ **initially** $P = x$ **if** $Init\_P = x$.
$\qquad$ **caused** $\{\sim P\}^{\mathrm{ch}}$ **if** $\top$ **after** $\sim P$,

($\{P\}^{\mathrm{ch}}$ is a choice formula standing for $P \lor \neg P$.)

5.2   Semantics of $p\mathcal{BC}+$

Given a non-negative integer m denoting the maximum length of histories, the semantics of an action description D in $p\mathcal{BC}+$ is defined by a reduction to multi-valued probabilistic program $Tr(D, m)$, which is the union of two subprograms D_m and D_{init} as defined below.

---

^{2} We require $0 < p_i < 1$ for each $i \in \{1, \ldots, n\}$ for the sake of simplicity. On the other hand, if $p_i = 0$ or $p_i = 1$ for some i, that means either v_i can be removed from the domain of pf or there is not really a need to introduce pf as a pf constant. So this assumption does not really sacrifice expressivity.
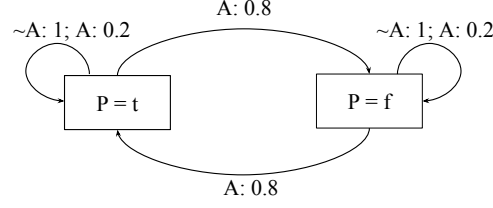
Figure 1 A transition system with probabilistic transitions.

For an action description D of a signature σ, we define a sequence of multi-valued probabilistic program $D_0, D_1, \ldots, D_m$ so that the stable models of D_m can be identified with the paths in the transition system described by D. The signature σ_m of D_m consists of atoms of the form $i : c = v$ such that

- for each fluent constant c of D, $i \in \{0, \ldots, m\}$ and $v \in Dom(c)$,
- for each action constant or pf constant c of D, $i \in \{0, \ldots, m-1\}$ and $v \in Dom(c)$.

We use σ_m^x, where $x \in \{act, fl, pf\}$, to denote the subset of σ_m

$$\{i : c = v \mid i : c = v \in \sigma_m \text{ and } c \in \sigma^x\}.$$

We define D_m to be the multi-valued probabilistic program $\langle PF, \Pi \rangle$, where Π is the conjunction of

$$i : F \leftarrow i : G \tag{8}$$

for every static law (4) in D and every $i \in \{0, \ldots, m\}$;

$$i{+}1 : F \leftarrow (i{+}1 : G) \wedge (i : H) \tag{9}$$

for every fluent dynamic law (5) in D and every $i \in \{0, \ldots, m-1\}$;

$$\{0 : c = v\}^{\mathrm{ch}} \tag{10}$$

for every regular fluent constant c and every $v \in Dom(c)$;

$$\{i : c = \mathbf{t}\}^{\mathrm{ch}}, \quad \{i : c = \mathbf{f}\}^{\mathrm{ch}} \tag{11}$$

for every action constant c; and PF consists of

$$p_1 :: \ i : pf = v_1 \mid \cdots \mid p_n :: \ i : pf = v_n \tag{12}$$

$(i = 0, \ldots, m-1)$ for each pf constant declaration (6) in D that describes the probability distribution of pf.

In addition, we define the program D_{init}, whose signature is $0 : \sigma^{initpf} \cup 0 : \sigma^{fl}$. D_{init} is the multi-valued probabilistic program

$$D_{init} = \langle PF^{init}, \Pi^{init} \rangle$$

where Π^{init} consists of the rule

$$\bot \leftarrow \neg(0 : F) \wedge 0 : G$$

for each initial static law (7), and PF^{init} consists of

$$p_1 :: \ 0\!:\!c = v_1 \mid \cdots \mid p_n :: \ 0\!:\!c = v_n$$

for each initpf constant declaration (6).

We define $Tr(D, m)$ to be the union of the two multi-valued probabilistic program $\langle PF \cup PF^{init}, \Pi \cup \Pi^{init} \rangle$.

▶ **Example 3.** For the action description PSD in Example 2, PSD_{init} is the following multi-valued probabilistic program ($x \in \{\mathbf{t}, \mathbf{f}\}$):

$$0.6 :: \ 0\!:\!Init\_P \ \mid \ 0.4 :: \ 0\!\sim\!Init\_P$$
$$\bot \leftarrow \neg(0\!:\!P\!=\!x) \wedge 0 : Init\_P\!=\!x.$$

and PSD_m is the following multi-valued probabilistic program (i is a schematic variable that ranges over $\{1, \ldots, m-1\}$):

$$0.8 :: \ i : Pf \ \mid \ 0.2 :: \ i\!:\!\sim\!Pf \qquad \{i{+}1 : P\}^{\mathrm{ch}} \leftarrow i : P$$
$$i{+}1 : P \leftarrow i\!:\!\sim\!P \wedge i : A \wedge i : Pf \qquad \{i{+}1\!:\!\sim\!P\}^{\mathrm{ch}} \leftarrow i\!:\!\sim\!P$$
$$i{+}1\!:\!\sim\!P \leftarrow i : P \wedge i : A \wedge i : Pf \qquad \{i : A\}^{\mathrm{ch}} \qquad \{i\!:\!\sim\!A\}^{\mathrm{ch}}$$
$$\{0\!:\!P\}^{\mathrm{ch}} \qquad \{0\!\sim\!P\}^{\mathrm{ch}}$$

5.3 $p\mathcal{BC}+$ Action Descriptions and Probabilistic Reasoning

In this section, we illustrate how the probabilistic extension of the reasoning tasks discussed in [11], i.e., prediction, postdiction and planning, can be represented in $p\mathcal{BC}+$ and automatically computed using LPMLN2ASP [16]. Consider the following probabilistic variation of the well-known Yale Shooting Problem: There are two (deaf) turkeys: a fat turkey and a slim turkey. Shooting at a turkey may fail to kill the turkey. Normally, shooting at the slim turkey has 0.6 chance to kill it, and shooting at the fat turkey has 0.9 chance. However, when a turkey is dead, the other turkey becomes alert, which decreases the success probability of shooting. For the slim turkey, the probability drops to 0.3, whereas for the fat turkey, the probability drops to 0.7.

The example can be modeled in $p\mathcal{BC}+$ as follows:

Notation: t range over $\{SlimTurkey, FatTurkey\}$.

Regular fluent constants:	Domains:
$Alive(t)$, $\quad Loaded$	Boolean
Statically determined fluent constants:	Domains:
$Alert(t)$	Boolean
Action constants:	Domains:
$Load$, $\quad Fire(t)$	Boolean
Pf constants:	Domains:
$Pf\_Killed(t)$, $\quad Pf\_Killed\_Allert(t)$	Boolean
InitPf constants:	
$Init\_Alive(t)$, $\quad Init\_Loaded$	Boolean

caused $Loaded$ **if** $\top$ **after** $Load$
caused $Pf\_Killed(SlimTurkey) = \{\mathbf{t} : 0.6, \mathbf{f} : 0.4\}$
caused $Pf\_Killed\_Alert(SlimTurkey) = \{\mathbf{t} : 0.3, \mathbf{f} : 0.7\}$
caused $Pf\_Killed(FatTurkey) = \{\mathbf{t} : 0.9, \mathbf{f} : 0.1\}$

caused $Pf\_Killed\_Alert(FatTurkey) = \{\mathbf{t} : 0.7, \mathbf{f} : 0.3\}$
caused $\sim Alive(t)$ **if** $\top$ **after** $Loaded \wedge Fire(t) \wedge \sim Alert(t) \wedge Pf\_Killed(t)$
caused $\sim Alive(t)$ **if** $\top$ **after** $Loaded \wedge Fire(t) \wedge Alert(t) \wedge Pf\_Killed\_Alert(t)$
caused $\sim Loaded$ **if** $\top$ **after** $Fire(t)$
default $\sim Alert(t)$
caused $Alert(t_1)$ **if** $\sim Alive(t_2) \wedge Alive(t_1) \wedge t_1 \neq t_2$
caused $\{Alive(t)\}^{\mathrm{ch}}$ **if** $\top$ **after** $Alive(t)$,
caused $\{Loaded\}^{\mathrm{ch}}$ **if** $\top$ **after** $Loaded$
caused $\{\sim Alive(t)\}^{\mathrm{ch}}$ **if** $\top$ **after** $\sim Alive(t)$
caused $\{\sim Loaded\}^{\mathrm{ch}}$ **if** $\top$ **after** $\sim Loaded$
caused $\bot$ **after** $a_1 \wedge a_2$
caused $Init\_Alive(t) = \{\mathbf{t} : 0.5, \mathbf{f} : 0.5\}$ **initially** $Alive(t) = b$ **if** $Init\_Alive(t) = b$
caused $Init\_Loaded = \{\mathbf{t} : 0.5, \mathbf{f} : 0.5\}$ **initially** $Loaded = b$ **if** $Init\_Loaded = b$

We translate the action description into an $\mathrm{LP}^{\mathrm{MLN}}$ program and use LPMLN2ASP to answer various queries about transition systems, such as prediction, postdiction and planning queries.

Prediction. For a prediction query, we are given a sequence of actions and observations that occurred in the past, and we are interested in the probability of a certain proposition describing the result of the history, or the most probable result of the history. Formally, we are interested in the conditional probability $Pr_{Tr(D,m)}(Result \mid Act, Obs)$ or the MAP inference $\operatorname*{argmax}_{Result} Pr_{Tr(D,m)}(Result \mid Act, Obs)$, where $Result$ is a proposition describing a possible outcome, Act is a set of facts of the form $i : a$ or $i : \sim a$ for $a \in \sigma^{act}$, and Obs is a set of facts of the form $i : c = v$ for $c \in \sigma^{fl}$ and $v \in Dom(c)$.

For example, in the Yale shooting example, such a query could be "Given that only the fat turkey is alive and the gun is loaded at the beginning, what is the probability that the fat turkey died after shooting is executed?". To answer this query, we manually translate the action description above into the input language of LPMLN2ASP and add the following action and observation as constraints:

```
:- not alive("slimTurkey", "f", 0). :- not alive("fatTurkey", "t", 0).
:- not loaded("t", 0). :- not fire("fatTurkey", "t", 0).
```

Executing the command

```
lpmln2asp -i yale-shooting.lpmln -q alive
```

yields

```
alive('fatTurkey', 'f', 1) 0.700000449318
```

Postdiction. In the case of postdiction, we infer a condition about the initial state given the history. Formally, we are interested in the conditional probability $Pr_{Tr(D,m)}(Initial\_State \mid Act, Obs)$ or the MAP inference $\operatorname*{argmax}_{Initial\_State} Pr_{Tr(D,m)}(Initial\_State \mid Act, Obs)$, where $Initial\_State$ is a proposition about the initial state; Act and Obs are defined as above.

For example, in the Yale shooting example, such a query could be "Given that the slim turkey was alive and the gun was loaded at the beginning, the person shot at the slim turkey and it died, what is the probability that the fat turkey was alive at the beginning?"

Formalizing the query and executing the command

```
lpmln2asp -i yale-shooting.lpmln -q alive
```

yields

```
alive('fatTurkey', 't', 1) 0.666661211973
```

Planning.   In this case, we are interested in a sequence of actions that would result in the highest probability of a certain goal. Formally, we are interested in

$$\underset{Act}{\mathrm{argmax}}\; Pr_{Tr(D,m)}(Goal \mid Initial\_State, Act)$$

where $Goal$ is a condition for a goal state, and Act is a sequence of actions $a \in \sigma^{act}$ specifying actions executed at each timestep.

For example, in the Yale shooting example, such query can be "given that both the turkeys are alive and the gun is not loaded at the beginning, generate a plan that gives best chance to kill both the turkeys with 4 actions".

Formalizing the query and executing the command

```
lpmln2asp -i yale-shooting.lpmln
```

finds the most probable stable model, which yields

```
load("t",0) fire("slimTurkey","t",1) load("t",2) fire("fatTurkey","t",3)
```

which suggests to first kill the slim turkey and then the fat turkey.

5.4   Extending $p\mathcal{BC}+$ to Allow Diagnosis

We define the following new constructs to allow probabilistic diagnosis in action domains. Note that these constructs are simply syntactic sugar that does not change the actual expressivity of the language.

- We introduce a subclass of regular fluent constants called *abnormal fluents*.
- When the action domain contains at least one abnormal fluent, we introduce a special statically determined fluent constant ab with Boolean domain, and we add
  default $\sim ab$.
- We introduce the expression

     caused_ab F **if** G **after** H

  where F and G are fluent formulas and H is a formula, provided that F does not contain statically determined constants and H does not contain initpf constants. This expression is treated as an abbreviation of

     caused F **if** $ab \wedge G$ **after** H.

Once we have defined abnormalities and how they affect the system, we can use

     caused ab

to enable taking abnormalities into account in reasoning.

We can answer the query in Example 1 by modeling the action domain with this extension. Due to lack of space, we skip the details.

6 Open Issues and Expected Achievements

The main open issue is that we do not have a compiler that automates the translation from $p\mathcal{BC}+$ to LP^{MLN}. As illustrated in Section 5.3, the action language $p\mathcal{BC}+$ can be executable through translation to LP^{MLN}. It is desirable to have a compiler that automates this translation, so that the user can directly write $p\mathcal{BC}+$ descriptions and does not need to worry about the translation detail. We plan to develop a compiler that translates action descriptions in $p\mathcal{BC}+$ into LP^{MLN} programs automatically.

The interface and usage of the compiler will be similar to the system CPLUS2ASP [1], which translates the action language $\mathcal{C}+$ to ASP.

Other future works include extending $p\mathcal{BC}+$ for hypothetical/counterfactual reasoning, exploring parameter learning in the setting of probabilistic action language, and empirically studying the performance of $p\mathcal{BC}+$ with weal-world applications.

References

1 Joseph Babb and Joohyung Lee. Cplus 2ASP: Computing Action Language $\mathcal{C}+$ in Answer Set Programming. In *LPNMR*, 2013.

2 Joseph Babb and Joohyung Lee. Action language $\mathcal{BC}+$. *Journal of Logic and Computation*, page exv062, 2015. `doi:10.1093/logcom/exv062`.

3 Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3:425–461, 2003.

4 Chitta Baral, Michael Gelfond, and Nelson Rushton. Probabilistic Reasoning With Answer Sets. In *Logic Programming and Nonmonotonic Reasoning*, pages 21–33, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

5 Chitta Baral, Sheila Mcilraith, and Tran Son. Formulating Diagnostic Problem Solving Using an Action Language With Narratives and Sensing. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 311–322, April 2000.

6 Chitta Baral, Nam Tran, and Le-Chi Tuan. Reasoning about actions in a probabilistic setting. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 507–512, 2002.

7 Fabio Aurelio D'Asaro, Antonis Bikakis, Luke Dickens, and Rob Miller. Foundations for a Probabilistic Event Calculus. *CoRR*, abs/1703.06815, 2017. `arXiv:1703.06815`.

8 Thomas Eiter and Thomas Lukasiewicz. Probabilistic Reasoning about Actions in Nonmonotonic Causal Theories. In *Proceedings Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI-2003)*, pages 192–199. Morgan Kaufmann Publishers, 2003.

9 Michael Gelfond and Vladimir Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.

10 Michael Gelfond and Vladimir Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3:195–210, 1998. URL: `http://www.ep.liu.se/ea/cis/1998/016/`.

11 Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1–2):49–104, 2004.

12 Enrico Giunchiglia and Vladimir Lifschitz. An action language based on causal explanation: Preliminary report. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pages 623–630. AAAI Press, 1998.

13 Gero Iwan. History-based diagnosis templates in the framework of the situation calculus. *AI Communications*, 15(1):31–45, 2002.

14   Joohyung Lee, Vladimir Lifschitz, and Fangkai Yang. Action Language $\mathcal{BC}$: Preliminary Report. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

15   Joohyung Lee and Yunsong Meng. Answer Set Programming Modulo Theories and Reasoning about Continuous Changes. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, 2013.

16   Joohyung Lee, Samidh Talsania, and Yi Wang. Computing LPMLN using ASP and MLN solvers. *Theory and Practice of Logic Programming*, 2017. `doi:10.1017/S1471068417000400`.

17   Joohyung Lee and Yi Wang. Weighted Rules under the Stable Model Semantics. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 145–154, 2016.

18   Anastasios Skarlatidis, Georgios Paliouras, George A Vouros, and Alexander Artikis. Probabilistic event calculus based on markov logic networks. In *Rule-Based Modeling and Computing on the Semantic Web*, pages 155–170. Springer, 2011.

19   Håkan LS Younes and Michael L Littman. PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects, 2004.

20   Weijun Zhu. *PLOG: Its Algorithms and Applications.* PhD thesis, Texas Tech University, 2012.

Explaining Actual Causation via Reasoning About Actions and Change

Emily C. LeBlanc

College of Computing and Informatics
Drexel University
Philadelphia, PA
leblanc@drexel.edu

Abstract

In causality, an actual cause is often defined as an event responsible for bringing about a given outcome in a scenario. In practice, however, identifying this event alone is not always sufficient to provide a satisfactory explanation of how the outcome came to be. In this paper, we motivate this claim using well-known examples and present a novel framework for reasoning more deeply about actual causation. The framework reasons over a scenario and domain knowledge to identify additional events that helped to "set the stage" for the outcome. By leveraging techniques from Reasoning about Actions and Change, the approach supports reasoning over domains in which the evolution of the state of the world over time plays a critical role and enables one to identify and explain the circumstances that led to an outcome of interest. We utilize action language $\mathcal{AL}$ for defining the constructs of the framework. This language lends itself quite naturally to an automated translation to Answer Set Programming, using which, reasoning tasks of considerable complexity can be specified and executed. We speculate that a similar approach can also lead to the development of algorithms for our framework.

1 Introduction and Problem Description

The comprehensive goal of this research has been to design, evaluate, and implement a novel causal reasoning framework to discover causal explanations that are in closer agreement with what common sense might lead one to conclude. Identifying actual causation concerns determining how a specified consequence came to be in a given scenario and has long been studied in a diversity of fields, including law, philosophy, and, more recently, computer science. Also referred to as *causation in fact*, actual causation is a broad term that encompasses all possible antecedents that have played a meaningful role in producing the consequence [5]. Consider the well-known Yale Shooting problem [16]:

> *Shooting a turkey with a loaded gun will kill it. Suzy loads the gun and then shoots the turkey. Why is the turkey dead?*

Intuition tells us that Suzy's shooting of the turkey is the *actual cause* of its death. However, if we know for certain that the gun was not loaded at the start of the story, then it is also important to recognize that Suzy's loading the gun played a key role in producing this consequence. On the other hand, if the gun was loaded from the start, then this point may

not be as significant. Moreover, if we build upon this example to say that Tommy handed Suzy the gun at the start of the scenario, then surely we want to identify Tommy's action as a contributory cause of the turkey's death. Hall [11] gives another classic example of actual causation in which two actors have each thrown a rock at a bottle and we wish to determine which actor's throw caused the bottle to break. It is easy to imagine similar extensions to the example that require deeper reasoning about causation to properly explain how the bottle broke – for example, did a third actor instruct the original two to throw their rocks in the first place? Literature examples aside, sophisticated actual causal reasoning has been prevalent in human society and continues to have an undeniable impact on the advancement of science, technology, medicine, and other important fields. From the development of ancient tools to modern root cause analysis in business and industry, reasoning about causal influence in a historical sequence of events enables us to diagnose the cause of an outcome of interest and gives us insight into how to bring about, or even prevent, similar outcomes in future scenarios. Consider problems such as explaining the occurrence of a set of suspicious observations in a monitoring system, reasoning about the efficiency actions taken in an emergency evacuation scenario, or verifying how an automatically generated workflow produces the expected results. It is easy to imagine that in cases such as these, determining surface-level causation (e.g., Suzy shot the turkey) may not be sufficient to provide a satisfactory explanation of how an outcome of interest to be.

In this dissertation work, we claim that reasoning about actual causation in complex scenarios requires the ability to identify more than the existence of a causal relationship. We may want a deeper understanding of the causal mechanism – was the outcome caused directly or indirectly? Did previously occurring events somehow *support* the causing event or the outcome's ability to be caused? To this end, the overall goal of the dissertation work is to investigate and demonstrate the suitability of action language and answer set programming to design and realize a novel approach to automated reasoning about actual causation as described above. The framework leverages techniques from Reasoning about Actions and Change (RAC) to support reasoning over domains that change over time in response to a sequence of events, as well as to answer queries for detailed causal explanations of an *outcome of interest* in a specific scenario. The language of choice for the formalization of knowledge is action language $\mathcal{AL}$ [2] which enables us to represent our knowledge of the direct and indirect effects of actions in a domain.

In the remainder of this summary, we present background on the action language $\mathcal{AL}$ and its semantics, provide an overview of the framework and its behavior on a novel actual causation scenario, survey existing literature, and finally discuss open issues and expected achievements for the dissertation.

2   Preliminaries

As we have already described, this work leverages techniques from Reasoning about Actions and Change [20] to support reasoning over domains that change over time. We assume that knowledge of a domain exists as a set of causal laws called an *action description* describing direct and indirect effects of actions using the action language $\mathcal{AL}$ [2]. These causal laws embody a transition diagram describing all possible world states of the domain and the events that trigger transitions between them. In the thesis investigation, we assume the existence of knowledge in this form, and while the work describes the formalization of the domain descriptions, the matter of the origin of knowledge is beyond the scope of the thesis.

The syntax of $\mathcal{AL}$ builds upon an alphabet consisting of a set $\mathcal{F}$ of symbols for *fluents* and a set $\mathcal{E}$ of symbols for *events*[1]. The $\mathcal{AL}$ is centered around a discrete-state-based representation of the evolution of the domain.

Fluents are boolean properties of the domain whose truth value may change over time. A *(fluent) literal* is a fluent f or its negation $\neg f$. Additionally, we define $\overline{f} = \neg f$ and $\overline{\neg f} = f$. A statement of the form

$$e \text{ \textbf{causes} } l_0 \text{ \textbf{if} } l_1, l_2, \ldots, l_n \tag{1}$$

is called *dynamic causal law*, and intuitively states that, if event e in $\mathcal{E}$ occurs in a state in which literals $l_1, \ldots, l_n$ hold, then l_0, the *consequence* of the law, will hold in the next state. A statement

$$l_0 \text{ \textbf{if} } l_1, \ldots, l_n \tag{2}$$

is called *state constraint* and says that, in any state in which $l_1, \ldots, l_n$ hold, l_0 also holds. This second kind of statement allows for an elegant and concise representation of indirect effects, which increases the flexibility of the language. Finally, an *executability condition* is a statement of the form:

$$e \text{ \textbf{impossible\_if} } l_1, \ldots, l_n \tag{3}$$

where e and $l_1, \ldots, l_n$ are as above. (3) states that e cannot occur if $l_1, \ldots, l_n$ hold. A set of statements of $\mathcal{AL}$ is called an *action description*. The semantics of an action description AD is defined by its *transition diagram* $\tau(AD)$, a directed graph $\langle N, E \rangle$ such that:

1. N is the collection of all states of AD;
2. E is the set of all triples $\langle \sigma, e, \sigma' \rangle$ where σ, σ' are states, e is an event executable in σ, and σ, e, σ' satisfy the *successor state equation* [17]:

$$\sigma' = Cn_Z(E(e, \sigma) \cup (\sigma \cap \sigma')) \tag{4}$$

where Z is the set of all state constraints of AD.

The argument of Cn_Z in (4) is the union of the set of direct effects $E(e, \sigma)$ of e, with the set $\sigma \cap \sigma'$ of the facts "preserved by inertia". The application of Cn_Z adds the "indirect effects" to this union. A triple $\langle \sigma, e, \sigma' \rangle \in E$ is called a *transition* of $\tau(AD)$ and σ' is a *successor state of* σ (under e). A sequence $\langle \sigma_1, \alpha_1, \sigma_2, \ldots, \alpha_k, \sigma_{k+1} \rangle$ is a *path of* $\tau(D)$ of length k if every $\langle \sigma_i, \alpha_i, \sigma_{i+1} \rangle$ is a transition in $\tau(D)$. We refer to state σ_1 of a path p as the *initial state* of p. A path of length 0 contains only an initial state. In the next section, we build upon this formalization to define a query to our framework for representing and reasoning about actual cause.

3 Framework Overview and Foundational Example

In this section, we provide an overview of the causal reasoning framework alongside a novel foundational example that showcases the reasoning capabilities and explanatory power of the framework. It is a straightforward scenario in which an outcome of interest, say θ_E, is not satisfied at the start of the scenario. After the occurrence of three events, say e_1, e_2,

---

[1] For convenience and compatibility with the terminology from RAC, in this paper we use *action* and *event* as synonyms.

and e_3, the outcome has been caused. Given the outcome of interest, the sequence of events, and knowledge of the domain in which they have occurred, our framework identifies causal explanations for how θ_E may have come to be. In order to explain actual causation, we will aim to characterize *transition events* which tell us the primary cause of an outcome and whether or not it was caused directly or indirectly, as well as *outcome* and *supporting events* which tell us which prior occurring events have contributed to causing the outcome.

Query

A query consists of an action description, a sequence of events, and the outcome of interest. The sequence of three scenario events and the outcome of interest for our example are represented by $v_E = \langle e_1, e_2, e_3 \rangle$, and $\theta_E = \{A, B, C, D, E, F\}$, respectively. The following action description AD_E characterizes events in the scenario's domain:

$$
\begin{cases}
e_1 \textbf{ impossible\_if } A & (5) \\
e_1 \textbf{ causes } E \textbf{ if } \neg E & (6) \\
e_2 \textbf{ causes } D \textbf{ if } \neg D & (7) \\
e_3 \textbf{ causes } A \textbf{ if } \neg A & (8) \\
e_3 \textbf{ causes } C \textbf{ if } \neg C & (9) \\
e_3 \textbf{ impossible\_if } \neg E & (10) \\
e_3 \textbf{ impossible\_if } \neg F & (11) \\
B \textbf{ if } C & (12)
\end{cases}
$$

Laws (5) and (6) describe event e_1, telling us that e_1 can only occur when A does not hold and e_1 will cause E if it does not already hold. Law (7) states that e_2 will cause D to hold if it does not already hold. Similar to causal laws (6) and (7), laws (8) and (9) tell us that e_3 will cause A and C to hold if they do not hold. The executability conditions (10) and (11) state that e_3 can only occur when both E and F hold. Finally, the state constraint (12) tells us that B holds whenever C holds. Given the action description AD_E, the sequence of events v_E, and the outcome of interest θ_E, the triple $\mathcal{Q}_E = \langle AD_E, v_E, \theta_E \rangle$ is the *query* for our example. Next, we introduce the concept of a *scenario path*, a unique mapping of the scenario described by a query to a representation of how the state of the world has changed in response to the events.

Scenario Path

Scenario paths represent a unique unfolding of a scenario and provide a convenient representation of how the domain changes over time in response to the events of the scenario. We reason over these paths to explain actual causation.

▶ **Definition 1.** Given a query $\mathcal{Q} = \langle AD, v, \theta \rangle$, a *scenario path* is a path $\rho = \langle \sigma_1, \alpha_1, \sigma_2, ..., \alpha_k, \sigma_{k+1} \rangle$ of $\tau(AD)$ satisfying the following conditions:
1. $\forall i, 1 \leq i \leq k, \alpha_i = e_i$
2. $\theta \nsubseteq \sigma_1$
3. $\exists i, 1 < i \leq k + 1, \theta \subseteq \sigma_i$

Condition 1 requires that the events in ρ correspond to the events of v, capturing the idea that each event of v represents a transition between states in ρ. Condition 2 requires that the set of fluent literals θ is not satisfied by the initial state of ρ, ensuring that the

■ **Table 1** Tabular representation of the scenario path $\rho_E \in P(\mathcal{Q}_E)$.

