Proceedings of the
25th Automated Reasoning Workshop
Bridging the Gap between Theory and Practice

ARW 2018

12th – 13th April 2018
University of Cambridge
Cambridge, United Kingdom

Editors:
Mateja Jamnik,
Angeliki Koutsoukou-Argyraki,
Edward Ayers,
Chaitanya Mangla

Department of Computer Science and Technology
University of Cambridge

UNIVERSITY OF
CAMBRIDGE

Preface

This volume contains the proceedings of ARW 2018, the twenty-fifth Workshop on Automated Reasoning, held on 12th–13th April 2018, in Cambridge, UK. As in previous events in the series, this workshop provides an informal forum for the automated reasoning community to discuss recent work, new ideas and current trends. It aims to bring together theoreticians and practitioners from all areas of automated reasoning in order to foster links between them and to encourage cross-fertilisation across various disciplines.

To mark this milestone 25th ARW, we organised a discussion panel on "Natural language processing and information retrieval for automated reasoning", invited two keynote speakers, presented 23 extended abstracts contributed by participants of the workshop, and screened a documentary about Bletchley Park.

Our invited speaker Ekaterina Komendantskaya (Heriot-Watt University), presented her work on "Machine learning for mining, understanding and automating computer proofs". Our invited speaker Larry Paulson (University of Cambridge), celebrated the contributions of one of the most distinguished researchers in Automated Reasoning and spoke about "A Career in Research: Mike Gordon and Hardware Verification". The abstracts covered a wide range of topics including various theorem provers for different domains and logics, their design and theoretical properties, novel extensions to calculi and logics, ontologies and knowledge bases, and security aspects of automated reasoning systems.

I would like to thank the members of the ARW Organising Committee for their advice. I would also like to express my sincere gratitude to all the colleagues who have helped with the local organisation, namely, Angeliki Koutsoukou-Argyraki, Larry Paulson, Edward Ayers, Wenda Li, Chaitanya Mangla and Zohreh Shams.

Cambridge
April 2018

Mateja Jamnik

Local Organisers

Mateja Jamnik (Chair)	(University of Cambridge)
Edward Ayers	(University of Cambridge)
Angeliki Koutsoukou-Argyraki	(University of Cambridge)
Wenda Li	(University of Cambridge)
Chaitanya Mangla	(University of Cambridge)
Lawrence Paulson	(University of Cambridge)
Zohreh Shams	(University of Cambridge)

ARW Organising Committee

Alexander Bolotov (Chair)	(University of Westminster)
Jacques Fleuriot (Secretary/Treasurer)	(University of Edinburgh)
Simon Colton	(Goldsmiths College, University of London)
Louise Dennis	(University of Liverpool)
Ullrich Hustadt	(University of Liverpool)
Mateja Jamnik	(University of Cambridge)
Florian Kammueller	(Middlesex University)
Ekaterina Komendantskaya	(Heriot-Watt University)
Alice Miller	(University of Glasgow)
Oliver Ray	(University of Bristol)
Renate Schmidt	(University of Manchester)

Workshop Website

`www.cl.cam.ac.uk/events/arw2018`

Programme

Thursday 12 April

9:00 Registration

10:00 Invited Talk (Ekaterina Komendantskaya)

11:00 Coffee Break

11:30 Short Talks (see pages 4-13 of online booklet)

Meta-unification Juan Casanova and Alan Bundy

Separated Normal Form Transformation Revisited Cláudia Nalon, Ullrich Hustadt and Clare Dixon

Models of Coinductive First-order Horn Clauses Yue Li

Pattern-Based Reasoning to Investigate the Correctness of Knowledge Graphs Kemas Wiharja, Jeff Z. Pan, Martin Kollingbaum and Yu Deng

A tree-style one-pass tableau for an extension of ECTL^{+} Alexander Bolotov, Montserrat Hermo and Paqui Lucio

12:30 Poster Session

13:00 Group Photo and Lunch

14:00 Short Talks (see pages 14-28 of online booklet)

Leo-III: A Theorem Prover for Classical and Non-Classical Logic Alexander Steen

The Elfe Prover — Verifying mathematical proofs of undergraduate students Maximilian Doré

Higher-order Reasoning Vampire Style Ahmed Bhayat and Giles Reger

Proving security properties of CHERI-MIPS Kyndylan Nienhuis, Alexandre Joannou and Peter Sewell

Extending the K$_{S}$P Prover to More Expressive Modal Logics Fabio Papacchini, Cláudia Nalon, Ullrich Hustadt and Clare Dixon

Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics Alasdair Armstrong, Thomas Bauereiss, Kathryn E. Gray, Prashanth Mundkur,Alastair Reid, Peter Sewell, Brian Campbell, Shaked Flur, Robert M. Norton, Christopher Pulte, Ian Stark and Mark Wassell

Verifying Strong Eventual Consistency for Conflict-free Replicated Data Types Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan and Alastair R. Beresford

Formalisation of MiniSail in the Isabelle Theorem Prover Alasdair Armstrong, Neel Krishnaswami, Peter Sewell and Mark Wassell

15:30 Poster Session and Coffee

16:30 Screening of a documentary about Bletchley Park. Finishing at 17:30.

19:00 Dinner (Clare College Old Court). Dinner is served at 19:00. Please arrive before 18:50.

Friday 13 April

9:00 Short Talks (see pages 29-38 of online booklet)

Real-world formal documentation Thomas Tuerk

Tuning Natural Deduction Proof Search by Analytic Methods. Alexander Bolotov and Alexander Gorchakov

Instantiation for Theory Reasoning in Vampire Giles Reger and Martin Riener

Towards Polynomial Time Forgetting and Instance Query; Rewriting in Ontology Languages Sen Zheng and Renate A.Schmidt

Iterative Abduction using Forgetting Warren Del-Pinto and Renate Schmidt

10:00 Coffee and Poster Session

10:30 Short Talks (see pages 39-48 of online booklet)

Equivariant ZFA with Choice: a position paper Murdoch J. Gabbay

BREU with Connections Peter Backeman

Designing a proof calculus for the application of learned search heuristics Michael Rawson and Giles Reger

Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction Yang XU, Jun LIU, Shuwei CHEN, Xiaomei ZHONG and Xingxing HE

Reasoning with Large Theories Using Over-Approximation Abstraction-Refinement Julio Cesar Lopez Hernandez and Konstantin Korovin