State	Event	State Affecting Law(s)
$\sigma_1 = \{\neg A, \neg B, \neg C, \neg D, \neg E, F\}$	$\alpha_1 = e_1$	e_1 **causes** E **if** $\neg E$
$\sigma_2 = \{\neg A, \neg B, \neg C, \neg D, E, F\}$	$\alpha_2 = e_2$	e_2 **causes** D **if** $\neg D$
$\sigma_3 = \{\neg A, \neg B, \neg C, D, E, F\}$	$\alpha_3 = e_3$	e_3 **causes** A **if** $\neg A$, e_3 **causes** C **if** $\neg C$, B **if** C
$\sigma_4 = \{A, B, C, D, E, F\}$	–	–

outcome has not already been caused prior to the known events of the story. Condition 3 requires that θ is satisfied in at least one state after the initial state in ρ. Conditions 2 and 3 together ensure that at least one event is responsible for causing θ to hold in ρ. The successor state equation (4) tells us some event in the scenario path must have directly or indirectly caused θ to be satisfied at some point after the initial state. The set of all scenario paths with respect to the query $\mathcal{Q}$ is denoted by $P(\mathcal{Q}) = \{\rho_1, \rho_2, \ldots, \rho_m\}$.

It is clear that there are multiple valid scenario paths in the set $P(\mathcal{Q}_E)$, each representing a valid evolution of state in response to the scenario's events in the domain given by AD_E. For the purposes of this discussion, we choose a path with a complex causal mechanism that will exercise the causal reasoning framework. We will refer to this path as ρ_E. Table 1 shows the evolution of state in ρ_E in response to the events of v_E. The first column lists each state σ_i of ρ_E, and the second column gives the event α_i that caused a transition to the state σ_{i+1}. It is easy to see that ρ_E satisfies the conditions of Definition 1 with respect to AD_E, v_E, and θ_E.

Transition Event

A *transition event* is an event in a scenario path that causes a transition from a state of the world where the outcome θ is not satisfied to a state of the world where θ is satisfied. In this section, we identify transition events and their direct and indirect effects on the outcome.

▶ **Definition 2.** Given a scenario path $\rho = \langle \sigma_1, \alpha_1, \sigma_2, \ldots, \alpha_k, \sigma_{k+1} \rangle$ and an outcome θ, event α_j, where $1 \leq j \leq k$, is a *transition event* of θ in ρ if the following conditions are satisfied by the transition $\langle \sigma_j, \alpha_j, \sigma_{j+1} \rangle$ of ρ:

1. $\theta \nsubseteq \sigma_j$
2. $\theta \subseteq \sigma_{j+1}$

Intuitively, event α_j is a transition event of outcome θ if the outcome was not satisfied when α_j occurred but *was* satisfied after its occurrence. Note that we have defined transition events in such a way that there can be multiple transition events for θ in ρ. Using Table 1, it is straightforward to verify that event e_3 is the only transition event of θ_E in the example scenario path ρ_E, clearly satisfying Conditions 1 and 2 of Definition 2.

Given a query $\mathcal{Q} = \langle AD, v, \theta \rangle$, a scenario path $\rho = \langle \sigma_1, \alpha_1, \sigma_1, \ldots, \alpha_k, \alpha_{k+1} \rangle$ in $P(\mathcal{Q})$, and a transition event α_j for θ, the set of *direct effects of α_j in θ* is $d_\theta(\alpha_j, \rho) = \theta \cap E(\alpha_j, \sigma_j)$. Recall that $E(\alpha_j, \sigma_j)$ is the set of all direct effects of event α_j given that it occurs in state σ_j. The set of all direct effects of e_3 with respect to σ_3, then, is $E(e_3, \sigma_3) = \{A, C\}$, in accordance with laws (8) and (9) in AD_E. The direct effects of e_3 in θ_E, then, is given by $d_{\theta_E}(e_3, \rho_E) = \theta_E \cap E(e_3, \sigma_3) = \{A, B, C, D, E, F\} \cap \{A, C\} = \{A, C\}$.

To determine the indirect effects of an event with respect to the outcome, first let $S = E(\alpha_j, \sigma_j) \cup (\sigma_j \cap \sigma_{j+1})$ represent the set of all literals directly caused by the transition event α_j and those preserved by inertia. Given a query $\mathcal{Q} = \langle AD, v, \theta \rangle$, a scenario path

$\rho = \langle \sigma_1, \alpha_1, \sigma_1, \ldots, \alpha_k, \alpha_{k+1} \rangle$ in $P(\mathcal{Q})$, and a transition event α_j for θ, the set of *indirect effects of α_j in θ* is $i_\theta(\alpha_j, \rho) = \theta \cap (\sigma_{j+1} \setminus S)$. Given the set $S_E = E(e_3, \sigma_3) \cup (\sigma_3 \cap \sigma_4) = \{A, C\} \cup \{D, E, F\} = \{A, C, D, E, F\}$ representing the direct effects of e_3 and the literals preserved by inertia, the indirect effects of e_3 in θ_E is

$$
\begin{aligned}
i_{\theta_E}(e_3, \rho_E) =& \theta_E \cap (\sigma_4 \setminus S_E) \\
=& \{A, B, C, D, E, F\} \cap (\{A, B, C, D, E, F\} \setminus \{A, C, D, E, F\}) \\
=& \{A, B, C, D, E, F\} \cap \{B\} \\
=& \{B\}
\end{aligned}
$$

This result is intuitive because e_3 directly caused C to hold by law (9) and we know from law (12) that whenever C holds in a certain state, then B holds. We claim that under these conditions, it must be the case the e_3 caused B indirectly.

First Causal Explanation

Both the knowledge of the transition event and its effects on the outcome are represented by the *first causal explanation*. Given the query $\mathcal{Q}_E = \langle AD_E, v_E, \theta_E \rangle$, the scenario path $\rho_E \in P(\mathcal{Q}_E)$, the transition event e_3 in ρ_E, and e_3's direct and indirect effects, $d_{\theta_E}(\rho_E, \theta_E)$ and $i_{\theta_E}(\rho_E, \theta_E)$, respectively, the *first causal explanation* for θ_E in ρ_E is the tuple

$$
\begin{aligned}
C_E^1 &= \langle \rho_E, e_3, d_{\theta_E}(\rho_E, \theta_E), i_{\theta_E}(\rho_E, \theta_E) \rangle \\
&= \langle \rho_E, e_3, \{A, C\}, \{B\} \rangle
\end{aligned}
$$

Explanation C_E^1 summarizes our initial findings – the event e_3 caused a transition from a state where the outcome $\{A, B, C, D, E, F\}$ did not hold to a state where it did hold in the scenario path ρ_E. Specifically, literals A and C were direct effects of e_3's occurrence while e_3 caused B indirectly.

While C_E^1 tells us how the set of literals $\{A, B, C\}$ of θ_E were made to hold in scenario path ρ_E, we are still missing information about which, if any, events prior to e_3 caused the remaining literals $\{D, E, F\}$ to hold in state σ_4. We also do not know if any prior occurring events influenced e_3's ability to be a transition event of θ_E. In this work, supporting events are events that have occurred prior to a transition event α_j that enable α_j to be a transition event for the outcome θ. We identify two types of supporting events, *outcome supporting event* (OSEs) and *transition supporting events* (TSEs), both which are presented in the following sections. In order to identify both OSEs and TSEs in a scenario path ρ, we must first introduce the notion that an event α_i *ensures* that a literal l will hold in a specified state σ_j if it is the most recent transition event for l.

▶ **Definition 3.** Given a scenario path $\rho = \langle \sigma_1, \alpha_1, \sigma_2, \ldots, \alpha_k, \alpha_{k+1} \rangle$, event α_i is an *ensuring event* of $l \in \sigma_j$ in ρ if:
1. α_i is a transition event of $\{l\}$ in ρ
2. $i < j$
3. $j - i$ is minimal

Condition 1 leverages Definition 2 to require that event α_i responsible for l holding in some state of ρ. Condition 2 requires that α_i occurs before α_j in ρ. Condition 3 requires that α_i is the most recent transition event of l in ρ. We claim that if no event ensures $l \in \sigma_j$ for a path ρ, this implies that l holds in every state of ρ because there exists no transition $\langle \sigma_i, \alpha_i, \sigma_{i+1} \rangle$ in the path such that $l \notin \sigma_i$. Therefore, l must have held in the initial state and

was never changed by a subsequent event prior to α_j's occurrence. Note that because ensuring events are also transition events, it is straightforward to leverage the characterizations of direct and indirect effects of transition events from Section 3 to learn if events ensured l in some state σ due to its direct or indirect effects.

Outcome Supporting Events

In the case where α_j does not set all of the literals of θ, OSEs can be responsible for ensuring that these remaining literals hold by the time α_j occurs in ρ. Finding OSEs requires first identifying if any literals in θ were not set as an effect of the transition event α_j. The set of remaining literals of an outcome θ is given by $R_\theta = \theta \setminus (d_\theta(\alpha_j, \rho) \cup i_\theta(\alpha_j, \rho))$. If $|R_\theta| > 0$, then a previously occurring event may have supported the outcome θ by ensuring that the remaining literals held in state σ_{j+1}.

▶ **Definition 4.** Given a query $\mathcal{Q}$, a factual path $\rho \in P(\mathcal{Q})$, a transition event α_j of θ, and a literal $l \in R_\theta$, α_i is an *outcome supporting event (OSE) via l* if α_i ensures $l \in \sigma_{j+1}$.

We denote by O^{supp} the set of OSEs and the literals they ensure. Formally, the tuple $\langle \alpha_i, l \rangle \in O^{supp}$ if α_i is a OSE via l. We denote by O^{init} the set of literals in R_θ that were not ensured by an event in ρ. Given a literal $l \in R_\theta$, $l \in O^{init}$ if:

$$\neg \exists \langle \alpha, l' \rangle \in O^{supp} \ s.t. \ l' = l$$

Intuitively, a literal l is in O^{init} when l has is no outcome supporting event in O^{supp}. In our example, we already know that we require additional causal information about the set of remaining outcome literals D, E, and F. Formally, the following literals in the outcome θ_E have not been explained by C_E^1:

$$\begin{aligned}
R_{\theta_E} &= \theta_E \setminus (d_{\theta_E}(e_3, \rho_E) \cup i_{\theta_E}(e_3, \rho_E)) \\
&= \{A, B, C, D, E, F\} \setminus (\{A, C\} \cup \{B\}) \\
&= \{A, B, C, D, E, F\} \setminus \{A, C, B\} \\
&= \{D, E, F\}
\end{aligned}$$

Because $|R_{\theta_E}| > 0$, there is more causal information to uncover. As covered in the earlier discussion on ensuring events, each literal in R_{θ_E} must either be ensured to hold in state σ_4 by an outcome supporting event or the literal has held consistently from the start of the scenario. Event e_2 is an outcome supporting event because it ensures that literal D held in σ_4. This event meets the three conditions of ensuring $D \in \sigma_4$. First, it is a transition event of $\{D\}$ because the literal D did not hold in state σ_2 but it did hold in σ_3 after e_2's occurrence. It clearly satisfies Conditions 2 because here $i = 2$ and $j = 4$, and so $i < j$. Finally, it satisfies Condition 3 because event e_i is the most recent transition event of $\{D\}$, and so $j - i$ is minimal. Similarly, it is straightforward to verify that e_1 is an outcome supporting event by ensuring that E holds in state σ_4. The set of outcome supporting events is given by $O_E^{supp} = \{\langle e_2, D \rangle, \langle e1, E \rangle\}$. Finally, the set $O_E^{supp} = \{F\}$ because there exists no tuple $\langle \alpha, F \rangle \in O_E^{supp}$, and so F must have held in the initial state of ρ_E and never changed value.

Second Causal Explanation

Knowledge of outcome supporting events and remaining outcome literals that held from the start is represented by the *second causal explanation*. Given the query $\mathcal{Q}_E = \langle AD_E, v_E, \theta_E \rangle$,

the scenario path $\rho_E \in P(\mathcal{Q}_\mathcal{E})$, and the transition event e_3 for θ_E, the *second causal explanation* for θ_E in ρ_E is

$$C_E^2 = \langle O_E^{supp}, O_E^{init} \rangle$$
$$= \langle \{\langle e_2, D \rangle, \langle e_1, E \rangle\}, \{F\} \rangle$$

Explanation C_E^2 provides us with information about how the remaining outcome literals $\{D, E, F\} \in \theta_E$ came to hold in the state σ_4. Of these remaining literals, D and E were ensured by events e_2 and e_1, respectively. The remaining literal F held in the initial state and was not ensured in σ_4 by any event prior to e_1.

C_E^2 tells us how the remaining outcome literals came to hold in σ_4, but there is even more causal information to be revealed in this example. Next, we discuss an approach to determining if any other events in scenario path ρ_E contributed to e_3's ability to be a transition event of θ_E.

Transition Supporting Events

TSEs ensure that the preconditions of α_j are satisfied in state σ_j so that α_j could occur and cause θ to be satisfied in σ_{j+1}. The approach to identifying TSEs is conveniently similar to identifying outcome supporting events, and so we will omit the majority of technical details in favor of working out the example in the interest of space. To determine whether or not any prior events supported the transition event e_3, we begin by identifying all preconditions for e_3's occurrence and its ability to produce its effects in ρ_E. We obtain α_j's *preconditions* in ρ by reasoning over the of laws in AD. In the dissertation work, we introduce notation to allow reasoning over the components of laws in an action description AD. For example, given a dynamic causal law λ in AD of form (1), let $e(\lambda) = e$, $c(\lambda) = l_0$, and $p(\lambda) = \{l_1, l_2, \ldots, l_n\}$. We denote by $\mathcal{D}(AD)$ the set of all dynamic causal laws in AD. We use a similar representation for executability conditions, and we introduce a set of conditions under which preconditions can be extracted from these laws. In our example, the literals $\neg A$ and $\neg C$ are in $prec(e_3, \rho_E)$ because of laws (8) and (9) in the action description AD_E. By our definition of precondition, the literals E and F are also in $prec(e_3, \rho_E)$ because of laws (10) and (11) in AD_E. Therefore, the set of preconditions of e_3 in ρ_E is $prec(e_3, \rho_E) = \{\neg A, \neg C, E, F\}$.

Similar to our definition of outcome supporting events, a *transition supporting event* is the most recent transition event for a precondition of the transition event. It is straightforward to verify that the set of transition supporting events is given by $T_E^{supp} = \langle e_1, E \rangle$ and the set of initially set literals is $T_E^{init} = \{\neg A, \neg C, F\}$.

Third Causal Explanation

Knowledge of transition supporting events and precondition literals that held from the start is represented by the *third causal explanation*. Given the scenario path $\rho_E \in P(\mathcal{Q}_\mathcal{E})$, the transition event e_3, the set of transition supporting events T_E^{supp}, and the set of uncaused literals T_E^{init} the *third causal explanation* for θ_E in ρ_E is

$$C_E^3 = \langle T_E^{supp}, T_E^{init} \rangle$$
$$= \langle \{\langle e_1, E \rangle\}, \{\neg A, \neg C, F\} \rangle$$

Explanation C_E^3 tells us about the transition event e_3's preconditions and how they were met by state σ_3. The preconditions literals of event e_3 were $\neg A$, $\neg C$, E, and F. Of these precondition literals, E was ensured in σ_3 by the occurrence of event e_1. The remaining

literals $\neg A$, $\neg C$, and F were not ensured in σ_3 by any scenario event. For relative brevity, we will not query further for details about the outcome and transition supporting events. It is easy to see, however, that the framework could tell us that the precondition literal E for e_3 was made to hold as a *direct effect* of e_1's occurrence.

Actual Causal Explanation

As the research intends to prove, there exists a space of possible structures for causal explanation. Recall that when there are remaining outcome literals to explain, there is a second causal explanation. However, if a transition event has no preconditions in the scenario path, then there is no third causal explanation. This implies that the structure of the explanation depends on the information encoded by the corresponding scenario path. We intend to characterize this space of structures in the dissertation. The framework can identify all three causal explanations in our example (i.e., C_E^1, C_E^2, and C_E^3). To summarize, the framework has explained that e_3 was a transition event for θ_E through both direct and indirect effects, e_1 and e_2 were outcome supporting events, and e_1 was a transition supporting event in the scenario path ρ_E.

4   Overview of Existing Literature

While actual causation has been treated in numerous ways in the Artificial Intelligence literature, the most relevant of which we will cover briefly in this section, existing approaches do not possess the fine-granularity of reasoning and explanation required to meet the reasoning needs of the examples discussed here. Many approaches to reasoning about actual cause have been inspired by the human intuition that cause can be determined by hypothesizing about whether or not a removing X from a scenario would prevent Y from being true [19]. Attempts to mathematically characterize actual causation have largely pursued counterfactual analysis of structural equations [22, 13, 15], neuron diagrams [12], and other logical formalisms [18, 23, 4]. It has been widely documented, however, that the counterfactual criteria alone is problematic and fails to recognize causation in some common cases such as preemption, overdetermination, and contributory cause [21, 10]. More recent approaches such as [14] have addressed some of these shortcomings by modifying the existing definitions of actual cause or by modeling change over time with some improved results. However, there is still no widely agreed upon counterfactual definition of actual cause in spite of a considerably large body of work aiming to find one.

The work of [3] departs from the counterfactual approach, using a similar insight to our own that actual causation can be determined by inspecting a specific scenario. Leveraging the Situation Calculus (SC) to formalize knowledge, the approach uses a single step regression approach to identify events deemed relevant to a logical statement becoming true. Although the conceptual approach is similar to our own, the technical approaches differ significantly. For example, [3] identifies a single sequence of causal events without explanation. There are also ramifications due to the choices for the formalization of the domain. Compared to $\mathcal{AL}$ formalizations, SC formalizations incur limitations when it comes to the representations of indirect effects of actions, which play an essential role in our work, and the elaboration tolerance of the formalization. Additionally, SC relies on First-Order Logic, while $\mathcal{AL}$ features an independent and arguably simpler semantics.

5   Open Issues and Expected Achievements

While the core of this framework is fairly well-developed at this stage, there remain some open issues that will be addressed in the dissertation. Evaluation of the framework is a crucial next step, and meaningful progress has been made towards demonstrating the framework's reasoning process when solving examples from causality literature in addition to novel scenarios. We expect to demonstrate that the framework can solve numerous classic examples with finer-grained causal explanations than the current state of the art. Moreover, the dissertation will present a number of empirical studies to compare and evaluate the ability of related approaches to solve the novel example presented in this paper. We expect that related approaches will not be able to explain the causal mechanism of our example in comparable detail. The dissertation will also present a novel set of identified open problems whose investigation can advance the capabilities of the causal reasoning framework. Regarding implementation, the choice of $\mathcal{AL}$ as the underlying formalism has useful practical implications. As demonstrated by a substantial body of literature (see, e.g., [1]), $\mathcal{AL}$ lends itself quite naturally to an automated translation to Answer Set Programming [8, 9], using which, complex reasoning tasks can be specified and executed (see, e.g., [6, 7]). We speculate that a similar approach can also lead to the development of algorithms for our framework, and have begun translating $\mathcal{AL}$ queries, scenario paths, and transition events to ASP.

───── **References** ─────

1   Marcello Balduccini and Michael Gelfond. Diagnostic reasoning with A-Prolog. *arXiv preprint cs/0312040*, 2003.

2   Chitta Baral and Michael Gelfond. Reasoning agents in dynamic domains. In *Logic-based artificial intelligence*, pages 257–279. Springer, 2000.

3   Vitaliy Batusov and Mikhail Soutchanski. Situation calculus semantics for actual causality. In *13th International Symposium on Commonsense Reasoning. University College London, UK. Monday, November*, volume 6, 2017.

4   Sander Beckers and Joost Vennekens. A general framework for defining and extending actual causation using CP-logic. *International Journal of Approximate Reasoning*, 77:105–126, 2016.

5   Charles E Carpenter. Concurrent Causation. *University of Pennsylvania Law Review and American Law Register*, 83(8):941–952, 1935.

6   Thomas Eiter, Wolfgang Faber, Nicola Leone, Gerald Pfeifer, and Axel Polleres. Answer set planning under action costs. *Journal of Artificial Intelligence Research*, 19:25–71, 2003.

7   Esra Erdem, Michael Gelfond, and Nicola Leone. Applications of Answer Set Programming. *AI Magazine*, 37(3), 2016.

8   Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

9   Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

10   Clark Glymour and David Danks. Actual causation: a stone soup essay. *Synthese*, 175(2):169–192, 2010.

11   Ned Hall. Two concepts of causation. *Causation and counterfactuals*, pages 225–276, 2004.

12   Ned Hall. Structural equations and causation. *Philosophical Studies*, 132(1):109–136, 2007.

13   Joseph Y Halpern. Axiomatizing causal reasoning. *Journal of Artificial Intelligence Research*, 12:317–337, 2000.

14   Joseph Y Halpern. *Actual causality*. MIT Press, 2016.

15   Joseph Y Halpern and Judea Pearl. Causes and explanations: A structural-model approach. Part I: Causes. *The British journal for the philosophy of science*, 56(4):843–887, 2005.

16   Steve Hanks and Drew McDermott. Nonmonotonic logic and temporal projection. *Artificial intelligence*, 33(3):379–412, 1987.

17   Patrick J. Hayes and John McCarthy. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 463–502. Edinburgh University Press, 1969.

18   Mark Hopkins and Judea Pearl. Causality and counterfactuals in the situation calculus. *Journal of Logic and Computation*, 17(5):939–953, 2007.

19   David Lewis. Causation. *The journal of philosophy*, 70(17):556–567, 1974.

20   J. McCarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence*, pages 431–450, 1969.

21   Peter Menzies. Counterfactual theories of causation. *The Stanford Encyclopedia of Philosophy*, 2001.

22   Judea Pearl. On the definition of actual cause, 1998.

23   Joost Vennekens. Actual causation in CP-logic. *Theory and Practice of Logic Programming*, 11(4-5):647–662, 2011.

Translating P-log, LP^{MLN}, LPOD, and CR-Prolog2 into Standard Answer Set Programs

Zhun Yang

School of Computing, Informatics, and Decision Systems Engineering, Arizona State University
Arizona State University, P.O. Box 878809, Tempe, AZ 85287, United States
zyang90@asu.edu

───── **Abstract** ─────

Answer set programming (ASP) is a particularly useful approach for nonmonotonic reasoning in knowledge representation. In order to handle quantitative and qualitative reasoning, a number of different extensions of ASP have been invented, such as quantitative extensions LP^{MLN} and P-log, and qualitative extensions LPOD, and CR-Prolog$_2$.

Although each of these formalisms introduced some new and unique concepts, we present reductions of each of these languages into the standard ASP language, which not only gives us an alternative insight into the semantics of these extensions in terms of the standard ASP language, but also shows that the standard ASP is capable of representing quantitative uncertainty and qualitative uncertainty. What's more, our translations yield a way to tune the semantics of LPOD and CR-Prolog$_2$. Since the semantics of each formalism is represented in ASP rules, we can modify their semantics by modifying the corresponding ASP rules.

For future work, we plan to create a new formalism that is capable of representing quantitative and qualitative uncertainty at the same time. Since LPOD rules are simple and informative, we will first try to include quantitative preference into LPOD by adding the concept of weight and tune the semantics of LPOD by modifying the translated standard ASP rules.

1 Introduction and Problem Description

In answer set programming, each answer set encodes a solution to the problem that is being modeled. There is often a need to express how likely a solution is, so several extensions of answer set programs, such as LP^{MLN} [19] and P-log [7], were made to express a quantitative uncertainty for each answer set. LP^{MLN} extends answer set programs by adopting the log-linear weight scheme of Markov Logic. P-log is a probabilistic extension of ASP with sophisticated semantics. Similarly, since there is often a need to express that one solution is preferable to another, several extensions of answer set programs, such as Logic Programs with Ordered Disjunction (LPOD) [8], CR-Prolog [5], and CR-Prolog$_2$ [6], were made to express a qualitative preference over answer sets. In LPOD, the qualitative preference is introduced by the construct of ordered disjunction in the head of a rule: $A \times B \leftarrow Body$ intuitively means, when $Body$ is true, if possible then A, but if A is not possible, then at least B. CR-Prolog$_2$ also has order rules as in LPOD, and it introduces consistency-restoring rules – rules that can be added only when they can make an inconsistent program consistent.

It remains an open question whether these formalisms can be reduced back to standard answer set programs. In other words, whether ASP is expressive enough to express the semantics of all these extensions? There were few attentions to this question where no positive answer had been proposed. Lee et al. [19] showed that a subset of P-log can be represented by LP^{MLN}, which is very similar to ASP except the introducing of weight for each rule. However, the feature of dynamic probability assignment in P-log is not preserved, and the reduction from LP^{MLN} to ASP was still unclear. Proposition 2 from [8] states that there is no reduction of LPOD to disjunctive logic programs [17] based on the fact that the answer sets of disjunctive logic programs are subset-minimal whereas LPOD answer sets are not necessarily so. However, this justification is limited to translations that preserve the underlying signature. Indeed, our paper "LP^{MLN}, Weak Constraints, and P-log" [20] and our ICLP paper that is being evaluated provides a positive answer to this question.

We present a reduction of P-log to LP^{MLN} and a reduction of LP^{MLN} to answer set programs with weak constraints. These translations show how the weights in the weak constraints can be used to denote quantitative uncertainty and, further, to represent probabilities. We also present a reduction of LPOD and CR-Prolog$_2$ to standard answer set programs by compiling away ordered disjunctions and consistency-restoring rules. These translations show how qualitative uncertainty is handled by the "definition" rules in ASP.

Since our research shows that ASP is capable of representing quantitative and qualitative uncertainty, it naturally follows a question that: can we combine quantitative uncertainty and qualitative preference in a single formalism? We are looking forward to answering this question in our future work.

The paper will give a summary of my research, including some background knowledge and reviews of existing literature (Section 2), goal of my research (Section 3), the current status of my research (Section 4), the preliminary results we accomplished (Section 5), and some open issues and expected achievements (Section 6).

2   Background and Overview of the Existing Literature

We only review the syntax and semantics of LP^{MLN} and LPOD. Please refer to [7] and [6] for the syntax and semantics of P-log and CR-Prolog$_2$, whose semantics are all based on a long translation to answer set programs.

2.1   Review: LP^{MLN}

We review the definition of LP^{MLN} from [19]. In fact, we consider a more general syntax of programs than the one from [19], but this is not an essential extension. We follow the view of [15] by identifying logic program rules as a special case of first-order formulas under the stable model semantics. For example, rule $r(x) \leftarrow p(x), not\ q(x)$ is identified with formula $\forall x(p(x) \land \neg q(x) \rightarrow r(x))$. An LP^{MLN} program is a finite set of weighted first-order formulas $w : F$ where w is a real number (in which case the weighted formula is called *soft*) or α for denoting the infinite weight (in which case it is called *hard*). An LP^{MLN} program is called *ground* if its formulas contain no variables. We assume a finite Herbrand Universe. Any LP^{MLN} program can be turned into a ground program by replacing the quantifiers with multiple conjunctions and disjunctions over the Herbrand Universe. Each of the ground instances of a formula with free variables receives the same weight as the original formula.

For any ground LP^{MLN} program Π and any interpretation I, $\overline{\Pi}$ denotes the unweighted formula obtained from Π, and Π_I denotes the set of $w : F$ in Π such that $I \models F$, and SM[Π] denotes the set $\{I \mid I$ is a stable model of $\overline{\Pi_I}\}$ (We refer the reader to the stable model

semantics of first-order formulas in [15]). The *unnormalized weight* of an interpretation I under Π is defined as LP^{MLN}

$$W_\Pi(I) = \begin{cases} exp\left(\sum\limits_{w:F \,\in\, \Pi_I} w\right) & \text{if } I \in \text{SM}[\Pi]; \\ 0 & \text{otherwise.} \end{cases}$$

The *normalized weight* (a.k.a. *probability*) of an interpretation I under Π is defined as

$$P_\Pi(I) = \lim_{\alpha \to \infty} \frac{W_\Pi(I)}{\sum\limits_{J \in \text{SM}[\Pi]} W_\Pi(J)}.$$

I is called a *(probabilistic) stable model* of Π if $P_\Pi(I) \neq 0$.

2.2 Review LPOD

We review the definition of LPOD from [8], which assumes propositional programs.

Syntax. A (propositional) LPOD Π is $\Pi_{reg} \cup \Pi_{od}$, where its *regular part* Π_{reg} consists of usual ASP rules $Head \leftarrow Body$, and its *ordered disjunction part* Π_{od} consists of *LPOD rules* of the form

$$C^1 \times \cdots \times C^n \leftarrow Body \tag{1}$$

in which C^i are atoms, n is at least 2, and *Body* is a conjunction of atoms possibly preceded by *not*.[1] Rule (1) says "when *Body* is true, if possible then C^1; if C^1 is not possible then C^2; ...; if all of $C^1, \ldots, C^{n-1}$ are not possible then C^n".

Semantics. For an LPOD rule (1), its *i-th option*, where $i \in \{1, \ldots, n\}$, is defined as $C^i \leftarrow Body, not\ C^1, \ldots, not\ C^{i-1}$.

Let Π be an LPOD. A *split program* of Π is obtained from Π by replacing each rule in Π_{od} by one of its options. A set S of atoms is a *candidate answer set* of Π if it is an answer set of a split program of Π. A split program of Π may be inconsistent (i.e., may not necessarily have an answer set).

A candidate answer set S of Π is said to *satisfy* rule (1)

- to degree 1 if S does not satisfy *Body*;
- to degree j ($1 \leq j \leq n$) if S satisfies *Body* and $j = min\{k \mid C^k \in S\}$.

For a set S of atoms, let $S^i(\Pi)$ denote the set of rules in Π_{od} satisfied by S to degree i. For candidate answer sets S_1 and S_2 of Π, [9] introduces the following four preference criteria.

1. **Cardinality-Preferred:** S_1 is *cardinality-preferred* to S_2 ($S_1 >^c S_2$) if there is a positive integer i such that $|S_1^i(\Pi)| > |S_2^i(\Pi)|$, and $|S_1^j(\Pi)| = |S_2^j(\Pi)|$ for all $j < i$.
2. **Inclusion-Preferred:** S_1 is *inclusion-preferred* to S_2 ($S_1 >^i S_2$) if there is a positive integer i such that $S_2^i(\Pi) \subset S_1^i(\Pi)$, and $S_1^j(\Pi) = S_2^j(\Pi)$ for all $j < i$.

---

[1] In [8], a usual ASP rule is viewed as a special case of a rule with ordered disjunction when $n = 1$ but in this paper, we distinguish them. This simplifies the presentation of the translation and also allows us to consider LPOD programs that are more general than the original definition by allowing modern ASP constructs such as aggregates.

3. **Pareto-Preferred:** S_1 is *pareto-preferred* to S_2 ($S_1 >^p S_2$) if there is a rule that is satisfied to a lower degree in S_1 than in S_2, and there is no rule that is satisfied to a lower degree in S_2 than in S_1.

4. **Penalty-Sum-Preferred:** S_1 is *penalty-sum-preferred* to S_2 ($S_1 >^{ps} S_2$) if the sum of the satisfaction degrees of all rules is smaller in S_1 than in S_2.

A set S of atoms is a *k-preferred* ($k \in \{c, i, p, ps\}$) *answer set* of an LPOD Π if S is a candidate answer set of Π and there is no candidate answer set S' of Π such that $S' >^k S$.

2.3   Existing Literature

There are quite a lot of formalisms made to represent quantitative uncertainty.

LP^{MLN} [19] is a probabilistic logic programming language that extends answer set programs [16] with the concept of weighted rules, whose weight scheme is adopted from that of Markov Logic [23], a probablistic extension of SAT. It is shown in [19, 18] that LP^{MLN} is expressive enough to embed Markov Logic and several other probabilistic logic languages, such as ProbLog [13], Pearls' Causal Models [22], and a fragment of P-log [7]. On the other hand, [2] proposed an embedding from LP^{MLN} into P-log.

Another famous quantitative extension of ASP are weak constraints [12], which are to assign a quantitative preference over the stable models of non-weak constraint rules: weak constraints cannot be used for deriving stable models.

Many formalisms are made to represent qualitative uncertainty. Most of them are extensions of ASP, where their semantics or implementations are also based on answer set programs.

In [11], LPOD is implemented using SMODELS. The implementation interleaves the execution of two programs–a generator which produces candidate answer sets and a tester which checks whether a given candidate answer set is maximally preferred or produces a more preferred candidate if it is not. An implementation of CR-Prolog reported in [3] uses a similar algorithm.

[14] finds the "order preserving answer sets" of an ordered logic program (where a strict partial order is assigned among some rules) by meta-programming. Our translations are similar to the meta-programming approach to handle preference in ASP in that we turn LPOD and CR-Prolog$_2$ into answer set programs that do not have the built-in notion of preference.

In contrast, the reductions shown in this paper can be computed by calling an answer set solver one time without the need for iterating the generator and the tester. This feature may be useful for debugging LPOD and CR-Prolog$_2$ programs because it allows us to compare all candidate and preferred answer sets globally.

Asprin [10] provides a flexible way to express various preference relations over answer sets and is implemented in CLINGO. Similar to the existing LPOD solvers, CLINGO makes iterative calls to find preferred answer sets, unlike the one-shot execution as we do.

In [1], Asuncion *et al.* extended propositional LPODs to the first order case, where the candidate answer sets of a first order LPOD can be obtained by finding the models of a second-order formula.

3 Goal of the Research

The following are our research objectives.

- **Find a translation *plog2asp* from P-log to answer set programs.** We design a one-time translation *plog*2*asp* such that for any P-log Π, the answer sets of the answer set program *plog*2*asp*(Π) agree with (i.e., their explanation to the domain are the same) the possible worlds of Π.

- **Find a translation *lpmln2asp* from LP^{MLN} to answer set programs.** We design a one-time translation *lpmln2asp* such that for any LP^{MLN} program Π, the answer sets of the answer set program *lpmln2asp*(Π) agree with the probabilistic answer sets of Π.

- **Analyze how quantitative uncertainty can be expressed in standard answer set programs.** We compare the two translations *plog*2*asp* and *lpmln2asp*, and analyze how quantitative uncertainty represented by weight (in LP^{MLN}) and sophisticated probability assignment (in P-log) can be expressed in standard answer set programs.

- **Find a translation *lpod2asp* from LPOD to answer set programs.** We design a one-time translation *lpod2asp* such that for any LPOD Π, the optimal answer sets of the answer set program lpod2asp(Π) "report" all the candidate answer sets of Π in different name spaces and whether each of them is a preferred answer set.

- **Find a translation *crpt2asp* from CR-Prolog$_2$ to answer set programs.** We design a one-time translation *crpt*2*asp* such that for any CR-Prolog$_2$ program Π, the optimal answer sets of the answer set program *crpt*2*asp*(Π) "report" all the generalized answer sets of Π in different name spaces and whether each of them is also a candidate answer sets or a preferred answer sets.

- **Analyze how qualitative uncertainty can be expressed in standard answer set programs.** We compare the two translations *lpod*2*asp* and *crpt*2*asp*, and analyze how qualitative preference represented by ordered disjunction and consistency-restoring rules can be expressed in standard answer set programs.

- **Design a single formalism to represent both quantitative and qualitative uncertainty.** We design a new formalism that can be used to represent quantitative and qualitative uncertainty at the same time. The semantics of the new formalism is defined as a reduction to standard answer set programs as we did for those four formalisms.

4 Current Status of the Research

This research is at a middle phase.

The first 2 bullets of our goals are done in our paper accepted by AAAI 2017 [20], where we proposed a translation *plog*2*lpmln* from P-log to LP^{MLN}, and a translation *lpmln*2*wc* from LP^{MLN} to answer set programs with weak constraints. The translations *lpod*2*asp* and *crpt*2*asp* are also completed in our paper accepted by ICLP 2018 [21]. We also compared all these four translations and have some ideas about how standard answer set programs handle quantitative and qualitative uncertainty.

Currently, we are testing our ideas by introducing quantitative uncertainty into LPOD. The experiments are based on our reduction from LPOD to answer set programs. We are tuning the semantics of LPOD by modifying on the translated rules.

In this section, we will present our main theorems, along with some examples to illustrate how our translations work.

5.1   From LP^{MLN} to Answer Set Programs

▶ **Theorem 1.** *(from [20]) For any LP^{MLN} program Π, the most probable stable models (i.e., the stable models with the highest probability) of Π are precisely the optimal stable models of the program with weak constraints* $\mathsf{lpmln2wc}(\Pi)$.

▶ **Example 2.** Consider the LP^{MLN} program Π_1 in Example 1 from [19].

$$\begin{array}{llr}
\alpha : & Bird(Jo) \leftarrow ResidentBird(Jo) & (r1) \\
\alpha : & Bird(Jo) \leftarrow MigratoryBird(Jo) & (r2) \\
\alpha : & \bot \leftarrow ResidentBird(Jo), MigratoryBird(Jo) & (r3) \\
2 : & ResidentBird(Jo) & (r4) \\
1 : & MigratoryBird(Jo) & (r5)
\end{array}$$

The (simplified) translation $\mathsf{lpmln2wc}(\Pi_1)$ is as follows, which simply removes α from each hard rule and turns each soft rule into a choice rule and a weak constraint.

$$\begin{array}{l}
Bird(Jo) \leftarrow ResidentBird(Jo) \\
Bird(Jo) \leftarrow MigratoryBird(Jo) \\
\bot \leftarrow ResidentBird(Jo), MigratoryBird(Jo) \\
\{ResidentBird(Jo)\}^{\mathrm{ch}} \\
\{MigratoryBird(Jo)\}^{\mathrm{ch}}
\end{array}$$

$$\begin{array}{ll}
{:}{\sim}\ ResidentBird(Jo) & [-2@0] \\
{:}{\sim}\ MigratoryBird(Jo) & [-1@0]
\end{array}$$

There are three probabilistic stable models of Π_1: $\emptyset$, $\{Bird(Jo), ResidentBird(Jo)\}$, and $\{Bird(Jo), MigratoryBird(Jo)\}$. Among them, $\{Bird(Jo), ResidentBird(Jo)\}$ is the most probable stable model of Π_1 since it is associated with a highest weight. It is also an optimal stable model of $\mathsf{lpmln2wc}(\Pi_1)$ since it has the least penalty -2 at level 0.

5.2   From P-log to LP^{MLN}

▶ **Theorem 3.** *(from [20]) Let Π be a consistent P-log program. There is a 1-1 correspondence ϕ between the set of the possible worlds of Π with non-zero probabilities and the set of probabilistic stable models of* $\mathsf{plog2lpmln}(\Pi)$.

▶ **Example 4.** Consider a variant of the Monty Hall Problem encoding in P-log from [7] to illustrate the probabilistic nonmonotonicity in the presence of assigned probabilities. There are four doors, behind which are three goats and one car. The guest picks door 1, and Monty, the show host who always opens one of the doors with a goat, opens door 2. Further, while the guest and Monty are unaware, the statistics is that in the past, with 30% chance the prize was behind door 1, and with 20% chance, the prize was behind door 3. Is it still better to switch to another door? This example can be formalized in P-log program Π_2, using both

assigned probability and default probability, as

$$\sim CanOpen(d) \leftarrow Selected = d. \quad (d \in \{1, 2, 3, 4\})$$
$$\sim CanOpen(d) \leftarrow Prize = d.$$
$$CanOpen(d) \leftarrow not \ \sim CanOpen(d).$$
$$random(Prize). \qquad random(Selected).$$
$$random(Open : \{x : CanOpen(x)\}).$$
$$pr(Prize = 1) = 0.3. \quad pr(Prize = 3) = 0.2.$$
$$Obs(Selected = 1). \ Obs(Open = 2). \ Obs(Prize \neq 2).$$

Intuitively, the translation $\mathsf{plog2lpmln}(\Pi_2)$ (i) reifies each atom $c = v$ in Π_2 into a form of $Poss(c = v)$, $PossWithAssPr(c = v)$, and $PossWithDefPr(c = v)$; (ii) defines each of these reified atoms by hard rules, e.g., $\alpha : Poss(Prize = d) \leftarrow not \ Intervene(Prize)$; and (iii) assigns the probabilities by soft rules, e.g., $ln(0.3) : \bot \leftarrow not \ AssPr(Prize = 1)$. The full translation is too long to be put here, please refer to Example 3 in [20] for details.

5.3 From LPOD to Answer Set Programs

▶ **Theorem 5.** *(from [21]) Under any of the four preference criteria, the preferred answer sets of an LPOD Π of signature σ are exactly the preferred answer sets on σ of $\mathsf{lpod2asp}(\Pi)$.*

▶ **Example 6.** Consider the following LPOD Π_3 about picking a hotel near the Grand Canyon. $hotel(1)$ is a 2-star hotel but is close to the Grand Canyon, $hotel(2)$ is a 3-star hotel and the distance is medium, and $hotel(3)$ is a 4-star hotel but is too far.

$$
\begin{array}{ll}
close \times med \times far \times tooFar. & \leftarrow hotel(2), \ not \ med. \\
star4 \times star3 \times star2. & \leftarrow hotel(2), \ not \ star3. \\
1\{hotel(X) : X = 1..3\}1. & \leftarrow hotel(3), \ not \ tooFar. \\
\leftarrow hotel(1), \ not \ close. & \leftarrow hotel(3), \ not \ star4. \\
\leftarrow hotel(1), \ not \ star2. &
\end{array}
$$

The translation $\mathsf{lpod2asp}(\Pi_3)$ is based on the definition of the *assumption program*, $AP(x_1, x_2)$, where $x_1 \in \{0, \ldots, 4\}$ and $x_2 \in \{0, \ldots, 3\}$. Intuitively, the value of x_i denotes an assumption about LPOD rule i: if $x_i = 0$, the body of rule i is false, thus no atom will be derived by rule i; if $x_i > 0$, the boy of rule i is true, and the x_i-th atom will be derived by rule i (which requires that all atoms in the head of rule i with a index lower than x_i must be false). An assumption program $AP(x_1, x_2)$ is initialized by a choice rule and a weak constraint (which makes sure that all consistent assumption programs are considered).

```
{ap(X1,X2): X1=0..4, X2=0..3}.                :~ ap(X1,X2). [-1, X1, X2]
```

The assumption programs include all regular rules in Π. Note that (i) we turn each atom a in Π into $a(X_1, X_2)$ so that the answer sets of assumption program $AP(x_1, x_2)$ are in its own name space (x_1, x_2); (ii) we add $ap(X_1, X_2)$ in the body of each rule so that these rules will not be "effective" if the assumption program $AP(X_1, X_2)$ is inconsistent.

```
1{hotel(H,X1,X2): H=1..3}1 :- ap(X1,X2).
:- ap(X1,X2), hotel(1,X1,X2), not close(X1,X2).
:- ap(X1,X2), hotel(1,X1,X2), not star2(X1,X2).
:- ap(X1,X2), hotel(2,X1,X2), not med(X1,X2).
:- ap(X1,X2), hotel(2,X1,X2), not star3(X1,X2).
:- ap(X1,X2), hotel(3,X1,X2), not tooFar(X1,X2).
:- ap(X1,X2), hotel(3,X1,X2), not star4(X1,X2).
```

Besides, the assumption programs include all assumptions that we record in (x_1, x_2).

```
% close * med * far * tooFar.
body_1(X1,X2) :- ap(X1,X2).
:- ap(X1,X2), X1=0, body_1(X1,X2).
:- ap(X1,X2), X1>0, not body_1(X1,X2).

close(X1,X2) :- body_1(X1,X2), X1=1.
med(X1,X2) :- body_1(X1,X2), X1=2.
far(X1,X2) :- body_1(X1,X2), X1=3.
tooFar(X1,X2) :- body_1(X1,X2), X1=4.

X1=1 :- body_1(X1,X2), close(X1,X2).
X1=2 :- body_1(X1,X2), med(X1,X2), not close(X1,X2).
X1=3 :- body_1(X1,X2), far(X1,X2), not close(X1,X2), not med(X1,X2).
X1=4 :- body_1(X1,X2), tooFar(X1,X2), not close(X1,X2),
        not med(X1,X2), not far(X1,X2).

% star4 * star3 * star2.
body_2(X1,X2) :- ap(X1,X2).

:- ap(X1,X2), X2=0, body_2(X1,X2).
:- ap(X1,X2), X2>0, not body_2(X1,X2).

star4(X1,X2) :- body_1(X1,X2), X2=1.
star3(X1,X2) :- body_1(X1,X2), X2=2.
star2(X1,X2) :- body_1(X1,X2), X2=3.

X2=1 :- body_1(X1,X2), star4(X1,X2).
X2=2 :- body_1(X1,X2), star3(X1,X2), not star4(X1,X2).
X2=3 :- body_1(X1,X2), star2(X1,X2), not star4(X1,X2),
        not star3(X1,X2).
```

To calculate the satisfaction degrees D_1, D_2 of two LPOD rules, lpod2asp(Π_3) contains

```
degree(ap(X1,X2), D1, D2) :- ap(X1,X2), D1=#max{1;X1}, D2=#max{1;X2}.
```

Note that all answer sets of $AP(x_1, x_2)$ will have a same satisfaction degree for each LPOD rule. Thus we also use $ap(x_1, x_2)$ to denote an answer set of $AP(x_1, x_2)$ in the following set of rules. To compare two candidate answer set S_1 and S_2 according to, say, Pareto-preference, and to determine whether an answer set of $AP(x_1, x_2)$ is a Pareto-preferred answer set, lpod2asp(Π_3) contains

```
equ(S1,S2) :- degree(S1,D1,D2), degree(S2,D1,D2).

prf(S1,S2) :- degree(S1,D11,D12), degree(S2,D21,D22), not equ(S1,S2),
              D11<=D21, D12<=D22.

pAS(X1, X2) :- ap(X1, X2), {prf(S, ap(X1,X2))}0.
```

5.4   From CR-Prolog₂ to Answer Set Programs

▶ **Theorem 7.** *(from [21]) For any CR-Prolog₂ program Π of signature σ, (a) the projections of the generalized answer sets of Π onto σ are exactly the generalized answer sets on σ of* crp2asp(Π). *(b) the projections of the candidate answer sets of Π onto σ are exactly the candidate answer sets on σ of* crp2asp(Π). *(c) the preferred answer sets of Π are exactly the preferred answer sets on σ of* crp2asp(Π).*

▶ **Example 8.** (From [4]) Consider the following CR-Prolog$_2$ program Π_4:

$$q \leftarrow t. \qquad\qquad p \leftarrow not\ q. \qquad\qquad 1: \quad t \overset{+}{\leftarrow} .$$
$$s \leftarrow t. \qquad\qquad r \leftarrow not\ s. \qquad\qquad 2: \quad q \times s \overset{+}{\leftarrow} .$$
$$\leftarrow p, r.$$

The idea behind crp2asp is similar to that for lpod2asp. crp2asp(Π_4) consists of
(i) all consistent assumption programs

```
{ap(X1,X2): X1=0..1, X2=0..2}.          :~ ap(X1,X2). [-1,X1,X2]

q(X1,X2) :- ap(X1,X2), t(X1,X2).
s(X1,X2) :- ap(X1,X2), t(X1,X2).
p(X1,X2) :- ap(X1,X2), not q(X1,X2).
r(X1,X2) :- ap(X1,X2), not s(X1,X2).
:- ap(X1,X2), p(X1,X2), r(X1,X2).

% 1: t <+-.
t(X1,X2) :- ap(X1,X2), X1=1.

% 2: q*s <+-.
q(X1,X2) :- ap(X1,X2), X2=1.
s(X1,X2) :- ap(X1,X2), X2=2.
```

(ii) the definition of dominate as well as the definition of candidate answer set

```
dominate(ap(X1,X2), ap(Y1,Y2)) :- ap(X1,X2), ap(Y1,Y2), 0<X1, X1<Y1.
dominate(ap(X1,X2), ap(Y1,Y2)) :- ap(X1,X2), ap(Y1,Y2), 0<X2, X2<Y2.

candidate(X1,X2) :- ap(X1,X2), {dominate(SP,ap(X1,X2))}0.
```

(iii) the definition of lessCrRuleApplied as well as the definition of preferred answer set

```
lessCrRuleApplied(ap(X1,X2), ap(Y1,Y2)) :- candidate(X1,X2),
        candidate(Y1,Y2), 1{X1!=Y1;X2!=Y2}, X1<=Y1, X2<=Y2.

pAS(X1,X2) :- candidate(X1,X2), {lessCrRule(SP,ap(X1,X2))}0.
```

6 Open Issues and Expected Achievements

One issue is that, among the 4 translations, only lpmln2wc has an implemented compiler. So, for now, most translations must be done manually. However, we may not implement the compilers for the translations lpod2asp and crpt2asp, since they are exponential to the number of non-regular rules.

Another issue is, currently, we are working on combining quantitative and qualitative uncertainty in a single formalism, but it is still not clear how these two kinds of uncertainty merge together. For example, if there is a preference rule saying "football > ping-pong > basketball" with a quantitative confidence 5, and there is another preference rule saying "indoor game > outdoor game" with confidence 10, what should be the order of these activities? To answer this question, we should first answer "how should the confidence be arranged in a rule without loss of generality?" The follow-up question is "what is the confidence of basketball if there is a probability of 70% that it is an indoor game?"

As for the future work, we will check whether the recent approach, Asprin [10], can be used to implement LPOD, CR-Prolog$_2$, LP^{MLN}, and even P-log. At the meantime, we will start to combine quantitative and qualitative uncertainty from tuning the semantics of LPOD to include quantitative uncertainty in its syntax and semantics. After the formalism is created and well defined, we will prove its expressivity and implement a compiler for it.

References

1 Vernon Asuncion, Yan Zhang, and Heng Zhang. Logic programs with ordered disjunction: first-order semantics and expressiveness. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, pages 2–11. AAAI Press, 2014.

2 Evgenii Balai and Michael Gelfond. On the Relationship between P-log and LP^{MLN}. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 915–921, 2016.

3 Marcello Balduccini. CR-MODELS: an inference engine for CR-Prolog. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 18–30. Springer-Verlag, 2007.

4 Marcello Balduccini, Marcello Balduccini, and Veena Mellarkod. CR-Prolog with Ordered Disjunction. In *In ASP03 Answer Set Programming: Advances in Theory and Implementation, volume 78 of CEUR Workshop proceedings*, 2003.

5 Marcello Balduccini and Michael Gelfond. Logic Programs with Consistency-Restoring Rules. In *International Symposium on Logical Formalization of Commonsense Reasoning, AAAI 2003 Spring Symposium Series*, pages 9–18, 2003.

6 Marcello Balduccini and Veena Mellarkod. A-prolog with cr-rules and ordered disjunction. In *Intelligent Sensing and Information Processing, 2004. Proceedings of International Conference on*, pages 1–6. IEEE, 2004.

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

8 Gerhard Brewka. Logic programming with ordered disjunction. In *AAAI/IAAI*, pages 100–105, 2002.

9 Gerhard Brewka. Preferences in answer set programming. In *CAEPIA*, volume 4177, pages 1–10. Springer, 2005.

10 Gerhard Brewka, James P Delgrande, Javier Romero, and Torsten Schaub. asprin: Customizing Answer Set Preferences without a Headache. In *AAAI*, pages 1467–1474, 2015.

11 Gerhard Brewka, Ilkka Niemelä, and Tommi Syrjänen. Implementing ordered disjunction using answer set solvers for normal programs. In *European Workshop on Logics in Artificial Intelligence*, pages 444–456. Springer, 2002.

12 Francesco Buccafurri, Nicola Leone, and Pasquale Rullo. Enhancing disjunctive datalog by constraints. *IEEE Transactions on Knowledge and Data Engineering*, 12(5):845–860, 2000.

13 Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, pages 2462–2467, 2007.

14 James P Delgrande, Torsten Schaub, and Hans Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.

15 Paolo Ferraris, Joohyung Lee, and Vladimir Lifschitz. Stable models and circumscription. *Artificial Intelligence*, 175:236–263, 2011.

16 Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080. MIT Press, 1988.

17   Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunct-
ive databases. *New Generation Computing*, 9:365–385, 1991.

18   Joohyung Lee, Yunsong Meng, and Yi Wang. Markov Logic Style Weighted Rules under the
Stable Model Semantics. In Technical Communications of the 31st International Conference
on Logic Programming, 2015.

19   Joohyung Lee and Yi Wang. Weighted Rules under the Stable Model Semantics. In
*Proceedings of International Conference on Principles of Knowledge Representation and
Reasoning (KR)*, pages 145–154, 2016.

20   Joohyung Lee and Zhun Yang. LPMLN, weak constraints, and P-log. In *Proceedings of the
AAAI Conference on Artificial Intelligence (AAAI)*, pages 1170–1177, 2017.

21   Joohyung Lee and Zhun Yang. Translating LPOD and CR-Prolog2 into Standard Answer
Set Programs. *arXiv preprint arXiv:1805.00643*, 2018. `arXiv:1805.00643`.

22   Judea Pearl. *Causality: models, reasoning and inference*, volume 29. Cambridge Univ
Press, 2000.

23   Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine Learning*,
62(1-2):107–136, 2006.

Proof-Relevant Resolution for Elaboration of Programming Languages

František Farka
University of St Andrews, UK, and
Heriot-Watt University, Edinburgh, UK
ff32@st-andrews.ac.uk
https://orcid.org/0000-0001-8177-1322

Abstract

Proof-relevant resolution is a new variant of resolution in Horn-clause logic and its extensions. We propose proof-relevant resolution as a systematic approach to elaboration in programming languages that is close to formal specification and hence allows for analysis of semantics of the language. We demonstrate the approach on two case studies; we describe a novel, proof-relevant approach to type inference and term synthesis in dependently types languages and we show how proof-relevant resolution allows for analysis of inductive and coinductive soundness of type class resolution. We conclude by a discussion of overall contributions of our current work.