11:30 Coffee and Poster Session

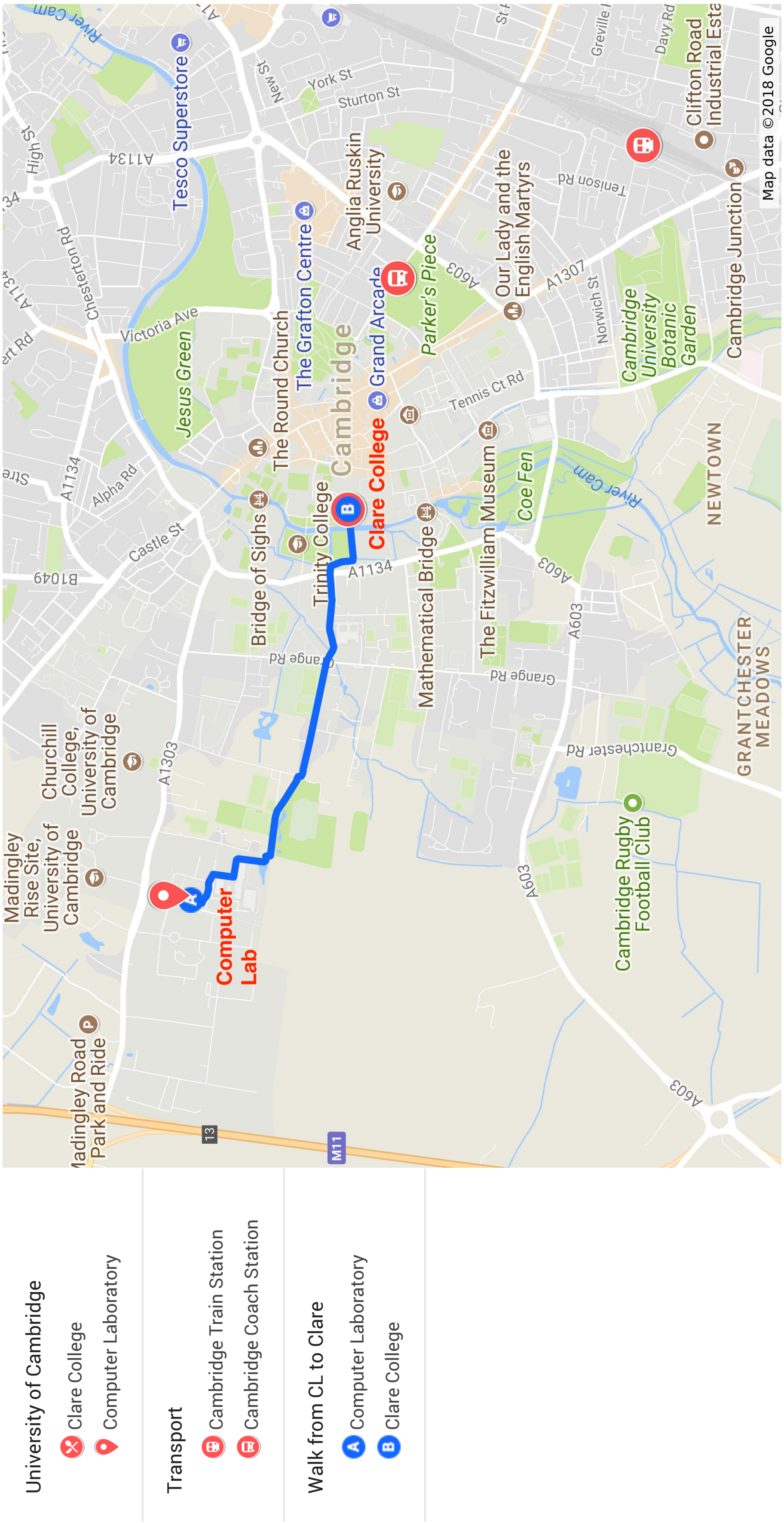
12:00 Discussion Panel

13:00 Lunch

14:00 Invited Talk (Larry Paulson)

15:00 Coffee and Business meeting

Local Map


University of Cambridge
Clare College
Computer Laboratory
Transport
Cambridge Train Station
Cambridge Coach Station
Walk from CL to Clare
A Computer Laboratory
B Clare College
Computer Lab
Clare College
Cambridge
Trinity College
Bridge of Sighs
The Round Church
The Grafton Centre
Grand Arcade
Parker's Piece
Anglia Ruskin University
Our Lady and the English Martyrs
Mathematical Bridge
The Fitzwilliam Museum
Coe Fen
Jesus Green
Tesco Superstore
Victoria Ave
Chesterton Rd
Castle St
Alpha Rd
Madingley Rise Site, University of Cambridge
Churchill College, University of Cambridge
Madingley Road Park and Ride
Cambridge University Botanic Garden
Cambridge Junction
Clifton Road Industrial Esta
Cambridge Rugby Football Club
NEWTOWN
GRANTCHESTER MEADOWS
River Cam
Grange Rd
Grantchester Rd
Tennis Ct Rd
Tenison Rd
Norwich St
York St
Sturton St
New St
High St
Davy Rd
Greville
A1134
A1303
A603
A1307
B1049
M11
13
Map data ©2018 Google

List of Participants

Name	Affiliation
Alasdair Armstrong	University of Cambridge
Edward Ayers	University of Cambridge
Peter Backeman	Uppsala University
Thomas Bauereiss	University of Cambridge
Ahmed Bhayat	University of Manchester
Alexander Bolotov	University of Westminster
Juan Casanova	University of Edinburgh
Julio Cesar Lopez Hernandez	University of Manchester
Cristina David	University of Cambridge
Warren Del-Pinto	University of Manchester
Maximilian Doré	University of Munich
Shaked Flur	University of Cambridge
Murdoch Gabbay	Heriot-Watt University
Victor Gomes	University of Cambridge
Ullrich Hustadt	University Of Liverpool
Mateja Jamnik	University of Cambridge
Ekaterina Komendantskaya	Heriot-Watt University
Konstantin Korovin	University of Manchester
Angeliki Koutsoukou Argyraki	University of Cambridge
Pauline Kra	Yeshiva University
Ramana Kumar	DeepMind
Ben Laurie	DeepMind
Wenda Li	University of Cambridge
Yue Li	University of Edinburgh
Jun Liu	Ulster University
Paqui Lucio	University of the Basque Country
Chaitanya Mangla	University of Cambridge
Kyndylan Nienhuis	University of Cambridge
Fabio Papacchini	University of Liverpool
Lawrence Paulson	University of Cambridge
Francis Priestland	ARM
Kasper Priskorski	GRAKN.AI
Daniel Raggi	University of Edinburgh
Michael Rawson	University of Manchester
Giles Reger	University of Manchester
Alastair Reid	ARM Research
Martin Riener	University of Manchester
Renate Schmidt	University of Manchester
Peter Sewell	University of Cambridge
Zohreh Shams	University of Cambridge
Yiannos Stathopoulos	University of Cambridge
Alexander Steen	Freie Universität Berlin
Aaron Stockdill	University of Cambridge
Thomas Tuerk	
Duo Wang	University of Cambridge
Mark Wassell	University of Cambridge
Robert Watson	University of Cambridge
Kemas Wiharja	University of Aberdeen
Sen Zheng	University of Manchester

Invited Talks

Contributed Abstracts

Machine learning for mining, understanding and automating computer proof

Ekaterina Komendantskaya, Heriot-Watt University

As software for mechanised proofs flourishes, we are about to enter the age of "Big Proof" (i.e. a Big Data stage of mechanised proof development). Large corpora of computer proofs, written in a range of programming languages, is already available on the Web. The question is: How much of Big Data technology is applicable in "Big Proof" domain?

In this talk, I will give a comparative overview of several machine learning methods used to mine, analyse and understand the existing corpora of mechanised proofs, and to automate new proofs. I will use the Machine Learning for Proof General (ML4PG) tool for Demos.

A Career in Research: Mike Gordon and Hardware Verification

Lawrence Paulson, University of Cambridge

The story of Mike Gordon's scientific career is instructive. Mike conceived the radical idea of hardware verification in the late 70s, a surprising choice given the number of other new fields he could have joined. Mike talked to researchers in the systems side of his Department (at Edinburgh). With impressive method, he learned about hardware and designed a small microcoded computer. Then he investigated the problem of how to verify this computer.

First he wanted to model behaviours using recursive domain equations. Then he opted for a CCS-like process calculus and implemented it on top of LCF. Finally he opted for higher-order logic, again a radical choice compared with the favoured alternatives of first-order logic and dependent type theory. Ultimately he realised his ambitions on a grand scale, with the verification of the ARM6 processor and landmark work on verifying assembly language code and proof-producing compilation. His foresight and boldness allowed him to transform the practices of verification and hardware design.

Discussion Panel: Natural language processing and information retrieval for automated reasoning

Ekaterina Komendantskaya, Lawrence Paulson and Yiannos Stathopoulos

The focus of the discussion will be on large-scale mathematical knowledge available in formalised libraries; In particular, on how these libraries can support sophisticated searches using natural language processing and information retrieval technologies. Amongst others, an important aim of developing sophisticated search tools is to provide automated support for construction of formal proofs, for example, by mining libraries for proof patterns and clustering lemmas based on the similarity of the proofs they are used in.

Meta-unification: An algorithm for finding solutions to constraint systems over unifiers with meta-variables

Juan Casanova juan.casanova@ed.ac.uk   Alan Bundy a.bundy@ed.ac.uk

University of Edinburgh, 10 Crichton St, Edinburgh EH8 9AB

Abstract: Certain approaches to detecting faults in ontologies rely on the ability to obtain provable instantiations of schematic formulas containing meta-variables (variables ranging over sub-formulas). One way to obtain such instantiations is to solve systems of constraints over unifiers arising from a generalization of resolution. A system of constraints over unifiers is a set of equations including first-order logic terms and atoms, as well as *unifier variables*. Such a system is satisfied by a set of substitutions (one for each unifier variable) if the equations hold for those substitutions and if each substitution is the most general unifier for the equation of which it is the outer-most unifier. Without meta-variables, each well formed system has at most one solution, which can be obtained by the well known most general unifier algorithm. However, when a system includes meta-variables, which are instantiated *before* solving the system, there may be different instantiations of the meta-variables that give rise to different most general unifiers. We implemented an algorithm designed to solve systems with meta-variables, giving an enumeration of both most general unifiers and meta-variable instantiations in an efficient way.