1 Introduction

First order resolution is a widely utilised technique in type inference. Hindley and Milner [13] were the first to notice that type inference in simply typed lambda calculus can be expressed as a first-order unification problem. This general scheme allows a multitude of extensions. For example, Hindley-Milner type system can be extended to constrained types system. For this extension, a constraint logic programming [15], was suggested, in which constraint solving over a certain domain was added to the existing first-order unification and resolution algorithms. Haskell type classes are another example of the application of logic programming. It is widely understood that type class resolution is in fact implemented as first-order resolution on Horn clauses. However, there is a caveat with respect to the traditional logic programming – a *dictionary* (that is, a proof term) needs to be constructed [14]. The research in the area of type classes is on-going: various extensions to the syntax of type classes are still being investigated [10].

Fu and Komendantskaya analysed type class resolution and proposed *proof-relevant* Horn clause logic [5] as the appropriate formalism. In this logic, Horn clauses are seen as types and proof witnesses as terms inhabiting the types. Given a proposition – a *goal* – and a set of Horn clauses – a *logic program* – the resolution process is captured by an explicit proof term construction. To briefly illustrate the proof-relevant approach, let us state the usual (generalised) modus ponens inference rule in a proof-relevant way:

$$\kappa : A \leftarrow B_1, \ldots, B_n \in \mathcal{P} \quad \frac{\mathcal{P} \vdash \delta_1 : \theta B_1 \quad \ldots \quad \mathcal{P} \vdash \delta_n : \theta B_n}{\mathcal{P} \vdash \kappa \delta_1 \ldots \delta_n : \theta A}$$

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 18; pp. 18:1–18:9
OpenAccess Series in Informatics
OASIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

That is, for a substitution θ and an atom A, an instance θA can deduced in a program $\mathcal{P}$ assuming that there is a Horn clause $A \leftarrow B_1, \ldots, B_n$ in $\mathcal{P}$ and that each θ-instance of the atom B_i in the body of the clause can deduced in $\mathcal{P}$. Moreover, the clause is equipped with an atomic symbol κ (which is unique for each clause in the program). Assuming that deduction of the instance θB_i is witnessed by a proof term δ_i the overall deduction of θA is witnessed by a compound proof-term $\kappa \delta_1 \ldots \delta_n$.

In our work, we investigate proof-relevant resolution and we propose it as a systematic approach to elaboration of programming languages. In order to avoid an excessive technical detail, in this short paper we focus on demonstrating assets of proof-relevant resolution by means of an example. In particular, in Section 2 we show that proof-relevant resolution is a convenient calculus to formulate type inference and term synthesis for dependently typed languages, and in Section 3 we build on such treatment of type-class resolution and show that such approach is convenient for working with the semantics of the language as well; namely we show that proof-relevant treatment of type classes is sound both inductively and coinductively. We refer the reader to our published work [3, 4] for technical details.

2   Type Inference and Term Synthesis

In the last decade, dependent types [17, 1] have gained popularity in the programming language community. They allow reasoning about program values within the types, and thus give more general, powerful and flexible mechanisms to enable verification of code. However, such verification comes for a price. There are many proof obligations in form of computationally irrelevant terms that manifest that the code has properties that are stated in types, that themselves beige a specification become very complicated. Extensive automation of type inference and term (proof obligation) synthesis is a necessity for any system that aims to be usable in practice. We propose a novel approach for such automation. We use the notion *refinement* to refer to the combined problem of type inference and term synthesis.

Using an abstract syntax that closely resembles existing functional programming language we define $\mathtt{maybe_A}$, an option type over a fixed type $\mathtt{A}$, indexed by a Boolean:

▶ **Example 1.**

```
data maybeA (a : A)   : bool → type where
     nothing          : maybeA ff
     just             : A → maybeA tt
```

Here, `nothing` and `just` are the two *constructors* of the `maybe` type. The type is indexed by `ff` when the `nothing` constructor is used, and by `tt` when the `just` constructor is used (`ff` and `tt` are constructors of `bool`). A function `fromJust` extracts the value from the `just` constructor:

```
fromJust : maybeA tt → A
fromJust (just x) = x
```

Note that the value `tt` appears within the type $\mathtt{maybe_A}\ \mathtt{tt} \to \mathtt{A}$ of this function (the type *depends* on the value), allowing for a more precise function definition that omits the redundant case when the constructor of type $\mathtt{maybe_A}$ is `nothing`. The challenge for the type checker is to determine that the missing case `fromJust nothing` is contradictory (rather than being omitted by mistake). Indeed, the type of `nothing` is $\mathtt{maybe_A}\ \mathtt{ff}$. However, the function specifies its argument to be of type $\mathtt{maybe_A}\ \mathtt{tt}$.

```
A : type
bool : type
ff tt : bool

(≡bool) : bool → bool → type
refl : Π(b:bool). b ≡bool b
elim≡bool: tt ≡bool ff → A

maybeA : bool → type
nothing : maybeA ff
just : A → maybeA tt
elim maybeA : Π(b:bool).maybeA b
→ (b ≡bool ff → A)
→ (b ≡bool tt → A → A)
→ A
```

■ **Figure 1** Signature of t_{fromJust}.

To type check such functions, the compiler translates them into terms in an *internal*, type-theoretic *calculus.* We rely on the calculus LF [9], a standard and well-understood first-order dependent type theory as a choice of such internal calculus. A signature that we use to encode our example in LF is give in Figure 1. We employ $A \to B$ as an abbreviation for $\Pi(a : A).B$ where a does not occur free in B. The final goal of type checking of the function `fromJust` in the programming language is to obtain the following encoding in the internal calculus:

▶ **Example 2.**

```
t fromJust := λ ( m:maybeA tt).elim maybeA tt m
(λ (w:tt≡boolff).elim≡bool w)
(λ (w:tt≡booltt).λ (x:A).x)
```

The missing case for `nothing` must be accounted for (*cf.* the line (λ (w:tt≡boolff).elim≡bool w) above). In this example (as is generally the case), only partial information is given in the programming language. To address this problem, we extend the internal language with term level metavariables, denoted by $?_a$, and type level metavariables, denoted by $?_A$. These stand for the parts of a term in the internal language that are not yet known. Using metavariables, the term that directly corresponds to `fromJust` is:

▶ **Example 3.**

```
t fromJust := λ ( m:maybeA tt).elim maybeA ?a m
(λ ( w: ?A ). ?b )
(λ ( w: ?B ).λ ( x:A).x)
```

The missing information comprises the two types $?_A$ and $?_B$ and the term $?_b$ for the constructor `nothing`. Obtaining types $?_A$, $?_B$ amounts to type inference whereas obtaining the term $?_b$ amounts to term synthesis. We translate *refinement problems* into the syntax of logic programs. The *refinement* algorithm that we propose takes a signature and a term with metavariables in the extended internal calculus to a logic program and a goal in proof-relevant Horn clause logic. The unifiers that are computed by resolution give an assignment of types

to type-level metavariables. At the same time, the computed proof terms are interpreted as an assignment of terms to term-level metavariables. We illustrate the process in the following paragraphs.

Consider the inference rule Π-T-Elim in LF for application of a dependent function to an argument:

$$\frac{\Gamma \vdash M : \Pi x : A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]} \text{ Π-T-Elim}$$

When type checking the term $t_{\texttt{fromJust}}$ an application of $\texttt{elim}_{\texttt{maybe}_{\texttt{A}}}$ $\texttt{tt}$ m to the term $\lambda(w :?_A).?_b$ in the context $m : \texttt{maybe}_{\texttt{A}}$ $\texttt{tt}$ needs to be type checked. This amounts to providing a derivation of the typing judgement that contains the following instance of the rule Π-T-Elim:

$$\frac{\begin{array}{c} m : \texttt{maybe}_{\texttt{A}} \texttt{ tt} \vdash \texttt{elim}_{\texttt{maybe}_{\texttt{A}}} \texttt{ tt} \ m \\ : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \rightarrow \texttt{A}) \rightarrow \cdots \rightarrow \texttt{A} \qquad m : \texttt{maybe}_{\texttt{A}} \texttt{ tt} \vdash \lambda(w :?_A).?_b :?_A \rightarrow ?_B \end{array}}{m : \texttt{maybe}_{\texttt{A}} \texttt{ tt} \vdash (\texttt{elim}_{\texttt{maybe}_{\texttt{A}}} \texttt{ tt} \ m) \ (\lambda(w :?_A).?_b) : (\texttt{tt} \equiv_{\texttt{bool}} \texttt{tt} \rightarrow \texttt{A} \rightarrow \texttt{A}) \rightarrow \texttt{A}}$$

For the above inference step to be a valid instance of the inference rule Π-T-Elim, it is necessary that $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A$ and $\texttt{A} = ?_B$. This is reflected in the following goal:

$$((\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) = ?_A) \wedge (\texttt{A} = ?_B) \wedge G_{(\texttt{elim}_{\texttt{maybe}_{\texttt{A}}} \texttt{ tt} \ m)} \wedge G_{\lambda(w:?_A).?_b} \tag{1}$$

The additional goals $G_{(\texttt{elim}_{\texttt{maybe}_{\texttt{A}}} \texttt{ tt} \ m)}$ and $G_{\lambda(w:?_A).?_b}$ are recursively generated by the algorithm for the terms $\texttt{elim}_{\texttt{maybe}_{\texttt{A}}}$ $\texttt{tt}$ m and $\lambda(w :?_A).?_b$, respectively. Similarly, assuming the term $\lambda(w :?_A).?_b$ is of type $(\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}) \rightarrow \texttt{A}$, type checking places restrictions on the term $?_b$:

$$\frac{m : \texttt{maybe}_{\texttt{A}} \texttt{ tt} \vdash \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} : \texttt{type} \qquad m : \texttt{maybe}_{\texttt{A}} \texttt{ tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \vdash ?_b : \texttt{A}}{m : \texttt{maybe}_{\texttt{A}} \texttt{ tt} \vdash \lambda(w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}).?_b : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \rightarrow \texttt{A}}$$

That is, $?_b$ needs to be a well-typed term of type $\texttt{A}$ in a context consisting of m and w. Recall that in the signature there is a constant $\texttt{elim}_{\equiv_{\texttt{bool}}}$ of type $\texttt{tt} \equiv_{\texttt{bool}} \texttt{ff} \rightarrow \texttt{A}$. Our translation will turn this constant into a clause in the generated logic program. There will be a clause that corresponds to the inference rule for elimination of a Π type as well:

$$\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}} : \ term(\texttt{elim}_{\equiv_{\texttt{bool}}}, \Pi x : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}. \texttt{A}, ?_\Gamma) \leftarrow$$
$$\kappa_{\texttt{elim}} : \ term(?_M ?_N, ?_B, ?_\Gamma) \leftarrow term(?_M, \Pi x :?_A.?_{B'}, ?_\Gamma) \wedge term(?_N, ?_A, ?_\Gamma) \wedge ?_{B'}[?_N/x] \equiv ?_B$$

The above clauses are written in the proof-relevant Horn clause logic, and thus $\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}$ and $\kappa_{\texttt{elim}}$ now play the role of proof-term symbols ("witnesses" for the clauses). In this clause, $?_M, ?_N, ?_A, ?_B, ?_{B'}$ and $?_\Gamma$ are *logic variables*, i.e. variables of the first-order logic. By an abuse of notation, we use the same symbols for metavariables of the internal calculus and logic variables in the logic programs generated by the refinement algorithm. We also use the same notation for objects of the internal language and terms of the logic programs. This is possible since we represent variables using *de Bruijn indices*.

The presence of $w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}$ in the context allows us to use the clause $\texttt{elim}_{\equiv_{\texttt{bool}}}$ to resolve the goal $term(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_{\texttt{A}} \texttt{ tt}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}])$:

$$term(?_M ?_N, \texttt{A}, [m : \texttt{maybe}_{\texttt{A}}, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \rightsquigarrow_{\kappa_{\texttt{elim}}}$$
$$term(?_M, \Pi x :?_A. \texttt{A}, [\ldots]) \wedge term(?_N, ?_A, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{elim}_{\equiv_{\texttt{bool}}}}}$$
$$term(?_N, \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}, [\ldots, w : \texttt{tt} \equiv_{\texttt{bool}} \texttt{ff}]) \wedge \texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{proj}_w}}$$
$$\texttt{A}[?_N/x] \equiv ?_B \rightsquigarrow_{\kappa_{\texttt{subst}_{\texttt{A}}}} \bot \tag{2}$$

The resolution steps are denoted by $\rightsquigarrow$. Each step is indexed by the name of the clause that was used. First, the goal is resolved in one step using the clause κ_{elim}. A clause κ_{proj_w} is used to project the variable w from the context. We omit a discussion of the exact shape of the clauses since it depends on the representation we use. In this presentation, we are just interested in composing the proof terms occurring in these resolution steps into one composite proof term: κ_{elim} $\kappa_{\text{elim}_{\equiv_{\text{bool}}}}$ $\kappa_{\text{proj}_w}\kappa_{\text{subst}_A}$. Note that, by resolving the goal (1), we obtain a substitution θ that assigns the type A to the logic variable $?_B$, *i.e.* $\theta(?_B) = A$. At the same time, the proof term computed by the the derivation (2) is interpreted as a solution $(\text{elim}_{\equiv_{\text{bool}}} \ w)$ for the term-level metavariable $?_b$. However, the proof term can be used to reconstruct the derivation of well-typedness of the judgement $m : \text{maybe}_A \ \text{tt}, w : \text{tt} \equiv_{\text{bool}} \text{ff} \vdash \text{elim}_{\equiv_{\text{bool}}} \ w : A$ as well. In general, a substitution is interpreted as a solution to a type-level metavariable and a proof term as a solution to a term-level metavariable. The remaining solution for $?_A$ is computed using similar methodology, and we omit the details here. Thus, we have computed values for all metavariables in Example 3, *i.e.* we inferred all types and synthesised all terms.

3   Type Class Resolution

Type classes are a versatile language construct for implementing ad-hoc polymorphism and overloading in functional languages. The approach originated in Haskell [16, 8] and has been further developed in dependently typed languages [7, 2]. For example, it is convenient to define equality for all data structures in a uniform way. In Haskell, this is achieved by introducing the equality class `Eq`:

▶ **Example 4.**

```
class Eq x where
  eq :: Eq x ⇒ x → x → Bool
```

and then declaring any necessary instances of the class, e.g. for pairs and integers:

```
instance (Eq x, Eq y) ⇒ Eq (x, y) where
  eq (x1, y1) (x2, y2) = eq x1 x2 && eq y1 y2
instance Eq Int where
  eq x y = primtiveIntEq x y
```

Type class resolution is performed by the Haskell compiler and involves checking whether all the instance declarations are valid. For example, the following function triggers a check that `Eq (Int, Int)` is a valid instance of type class `Eq`:

```
test :: Eq (Int, Int) ⇒ Bool
test = eq (1,2) (1,2)
```

It is folklore that type class instance resolution resembles SLD-resolution from logic programming. The type class instance declarations above could, for example, be viewed as the following two Horn clauses:

▶ **Example 5** (Logic program P_{Pair}).
$$\kappa_{\text{pair}} : \text{eq}(x), \ \text{eq}(y) \ \Rightarrow \text{eq}(\text{pair}(x,y))$$
$$\kappa_{\text{int}} : \qquad\qquad\qquad \Rightarrow \text{eq}(\text{int})$$

Then, given the query $\mathtt{eq}(\mathtt{pair}(\mathtt{int},\mathtt{int}))$, resolution terminates successfully with the following sequence of inference steps:

$$\mathtt{eq}(\mathtt{pair}(\mathtt{int},\mathtt{int})) \rightarrow_{\kappa_{\mathrm{pair}}} \mathtt{eq}(\mathtt{int}), \mathtt{eq}(\mathtt{int}) \rightarrow_{\kappa_{\mathrm{int}}} \mathtt{eq}(\mathtt{int}) \rightarrow_{\kappa_{\mathrm{int}}} \emptyset$$

The proof witness $\kappa_{\mathrm{pair}}\kappa_{\mathrm{int}}\kappa_{\mathrm{int}}$ (called a "dictionary") is constructed by the compiler. This is treated internally as an executable function.

Despite the apparent similarity of type class syntax and type class resolution to Horn clause resolution they are not, however, identical. At a syntactic level, type class instance declarations correspond to a restricted form of Horn clauses, namely ones that:

(i) do not *overlap* (*i.e.* whose heads do not unify); and that

(ii) do not contain existential variables (*i.e.* variables that occur in the bodies but not in the heads of the clauses). At an algorithmic level,

(iii) type class resolution corresponds to Horn-clause resolution in which unification is restricted to term-matching.

Assuming there is a clause $B_1, \ldots B_n \Rightarrow A'$, then a query $?\ A'$ can be resolved with this clause only if A can be matched against A', *i.e.* if a substitution σ exists such that $A = \sigma A'$. In comparison, Horn-clause resolution incorporates *unifiers*, as well as *matchers*, *i.e.* it also proceeds to resolve the above query and clause in all the cases where $\sigma A = \sigma A'$ holds.

These restrictions guarantee that type class inference computes the *principal* (most general) type. Restrictions (i) and (ii) amount to deterministic inference by resolution, in which only one derivation is possible for every query. Restriction (iii) means that no substitution is applied to a query during inference, *i.e.* we prove the query in an implicitly universally quantified form. It is common knowledge that (as with Horn-clause resolution) type class resolution is *inductively sound*, *i.e.* that it is sound relative to the least Herbrand models of logic programs [12]. Moreover we established [3], for the first time, that it is also *universally inductively sound*, *i.e.* that if a formula A is proved by type class resolution, every ground instance of A is in the least Herbrand model of the given program. In contrast to Horn-clause resolution, however, type class resolution is *inductively incomplete*, *i.e.* it is incomplete relative to least Herbrand models, even for the class of Horn clauses that is restricted by conditions i and ii. For example, given a clause $\Rightarrow \mathtt{q}(\mathtt{f}(x))$ and a query $?\ \mathtt{q}(x)$, Horn-clause resolution is able to find a proof (by instantiating x with $\mathtt{f}(x)$), but type class resolution fails. Lämmel and Peyton Jones have suggested [11] an extension to type class resolution that accounts for some non-terminating cases of type class resolution. Consider, for example, the following mutually defined data structures:

▶ **Example 6.**

```
data OddList a = OCons a (EvenList a)
data EvenList a = Nil | ECons a (OddList a)
```

which give rise to the following instance declarations for the `Eq` class:

```
instance (Eq a, Eq (EvenList a)) ⇒ Eq (OddList a) where
 eq (OCons x xs) (OCons y ys) = eq x y && eq xs ys

instance (Eq a, Eq (OddList a)) ⇒ Eq (EvenList a) where
 eq Nil Nil = True
 eq (ECons x xs) (ECons y ys) = eq x y && eq xs ys
 eq _ _ = False
```

The `test` function below triggers type class resolution in the Haskell compiler:

```
test :: Eq (EvenList Int) ⇒ Bool
test = eq Nil Nil
```

However, inference by resolution does not terminate in this case. Consider the Horn clause representation of the type class instance declarations:

▶ **Example 7** (Logic program $P_{EvenOdd}$)**.**

$$\kappa_{\mathrm{odd}} : \mathtt{eq}(x),\ \mathtt{eq}(\mathtt{evenList}(x))\ \Rightarrow \mathtt{eq}(\mathtt{oddList}(x))$$
$$\kappa_{\mathrm{even}} : \mathtt{eq}(x),\ \mathtt{eq}(\mathtt{oddList}(x))\ \ \Rightarrow \mathtt{eq}(\mathtt{evenList}(x))$$
$$\kappa_{\mathrm{int}} : \qquad\qquad\qquad\qquad\quad\ \Rightarrow \mathtt{eq}(\mathtt{int})$$

The non-terminating resolution trace is given by:

$$\underline{\mathtt{eq}(\mathtt{evenList}(\mathtt{int}))} \to_{\kappa_{\mathrm{even}}} \mathtt{eq}(\mathtt{int}),\ \mathtt{eq}(\mathtt{oddList}(\mathtt{int})) \to_{\kappa_{\mathrm{int}}} \mathtt{eq}(\mathtt{oddList}(\mathtt{int}))$$
$$\to_{\kappa_{\mathrm{int}}} \mathtt{eq}(\mathtt{int}),\ \mathtt{eq}(\mathtt{evenList}(\mathtt{int})) \to_{\kappa_{\mathrm{int}}} \underline{\mathtt{eq}(\mathtt{evenList}(\mathtt{int}))} \to_{\kappa_{\mathrm{even}}} \cdots$$

A goal `eq(evenList(int))` is simplified using the clause κ_{even} to two new goals `eq(int)` and `eq(oddList(int))`. The first of these is discarded using the clause κ_{int}. Resolution continues using κ_{odd} and κ_{int}, resulting in the original goal `eq(evenList(int))`. It is easy to see that such a process could continue infinitely and that this goal constitutes a *cycle* (underlined above). As suggested by Lämmel and Peyton Jones [11], the compiler can terminate the infinite inference process as soon as it detects the underlined cycle. Moreover, it can also construct the corresponding proof witness in a form of a recursive function. For the example above, such a function is given by the fixed point term $\nu\alpha.\kappa_{\mathrm{even}}\kappa_{\mathrm{int}}(\kappa_{\mathrm{odd}}\kappa_{\mathrm{int}}\alpha)$, where ν is a fixed point operator. The intuitive reading of such a proof is that an infinite proof of the query `eq (evenList(int))` exists, and that its shape is fully specified by the recursive proof witness function above. We say that the proof is given by *corecursive type class resolution*.

Corecursive type class resolution is not inductively sound. For example, the formula `eq(evenList(int))` is not in the least Herbrand model of the corresponding logic program. However, we proved [3] that it is *(universally) coinductively sound*, *i.e.* it is sound relative to the greatest Herbrand models. For example, `eq(evenList(int))` is in the greatest Herbrand model of the program $P_{EvenOdd}$. Similarly to the inductive case, corecursive type class resolution is coinductively incomplete. Consider the clause $\kappa_{\mathrm{inf}} : \mathtt{p}(x) \Rightarrow \mathtt{p}(\mathtt{f}(x))$. This clause may be given an interpretation by the greatest (complete) Herbrand models. However, corecursive type class resolution does not yield infinite proofs.

Unfortunately, this simple method of cycle detection does not work for all non-terminating programs. Consider the following example, which defines a data type `Bush` (for bush trees), and its corresponding instance for `Eq`:

```
data Bush a = Nil | Cons a (Bush (Bush a))
instance Eq a, Eq (Bush (Bush a)) ⇒ Eq (Bush a) where { ... }
```

Here, type class resolution does not terminate. However, it does not exhibit cycles either. Consider the Horn clause translation of the problem:

▶ **Example 8** (Logic program P_{Bush})**.**

$$\kappa_{\mathrm{int}} : \qquad\qquad\qquad\qquad\quad\ \Rightarrow \mathtt{eq}(\mathtt{int})$$
$$\kappa_{\mathrm{bush}} : \mathtt{eq}(x),\ \mathtt{eq}(\mathtt{bush}(\mathtt{bush}(x))) \Rightarrow \mathtt{eq}(\mathtt{bush}(x))$$

The derivation below shows that no cycles arise when we resolve the query ? $\texttt{eq}(\texttt{bush}(\texttt{int}))$ against the program P_{Bush}:

$$\texttt{eq}(\texttt{bush}(\texttt{int})) \rightarrow_{\kappa_{\text{bush}}} \texttt{eq}(\texttt{int}), \texttt{eq}(\texttt{bush}(\texttt{bush}(\texttt{int}))) \rightarrow_{\kappa_{\text{int}}} \dots \rightarrow_{\kappa_{\text{bush}}}$$
$$\texttt{eq}(\texttt{bush}(\texttt{int})), \texttt{eq}(\texttt{bush}(\texttt{bush}(\texttt{bush}(\texttt{int})))) \rightarrow_{\kappa_{\text{int}}} \dots$$

Fu *et al.* [6] have introduced an extension to corecursive type class resolution that allows implicative queries to be proved by corecursion and uses the recursive proof witness construction. Implicative queries require the language of proof terms to be extended with λ-abstraction. For example, in the above program the Horn formula $\texttt{eq}(x) \Rightarrow \texttt{eq}(\texttt{bush}(x))$ can be (coinductively) proven with the recursive proof witness $\kappa_{\text{new}} = \nu\alpha.\lambda\beta.\kappa_{\text{bush}}\beta(\alpha(\alpha\beta))$. If we add this Horn clause as a third clause to our program, we obtain a proof of $\texttt{eq}(\texttt{bush}(\texttt{int}))$ by applying κ_{new} to κ_{int}. In this case, it is even more challenging to understand whether the proof $\kappa_{\text{new}}\kappa_{\text{int}}$ of $\texttt{eq}(\texttt{bush}(\texttt{int}))$ is indeed sound: whether inductively, coinductively or in any other sense. We established [3], for the first time, *coinductive soundness* for proofs of such implicative queries, relative to the greatest Herbrand models of logic programs. Namely, we determined that proofs that are obtained by extending the proof context with coinductively proven Horn clauses (such as κ_{new} above) are coinductively sound but inductively unsound. This result demonstrates feasibility of proof-relevant approach for study of the semantic properties of elaboration of programming language constructs.

4   Contributions

In our work, we study proof-relevant resolution as a systematic approach to elaboration in programming languages. We argue that proof-relevant resolution is an appropriate technique for elaboration while it stays very close to formal specification and allows for analysis of semantics. In this short paper, we demonstrate this claim on two examples. We discuss refinement if dependently typed languages and soundness of type class resolution.

The main contributions of our work are:

1. We present a novel approach to refinement for a first-order type theory with dependent types;

2. we prove that our approach (*i.e.* generation of goals and logic programs) is decidable and hence can serve as a basis for a verified implementation;

3. we show that proof-relevant first-order Horn clause resolution gives an appropriate inference mechanism for dependently typed languages: firstly, it is sound with respect to type checking in LF; secondly, the proof term construction alongside the resolution trace allows to reconstruct the derivation of well-typedness judgement.

4. We show that proof-relevant approach to type class resolution and its two recent corecursive extensions [6, 11] are sound relative to the standard (Herbrand model) semantics of logic programming; and

5. we show that these new extensions are indeed "corecursive", *i.e.* that they are modelled by the greatest Herbrand model semantics rather than by the least Herbrand model semantics.

References

1 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. `doi:10.1017/S095679681300018X`.

2 Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In *Proc. of ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 143–155, 2011. `doi:10.1145/2034773.2034796`.

3 Frantisek Farka, Ekaterina Komendantskaya, and Kevin Hammond. Coinductive Soundness of Corecursive Type Class Resolution. In Manuel V. Hermenegildo and Pedro López-García, editors, *Logic-Based Program Synthesis and Transformation - 26th International Symposium, LOPSTR 2016, Edinburgh, UK, September 6-8, 2016, Revised Selected Papers*, volume 10184 of *Lecture Notes in Computer Science*, pages 311–327. Springer, 2016. `doi:10.1007/978-3-319-63139-4_18`.

4 František Farka, Ekaterina Komendantskaya, and Kevin Hammond. Proof-relevant Horn Clauses for Dependent Type Inference and Term Synthesis. In *accepted to ICLP 2018*, 2018.

5 Peng Fu and Ekaterina Komendantskaya. Operational semantics of resolution and productivity in Horn clause logic. *Formal Asp. Comput.*, 29(3):453–474, 2017. `doi:10.1007/s00165-016-0403-1`.