1 Introduction

In the context of automated fault detection in logical ontologies (e.g. OWL), we conducted previous work involving meta-logical reasoning [1] and, in particular, finding instantiations of meta-variables in schematic formulas (representing typical error patterns in ontology design) that make them inferable (provable) in the ontology. Under this framework, we can *automate* the process of finding these patterns, a common approach in the ontology community (see, for example, [2]). Our first implementation of this framework using a naive algorithm proved to be intractable even for the simplest cases.

To the best of our knowledge, theorem proving over formulas with meta-variables has not been thoroughly explored in the literature, and as a consequence, we designed and implemented an algorithm which solves unification with meta-variables with promising performance. This can be paired with a simple generalization of resolution to automatically find provable instantiations of a formula with meta-variables.

This is work in progress. The algorithm has been implemented and has passed a set of unit tests for correction, but its performance has not yet been thoroughly evaluated and it still lacks a robust theoretical background. Here, we present the ideas behind the algorithm and its current results.

2 Systems of unifier constraints with meta-variables

A unification problem in first-order logic consists in finding the most general substitution σ that, when applied to two different terms[1], makes them equal. This can be expressed as an equation:

$$\sigma t_1 = \sigma t_2 \tag{1}$$

where σ is a *unifier variable*, t_1 and t_2 are given terms and there is an implicit constraint that σ is the most general substitution satisfying the equation.

In the context of resolution theorem proving, several unification problems are solved, with a different unifier applied to each, and where results from previous unifications can be arguments to latter ones. Situations like this can be expressed as systems of constraints, such as:

$$\begin{array}{l} \sigma_1 x = \sigma_1 f(y,z) \\ \sigma_2 g(w,w) = \sigma_2 \sigma_1 g(x,w) \\ \sigma_3 \sigma_1 x = \sigma_3 \sigma_2 w \end{array} \tag{2}$$

where f, g are function symbols in the logic signature.

This representation easily allows to include *meta-variables*. Each instantiation of meta-variables gives rise to a new system of constraints with a different set of most general unifiers, and corresponds to a different theorem in the proof that generated the constraints. We will use upper case letters starting from A to represent meta-variables.

A solution to such a problem is a set (an enumeration, as the set is in many cases infinite) of meta-variable instantiations that make the resulting system of constraints satisfiable. For example, the system:

$$\begin{array}{l} \sigma_1 A = \sigma_1 f(y) \\ \sigma_2 \sigma_1 B = \sigma_2 f(y) \end{array} \tag{3}$$

[1] Or atoms. For the rest of the text, we will refer only to terms, because atom unification can be easily reduced to term unification, even with meta-variables.

has as a possible solution the instantiation $A := f(y), B := f(y)$, but also $A := x, B := f(z)$.

3 Dependency factorization unification

We define a *dependent* to be one or more unifier variables applied to a variable. For example, $\sigma_2\sigma_1 x$. An equation like $\sigma_1 x = \sigma_1 f(y, z)$ establishes a relation between the dependents $\sigma_1 x, \sigma_1 y$ and $\sigma_1 z$. Namely, $\sigma_1 x = f(\sigma_1 y, \sigma_1 z)$. We represent these relations in a *dependency graph* where nodes are dependents and edges are dependencies. Note that this is a multi-graph where edges may have multiple sources. For example, the edge $\sigma_1 x = f(\sigma_1 y, \sigma_1 z)$ has two sources: $\sigma_1 y, \sigma_1 z$ and one target: $\sigma_1 x$.

The most general unifier algorithm acts by matching the outer-most function symbol of the two sides of an equation, if they have one, and recursively unifying their sub-terms. Inspired by this idea, we can solve unification problems by a process of*factorization* of dependencies in a dependency graph, where equations have been previously simplified to express dependencies. For example, the equation

$$\sigma_1 f(x, g(x)) = \sigma_1 f(g(y), g(g(h(z)))) \tag{4}$$

can be simplified to the two equations:

$$\begin{aligned} \sigma_1 x &= g(\sigma_1 y) \\ \sigma_1 x &= g(h(\sigma_1 z)) \end{aligned} \tag{5}$$

corresponding to dependencies in a dependency graph. We can then *factorize* this graph[2] by merging all incoming dependencies to the dependent $\sigma_1 x$ (as they must be equal), which gives us the new dependency $\sigma_1 y = h(\sigma_1 z)$.

The dependency graph has several properties that relate to those of the unification problem. For example, a cycle in this directed graph indicates an occurs check. Perhaps more interestingly, a root in this graph (for example, $\sigma_1 z$) indicates a dependent which *does not depend on anything*. This is where the most general unifier constraint becomes important, as if the dependent is independent, then the most general unifier must necessarily replace it with a fresh variable, call it z_{σ_1}. This means that we can *solve* this node as $\sigma_1 z = z_{\sigma_1}$ and then *propagate* this value through the dependencies.

4 Enter meta-variables

The dependency graph representation is particularly useful for dealing with meta-variables. We may include a new kind of dependents corresponding to meta-variables. For example, $\sigma_1 A$, and generate, factorize and propagate dependencies in the same way that we did when there were no meta-variables.

The catch, however, is that a root in a graph with meta-variables does not necessarily mean that we can assume that dependent to be independent. For example, the graph with the two dependencies $\sigma_1 A = f(\sigma_1 x)$, $\sigma_1 y = f(\sigma_1 z)$ has $\sigma_1 z$ as one of its roots, but if A were instantiated to $A := z$, then $\sigma_1 z$ would no longer be independent. Not all is lost, though, and especially when considering systems of constraints with plenty of unifiers (such as the ones arising from a resolution proof), meta-variables may appear with different unifiers and there may be variables (like z_{σ_1}) which cannot be part of a meta-variable instantiation, and so certain parts of the graph may be solved before enumerating possible values for meta-variables.

Calculating which root dependents can be safely given a value has low complexity (at most quadratic). If there were none of these, then we are left with the inevitable task of enumerating all possible values of a meta-variable, resulting in a dependency graph for each possible instantiation.

5 Implementation

We currently have a working implementation in Haskell. This has given correct results for an extensive set of test cases. The algorithm's performance is also promising (outputting several hundreds of solutions to systems with two meta-variables in one minute). However, we have not yet had the time to properly evaluate it and compare it with more naive approaches to the problem, or to provide theoretical results in this regard.

6 Conclusion

We believe that, in conjunction with a resolution procedure, our algorithm is a promising approach to finding provable instantiations of formulas with meta-variables in an efficient manner. It is designed to enumerate meta-variables only when, as far as we understand it, it is absolutely necessary to do so, therefore reducing inefficiencies as much as the problem itself permits.

Future work includes thorough evaluation of the algorithm, building the resolution procedure on top of the meta-unification algorithm, developing theoretical justifications for the algorithms workings and, of course, application to the original motivating problem of ontology fault detection.

References

[1] Juan Casanova. Meta-ontology fault detection. Master's thesis, University of Edinburgh, 2017. (https://tinyurl.com/yb72r3ch).

[2] Alan Rector et al. *OWL Pizzas: Practical Experience of Teaching OWL-DL: Common Errors & Common Patterns.* Springer Berlin Heidelberg, 2004.

[2] Note that this does **not** correspond the usual notion of graph factorization.

Separated Normal Form Transformation Revisited

Cláudia Nalon[1] Ullrich Hustadt[2] Clare Dixon[2]

[1] Department of Computer Science, University of Brasília, Brasília, Brazil
nalon@unb.br
[2] Department of Computer Science, University of Liverpool, Liverpool, UK
{U.Hustadt, CLDixon}@liverpool.ac.uk

Abstract: We consider the transformation of propositional linear time temporal logic formulae into a clause normal form, called *separated normal form*, suitable for resolution calculi. In particular, we investigate the effect of applying various pre-processing techniques on characteristics of the normal form and determine the best combination of techniques on a large collection of benchmark formulae.