6 Peng Fu, Ekaterina Komendantskaya, Tom Schrijvers, and Andrew Pond. Proof Relevant Corecursive Resolution. In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture Notes in Computer Science*, pages 126–143. Springer, 2016. `doi:10.1007/978-3-319-29604-3_9`.

7 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *Proc. of ICFP 2011, Tokyo, Japan, September 19-21, 2011*, pages 163–175, 2011. `doi:10.1145/2034773.2034798`.

8 Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18(2):109–138, 1996. `doi:10.1145/227699.227700`.

9 Robert Harper and Frank Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Trans. Comput. Log.*, 6(1):61–101, 2005. `doi:10.1145/1042038.1042041`.

10 Georgios Karachalias and Tom Schrijvers. Elaboration on functional dependencies: functional dependencies are dead, long live functional dependencies! In Iavor S. Diatchki, editor, *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell, Oxford, United Kingdom, September 7-8, 2017*, pages 133–147. ACM, 2017. `doi:10.1145/3122955.3122966`.

11 Ralf Lämmel and Simon L. Peyton Jones. Scrap your boilerplate with class: extensible generic functions. In *Proc. of ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pages 204–215, 2005. `doi:10.1145/1086365.1086391`.

12 John W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

13 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

14 Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Haskell workshop*, Amsterdam, January 1997. URL: `https://www.microsoft.com/en-us/research/publication/type-classes-an-exploration-of-the-design-space/`.

15 Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283, 2008. `doi:10.1017/S0956796807006569`.

16 P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proc. of POPL '89*, pages 60–76, New York, NY, USA, 1989. ACM. `doi:10.1145/75277.75283`.

17 Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. A specification for dependent types in Haskell. *PACMPL*, 1(ICFP):31:1–31:29, 2017. `doi:10.1145/3110275`.

The Learning-Knowledge-Reasoning Paradigm for Natural Language Understanding and Question Answering

Arindam Mitra[1]
Arizona State University
Tempe, USA
amitra7@asu.edu
 https://orcid.org/0000-0003-0089-510X

─── **Abstract** ──────────────────────────────

Given a text, several questions can be asked. For some of these questions, the answer can be directly looked up from the text. However for several other questions, one might need to use additional knowledge and sophisticated reasoning to find the answer. Developing AI agents that can answer these kinds of questions and can also justify their answer is the focus of this research. Towards this goal, we use the language of Answer Set Programming as the knowledge representation and reasoning language for the agent. The question then arises, is how to obtain the additional knowledge? In this work we show that using existing Natural Language Processing parsers and a scalable Inductive Logic Programming algorithm it is possible to learn this additional knowledge (containing mostly commonsense knowledge) from question-answering datasets which then can be used for inference.

1 Introduction

Developing agents that can understand text is one of the long term goals of Artificial Intelligence. To track the progress towards this goal, several question-answering challenges have been proposed, such as, the science question answering challenge *aristo* [1], project *euclid*'s math word problem solving [3, 4] and *facebook research*'s *bAbI* question answering challenge [9]. In all these challenges, a small text is provided describing a scenario and one or more questions based on that scenario. Table 1 shows an example from each of these three tasks.

It should be noted that answering these questions (Table 1) requires knowledge that goes beyond the text. For example, to answer the questions from the bAbI task (Table 1) one needs to know the effect of certain actions. Similarly, answering the math question requires the knowledge that the games one has won or lost is a subset of the games one has played and also that the value of a whole is equal to the sum of its parts. The later is popularly known as the *part-whole* formula. The science question on the other hand requires one to know the dynamics of predator-prey population. Some of this knowledge such as the math

---

[1] The author is advised by Dr. Chitta Baral from Arizona State University.

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 19; pp. 19:1–19:6
 OpenAccess Series in Informatics
 OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

■ **Table 1** shows an example problem from the datasets of bAbI, word math problems and Aristo.

| Mary grabbed the football. |
| Mary traveled to the office. |
| Mary took the apple there. |
| What is Mary carrying? A:football,apple |
| Mary left the football. |
| Daniel went back to the bedroom. |
| What is Mary carrying? A:apple |

(a) An example from a bAbI challenge.

| Sara's high school played 12 basketball games this year. The team won most of their games. They were defeated during 4 games. How many games did they win ? |

(b) An example of a word arithmetic problem.

| In one area, a large source of prey for eagles is rabbits. If the number of rabbits suddenly decreases, what effect will it most likely have on the eagles? (A) Their numbers will increase. (B) Their numbers will decrease. (C) They will adapt new behaviors. (D) They will migrate to new locations. |

(c) An example of a science question.

formula or the prey-predator population dynamics, can be easily collected from books and can be provided to the agent as a background knowledge. However, some types of knowledge such as the affect of the actions or the commonsense knowledge about part whole relations between verbs might be difficult to write down manually as there exists a vast amount of such knowledge. In this research, thus we aim to learn such knowledge from question-answering dataset.

The proposed QA-architecture namely the Learning-Knowledge-Reasoning paradigm, has three components: 1) A semantic parser, T that converts the text into the required logical form, 2) An Inductive Logic Programming module, L that learns missing knowledge from the training data and 3) A reasoning engine, R which computes the answer given the query. In the training phase, given some background knowledge B and a training dataset D the Inductive Logic Programming module uses the semantic parser T and a rule learning algorithm to learn the necessary knowledge H from D. In the test phase, both B and H are used to answer a given question. We have used the language of Answer Set Programming for the purpose of knowledge representation and reasoning.

2 Background

2.1 Answer Set Programming

An answer set program is a collection of rules of the form,

$$L_0 \leftarrow L_1, ..., L_m, \textbf{not } L_{m+1}, ..., \textbf{not } L_n$$

where each of the L_i's is a literal in the sense of a classical logic. Intuitively, the above rule means that if $L_1, ..., L_m$ are true and if $L_{m+1}, ..., L_n$ can be safely assumed to be false then L_0 must be true. The left-hand side of an ASP rule is called the *head* and the right-hand side is called the *body*. Predicates and ground terms in a rule start with a lower case letter, while variable terms start with a capital letter. We will follow this convention throughout the paper. A rule with no *head* is called a *constraint*. A rule with empty *body* is referred to as a *fact*. The semantics of ASP is based on the stable model semantics of logic programming [2].

Table 2 The basic predicates and axioms of Simple Discrete Event Calculus (SDEC).

Predicate	Meaning
happensAt(F, T)	Event E occurs at time T
initiatedAt(F, T)	At time T a period of time for which fluent F holds is initiated
terminatedAt(F, T)	At time T a period of time for which fluent F holds is terminated
holdsAt(F, T)	Fluent F holds at time T
Axioms	
$holdsAt(F, T+1)$ $\leftarrow initiatedAt(F, T).$	$holdsAt(F, T+1) \leftarrow$ $holdsAt(F, T),$ $not\ terminatedAt(F, T).$

In this work, both the background knowledge B and the learned knowledge H are a collection of such ASP rules.

2.2 Event Calculus

Event calculus is a temporal logic for reasoning about the events and their effects. The ontology of the Event calculus comprises of *time points*, *fluents* (i.e. properties which have certain values at a time point) and *events* (i.e. occurrences in time that may affect fluents and alter their value). The formalism also contains two domain-independent axioms to incorporate the commonsense *law of inertia*, according to which fluents persist over time unless they are affected by an event. The building blocks of Event calculus and its domain independent axioms are presented in Table 2.

3 Inductive Logic Programming for Mutually Distinct Examples

Inductive Logic Programming (ILP) [7] is a subfield of Machine learning that is focused on learning logic programs. Given a set of positive examples $\mathcal{E}^+$, negative examples $\mathcal{E}^-$ and some background knowledge $\mathcal{B}$, an ILP algorithm finds an Hypothesis $\mathcal{H}$ (answer set program) such that $\mathcal{B} \cup \mathcal{H} \models \mathcal{E}^+$ and $\mathcal{B} \cup \mathcal{H} \not\models \mathcal{E}^-$. The possible hypothesis space is often restricted with a language bias that is specified by a series of mode declarations $\mathcal{M}$ [8].

This definition however does not consider the fact that a statistical machine learning dataset contains several *context dependent* examples. We recently proposed a variation of the standard ILP task namely, Inductive Logic Programming for "mutually Distinct Examples" [6] which is more suitable for working with this machine learning datasets. An ILP task for "mutually Distinct Examples" [6] (denoted as ILP^{DE}) is defined as follows:

▶ **Definition 1 (Inductive Logic Programming for Mutually Distinct Examples).** An ILP task for *Distinct Examples* (denoted as ILP^{DE}) is a tuple $\langle B, M, D \rangle$, where B is an Answer Set Program, called the background knowledge, M defines the set of rules allowed in hypotheses (the hypothesis space) and D is the dataset containing a series of mutually distinct examples $\langle E_1, E_2, ..., E_n \rangle$. Here each E_i is a tuple $\langle O_i, E_i^+, E_i^- \rangle$ where, O_i is a logic program, called *observation*, E^+ is a set of positive ground literals and E^- is a set of negative ground literals.

```
(g / grab
    :ARG0 (p / person
        :name (m / name :op1 Mary ))
    :ARG1 (f / football ))
```

Figure 1 AMR representation of "Mary grabbed the football."

```
(c / carry
    :ARG0 (m / Marry )
    :ARG1 (a / amr-unknown)))
```

Figure 2 AMR representation of "What is Mary carrying?"

A hypothesis H is an inductive solution of T (written as $H \in ILP^{DE}(B, M, D)$) *iff,*

$$H \cup B \cup O_i \vdash E_i^+, \ \forall i = 1...n$$

$$H \cup B \cup O_i \nvdash E_i^-, \ \forall i = 1...n$$

An iterative and incremental algorithm, has also been developed [6] to compute the solution of an ILP^{DE} task.

4   Learning Knowledge from dataset

To learn the missing knowledge H from the training dataset D, first an instance of the ILP^{DE} task is created. The iterative and incremental algorithm for ILP^{DE} in [6] is then used which outputs the desired H. In this section we describe this procedure with the example of the bAbI question answering challenge.

Background Knowledge B

The background knowledge contains the two commonsense *laws of inertia* from Event calculus, according to which fluents persist over time unless they are affected by an event.

Mapping an bABI Example to an ILP^{DE} Example

The bAbI challenge contains 20 different question answering tasks. One of such task is about reasoning with sets. An example of that which is shown in table 1. The training dataset for each tasks contains 1000 of such examples. Each of such example is translated into an ILP^{DE} example $E_i = <O_i, E_i^+, E_i^->$ in the following manner.

Given a question-answer text such as the one shown in Table 1(a), the translation module first converts the natural language sentences to the syntax of Event calculus. While doing so, it first obtains the Abstract Meaning Representation (AMR) of the sentence from the AMR parser in the statistical NLP layer and then applies a rule-based procedure to convert the AMR graph to the syntax of Event calculus. Figure 1 & 2 show two AMR representations for the sentence "Mary grabbed the football." and the question "What is Mary carrying?". The representation of the question-answer text in $<O_i, E_i^+, E_i^->$ form is shown in Table 3. The narratives in O_i (Table 3) describe that the event of grabbing a football by Mary has happened at time point 1, then another event named *travel* has happened at time point 2 and so on. The first two annotations in E_i^+ state that both the fluents specifying Mary is carrying an apple and Mary is carrying a football holds at time point 4. The *not holdsAt* annotation in E_i^- states that at time point 7 Mary is not carrying a football.

Table 3 Representation of the Example in Table 1(a) in ILP^{DE} format.

O_i	$happensAt(grab(mary, football), 1).$ $happensAt(travel(mary, office), 2).$ $happensAt(take(mary, apple), 3).$ $happensAt(leave(mary, footbal;), 5).$ $happensAt(go\_back(daniel, bedroom), 6).$
E_i^+	holdsAt(carry(mary,football),4). holdsAt(carry(mary,apple),4). holdsAt(carry(mary,apple),7).
E_i^-	not holdsAt(carry(mary,football),7).

Table 4 Rules learned from the task 8 of bABI dataset.

$$initiatedAt(carry(P,O),T) \leftarrow happensAt(get(P,O),T).$$
$$initiatedAt(carry(P,O),T) \leftarrow happensAt(take(P,O),T).$$
$$terminatedAt(carry(P,O),T) \leftarrow happensAt(drop(P,O),T).$$
$$initiatedAt(carry(P,O),T) \leftarrow happensAt(pick\_up(P,O),T).$$
$$initiatedAt(carry(P,O),T) \leftarrow happensAt(grab(P,O),T).$$
$$terminatedAt(carry(P,O),T) \leftarrow happensAt(discard(P,O),T).$$
$$terminatedAt(carry(P,O),T) \leftarrow happensAt(put\_down(P,O),T).$$
$$terminatedAt(carry(P,O),T) \leftarrow happensAt(leave(P,O),T).$$

Computing the Inductive Solution

The algorithm [6] that computes the solution roughly works as follows: Given an instance of the ILP^{DE} task, it first finds a solution H_1 of E_1. Then it expands H_1 minimally to solve only E_2 and obtains H_2. In the next iteration it again expands H_2 minimally to solve E_1 and it continues expanding until it finds a hypothesis that solves both E_1 and E_2. Next it starts with a solution of $\langle E_1, E_2 \rangle$ and tries to expand it iteratively until it solves all of E_1, E_2 and E_3. The process continues until a hypothesis is found that explains all the examples. The algorithm is shown to be sound and complete when $H \cup B \cup O_i$ is *stratified* for all $i = 1, ..., n$, [6]. Table 4 shows the 8 rules that are learned for this task. Our system following this learning-knowledge-reasoning method outperforms all the deep learning systems for the bAbI challenge. [5].

5   Current State of Research

Currently we are trying to apply this framework of learning-knowledge-reasoning to the task of word arithmetic problem solving, where the goal is to learn human readable knowledge which can help the question answering agent to decide which arithmetic formulas to apply for a particular problem and in which order.

6   Conclusion

Earlier days of Artificial Intelligence have seen many handwritten rule based systems. Later those were replaced by better performing machine learning based systems. With the advancements of knowledge representation and reasoning languages, a natural question arises, "if machines can learn logic programs, can they achieve better accuracy than existing statistical

machine learning methods such neural networks?" It should be noted that the system of [5] achieved better results than the existing deep learning models on the bAbI dataset. To further explore this possibility we need to focus on the task of learning of logic programs and need to develop systems that can learn from large datasets. In this research, we have made an attempt towards that.

References

1   Peter Clark. Elementary school science and math tests as a driver for AI: take the aristo challenge! *Innovative Applications of Artificial Intelligence*, 2015.

2   Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

3   Mohammad Javad Hosseini, Hannaneh Hajishirzi, Oren Etzioni, and Nate Kushman. Learning to solve arithmetic word problems with verb categorization. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 523–533, 2014.

4   Rik Koncel-Kedziorski, Hannaneh Hajishirzi, Ashish Sabharwal, Oren Etzioni, and Siena Dumas Ang. Parsing algebraic word problems into equations. *Transactions of the Association for Computational Linguistics*, 3:585–597, 2015.

5   Arindam Mitra and Chitta Baral. Addressing a Question Answering Challenge by Combining Statistical Methods with Inductive Rule Learning and Reasoning. In *AAAI*, pages 2779–2785, 2016.

6   Arindam Mitra and Chitta Baral. Incremental and Iterative Learning of Answer Set Programs from Mutually Distinct Examples. *CoRR*, abs/1802.07966, 2018. `arXiv:1802.07966`.

7   Stephen Muggleton. Inductive logic programming. *New generation computing*, 8(4):295–318, 1991.

8   Stephen Muggleton. Inverse entailment and Progol. *New generation computing*, 13(3–4):245–286, 1995.

9   Jason Weston, Antoine Bordes, Sumit Chopra, and Tomas Mikolov. Towards AI-complete question answering: a set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015. `arXiv:1502.05698`.

Speeding up Lazy-Grounding Answer Set Solving

Richard Taupe

Alpen-Adria-Universität, Klagenfurt, Austria

Siemens AG Österreich, Vienna, Austria

rtaupe@edu.aau.at

https://orcid.org/0000-0001-7639-1616

Abstract

The grounding bottleneck is an important open issue in Answer Set Programming. Lazy grounding addresses it by interleaving grounding and search. The performance of current lazy-grounding solvers is not yet comparable to that of ground-and-solve systems, however. The aim of this thesis is to extend prior work on lazy grounding by novel heuristics and other techniques like non-ground conflict learning in order to speed up solving. Parts of expected results will be beneficial for ground-and-solve systems as well.

1 Introduction

Answer Set Programming (ASP) is an approach to declarative problem solving [3, 22], in which problems to be solved by a computer are encoded as logic programs, which are sets of rules that can contain variables. Most ASP systems follow the *ground-and-solve* paradigm and split the solving process into two steps: First, a *grounder* transforms the input program containing variables into a propositional encoding [4, 12, 14, 19]. Then, solutions for the resulting variable-free program are found by a *solver* [16]. The grounding step can result in an exponential blow-up in space in the worst case [10].

This *grounding bottleneck* is a major problem of traditional approaches to ASP. For example, the rule

$$partnerunits(U, P) \leftarrow unit2zone(U, Z), unit2sensor(P, S), zone2sensor(Z, S), U \neq P.$$

has to be replaced by up to $|U| \cdot |P| \cdot |Z| \cdot |S|$ ground rules, where $|U|$, $|P|$, $|Z|$ and $|S|$ are the number of constants the respective variable may be substituted with. Many of these ground rules may be irrelevant because they are not needed to build a specific answer set anyway.

Problems that are actually easy to solve thus become prohibitive as soon as their grounding ceases to fit into working memory. This makes ASP, an otherwise powerful and versatile approach, unsuitable for large-scale problem instances frequently occurring in practice.

Lazy grounding interleaves grounding and solving to avoid storing the entire ground program in memory. By this, lazy grounding addresses the limitations of state-of-the-art grounders like GRINGO [14] and $\mathcal{I}$-DLV [4,12]. These grounders employ sophisticated grounding techniques to omit irrelevant ground rules, but these can only mitigate and not eliminate the blow-up in space. Known approaches to lazy grounding are ASPeRiX [20], GASP [25], OMiGA [7], and, most recently, ALPHA [30]. Lazy grounding methods have also been proposed for FO($\cdot$), a knowledge representation formalism whose foundations are similar to those of ASP [8]. Also related, though not a lazy-grounding system as such, are *lazy constraints*, a technique that removes constraints that consume much space in grounding from the input program and adds only relevant ground ones again when a potential answer set has been found that needs to be checked for constraint violations [6].

While lazy-grounding systems are able to limit their memory usage, their time consumption is not yet comparable to that of state-of-the-art solvers. One reason for this is that most of these systems do not exploit conflict-driven nogood learning (CDNL), which is a key success factor of state-of-the-art ASP solvers. ALPHA has been the first lazy-grounding system to employ CDNL [30]. The system consists of a grounder and a solver which, however, do not work in sequence (as in ground-and-solve), but interact cyclically. Still, ALPHA also does not reach the performance of traditional solvers yet. One reason for this is that ALPHA (and all other lazy-grounding systems) lack powerful search heuristics to guide the exploration of the search space, which are another major success factor of traditional systems.

The remainder of this research summary is structured as follows: In section 2, we recall preliminaries on ASP and lazy grounding. In section 3, we state the research questions addressed by this thesis, after which we report on its current status in section 4. Preliminary results are presented in section 5, before open issues and expected achievements are put forward in section 6. This paper is then briefly concluded in section 7.

2   Preliminaries

In this section, we present a brief account of ASP syntax and semantics and of the general idea of the lazy-grounding ASP solver ALPHA.

2.1   Syntax

An answer-set program P is a finite set of rules of the form

$$h_1; \ldots; h_d \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n.$$

where $h_1, \ldots, h_d$ and $b_1, \ldots, b_m$ are positive literals (i.e. atoms) and $not\ b_{m+1}, \ldots, not\ b_n$ are negative literals. An atom is either a classical atom or a cardinality atom[1]. A classical atom is an expression $p(t_1, \ldots, t_n)$ where p is an n-ary predicate and $t_1, \ldots, t_n$ are terms. A term is either a variable or a constant. A literal is either an atom a or its default negation $not\ a$. Default negation refers to the absence of information, i.e. an atom is assumed to be false as long as it is not proven to be true. A *cardinality atom* is of the form

$$l\ \{a_1 : l_{1_1}, \ldots, l_{1_m}; \ldots; a_n : l_{n_1}, \ldots, l_{n_o}\}\ u$$

---

[1] Other types of atoms are supported in the language standard ASP-Core-2 [2], but these are not needed within the scope of this article.

where each structure $a_i : l_{i_1}, \ldots, l_{i_m}$ is a *conditional literal* in which a_i (the head of the conditional literal) and all l_{i_j} are classical literals, and l and u are terms representing non-negative integers indicating lower and upper bound. If one or both of the bounds are not given, their defaults are used, which are 0 for l and ∞ for u.

$H(r) = \{h_1, \ldots, h_d\}$ is called the *head*, and $B(r) = \{b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n\}$ the *body* of the rule. A rule r with $H(r)$ consisting of a cardinality atom is called *choice rule*. A rule with a head consisting of more than one classical atom is called *disjunctive rule*. A rule whose head consists of at most one classical atom is called a *normal rule*. A normal rule with empty head, e.g. $\leftarrow b.$, is called *(integrity) constraint*. A normal rule with empty body, e.g. $h \leftarrow .$, is called *fact*.

2.2 Semantics

There are several ways to define the semantics of an answer-set program, i.e. to define the set of answer sets of an answer-set program. An overview is provided by [23]. Probably the best-known semantics is based on the *Gelfond-Lifschitz reduct* [17]. A variant that applies to choice rules also is presented in [5].

Informally, an answer set A of a program P is a subset-minimal model of P (i.e. a set of atoms interpreted as *true*) which satisfies the following conditions: All rules in P are satisfied by A; and all atoms in A are "derivable" by rules in P. A rule is satisfied if its head is satisfied or its body is not. The disjunctive head of a rule is satisfied if at least one of its atoms is. A cardinality atom is satisfied if $l \leq |C| \leq u$ holds, where C is the set of head atoms in the cardinality literal whose conditions (e.g. $l_{i_1}, \ldots, l_{i_m}$ for a_i) are satisfied and which are satisfied themselves. In the presence of choice rules, the semantics is adjusted to allow non-minimal subsets satisfying the cardinality atom to appear in answer sets.

2.3 Lazy Grounding and Solving

ASP systems employing lazy grounding, such as Alpha, are based on so-called computation sequences, which are sequences of firing rules. Starting from facts, rules are fired one after the other by choosing in each step among the set of *applicable* rules, which are ground rules whose positive body is already satisfied and whose negative body is false or unassigned. This implies that the solver guesses whether an applicable rule fires, while traditional CDNL-based search guesses whether an arbitrary atom is *true* or *false*. Lazy-grounding solvers need an additional truth value *must-be-true* to distinguish whether an atom was derived by a firing rule or by a constraint [30]. When a conflicting assignment is reached, the solver backtracks. In this case, CDNL can learn new information from the conflict that is then used to avoid encountering similar conflicts in the future [16].

3 Research Questions

The remaining performance issues in lazy-grounding ASP solving lead us to the central research questions of this thesis:

1. How can lazy-grounding solvers be enabled to solve large-scale (industrial) problem instances as efficiently as traditional solvers solve smaller instances?
2. How can conflict learning contribute to that goal, and can conflicts be reused across problem instances?
3. How can various forms of heuristics, e.g. domain-independent or domain-specific search heuristics, contribute to that goal?

Within the scope of the DynaCon research project[2], lazy grounding methods will be evaluated on real-world industrial problem instances from domains like cyber-physical systems, road traffic control, and railway operation [11].

4   Current Status of the Research

To date, our research efforts have focused on research question 3, i.e. on heuristics. Preliminary results for domain-independent search heuristics have been reported at the 1st International Workshop on Practical Aspects of Answer Set Programming [28] and at the LPNMR 2017 Doctoral Consortium [27]. Lazy grounding and other ASP-based approaches to large-scale product configuration problems have been investigated in a contribution to the 19th International Configuration Workshop [26]. Since then, our research focus has shifted to domain-specific heuristics. A conference paper on this topic is currently being written.

5   Preliminary Results Accomplished

Source code has been contributed to Alpha, a lazy-grounding system introduced by Antonius Weinzierl [30]. Contributions are made under an open-source license and are freely available at `https://github.com/alpha-asp`.

While initial work aimed at comprehending the solver's inner workings and making small improvements and extensions on the go, our focus has soon shifted to the development of novel search heuristics.

5.1   Heuristics for Lazy-Grounding ASP Solving

Alpha takes ideas from state-of-the-art ASP solvers that work on a full grounding. Therefore it is natural to investigate heuristics from such systems and try to apply them in the solver component of a lazy-grounding system like Alpha. Heuristics for answer-set solving can roughly be classified into *domain-independent heuristics*, which are designed without a concrete application domain in mind, and *domain-specific heuristics*, which have to be tailored to a specific problem. For the class of domain-independent heuristics, two prominent examples are *VSIDS* [24] and *BerkMin* [18], which have originally been developed for SAT but are also successfully employed by ASP solvers (such as clasp [16] and wasp [1]). Both assign a so-called *activity* counter to every variable that counts the number of clauses involving this variable that are responsible for at least one conflict. These counters are divided by a constant ("decayed") periodically to reduce the influence of "aged" clauses. When the heuristic is asked for an atom, it chooses the most active unassigned atom. This is done to regard the fact that the set of variables responsible for conflicts may change very quickly. *BerkMin* additionally organizes the set of conflict clauses as a chronologically ordered stack, thereby preferring variables in recent conflicts. Other counters are maintained for picking truth values.

A direct application of *BerkMin* or *VSIDS* to a lazy-grounding ASP system like Alpha seems unnatural because such a solver differs in many important ways from a solver adhering to the classical ground-and-solve paradigm. One major difference is that not all ground rules, and consequently not all ground atoms, are known at any time to a lazy-grounding solver. Because of this, a heuristic that applies ideas from *BerkMin* or *VSIDS* to lazy grounding can

---

[2] Dynamic knowledge-based (re)configuration of cyber-physical systems, `https://isbi.aau.at/dynacon`

only incorporate atoms that are already grounded and thus known to the solver. Another major difference lies in the solving mechanism: while a traditional ASP solver can choose any atom to guess on, Alpha only guesses on atoms representing bodies of applicable rules (cf. section 2.3).

5.2 Domain-Independent Heuristics

A set of domain-independent heuristics inspired by BerkMin and incorporating new ideas has been developed for Alpha and is described in detail in [28], where the results of a basic experimental evaluation on a number of benchmark problems can also be found. Although the heuristics presented are still under development and only a brief experimental study was conducted, promising results can be seen.

The novel family of "dependency-driven" heuristics was repeatedly able to outperform Alpha's naive heuristic as well as two BerkMin-inspired heuristics. It extends the latter by expanding the scope of atoms considered by the heuristics: If the atom a chosen due to its activity and recency is a choice point, i.e. it represents the body of an applicable rule, it is immediately picked. If that is not the case, the set of choice points *depending* on a are considered for selection, where a choice point representing the body of a ground rule $r\sigma$ is said to depend on all atoms occurring in $H(r\sigma) \cup B(r\sigma)$.

While our results are encouraging, there is obviously more work to be done to improve the performance of these heuristics and the solver in general.

5.3 Domain-Specific Heuristics

Domain-specific heuristics have been proposed for pre-grounding solvers but are not directly transferable to a lazy-grounding system for the same reasons that domain-independent heuristics are not. hclasp [15] is an extension of the solver clasp that accepts heuristic predicates as part of the declarative problem specification. These predicates allow to modify priorities and truth preferences of atoms, i.e. the order in which atoms are guessed and the truth values assigned to them during solving. These modifications can be mixed with domain-independent heuristics like VSIDS. Heuristic predicates have since been replaced by heuristic directives in clasp [13].

hwasp [9], on the other hand, is an extension of the solver wasp that facilitates the integration of external heuristics implemented in a procedural language which are consulted at specific points during the solving process via a prespecified API.

Our current efforts are directed at devising an extension of the ASP language by annotations or directives to specify heuristics declaratively within the input program. Our preliminary proposal for such annotations is syntactically similar to *optimize statements* in ASP-Core-2 [2] or *weak constraints* in DLV [21].

A rule to which an heuristic annotation is attached could be of the following form:

$$h_1; \ldots; h_d \leftarrow b_1, \ldots, b_m, not\ b_{m+1}, \ldots, not\ b_n.\ [w@l, s\colon c_1, \ldots, c_o]$$

Here, w and l are terms denoting weight and level of the heuristics (together called *priority*, in which level is more important than weight), both defaulting to 1, and s is a sign (*true* or *false*) stating if the rule shall fire or not fire when selected.

During solving, all rules that have already been grounded and whose condition $c_1, \ldots, c_o$ is satisfied by the current partial assignment[3] are candidates for rule selection. From these

---

[3] To be in line with semantics of default negation, atoms that are still unassigned are assumed false.

candidates, the solver will choose the one with the highest priority $w@l$ and assign s to the atom representing its body. If multiple applicable rules have the same maximum priority, a fallback heuristics like BerkMin or VSIDS is used to break ties. If s is not specified, the sign is also determined by a fallback heuristics. If a condition is not satisfied, the corresponding rule stays applicable but has default weight and level ($1@1$).

This syntax is still preliminary. We are currently working on extending it with ways to specify preferences over disjunctive heads or elements of a choice head, and with support for randomness and restarts. A full proposal together with example programs and a performance study will form a forthcoming publication.

6   Open Issues and Expected Achievements

So far, we have concentrated on search heuristics, i.e. on branching strategies for the solver component of a lazy-grounding ASP system. Several other procedures of such a system could be equipped with heuristic decision-making as well. A prominent group of such heuristics is formed by grounding heuristics, which are relevant for lazy-grounding systems only. They deal with questions like how much to ground at a given point in time and when to save space by forgetting grounded nogoods or doing a restart. Some work in this direction has already been done for FO($\cdot$) [8]. Ways to adapt and extend this for use by a CDNL-based solver will be explored.

Furthermore, we plan to explore other ways of modifying or enhancing the algorithms of a lazy-grounding ASP system to improve performance by aiding heuristic decision-making. For example, program analysis techniques (atom dependencies, strongly connected components, etc.) could guide the selection or tuning of solver heuristics, and relaxing the restrictions which atoms we can guess on could open up new ways towards finding answer sets quickly. By these efforts, we expect the solver heuristics to have more information and operate on a smaller search space – and thus to perform even better then currently is the case.

Also, our research will not be limited to heuristics but include other important issues of lazy-grounding ASP systems as well. One such issue is to extend CDNL to generalise learnt nogoods to the non-ground level and reuse such non-ground nogoods for future problem instances. Preliminary work on non-ground rule learning has already been done in the scope of OMiGA [29]. There, a technique called *rule unfolding* is proposed, which is based on ideas from propositional resolution and derives new rules that are implied by the original program. A new rule that is a constraint containing variables can be seen as a non-ground nogood. To the best of our knowledge, non-ground rule learning has not yet been addressed in conjunction with CDNL, and the question how such learnt rules can be utilised to accelerate solving of future problem instances has also remained unstudied.

We expect that several aspects of our work will be beneficial not only for lazy-grounding systems, but for ground-and-solve systems as well.

7   Conclusion

Lazy grounding is an approach to solve the grounding bottleneck in ASP by interleaving grounding and solving. The performance of current lazy-grounding systems is not on par with that of systems producing the full grounding upfront, however. The goal of this thesis is to equip the lazy-grounding approach with novel heuristics and other techniques to make solving faster. Preliminary results have been accomplished in the area of domain-independent and domain-specific search heuristics. To the knowledge of the authors, this investigation of search heuristics in lazy-grounding ASP systems is the first of its kind. Future achievements on heuristics and conflict learning are expected to be transferable to ground-and-solve systems.

References

1 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`.

2 ASP Standardization Working Group. ASP-Core-2 input language format, 2012-12-13. URL: `https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf`.

3 Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011. `doi:10.1145/2043174.2043195`.

4 Francesco Calimeri, Davide Fuscà, Simona Perri, and Jessica Zangari. $\mathcal{I}$-dlv: The New Intelligent Grounder of dlv. 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 192–207. Springer, 2016. `doi:10.1007/978-3-319-49130-1_15`.

5 Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the Fifth Answer Set Programming Competition. *Artif. Intell.*, 231:151–181, 2016. `doi:10.1016/j.artint.2015.09.008`.

6 Bernardo Cuteri, Carmine Dodaro, Francesco Ricca, and Peter Schüller. Constraints, lazy constraints, or propagators in ASP solving: An empirical analysis. *TPLP*, 17(5-6):780–799, 2017. `doi:10.1017/S1471068417000254`.

7 Minh Dao-Tran, Thomas Eiter, Michael Fink, Gerald Weidinger, and Antonius Weinzierl. OMiGA : An open minded grounding on-the-fly answer set solver. In Luis Fariñas del Cerro, Andreas Herzig, and Jérôme Mengin, editors, *Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings*, volume 7519 of *Lecture Notes in Computer Science*, pages 480–483. Springer, 2012. `doi:10.1007/978-3-642-33353-8_38`.

8 Broes de Cat, Marc Denecker, Peter J. Stuckey, and Maurice Bruynooghe. Lazy Model Expansion: Interleaving Grounding with Search. *J. Artif. Intell. Res.*, 52:235–286, 2015. `doi:10.1613/jair.4591`.

9 Carmine Dodaro, Philip Gasteiger, Nicola Leone, Benjamin Musitsch, Francesco Ricca, and Konstantin Schekotihin. Combining Answer Set Programming and domain heuristics for solving hard industrial problems (Application Paper). *TPLP*, 16(5-6):653–669, 2016. `doi:10.1017/S1471068416000284`.

10 Thomas Eiter, Wolfgang Faber, Michael Fink, and Stefan Woltran. Complexity results for answer set programming with bounded predicate arities and implications. *Ann. Math. Artif. Intell.*, 51(2-4):123–165, 2007. `doi:10.1007/s10472-008-9086-5`.

11 Thomas Eiter, Gerhard Friedrich, Richard Taupe, and Antonius Weinzierl. Lazy Grounding for Dynamic Configuration. *KI - Künstliche Intelligenz*, May 2018. `doi:10.1007/s13218-018-0536-x`.

12 Wolfgang Faber, Nicola Leone, and Simona Perri. The Intelligent Grounder of DLV. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2012. `doi:10.1007/978-3-642-30743-0_17`.

13 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory Solving Made Easy with Clingo 5. In Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos, editors, *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016,*

New York City, USA, volume 52 of *OASICS*, pages 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. `doi:10.4230/OASIcs.ICLP.2016.2`.

14   Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* Series 3. In James P. Delgrande and Wolfgang Faber, editors, *Logic Programming and Non-monotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, volume 6645 of *Lecture Notes in Computer Science*, pages 345–351. Springer, 2011. `doi:10.1007/978-3-642-20895-9_39`.

15   Martin Gebser, Benjamin Kaufmann, Ramón Otero, Javier Romero, Torsten Schaub, and Philipp Wanko. Domain-Specific Heuristics in Answer Set Programming. In Marie des-Jardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA.*, pages 350–356. AAAI Press, 2013. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6278`.

16   Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012. `doi:10.1016/j.artint.2012.04.001`.

17   Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

18   Evguenii I. Goldberg and Yakov Novikov. BerkMin: A Fast and Robust Sat-Solver. In *2002 Design, Automation and Test in Europe Conference and Exposition (DATE 2002), 4-8 March 2002, Paris, France*, pages 142–149. IEEE Computer Society, 2002. `doi:10.1109/DATE.2002.998262`.

19   Benjamin Kaufmann, Nicola Leone, Simona Perri, and Torsten Schaub. Grounding and Solving in Answer Set Programming. *AI Magazine*, 37(3):25–32, 2016. URL: `http://www.aaai.org/ojs/index.php/aimagazine/article/view/2672`.

20   Claire Lefèvre, Christopher Béatrix, Igor Stéphan, and Laurent Garcia. ASPeRiX, a first-order forward chaining approach for answer set computing. *TPLP*, 17(3):266–310, 2017. `doi:10.1017/S1471068416000569`.

21   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`.

22   Vladimir Lifschitz. What Is Answer Set Programming? In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008. URL: `http://www.aaai.org/Library/AAAI/2008/aaai08-270.php`.

23   Vladimir Lifschitz. Thirteen Definitions of a Stable Model. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday*, volume 6300 of *Lecture Notes in Computer Science*, pages 488–503. Springer, 2010. `doi:10.1007/978-3-642-15025-8_24`.

24   Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. `doi:10.1145/378239.379017`.

25   Alessandro Dal Palù, Agostino Dovier, Enrico Pontelli, and Gianfranco Rossi. GASP: answer set programming with lazy grounding. *Fundam. Inform.*, 96(3):297–322, 2009. `doi:10.3233/FI-2009-180`.

26   Gottfried Schenner and Richard Taupe. Techniques for Solving Large-Scale Product Configuration Problems with ASP. In Linda L. Zhang and Albert Haag, edit-

ors, *Proceedings of the 19th International Configuration Workshop*, pages 12–19, La Défense, France, 2017. URL: `https://www.ieseg.fr/wp-content/uploads/2017/01/Proceedgins_FinalV2.pdf#page=12`.

27 Richard Taupe. Lazy Grounding and Heuristic Solving in Answer Set Programming. In Marina de Vos, editor, *LPNMR 2017 Doctoral Consortium*, Bath, 2017. URL: `http://lpnmr-dc.cs.bath.ac.uk/programme/Proceedings.pdf#page=41`.

28 Richard Taupe, Antonius Weinzierl, and Gottfried Schenner. Introducing Heuristics for Lazy-Grounding ASP Solving. In *1st International Workshop on Practical Aspects of Answer Set Programming*, 2017. URL: `https://sites.google.com/site/paoasp2017/Taupe-et-al.pdf`.

29 Antonius Weinzierl. Learning Non-Ground Rules for Answer-Set Solving. In David Pearce, Shahab Tasharrofi, Evgenia Ternovska, and Concepción Vidal, editors, *2nd Workshop on Grounding and Transformations for Theories With Variables*, pages 25–37, 2013. URL: `http://kr.irlab.org/sites/10.56.35.200.gttv13/files/gttv13.pdf#page=31`.

30 Antonius Weinzierl. Blending Lazy-Grounding and CDNL Search for Answer-Set Solving. In Marcello Balduccini and Tomi Janhunen, editors, *Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings*, volume 10377 of *Lecture Notes in Computer Science*, pages 191–204. Springer, 2017. `doi:10.1007/978-3-319-61660-5_17`.

Knowledge Authoring and Question Answering via Controlled Natural Language

Tiantian Gao

Department of Computer Science, Stony Brook University
Stony Brook, NY, USA
tiagao@cs.stonybrook.edu

Abstract

Knowledge acquisition from text is the process of automatically acquiring, organizing and structuring knowledge from text which can be used to perform question answering or complex reasoning. However, current state-of-the-art systems are limited by the fact that they are not able to construct the knowledge base with high quality as knowledge representation and reasoning (KRR) has a keen requirement for the accuracy of data. Controlled Natural Languages (CNLs) emerged as a technology to author knowledge using a restricted subset of English. However, they still fail to do so as sentences that express the same information may be represented by different forms. Current CNL systems have limited power to standardize sentences that express the same meaning into the same logical form. We solved this problem by building the Knowledge Authoring Logic Machine (KALM), which is a technology for domain experts who are not familiar with logic to author knowledge using CNL. The system performs semantic analysis of English sentences and achieves superior accuracy of standardizing sentences that express the same meaning to the same logical representation. Besides, we developed the query part of KALM to perform question answering, which also achieves very high accuracy in query understanding.

1 Introduction

Knowledge acquisition is the process of extracting, organizing, and structuring knowledge from data sources such that the constructed knowledge base can be used for question answering or performing complex reasoning. Traditional ways of knowledge acquisition largely reply on domain experts to encode the knowledge base in rule-based systems such as XSB [12] and Clingo [4]. However, this requires too much domain specific knowledge and eligible engineers are in very short supply. Information extraction systems emerged as the tools to extract knowledge frame text (i.e., OpenIE [1], SEMAFOR [2], Stanford CoreNLP/KBP [8], SLING [10]). They achieved admirable results in processing free text, however, their accuracy is far from meeting the requirement of knowledge representation. In addition, they are only designed to extract the knowledge from text, but not intended to represent it in a way suitable for reasoning. Controlled Natural Languages (CNLs) [7] emerged as a technology that bridges this gap. Representative systems include Attempto Controlled English (ACE) [3] and Processable English (PENG) [11]. They are designed to process English sentences with restricted grammar but unambiguous interpretations and translate the sentences into logic for reasoning. The main issue with CNLs is that they have
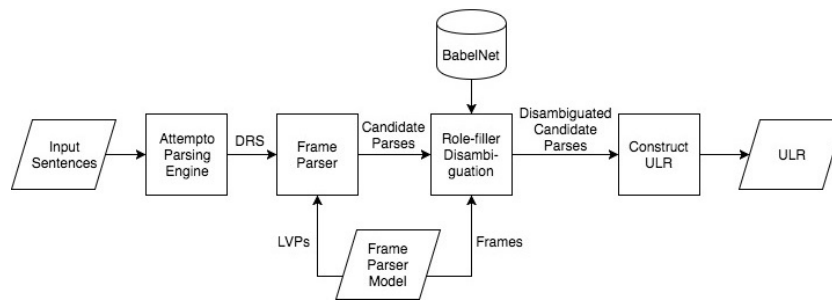
■ **Figure 1** Pipeline for translating a sentence into ULR.

limited power of standardizing sentences that express the same information via different syntactic forms into the same logical representation. For instance, the sentences *a customer buys a phone*, *a customer makes a purchase of a phone*, *a customer is a buyer of a phone* are mapped to different logical representations. Therefore, they are not suffice for question answering or complex logical reasoning.

In this work, we build Knowledge Authoring Logic Machine (KALM), which conducts semantic analysis of CNL sentences and achieves superior accuracy of standardizing English sentences that express the same information via different forms to the same logical form. The system is built based on utilizing linguistic knowledge bases (BabelNet [9] and FrameNet [5]) and our frame-based parsing and disambiguation algorithms. Besides, we developed the query part of KALM which supports high accuracy query parsing and answer retrieval. The following is organized as follows: Section 2 describes the KALM system for knowledge authoring, Section 3 describes the query part of KALM, Section 4 shows the evaluation results of KALM, Section 5 discusses the next steps of work, Section 6 concludes the paper.

2 Knowledge Authoring Logic Machine (KALM)

Figure 1 shows the pipeline of KALM that translates a CNL sentence into *unique logical representation* (*ULR*), the semantic form of CNL sentences. The KALM framework consists of five components:

Syntactic Parsing. We use Attempto Parsing Engine (APE)[1] to parse CNL sentences and translate them into Discourse Representation Structure (DRS) [6], which represents the syntactic and dependency information of the sentences. DRS relies on 7 predicates: `object/6`, `predicate/4`, `property/3`, `modifier_adv/3`, `modifier_pp/3`, `relation/3`, and `has_part/2`. For example, the `object`-predicate represents an entity which corresponds to a noun word in the sentences. A `predicate`-predicate represents an event and the subject and object of the events. `predicate`-predicate corresponds to a verb word in a sentence. For example, given the sentence *A customer buys a phone*, it is parsed into DRS as

```
object(A,customer,countable,na,eq,1)
object(B,phone,countable,na,eq,1)
predicate(C,buy,A,B)
```

---

[1] `https://github.com/Attempto/APE`

Frame-based Parsing. Based on the DRS, the frame-based parser generates a list of *candidate parses*, which represent the frame-semantic relations the sentences entail. For instance, given the sentence *A customer buys a phone*, the frame-based parser generates the following parse result: `Frame(Commerce_Buy, Roles: Buyer = `*customer*`, Goods = `*phone*`)`. The parse says there are two entities: *customer* and *phone*, which are involved in the `Commerce_Buy` relation. The *customer* serves as the `Buyer` role of this frame relation and *phone* serves as the `Goods` role of the frame relation. The parser is constructed based on two components: *logical frames* and *logical valence patterns* (*lvps*). The logical frames represent the definition of the frame relations via Prolog facts. For instance, the `Commerce_Buy`[2] frame is represented as

```
fp(Commerce_Buy,[
    role(Buyer, [bn:00014332n], []),
    role(Seller, [bn:00053479n], []),
    role(Goods, [bn:00006126n,bn:00021045n], []),
    role(Recipient, [bn:00066495n],[]),
    role(Money, [bn:00017803n], [currency])]).
```

where for each `role`-term, the first argument represents the name of the *frame role*, the second argument represents the BabelNet synsets associated which capture the meaning of the role, and the third argument specifies some data type constraints. The lvps represent the grammatical context of a sentence that could potentially entail a frame. Consider the following lvp for extracting an instance of the `Commerce_Buy` frame:

```
lvp(buy, v, Commerce_Buy, [
    pattern(Buyer, verb->subject, required),
    pattern(Goods, verb->object, required),
    pattern(Recipient, verb->pp(for)->dep, optnl),
    pattern(Money, verb->pp(for)->dep, optnl),
    pattern(Seller, verb->pp(from)->dep, optnl)]).
```

The first and second arguments represents a *lexical unit* (a word + *part-of-speech*) that could trigger an instance of the `Commerce_Buy` frame. Next, it comes with a list of `pattern`-terms, each represents the syntactical context between the lexical unit, frame role, and the actual *role-filler* word. The lvps are generated automatically by KALM based on the annotated training sentences, which contains the frame name, lexical unit, and frame elements information. When a new sentence comes, we check every word in the sentence and find whether there exists any lvp whose lexical unit matches the chosen word. If so, we apply the lvp to the sentence and extract an instance of the frame from the sentence.

Role-Filler Disambiguation. Doing frame-based parsing is not enough because the aforementioned frame-based parsing only replies the grammatical information of the sentence. This way of parsing may generate candidate parses that misidentify the frames, role-filler words, or assign the wrong roles to the role-filler words. To rule out the wrong parses, we perform role-filler disambiguation which checks whether the extracted role-filler words are semantically compatible with the frame roles. For each role-filler and role pair, we compute a semantic score. Based on the scores for the role-filler and role pairs, we score the

---

[2] `https://framenet2.icsi.berkeley.edu/fnReports/data/frame/Commerce_buy.xml`

entire candidate parse and removes the ones that falls below a threshold. To compute the semantic score, we first query the role-filler word against BabelNet and get a list of associated BabelNet synsets (called *candidate role-filler synsets*). Then, we traverse BabelNet semantic network and measure the semantic similarity between each candidate role-filler synset and the corresponding role-synset. Basically, we consider all semantic paths that connect the synset pair, and then use a heuristic scoring function to score the path. The candidate role-filler synset which achieves the highest semantic score is chosen and assigned as the disambiguated role-filler synset for the respective role-filler word.

Translating into ULR.   Based on the *disambiguated candidate parses* generated from the role-filler disambiguation step, we translate the parses into ULR. ULR uses `frame/2` and `role/2` predicates to represent instances of frames and roles. ULR uses `synset/2` and `text/2` predicates to represent the synset and text information for the role-filler words. For example, the sentence *a customer buys a phone* is translated into ULR as

```
frame(id_1, Commerce_buy).
role(id_1, Buyer, id_2).
role(id_1, Goods, id_3).
synset(id_2, bn:00022095n). % customer synset
text(id_2, customer).
synset(id_3, bn:00062020n). % phone synset
text(id_3, phone).
```

3   Question Answering

3.1   Issues in CNL-based Queries

The ACE query language[3] supports two types of queries: *true/false*- and *wh*-queries where the query words include *who, where, what,* and so on. A *true/false*-query is translated into DRS the same way as a definite sentence does. For *wh*-queries, APE uses a special predicate `query/2` to represent the *wh*-words. For instance, the query *who buys what?* is represented in DRS as

```
query(A,who)-1/1
query(B,what)-1/3
predicate(C,buy,A,B)-1/2
```

where the variables *who* and *what* are captured by the `query`-predicate.

However, APE only does shallow syntactic analysis of a query. There are a few issues to solve before we can precisely capture the meaning of a query and acquire the intended knowledge. Consider the following query sentences:

1. *Mary buys which car?*
2. *Who buys IBM's stocks?*
3. *Which person buys which car in which place at which price?*
4. *A $person buys a $car in a $place at a $price.*

First of all, as a *wh*-variable is a placeholder for the entities to be shown in the output result, the type of entities the variable represents must be disambiguated and also used for

---

[3] `http://attempto.ifi.uzh.ch/site/docs/ace/6.7/ace_constructionrules.html`
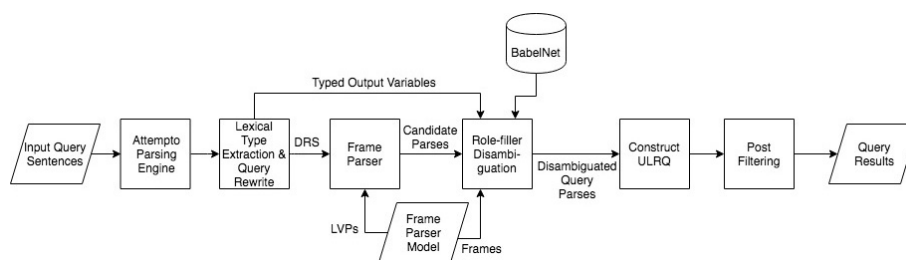
Figure 2 Pipeline for translating a query into ULRQ and answer retrieval and filtering.

acquiring the related information. As shown in Sentence (1) in the above example, we need to identify that the type of the entities associated with the *which*-variable is a *car*. Therefore, if we know *Mary* buys both a *Camry* and a *pen*, only *Camry* should be returned.

Second, ACE's query language has constrained power of denoting types in the query. As shown in Sentence (2) in the above example, emphWho could refer to either a *company* or a *person*. However, it is ambiguous whether the user intends to acquire *company* or *person* entities or both. One solution to that is to use the query word *which* and rewrite the sentence as *Which person buys IBM's stocks* if the user intends to acquire *person* entities. However, the sentence may become cumbersome when there are many such typed variables as shown in Sentence (3). To solve this problem, we introduce *typed output variables* in the query language as the form *$type*. Hence, Sentence (3) will be rewritten to Sentence (4) which is expressed in a more precise and concise way.

Third, to acquire the associated instances of frames from the knowledge base which is constructed in the knowledge authoring phase, we also need to perform frame-semantic parsing based on queries. However, as shown in the previous example, the DRS for query is not exactly the same as the DRS used to represent definite sentences. Therefore, the existing lvps are not applicable for parsing queries. One way to solve this problem is to construct an additional set of training sentences for queries. However, this will requires a lot of work. Besides, since FrameNet doesn't contain any sentences related to queries, it would require a lot of manual work to construct CNL queries. To solve this issue, we perform a DRS rewrite to queries such that we can reuse the existing lvps for definite sentences to parse queries.

3.2 Question Answering

Figure 2 shows the pipeline that translates a CNL query into the logical form, *Unique Logical Representation for Queries* (*ULRQ*), which is used to query the knowledge base to retrieve the answers. The question answering part consists of the following components:

Syntactic Parsing. This is the same as the knowledge authoring part.

Query Parsing. We also perform frame-based parsing to generate several candidate parses which represent the frame relations the query belongs to. However, as mentioned in the previous subsection, the DRS for queries are different from the DRS for definite sentences. Therefore, we perform a DRS adaptation of the DRS corresponding to the query such that the existing lvps for definite sentences can be reused to do frame-based parsing for queries. Besides, we perform a syntactic analysis of the queries and identify the lexical types of the query words (e.g., *which*).

Role-Filler Disambiguation.   This is the same as the knowledge authoring part.

Translating Queries into ULRQ.   Queries are represented in a similar way as definite sentences except that we use logical variables to denote instances of frames and roles. For instance, the query *Who buys a phone?* is translated into ULRQ as

```
?- frame(FrameV,'Commerce_Buy'),
   role(FrameV,'Buyer',BuyV), synset(BuyV,BuyerRoleFillerOutV),
   role(FrameV,'Goods',GoodV), synset(GoodV,GoodsRoleFillerOutV),
   check_type(BuyerRoleFillerOutV,bn:00046516n),  % person synset
   check_type(GoodsRoleFillerOutV,bn:00062020n).  % phone synset
```

Type Filtering of Query Results.   As is shown from the ULRQ above, the clauses from lines 1-3 retrieves all instances of frames and the associated roles from the knowledge base. However, not all role-fillers for `Buyer` and `Goods` may be related to *person* and *phone*. To rule out the unrelated ones, we perform type filtering of the query results, which calls the `check_type` predicate in the above ULRQ.

4   Evaluation

At present, KALM contains 50 logical frames with 213 logical valance patterns. We use the following metrics to measure the performance of the system:

FrSynC	all frames, roles & output variables are identified correctly; all role-filler words & variable types are disambiguated correctly
FrC	all frames, roles and output variables are identified correctly, but some disambiguation mistakes
Wrong	some frames, roles or output variables are misidentified

For knowledge authoring, we achieve an accuracy of 95.6% (FrSynC). This accuracy is far from the state-of-the-art information extraction systems including SEMAFOR, SLING, and Stanford CoreNLP. For understanding of the queries, we achieve an accuracy of 94.49% (FrSynC).

5   Next Steps

The current work focuses on authoring of definite knowledge from CNL sentences and question answering. The next step is to acquire rules from CNL sentences and perform more complex reasoning. This not only requires parsing individual sentences correctly, but also requires multi-sentence parsing and information in different sentences must be related to each other properly. This goes well beyond anaphora resolution, which ACE is already able to handle.
1. *Every bird is an animal.*
2. *Every bird flies.*
3. *Stella is a sea eagle.*
4. *Penguins do not fly.*
5. *A violet is not an animal.*
6. *Sparrow Daffy doesn't fly.*

Consider the above example: Sentences (1) and (2) denote rules which say that if we know there is a *bird*, we can infer the bird is an animal and flies. Therefore, based on Sentences (1), (2) and (3), we can infer *Stella* is an animal and flies. However, this does't hold for *Tweety*

because Sentence (4) is an exception to Sentence (2) and therefore refutes any conclusion derived from Sentence (2). Moreover, based on Sentence (1) and (5), since a violet is not an animal, we conclude that a violet is not a bird. But, this way of reasoning is not desired for Sentence (2) and (6) because Daffy may be injured and therefore not being able to fly.

To precisely capture the meaning of rules in CNL and perform reasoning for the above cases, the research issues are three-fold: the first issue is the development of CNL extensions that are suitable for representing rules and inter-sentence dependencies/references. For instance, in addition to the form "*every . . .*" as shown in Sentence (1) and (2), we can also use an "*if. . .then*" statement to represent a rule. Besides, we need a mechanism to indicate the inter-sentence dependencies as shown in Sentence (1) and (4) where Sentence (4) is an exception case to Sentence (2). This could be done either by specifying the inter-sentence dependencies explicitly or by an automatic mechanism to recognize these dependencies without explicit mentioning.

The second issue is the actual nature of the logic to be used for capturing rules. As shown in the above example, when there is a fact that a *violet* is not an animal, it is natural to infer that it is not an animal. But, it is not reasonable to infer the Daffy is not a bird if Daffy doesn't fly. To distinguish the differences, we can use a *first-order logic* rule to represent Sentence (1) where *contrapositive inference* is desired and use a *Prolog* rule to represent Sentence (2) where contrapositive inference is not required. As to the inter-sentence dependency between Sentence (2) and (4), we believe *defeasible logic* is a good fit. Basically, rules have priories in defeasible logic where the rule with a higher priority can defeat a default rule which has a lower priority. For the above example, we label Sentence (4) as a rule with a higher priority than the rule corresponding Sentence (2) and also defeat the low priority rule when incompatible conclusions are derived.

The third issue is standardization. Same as knowledge authoring for definite sentences and queries, we will also standardize rules that express the same meaning via different syntactic forms.

6 Conclusion

In this work, we described the KALM system, which achieves superior accuracy in knowledge authoring and question answering. The system is built on our frame-based parsing and disambiguation algorithms and the use of external linguistic knowledge bases including BabelNet and FrameNet. As the next step, we plan to work on extracting rules from sentences and perform common sense reasoning.

References

1 Gabor Angeli, Melvin Jose Johnson Premkumar, and Christopher D. Manning. Leveraging Linguistic Structure For Open Domain Information Extraction. In *53rd Annual Meeting of the Association for Computational Linguistics*, pages 344–354, Beijing, China, 2015.

2 Dipanjan Das, Desai Chen, André F. T. Martins, Nathan Schneider, and Noah A. Smith. Frame-Semantic Parsing. *Comp, Linguistics*, 40(1):9–56, 2014. `doi:10.1162/COLI_a_00163`.

3 Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In *Reasoning Web, 4th Intl. Summer School, Sept. 7-11*, pages 104–124, Venice, Italy, 2008.

4   Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Thomas Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011.

5   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.

6   Hans Kamp and Uwe Reyle. *From discourse to logic: Introduction to modeltheoretic semantics of natural language, formal logic and discourse representation theory*, volume 42. Springer Science & Business Media, 2013.

7   Tobias Kuhn. A Survey and Classification of Controlled Natural Languages. *Comp. Linguistics*, 40(1):121–170, 2014.

8   Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP Natural Language Processing Toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014. URL: `http://www.aclweb.org/anthology/P/P14/P14-5010`.

9   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.

10 Michael Ringgaard, Rahul Gupta, and Fernando C. N. Pereira. SLING: A framework for frame semantic parsing. *CoRR*, 1710.07032:1–9, 2017. `arXiv:1710.07032`.

11 Rolf Schwitter. English as a Formal Specification Language. In *13th Intl. Workshop on Database and Expert Systems Appl. (DEXA 2002)*, pages 228–232, Aix-en-Provence, France, 2002.

12 T. Swift and D.S. Warren. XSB: Extending the power of prolog using tabling. *Theory and Practice of Logic Programming*, 2011.

Natural Language Generation From Ontologies Using Grammatical Framework

Van Duc Nguyen

Computer Science Department
New Mexico State University, USA
vnguyen@cs.nmsu.edu

Abstract

The paper addresses the problem of automatic generation of natural language descriptions for ontology-described artifacts. The motivation for the work is the challenge of providing textual descriptions of automatically generated scientific workflows (e.g., paragraphs that scientists can include in their publications). The extended abstract presents a system which generates descriptions of sets of atoms derived from a collection of ontologies. The system, called `nlgPhylogeny`, demonstrates the feasibility of the task in the *Phylotastic* project, that aims at providing evolutionary biologists with a platform for automatic generation of phylogenetic trees given some suitable inputs. `nlgPhylogeny` utilizes the fact that the Grammatical Framework (GF) is suitable for the natural language generation (NLG) task; the abstract shows how elements of the ontologies in Phylotastic, such as web services, inputs and outputs of web services, can be encoded in GF for the NLG task.

1 Introduction

In many applications whose users are not proficient in computer programming, it is of the utmost important to be able to communicate the results of a computation to the users in an easily understandable way (e.g., text rather than a complex data structure). The problem of generating natural language explanations has been explored in several research efforts. For example, the problem has been studied in the context of question-answering systems[1], providing recommendations[2], etc. With the proliferation of spoken dialogue systems and conversational agents on mobile robots, phones, etc., verbal interfaces such as Amazon Echo and Google Home for human-robot-interaction, and the availability of text-to-speech programs such as the TTSReader Extension[3], the application arena of systems capable of generating natural language representation will just become larger.

In this paper, we describe a system called `nlgPhylogeny`for generating natural language descriptions of collections of atoms derived from a set of ontologies. The system is powered by Grammatical Framework.

---

[1] http://coherentknowledge.com
[2] http://gem.med.yale.edu/ergo/default.htm
[3] https://ttsreader.com

■ **Figure 1** Overview of `nlgPhylogeny`.

2   Methodology

In this section, we describe the `nlgPhylogeny` system. Figure 1 shows the overall architecture of `nlgPhylogeny`. The main component of the system is the *GF generator* whose inputs are the ontology and the elements necessary for the NLG task (i.e., the set of linearizations, the set of pre-define conjunctives, the set of vocabularies, and the set of sentence models) and whose output is a GF program, i.e., a pair of GF abstract and concrete syntax. This GF program is used for generating the descriptions of workflows via the GF runtime API. The adapter provides the GF generator with the information from the ontology, such as the classes, instances, and relations.

2.1   Web Service Ontology (WSO)

Phylotastic uses web service composition to generate workflows for the extraction/construction of phylogenetic trees. It makes use of two ontologies: WSO and PO. WSO encodes information about registered web services and their abstract classes. In the following discussion, we refer to a simplified version of the ASP encoding of the ontologies used in [3], to facilitate readability. In WSO, a service has a name and is associated with a list of inputs and a list of outputs. For example, the service which is named in ontology *phylotastic_FindScientificNamesFromWeb_GET* is an instance of the class *names_extraction_web*. The data that *phylotastic_FindScientificNamesFromWeb_GET* uses and produces are encoded by the following 3 atoms:

```
instance_operation_has_input_has_data_format(
 phylotastic_FindScientificNamesFromWeb_GET,
 resource_WebURL,
 url_format).

 instance_operation_has_output_has_data_format(
 phylotastic_FindScientificNamesFromWeb_GET,
 resource_SetOfSciName,
 scientific_names_format).
instance_operation_has_output_has_data_format(
```

```
phylotastic_FindScientificNamesFromWeb_GET,
resource_SetOfNames,
list_of_strings).
```

In regard the above atoms, the first argument is the name of the service, the second argument is the service input or output, and the last argument is the data type of the second argument.

The web service ontology of the Phylotastic project is exported to an ASP program (from its original OWL encoding) and an inference engine is provided for reasoning about classes, inheritance, etc. `nlgPhylogeny` employs this engine in identifying information related to the set of atoms whose description is requested by a user (e.g., What are the inputs of a service? What is the data type of an input x of a service y?).

2.2   GF generator

Each Phylotastic workflow is an acyclic directed graph, where the nodes are web services, each consumes some resources (inputs) and produces some resources (outputs). An example of the specification of workflow is as follows.[4]

```
occur_concrete(phylotastic_ExtractSpeciesNames_From_Gene_Tree_GET,1)
occur_concrete(phylotastic_ResolvedScientificNames_OT_TNRS_GET,3)
occur_concrete(phylotastic_GenerateGeneTree_From_Genes,0)
occur_concrete(phylotastic_GeneTree_Scaling,2)
```

This set of atoms is a partial description of the result of a web service composition process, as described in [3]. Intuitively, this set of atoms represents a plan consisting of 4 steps. At each step, a concrete instance of the service class named by the first argument of the atom `occur_concrete/2` is executed.

To generate the description of a workflow, we employ the framework described in [4]. This framework consists of three major processing phases: (1) document planning (content determination), (2) microplanning, and (3) surface realization. The document planning phase is used to determine the structure of the text to be generated. Based on the structure determined in the document planning phase, the microplanner makes lexical/syntactic choices to generate the content of the sentences, and the realization phase generates the actual sentences. In our work, we combine the microplanning and surface realization phase into a single phase due to the nature of the grammar definition and the capability of GF in sentence generation.

In the document planning step, we create – for each occurrence atom – a sentence which specifies the input(s) and output(s) of the service mentioned in the first argument of the atom. Optionally, to describe the service in more details, one or two more sentences about datatype of the service's inputs or outputs can be included. As we have mentioned in the previous subsection, the information about the inputs, outputs, and data types of the inputs and outputs of a service can be obtained via the ASP reasoning engine of the Phylotastic system. In general, we identify the following document planning structure:

---

[4] For simplicity, we use examples which are linear sequences of services.

relation:	IDENTITY
argument_1:	instance or class in ontology
argument_2:	list of service inputs
argument_3:	list of service outputs

(optional)	
relation:	IDENTITY
argument_1:	name of input or output of service
argument_2:	data type of argument_1

(optional)	
relation:	IDENTITY
argument:	actual data involved in the workflow

The document planning phase determines three messages for the sentence generation phase.

In the microplanning step, we focus on developing a GF generator that can produce a portable grammar format (**pgf**) file [1]. This file is able to encode and generate 3 types of sentences as mentioned above. The GF generator (see Fig. 1) accepts two flows of input data: The first one is the flow of data from the ontology which is maintained by an adapter. The *adapter* is the glue code that connects the ontology to the GF generator. Its main function is to extract classes and properties from the ontology. The second one is the flow of data from predefined resources that cannot be automatically obtained from the ontology – instead they require manual effort from both ontology experts and linguistic developers:

- A list of *linearizations*; these are essentially the translations of names of ontology entities into linguistic terms. This translation is performed by experts who have knowledge of the ontology domain. An important reason for the existence of this component is that some classes or terms used in the ontology might not be directly understandable by the end user. This may be the result of very specialized strings used in the encoding of the ontology by the ontology engineer (e.g., abbreviations), or the use of URIs for the representation of certain concepts. For example, the class *phylotastic_OTResolvedNames* can be meaningfully linearized to *OpenTree Name Resolution service*.

- Some *model sentences* which are principally Grammatical Framework syntax trees with meta-information. The meta-information denotes which part of syntax tree can be replaced by some *vocabulary* or *linearization*. As indicated above, we decided that each occurrence atom of a workflow will be described by at most three sentences. For example, in regards to the first message in the document planning structure, the generated sentence will have the inputs and the outputs of a service; the second message indicates a sentence about the data type of its first argument (input or output); the third message is about the actual data used during the execution of the workflow. However, the messages do not specify how many inputs and outputs should be included in the generated sentence. The structure of the sentence representing a service that requires one input and one output is different from the structure of sentence representing that a service that does not require any inputs. These variations in sentences are recorded in the *model sentence* component. An example of a model sentence, for the case of a service that has a single input is as follows:

```
{
"s": "mkS (mkCl subject_in p_in_1);",
"placeholder": {
"subject_in": ["input of subject", "subject's input"]
}
}
```

- A list of *pre-defined vocabularies* which are domain-specific for the ontology. A *pre-defined vocabulary* is different from linearizations, in the sense that some lexicon may not be present in the ontology but might be needed in the sentence construction; the predefined vocabulary is also useful to bring variety in word choices when parts of a *model sentence* are replaced by the GF generator.
- A configuration of *pre-defined conjunctives* which depend on the document planning result. Basically, this configuration defines which sentences accept a conjunctive adverb in order to provide generated text transition and smoothness.

To encode sentences, the GF generator defines 3 categories: Input, Output and Format in the abstract syntax.

```
abstract Phylo = {
 flags startcat = Message;
 cat
 Message; Input; Output; Format;
 ...
}
```

and the corresponding English concrete syntax:

```
concrete PhyloEng of Phylo = open
SyntaxEng, ParadigmsEng, ConstructorsEng in {
 lincat
 Message = S; Input = NP; Output = NP; Format = NP;
 ...
}
```

SyntaxEng, ParadigmsEng, ConstructorsEng are GF Resources Grammar libraries which provide some constructors for sentence components like Verb, Noun Phrase, etc.. in English.

The GF generator obtains information about the services (e.g., how many inputs/outputs has the service? what are the data types of the inputs/outputs? etc.) by querying the ontology (via the adapter). Each service will be mapped to several functions in GF:

- A function which encodes the meaning of the sentence used for describing the service. The GF generator will prefix the name of the service with $f\_$ to create this kind of function name.
- A function which encodes the meaning of each input. The GF generator will prefix the name of the input with $i\_$.
- A function which encodes the meaning of each output. The GF generator will prefix the name of the output with $o\_$.

Based on the number of inputs and outputs of a service, the GF generator determines how many parameters will be included in the GF abstraction function corresponding to the service. Furthermore, for each input or output of a service, the GF generator includes an *Input* or *Output* in the GF abstract function. As an example, the result of the encoding of the atom

```
occur_concrete(phylotastic_FindScientificNamesFromWeb_GET,1)
```

in the GF abstract syntax is

```
f_phylotastic_FindScientificNamesFromWeb_GET: Input -> Output -> Message;
i_resource_WebURL : Input;
o_resource_SetOfNames : Output;
```

Next, the GF generator looks up in the *sentence models* a model syntax tree whose structure is suitable for the number of inputs and outputs of the service. If such syntax tree exists, the GF generator will replace parts of the syntax tree with the GF service input and output functions, to create a new GF syntax tree which can be appended in the GF concrete function. The functions in the abstract syntax corresponds to the following functions in the GF concrete syntax:

```
f_phylotastic_FindScientificNamesFromWeb_GET i_resource_WebURL
o_resource_SetOfNames =
mkS and_Conj
 (mkS (mkCl phylotastic_FindScienticNamesFromWeb_GET_in
 (mkV2 "require")
 i_resource_WebURL))
 (mkS (mkCl phylotastic_FindScienticNamesFromWeb_GET_out
 (mkV2 "return" )
 o_resource_SetOfSciName ));

i_resource_WebURL = mkNP(mkCN (mkN "webURL"));
i_resource_SetOfNames = mkNP(mkCN (mkN "asetof names"));
```

The above functions consist of several syntactic construction functions which are implemented in the GF Resources Grammar library:
- mkN which creates a noun from a string;
- mkCN which creates a common noun from a noun;
- mkNP which creates a noun phrase from a common noun;
- mkV2 which creates a verb from a string;
- mkCl which creates a clause. Clause can be constructed from sequence of a noun phrase, a verb and another noun phrase (NP V2 NP);
- mkS which creates a sentence. Sentence can be constructed from a clause (Cl) or from 2 other sentences and a conjunction word (and_Conj S S).

   From the abstract and concrete syntax built by GF generator, the atom `occur_concrete(phylotastic_FindScientificNamesFromWeb_GET,1)` is translated into the sentence
The input of phylotastic_FindScientificNamesFromWeb_GET is a web link and its outputs are a set of species names and a set of scientific names.
We use the same technique to encode the other types of sentences indicated by the document planning structure.

3   Discussion and future works

To the best of our knowledge, we found the work in [2] that reports on generating natural language text from class diagrams highly related to what we are doing. In [2], authors developed a system to generate specifications for UML class design. The difference between our work and [2] is the design of the system to employ automation on text generation for a given ontology under some assumptions.

From our case study we have identified two directions of future work that we find interesting. The first direction is to generate descriptions from annotations in ontology. We observe that the annotations play an vital role in ontology development in the sense of recording notes and explanations about concepts. Ontology developers usually use annotations to define the concepts and to describe relations between the concepts in the ontology, so that they employ reusablitity of the ontology. It is possible to apply natural language processing techniques to extract information from the annotation and tie that information with which concept or relation the annotation describes to re-generate text when needed. We believe that extracting and re-generating process is useful for query-answer system and information retrieval system since the process reduces the effort of system developers to create a module to explain the result of query.

The second direction is to make more use of the Grammatical Framework. We also want to make more of GF's capacity for several concrete languages to share the same abstract syntax. In other words, given an annotated ontology, we would like to generate explanations in multiple languages for a query.

---- **References** ----

1 Krasimir Angelov, Björn Bringert, and Aarne Ranta. PGF: A Portable Run-time Format for Type-theoretical Grammars. *Journal of Logic, Language and Information*, 19:201–228, 2010.

2 Hakan Burden and Rogardt Heldal. Natural Language Generation from Class Diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 2011), Wellington, New Zealand, ACM*, 2011.

3 Thanh H. Nguyen, Tran Cao Son, and Enrico Pontelli. Automatic Web Services Composition for Phylotastic. In *Practical Aspects of Declarative Languages - 20th International Symposium*, pages 186–202, 2018. `doi:10.1007/978-3-319-73305-0_13`.

4 Ehud Reiter and Robert Dale. *Building natural language generation systems*. Cambridge university press, 2000.

Model Revision of Logical Regulatory Networks Using Logic-Based Tools

Filipe Gouveia[1]

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, 1000-029, Lisboa, Portugal
filipe.gouveia@tecnico.ulisboa.pt
ⓘ https://orcid.org/0000-0003-1852-2782

Inês Lynce[2]

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, 1000-029, Lisboa, Portugal
ines.lynce@tecnico.ulisboa.pt
ⓘ https://orcid.org/0000-0003-4868-415X

Pedro T. Monteiro[3]

INESC-ID/Instituto Superior Técnico, Universidade de Lisboa
Rua Alves Redol 9, 1000-029, Lisboa, Portugal
pedro.tiago.monteiro@tecnico.ulisboa.pt
ⓘ https://orcid.org/0000-0002-7934-5495

—— Abstract ——

Recently, biological data has been increasingly produced calling for the existence of computational models able to organize and computationally reproduce existing observations. In particular, biological regulatory networks have been modeled relying on the Sign Consistency Model or the logical formalism. However, their construction still completely relies on a domain expert to choose the best functions for every network component. Due to the number of possible functions for k arguments, this is typically a process prone to error. Here, we propose to assist the modeler using logic-based tools to verify the model, identifying crucial network components responsible for model inconsistency. We intend to obtain a model building procedure capable of providing the modeler with repaired models satisfying a set of pre-defined criteria, therefore minimizing possible modeling errors.

1 Introduction

Modeling biological regulatory networks is particularly useful to test hypotheses and to identify predictions *in silico*. With this aim, different qualitative formalisms have been introduced to model, analyze and simulate regulatory networks and their behaviors. However,

---

Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018).
Editors: Alessandro Dal Palu', Paul Tarau, Neda Saeedloei, and Paul Fodor; Article No. 23; pp. 23:1–23:10
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the simulation and analysis of such behaviors is hindered by the combinatorial explosion of the qualitative state space. To tackle this problem, formal verification techniques have been introduced in Systems Biology. These techniques include model-checking techniques to automatically verify reachability properties [14], model reduction techniques to reduce the size of the generated dynamics [15], SAT-based approaches to identify attractors [4], among others [17].

Given a complete model of a regulatory network, newly acquired experimental data may render it inconsistent, forcing the model to be revised and updated. The process of review and update a model is called model revision, which is still mainly a manual task performed by a modeler, typically an expert in the domain, and therefore prone to error.

Approaches to model revision relying on the Sign Consistency Model (SCM) have been implemented using logic-based tools such as Answer Set Programming (ASP)[8] and Boolean Satisfiability (SAT)[10]. However, the SCM lacks in expressiveness for regulatory functions, as it is based in sign algebra. This work aims to extend current approaches for model revision to the Logical formalism, and to provide a semi-automatic tool to assist the modeler throughout the model definition process [21].

An overview of some of the key concepts of regulatory networks is given in Section 2. In Section 3 it is mentioned some of the work done in System Biology, regarding regulatory networks. Section 4 describes the logic-based approach for Model Revision. Section 5 concludes the document with an overview of the directions of the future work.

2   Regulatory Networks

A biological regulatory network is a set of proteins and genes, that interact with each other or with other substances in the cell. Qualitative models have proven to be well adapted for the modeling of systems where quantitative information is generally incomplete or noisy. Typically, network components only affect other components above some concentration level. In this way, it is possible to consider discrete variables to model regulatory networks, corresponding to different levels of concentration, e.g. active/inactive.

2.1   Logical Model

Logical models where used to represent regulatory networks by Kauffman in 1969 [12], and Thomas in 1973 [20].

In the Logical Model the components of the network are represented by Boolean variables. A Boolean variable can either be *True (1, on, active)* or *False (0, off, inactive)*. If a component in a regulatory network is represented by a Boolean variable, then it has value *True* if it is present (or activated), and it has value *False* if it is absent (or inhibited).

Moreover, the interactions between components are described as Boolean functions [20]. This will allow to determine the state of a component based on the presence or absence of other components.

A Logical Model can be represented with a logical circuit since nodes have a Boolean value and regulatory functions are Boolean functions, as shown in Figure 1.

Figure 1 illustrates an example of a logical model with the correspondent regulatory functions. With this representation we can verify that, for example, component c is regulated by components d and a, and its regulatory function is a logical AND from these two inputs.

The (Boolean) Logical Model can be generalized [21]. It is possible to consider more than two values for each variable. For example, considering Figure 2, we can have a variable a that affects b above a concentration level threshold t_1, but only affects c above a concentration level threshold $t_2 > t_1$. In this case variable a can have three possible values:
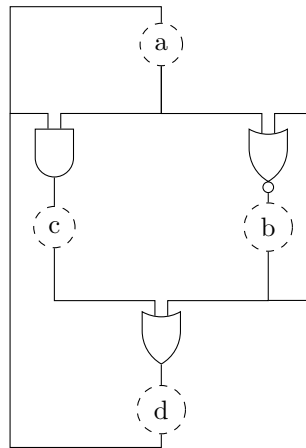
Figure 1 Example of a Logical Model represented as a logical circuit.
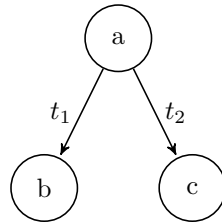
Figure 2 Example of a Generalization of the Logical Model.

- 0: concentration level below t_1 (not affecting any other variable);
- 1: concentration level between t_1 and t_2 (only affecting variable b);
- 2: concentration level above t_2 (affecting variables b and c);

Formally, we can define a Logical Model as a tuple (G, K) where:
- $G = \{g_1, g_2, ..., g_n\}$ is the set of components of the network. Each g_i is associated with an integer value in $\{0, ..., max_i\}$, representing the concentration level of the component. The state of the network is thus defined as a vector $s \in S = \prod_{g_i \in G} \{0, ..., max_i\}$.
- $K = \{K_1, K_2, ..., K_n\}$ is the set of regulatory functions where K_i is the regulatory function of g_i and $K_i : S \to \{0, ..., max_i\}$.

If all $max_i = 1$, then we have a Boolean Logical Model, since each $g_i \in \{0, 1\}$.

2.2 Probabilistic Boolean Networks

In a logical model, each component regulated by k other components can have 2^{2^k} possible regulatory Boolean functions. Additionally, in some cases experimental data is insufficient or there is incomplete knowledge to choose a single regulatory function, where several candidates are possible. In other words, possibly several regulatory functions could explain the experimental data. With this in mind, the logical model was extended in order to account for the uncertainty of the regulatory functions [18].

In a Probabilistic Boolean Network (PBN), each component has several regulatory functions, each with a given probability associated. These probabilities are determined based on the data available, such that it is compatible with prior knowledge of the network. Then, at each time step, and for each component, a regulatory function is selected according to the correspondent probabilities, in order to determine its target value.
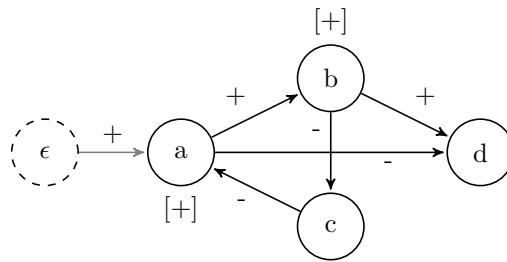
■ **Figure 3** Example of a Sign Consistency Model. Observed components a and b are labeled with the correspondent observation. Component ϵ represents an external stimulus.

2.3   Sign Consistency Model

Siegel *et al.* proposed a Sign Consistency Model (SCM) [19]. In this approach, it is only considered the difference in the expression levels between two situations: a value increase, or decrease.

The SCM is usually represented by a graph where each node represents a biological component, with a value + (increase of concentration) or − (decrease of concentration). The edges in the graph represent interactions between components and can be labeled "+" or "−". An edge with label "+" ("−") from a to b means that an increase of the concentration of a increases (decreases) the concentration of b.

Also, a component can be considered an *input*, having a stimulation from the exterior world (outside the regulatory network). If a node is an *input* then its regulatory function can be ignored, as there is an exterior stimulation increasing its concentration. In some representations, an extra generic node ϵ is added to represent the exterior world. For each *input* node, an edge is added from ϵ to that node.

The regulatory functions are then based on the sign algebra, where the value of each component is the sum of the products between the value of each regulator and the corresponding edge.

Figure 3 illustrates an example of a Sign Consistency Model of a network, where node a is an input, and therefore have an input edge from the generic node ϵ that represents the exterior world. Nodes a and b are observed nodes, where an increase of concentration was observed. In this example, node c is expected to have a negative (−) sign because it only has one regulator b, which has a negative interaction with c ($c = b \times (b \to c) = (+) \times (-) = -$). However in this example, node d receives a positive and a negative interaction. in this case we say that we have a competition and d can assume either value.

3   Related Work

The analysis and verification of biological regulatory networks provide opportunities for the application of several methodologies. From network identification and parametrization, model verification, attractors determination or to model revision. In this section, some of the main methods from the last decade are described, as well as the corresponding problem and technology.

3.1   Network and Model Inference

Building computational models to correctly represent regulatory networks is of great importance. In order to build such model, one first needs to infer the network topology from a given set of experimental data. Some of the difficulties of this task relies on the few samples

of observational data and in the incompleteness and inaccuracy of the experimental data. Then, one also needs to infer, for each component, the associated regulatory functions (model inference).

In regulatory networks inference, several statistical learning techniques are commonly used [2, 7]. Also, logic-based tools have been successfully used to learn biological models. *Caspo* [11] is a tool to identify the complete family of feasible models from a training Boolean logical model from prior knowledge and experimental data.

3.2 Reachability Verification

Given a model and a set of experimental data, it is interesting to verify if the model can explain the results obtained in the experiment. In particular, one may verify if the model is capable of generating behaviors from a set of initial states to a set of target states. These behaviors are typically represented by a State Transition Graph (STG), where nodes represent states of the network, and edges represent possible transitions between states. The generation of this STG can be made synchronously or asynchronously. In the synchronous approach, in a given state of the STG, all components can update their value simultaneously, i.e., each state as a single successor. In the asynchronous approach, in a given state of the STG, only one component can update their value to a successor state, i.e., each state has as many successors as components changing their values.

Model checking consists in the verification if a model satisfies a given (set of) property [3], and has been successfully used for the verification of regulatory networks. Here, biological observations are encoded in temporal logic formulas, and a model checker is used to verify the existence of particular behaviors [14].

Also of interest, is to know how can a system be influenced in order to avoid reaching unsafe or undesired states. Recently, the work in [6] introduces the notion of *bifurcation*, transitions after which a given goal is no longer reachable. This work presents a method using Answer Set Programming, to identify bifurcations given a model represented as a discrete finite-state of interacting components. However, since this method relies on under/over approximations, is not complete, i.e., does not guarantee the identification of all the bifurcations.

3.3 Attractors Identification

A key property of the dynamics of a regulatory network are *attractors*, which typically denote subsets of states of biological interest. There are two types of attractors: point attractors and cycle attractors. A point attractor, or a stable state, is a state from which there is no transition to any other state in the STG. A cycle attractor is a set of states, whose sequence repeats over time, from which no transition can leave, i.e., a terminal strongly connected component in the STG.

An efficient approach to determine point attractors in (multivalued) logical models uses Multi-values Decision Diagrams (MDDs) [16]. Also, some approaches consider the identification of point and cycle attractors in synchronous dynamics. The work in [5] uses Answer Set Programming (ASP) and allows the determination of all attractors considering a Markovian program in order to overcome the challenge of determining the number of time-steps needed to achieve an attractor. The work in [4] uses a SAT based bounded model checker to determine all the attractors of the network by incrementally determining the attractors of a given length.

3.4   Reduction

It is often the case where the generation of the network dynamics is intractable for large and complex regulatory networks, due to the state space combinatorial explosion. Reduction techniques can then be applied in order to reduce the model, and therefore the generated state space. It has been shown that reduction methods can be successfully applied preserving some dynamical properties of the network, such as attractors [15].

4   Model Revision Approach

During the iterative model construction procedure, as new data is acquired, the current model may not be able to explain the new data, and therefore need to be revised. Revision processes capable of suggesting addition/removal of networks interactions, and changes to variable values in order to make a model consistent with the available data have been proposed [13]. An approach was proposed considering the SCM and developed using the Answer Set Programming (ASP) paradigm [8]. Also, an approach was proposed using MaxSAT, a SAT extension used to solve optimization problems [10]. However, the SCM formalism relies on a simple rule regarding regulatory functions.

The logical formalism [20] has been widely used to model biological networks, and have been successfully implemented using ASP [9] and SAT [1], allowing to model the regulatory functions with increased expressiveness w.r.t. the SCM. Model repair usually operates under a minimal assumption as there can be several ways to make a model consistent. Such optimization criteria can be regarding the number of atomic repair operations [8, 10] or considering some properties found in the literature [13]. Nevertheless, existing approaches typically rely on repair operations that potentially change the topology of the network, invalidating previous domain knowledge.

As mentioned in Section 2, there can be several regulatory functions that can explain the experimental data. Avoiding changing the topology of the network and change regulatory functions leads to a minimal impact on the truth table of the variables of the model, and therefore a smaller impact on the associated dynamics.

Our idea is to develop a model revision procedure capable of building a consistent model iteratively as new data is acquired, relying on the logical formalism. Moreover, it is desired to avoid changing the topology of the network, and try to explain possible causes of inconsistencies with regulatory functions.

On a first phase of the work, one should be able to verify the consistency of a given model with a set of experimental data, i.e., if the model can explain the experimental data obtained. Model checking techniques should be used for this purpose. It is intended to implement this using different logic based tools, such as ASP, SAT and MaxSAT, in order to make a comparison with respect to the easiness of representation and computational efficiency.

On a second phase, if a model is not consistent with the experimental data, the causes of such inconsistencies must be identified. This is closely related to the identification of Minimal Unsatisfiable Subsets (MUSes) in SAT formulas, and therefore SAT-based tools should be used in the process. As there can be multiple concurrent reasons to explain the existence of inconsistencies, a biological meaningful measure should be provided in order to rank the possible explanations to be presented to a modeler.

On a final phase, considering the most plausible cause for inconsistency, a procedure for model revision should be defined. For this, SAT-based tools for the identification of Minimal Correction Subsets (MCSes) should be considered. This model revision process should be iterative, considering that multiple reasons for inconsistency may exist. We will first try

■ **Listing 1** Example of input.

```
edge(c1,c2,0).                    obs_vlabel(c1,1).
edge(c1,c3,1).                    obs_vlabel(c2,0).
edge(c2,c1,1).                    obs_vlabel(c3,0).
edge(c2,c3,1).                    obs_vlabel(c4,0).
edge(c4,c2,1).
edge(c4,c3,0).


functionOr(c1,1).                 functionOr(c3,1..2).
functionAnd(c1,1,c2).             functionAnd(c3,1,c1).
                                  functionAnd(c3,2,c2).
functionOr(c2,1).                 functionAnd(c3,2,c4).
functionAnd(c2,1,c1).
functionAnd(c2,1,c4).
```

to explain the causes of inconsistencies with regulatory functions. However, changing the regulatory functions may not be sufficient, and therefore one may need to consider changing the topology of the network. To achieve this, an iterative approach will be considered where different causes of inconsistency are taken into account.

In the revision process, not only the model consistency must be taken into account, but also other known properties about the network must hold, such as the existence of known attractors and its reachability. As the number of possible states for a network increases exponentially with the number of components, guaranteeing the existence of the known attractors, for example, can be a difficult task. For this, model reduction techniques may be necessary.

We start by considering only monotone non-degenerate functions. In a monotone function, each regulator has only one role, i.e., it is either strictly positive or negative. In a non-degenerate function, all regulators are functional, i.e., all regulators have an influence in the regulatory function.

Currently, we have an Answer Set Programming approach implemented for the logical formalism, and we are able to verify the consistency of a model given some experimental data at steady state, i.e., without considering any dynamics. Moreover, we are able, in case of inconsistency, to identify the regulatory functions that can explain such inconsistencies. We are working on the process of repairing such functions in order to validate if the proposed model solutions become consistent.

We represent the logical model as a directed graph and the regulatory functions in disjunctive normal form (DNF). As we only consider monotone functions and, therefore, each regulator only has one role (positive interaction or negative interaction), this role is defined by the edge. A positive (negative) edge represents a positive (negative) interaction. An example is presented in Listing 1 with the representation of the model and the observations.

The predicate `edge(A,B,S)` represents an edge from `A` to `B` with sign `S`. Predicate `functionOr(A,C)` indicates the number of clauses (`C`) in the regulatory function of `A` in the DNF. Predicate `functionAnd(A,C,B)` indicates that the clause `C` of the regulatory function of `A` contains variable `B`. The observations are represented by the predicate `obs_vlabel(A,S)`, which means that value `S` was observed in node `A`.

■ **Listing 2** Consistency check in Answer Set Programming.

```
sign(0;1).                          complement(T,S):-sign(S),sign(T),T!=S.

vertex(V):-edge(V,_,_).           vertex(V):-edge(_,V,_).

% generate
1{vlabel(V,S):sign(S)}1:-vertex(V).
{r_gen(V)} :- vertex(V).        {r_part(V)} :- vertex(V).

:-vlabel(V,S), obs_vlabel(V,T),complement(S,T).

% functions
% one positive or negative contribution in a clause
onePositive(V,Id):-functionAnd(V,Id,V2),edge(V2,V,S),vlabel(V2,S).
oneNegative(V,Id):-functionAnd(V,Id,V2),edge(V2,V,S),vlabel(V2,T),
        complement(S,T).

% none negative contribution in a clause
noneNegative(V,Id):-onePositive(V,Id),not oneNegative(V,Id).

vlabel(V,1):-1{noneNegative(V,Id):functionOr(V,Id)},vertex(V),
        not r_part(V).
vlabel(V,0):-{noneNegative(V,Id):functionOr(V,Id)}0,vertex(V),
        not r_gen(V).

repair(f,V) :- r_gen(V).         repair(f,V) :- r_part(V).
#minimize {1,V : repair(_,V)}.
```

The main idea behind the encoding presented in Listing 2 is that each node of the network (`vertex`) must have exactly one label that represents the expected value (`vlabel`), and it is not possible to have a label different from the observation. Each label is determined based on the contributions of each regulator in the associated regulatory function. To allow determining possible causes of inconsistencies, we defined the predicates `r_gen` and `r_part` indicating that a regulatory function should be generalized or particularized, respectively, justifying the inconsistency of the model. In order to achieve this, we allow a label of a vertex to be different than expected given the regulatory function, if that function is a possible cause of inconsistency.

5 Conclusions and Future Work

Qualitative formalisms have been used whenever information is scarce. In particular, the logical formalism has proved successful to model complex biological networks. Nevertheless, the construction of such models is still mainly a manual task, and therefore prone to errors and to interpretations of a specific modeler. Here, we focus on the problem of model revision, i.e., to assist the modeler in the process of revising the model associated functions in order to render the model consistent with the existing and new data.

Here, we propose to consider the logical formalism limiting to the set of monotone non-degenerate functions. Also, we start by verifying the consistency of models at steady state, i.e., without considering any dynamics. We consider an Answer Set Programming approach to identify which nodes are the causes for model inconsistency.

We intend to follow the work plan described in the previous section, and be able to present a procedure and corresponding tool capable of building a consistent model iteratively as new data is acquired.

―――― **References** ――――

1 Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

2 Atul J Butte and Isaac S Kohane. Mutual information relevance networks: functional genomic clustering using pairwise entropy measurements. In *Biocomputing 2000*, pages 418–429. World Scientific, 1999.

3 Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

4 Elena Dubrova and Maxim Teslenko. A SAT-based algorithm for finding attractors in synchronous boolean networks. *IEEE/ACM transactions on computational biology and bioinformatics*, 8(5):1393–1399, 2011.

5 Timur Fayruzov, Jeroen Janssen, Dirk Vermeir, Chris Cornelis, and Martine De Cock. Modelling gene and protein regulatory networks with answer set programming. *International journal of data mining and bioinformatics*, 5(2):209–229, 2011.

6 Louis Fippo Fitime, Olivier Roux, Carito Guziolowski, and Loïc Paulevé. Identification of bifurcation transitions in biological regulatory networks using Answer-Set Programming. *Algorithms for Molecular Biology*, 12(1):19, 2017.

7 Nir Friedman. Inferring cellular networks using probabilistic graphical models. *Science*, 303(5659):799–805, 2004.

8 Martin Gebser, Carito Guziolowski, Mihail Ivanchev, Torsten Schaub, Anne Siegel, Sven Thiele, and Philippe Veber. Repair and Prediction (under Inconsistency) in Large Biological Networks with Answer Set Programming. In *KR*, 2010.

9 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Answer set solving in practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3):1–238, 2012.

10 João Guerra and Inês Lynce. Reasoning over biological networks using maximum satisfiability. In *Principles and Practice of Constraint Programming*, pages 941–956. Springer, 2012.

11 Carito Guziolowski, Santiago Videla, Federica Eduati, Sven Thiele, Thomas Cokelaer, Anne Siegel, and Julio Saez-Rodriguez. Exhaustively characterizing feasible logic models of a signaling network using answer set programming. *Bioinformatics*, page btt393, 2013.

12 Stuart Kauffman. Homeostasis and differentiation in random genetic control networks. *Nature*, 224(5215):177, 1969.

13 Elie Merhej, Steven Schockaert, and Martine De Cock. Repairing inconsistent answer set programs using rules of thumb: A gene regulatory networks case study. *International Journal of Approximate Reasoning*, 83:243–264, 2017.

14 Pedro T Monteiro, Wassim Abou-Jaoudé, Denis Thieffry, and Claudine Chaouiya. Model Checking Logical Regulatory Networks. *IFAC Proceedings Volumes*, 47(2):170–175, 2014.

15 Aurélien Naldi, Elisabeth Remy, Denis Thieffry, and Claudine Chaouiya. Dynamically consistent reduction of logical regulatory graphs. *Theoretical Computer Science*, 412(21):2207–2218, 2011.

16 Aurélien Naldi, Denis Thieffry, and Claudine Chaouiya. Decision diagrams for the representation and analysis of logical models of genetic networks. In *CMSB*, volume 7, pages 233–247. Springer, 2007.

17 Loïc Paulevé. Reduction of Qualitative Models of Biological Networks for Transient Dynamics Analysis. *IEEE/ACM transactions on computational biology and bioinformatics*, 2017.

18   Ilya Shmulevich, Edward R Dougherty, Seungchan Kim, and Wei Zhang. Probabilistic Boolean networks: a rule-based uncertainty model for gene regulatory networks. *Bioinformatics*, 18(2):261–274, 2002.

19   Anne Siegel, Ovidiu Radulescu, Michel Le Borgne, Philippe Veber, Julien Ouy, and Sandrine Lagarrigue. Qualitative analysis of the relation between DNA microarray data and behavioral models of regulation networks. *Biosystems*, 84(2):153–174, 2006.

20   René Thomas. Boolean formalization of genetic control circuits. *Journal of theoretical biology*, 42(3):563–585, 1973.

21   René Thomas. Regulatory networks seen as asynchronous automata: a logical description. *Journal of theoretical biology*, 153(1):1–23, 1991.

Scalable Robotic Intra-Logistics with Answer Set Programming

Philipp Obermeier

Institute of Computer Science, University of Potsdam
Germany
phil@cs.uni-potsdam.de

─── **Abstract** ───

Over time, Answer Set Programming (ASP) has gained traction as a versatile logic programming semantics with performant processing systems, used by a growing number of significant applications in academia and industry. However, this development is threatened by a lack of commonly accepted design patterns and techniques for ASP to address dynamic application on a real-world scale. To this end, we identified *robotic intra-logistics* as representative scenario, a major domain of interest in the context of the fourth industrial revolution. For this setting, we aim to provide a scalable and efficient ASP-based solutions by (1) stipulating a standardized test and benchmark framework; (2) leveraging existing ASP techniques through new design patterns; and (3) extending ASP with new functionalities. In this paper we will expand on the subject matter as well as detail our current progress and future plans.

1 Introduction

Answer Set Programming (ASP; [4]) has come a long way, starting as a semantics for logic programming, over having increasingly performant systems, to a growing number of significant applications in academia and industry. In contrast to other solver paragdims, ASP offers an unprecedented degree of versatility and brevity, which is best put in perspective by solving multi-faceted problems. However, this development is threatened by a lack of commonly accepted design patterns and techniques for ASP to address dynamic application on a real-world scale. In addition, many industrial applications require the integration of multiple types of knowledge and forms of reasoning, a feature commonly neglected by existing approaches. As a first step to overcome these problems, we have identified *robotic intra-logistics* as representative scenario for our investigation. This domain is a major subject of interest in the context of the fourth industrial revolution, as witnessed by Amazon's Kiva, GreyOrange's Butler, and Swisslog's CarryPick systems.[1] All of them aim at automatizing warehouse operations (illustrated by Figure 1) by using robot vehicles that drive underneath mobile shelves and deliver them to picking stations. From there, workers pick and place the requested items in shipping boxes. For this setting, we aim to provide scalable and efficient

---

[1] www.amazonrobotics.com, www.greyorange.com/products/butler, www.swisslog.com/carrypick
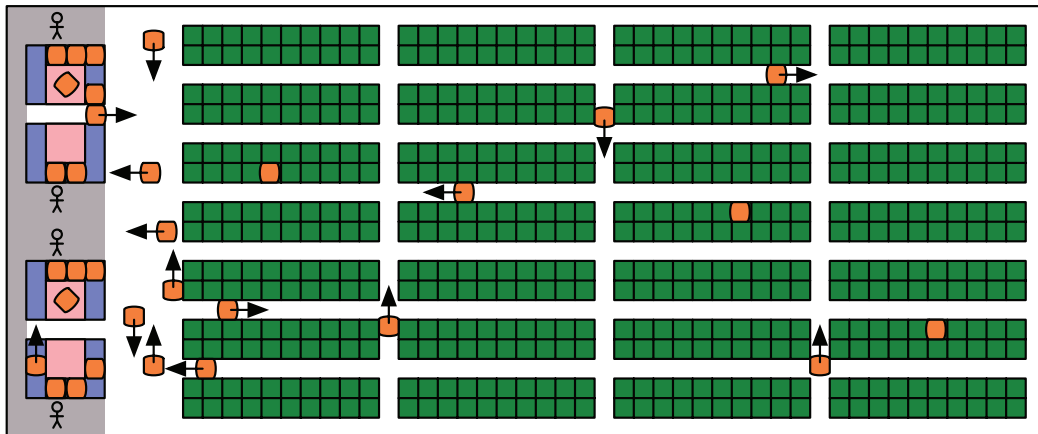
Figure 1 Layout of an Autonomous Warehouse System [Wurman *et al.*, 2008].

ASP-based solutions by (1) stipulating a standardized test and benchmark framework; (2) leveraging existing ASP techniques through new design patterns; and (3) extending ASP with new functionalities.

2   Related Work

What distinguishes robotic intra-logistics from other combinatorial problems is its multidimensional nature that necessitates the integration of a great many of aspects, most notably path finding and order fulfillment.

At the core of many path finding problems lies the search for a route for an agent from an initial to a final location. The *multi-agent path finding* (MAPF) problem asks for a collision-free route for each agent such that the total makespan is minimal. MAPF is related to many real-world applications but already computationally intractable [8]. While in MAPF each agent is assigned a unique destination, its *anonymous* variant requires no assignment of agents to destinations [9]. The problem domains of *asprilo* are obviously related to multi-agent path finding. More specifically, the *asprilo* domain **M** corresponds to anonymous MAPF. Each order is uniquely associated with a destination shelf and there is no pre-assignment of a robot to an order. Robots can freely reach any destination shelf. Clearly, **M** is easily extended to cover non-anonymous MAPF by relating robots and orders.

Task assignment and path finding (TAPF; [5]) is a generalization of MAPF. TAPF groups agents into teams. Although teams are (non-anonymously) pre-assigned to groups of destinations, any robot in the team can be (anonymously) selected for a destination in the assigned group. G-TAPF [7] is a generalization of TAPF aiming at more realistic settings by allowing the number of tasks to be greater than the number of agents and considering deadlines, orderings, and checkpoints. That is, deadlines are associated with order lines, orders are completed in a pre-defined ordering and all lines in a single order need to be fulfilled before any line of another order is completed, and while fulfilling an order, a robot is required to go through a sequence of locations, called checkpoints. Regarding previous uses of ASP, [1] address several aspects of multi-agent path finding problems.

[6] address an online version of path finding, where not all destination tasks[2] are given initially but may arrive over time.

---

[2]   Actually, this work also uses pick-up and delivery tasks to simulate a warehouse system.
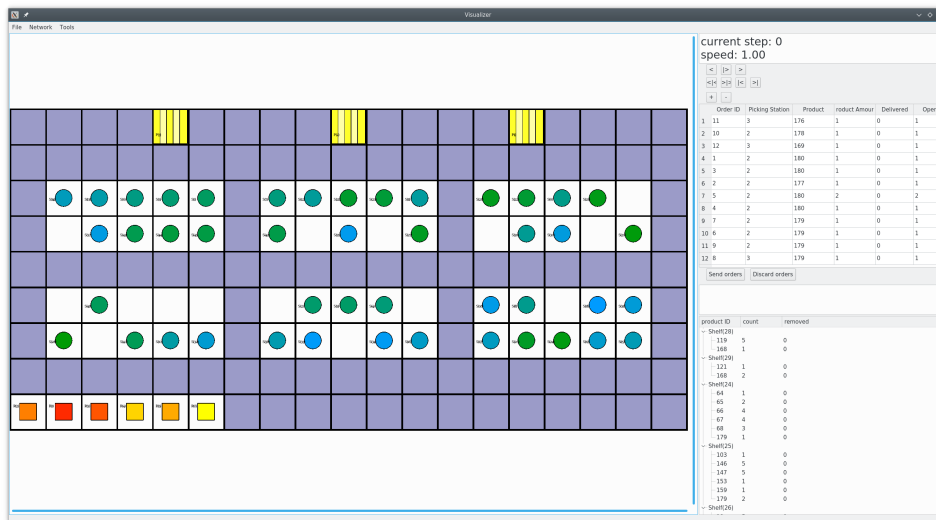
Figure 2 Exemplary *asprilo* screenshot: The main window gives a warehouse layout; no plan is loaded. Picking stations are represented by striped yellow squares, shelves by solid circles, and robots by solid squares. Highways are brought out in purple. The side windows provide controls for plan animation, and give details about the current orders and the warehouse inventory.

3 Current Research Progress

So far, we introduced a scalable approach for a generalized variant of TAPF, formally laid out the problem domain, and conducted a real-life case study for car assembly:

*asprilo***: Robotic Intra-Logistics Benchmark Suite [2].** We introduce the *asprilo*[3] framework to facilitate experimental studies of approaches addressing complex dynamic applications. For this purpose, we have chosen the domain of robotic intra-logistics. This domain is not only highly relevant in the context of today's fourth industrial revolution but it moreover combines a multitude of challenging issues within a single uniform framework. This includes multi-agent planning, reasoning about action, change, resources, strategies, etc. In return, *asprilo* allows users to study alternative solutions as regards effectiveness and scalability. Although *asprilo* relies on Answer Set Programming and Python, it is readily usable by any system complying with its fact-oriented interface format. This makes it attractive for benchmarking and teaching well beyond logic programming. More precisely, *asprilo* consists of a versatile benchmark generator, solution checker and visualizer (see Figure 2) as well as a bunch of reference encodings featuring various ASP techniques. Importantly, the visualizer's animation capabilities are indispensable for complex scenarios like intra-logistics in order to inspect valid as well as invalid solution candidates. Also, it allows for graphically editing benchmark layouts that can be used as a basis for generating benchmark suites. The *asprilo* framework is freely available at `https://potassco.org/asprilo`.

Generalized Target Assignment and Path Finding [7]. Both MAPF and TAPF models suffer from their limiting assumption that the number of agents and targets are equal. We propose the *Generalized TAPF* (*G-TAPF*) formulation that allows for (1) unequal

---

[3] *asprilo* stands for *Answer Set Programming for robotic intra-logistics*.
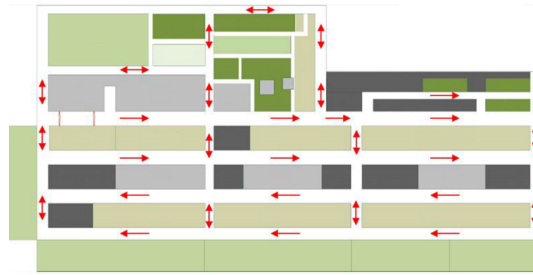
Figure 3 Real-world factory layout with transport corridors and directions indicated by arrows.

number of agents and tasks; (2) tasks to have deadlines by which they must be completed; (3) ordering of groups of tasks to be completed; and (4) tasks that are composed of a sequence of checkpoints that must be visited in a specific order. As different G-TAPF variants may be applicable in different domains, we model them using ASP, which allows one to easily customize the desired variant by choosing appropriate combinations of rules to enforce. Our experimental results show that the popular CBM (conflict-based min-flow) algorithm is better in simple TAPF problems with few conflicts, but worse in difficult problems with more conflicts. We also show that ASP technologies can easily exploit domain-specific information to improve its scalability and efficiency. The contributions in this paper thus make a notable jump towards deploying MAPF and TAPF algorithms in practical applications.

Routing Driverless Transport Vehicles in Car Assembly [3]. Automated storage and re-trieval systems are principal components of modern production and warehouse facilities. In particular, automated guided vehicles nowadays substitute human-operated pallet trucks in transporting production materials between storage locations and assembly stations. While low-level control systems take care of navigating such driverless vehicles along programmed routes and avoid collisions even under unforeseen circumstances, in the common case of multiple vehicles sharing the same operation area, the problem remains how to set up routes such that a collection of transport tasks is accomplished most effectively. We address this prevalent problem in the context of car assembly (see Figure 3) at Mercedes-Benz Ludwigsfelde GmbH, a large-scale producer of commercial vehicles, where routes for automated guided vehicles used in the production process have traditionally been hand-coded by human engineers. Such ad-hoc methods may suffice as long as a running production process remains in place, while any change in the factory layout or production targets necessitates tedious manual reconfiguration, not to mention the missing portability between different production plants. Unlike this, we propose a declarative approach based on Answer Set Programming to optimize the routes taken by automated guided vehicles for accomplishing transport tasks. The advantages include a transparent and executable problem formalization, provable optimality of routes relative to objective criteria, as well as elaboration tolerance towards particular factory layouts and production targets. Moreover, we demonstrate that our approach is efficient enough to deal with the transport tasks evolving in realistic production processes at the car factory of Mercedes-Benz Ludwigsfelde GmbH.

4 Open Issues and Expected Achievements

To sum up, we expect the following major achievements through our research:
1. A standardized framework for experimental studies of dynamic systems, specifically in the intra-logistics domains
2. Novel ASP design patterns and extensions for solving various problems in dynamic systems on an industrial scale

—— **References** ——

1 E. Erdem, D. Kisa, U. Öztok, and P. Schüller. A General Formal Framework for Pathfinding Problems with Multiple Agents. In M. desJardins and M. Littman, editors, *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, pages 290–296. AAAI Press, 2013. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6293`.

2 M. Gebser, P. Obermeier, T. Otto, T. Schaub, O. Sabuncu, V. Nguyen, and T. Son. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming*, 2018. To appear. URL: `http://arxiv.org/abs/1804.10247`.

3 M. Gebser, P. Obermeier, M. Rasch-Heitmann, and T. Schaub. Routing Driverless Transport Vehicles in Car Assembly with Answer Set Programming. *Theory and Practice of Logic Programming*, 2018. To appear. URL: `http://arxiv.org/abs/1804.10437`.

4 V. Lifschitz. Answer set planning. In D. de Schreye, editor, *Proceedings of the International Conference on Logic Programming (ICLP'99)*, pages 23–37. MIT Press, 1999.

5 H. Ma and S. Koenig. Optimal Target Assignment and Path Finding for Teams of Agents. In C. Jonker, S. Marsella, J. Thangarajah, and K. Tuyls, editors, *Proceedings of the Fifteenth International Conference on Autonomous Agents and Multiagent Systems (AAMAS'16)*, pages 1144–1152. ACM Press, 2016.

6 H. Ma, J. Li, T. Kumar, and S. Koenig. Lifelong Multi-Agent Path Finding for Online Pickup and Delivery Tasks. In *Proceedings of the Sixteenth Conference on Autonomous Agents and MultiAgent Systems (AAMAS'17)*, pages 837–845. ACM Press, 2017.

7 V. Nguyen, P. Obermeier, T. Son, T. Schaub, and W. Yeoh. Generalized Target Assignment and Path Finding Using Answer Set Programming. In C. Sierra, editor, *Proceedings of the Twenty-sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, pages 1216–1223. IJCAI/AAAI Press, 2017.

8 P. Surynek. An Optimization Variant of Multi-Robot Path Planning Is Intractable. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI'10)*, pages 1261–1263. AAAI Press, 2010. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1768`.

9 J. Yu and S. LaValle. Multi-agent Path Planning and Network Flow. In E. Frazzoli, T. Lozano-Pérez, N. Roy, and D. Rus, editors, *Proceedings of the Tenth Workshop on the Algorithmic Foundations of Robotics (WAFR'12)*, volume 86 of *Springer Tracts in Advanced Robotics*, pages 157–173. Springer-Verlag, 2012. `doi:10.1007/978-3-642-36279-8_10`.