1 Introduction

Clause normal forms are the foundation of most resolution calculi and such calculi exist for a wide range of logics, including propositional, first-order, modal and temporal logics. Given a formula, its clause normal form is typically computed using a combination of equivalence or satisfiability preserving rewrite steps, including simplification as a special case, and renaming, which replaces complex subformulae by new propositional variables. Variations in the normal form can greatly influence the performance of resolution-based reasoning systems and transformation procedures that compute clause normal forms typically aim to produce fewer and/or shorter clauses for a given formula. For propositional and first-order logic the computation of such 'small' clause normal forms is well-studied [4, 1, 5]. However, the problem has been not been investigated to the same extent for non-classical logics, one exception being the work by Dixon and Nalon on prenexing versus anti-prenexing for modal logics [3].

In this paper we revisit the problem of computing a clause normal form for propositional linear time temporal logic (PLTL), called *separated normal form* (SNF). In particular, we determine the effect of applying various pre-processing techniques during the computation.

2 PLTL and SNF

PLTL is an extension of propositional logic with unary operators $\bigcirc$ (in the next moment of time), $\square$ (always) and $\Diamond$ (eventually) and binary operators $\mathcal{U}$ (until) and $\mathcal{W}$ (unless). PLTL-formulae are interpreted over infinite sequences of states $\sigma = (s_i)_{i \in \mathbb{N}}$ such that each s_i, $0 \leq i$, is a propositional valuation. A PLTL-formula φ is *satisfiable* iff there exists an infinite sequence of states $\sigma = (s_i)_{i \in \mathbb{N}}$ such that φ holds at s_0 in σ. Two PLTL-formulae φ and ψ are *equi-satisfiable* iff φ is satisfiable if and only if ψ is satisfiable.

A PLTL-formula $\bigwedge_{1 \leq i \leq n} C_i$ is in *Separated Normal Form* (SNF) iff every conjunct (*clause*) C_i, $1 \leq i \leq n$, has one of the following three forms.

$$\bigvee_{i=1}^{m} l_i \qquad \text{(initial clause)}$$
$$\square(\bigvee_{i=1}^{m} l_i \vee \bigvee_{j=1}^{n} \bigcirc l'_j) \qquad \text{(global clause)}$$
$$\square(\bigvee_{i=1}^{m} l_i \vee \Diamond l'_1) \qquad \text{(eventuality clause)}$$

where $n, m \geq 0$ and for every $1 \leq i \leq m$, $1 \leq j \leq n$, l_i and l'_j are propositional variables. Such a finite set N is equivalent to the conjunction of its elements.

Every PLTL-formula φ can be transformed into an equi-satisfiable formula in SNF. Fisher, Dixon and Peim [2] describe functions τ_0 and τ_1 with $\tau_0(\varphi) = (q_1 \wedge \tau_1(\square(\neg q_1 \vee \varphi)))$, where q_1 is a fresh propositional variable not occurring in φ, such that $\tau_0(\varphi)$ is in SNF and equi-satisfiable to φ. The function τ_1 proceeds top-down and uses *renaming* to deal with subformulae that are not yet in normal form. Part of the inductive definition of τ_1 is shown in Figure 1.

It is easy to see that τ_1 will not always produce a normal form with the smallest number of clauses. For example, $\varphi_1 = \square(\neg q \vee \neg \bigcirc \neg p)$ is equivalent to $\square(\neg q \vee \bigcirc p)$ which is in normal form, but, according to (1), $\tau_1(\varphi_1)$ would produce a conjunction of two clauses. This could be avoided by converting formulae to *negation normal form* before applying τ_1. The formula $\varphi_2 = \square(\neg q \vee (p \,\mathcal{U}\, p))$ is equivalent to $\square(\neg q \vee p)$ which again is in normal form, but,

$$\tau_1(\square(\neg q \vee \neg \bigcirc \varphi)) = \tau_1(\square(\neg q \vee \bigcirc q')) \wedge \tau_1(\square(\neg q' \vee \neg \varphi)) \tag{1}$$
$$\tau_1(\square(\neg q \vee \square\, l)) = \square(\neg q \vee l) \wedge \square(\neg q \vee q') \wedge \square(\neg q' \vee \bigcirc l) \wedge \square(\neg q' \vee \bigcirc q') \tag{2}$$
$$\tau_1(\square(\neg q \vee (\varphi \,\mathcal{U}\, \psi))) = \tau_1(\square(\neg q \vee (q' \,\mathcal{U}\, \psi)) \wedge \tau_1(\square(\neg q' \vee \varphi)) \qquad \text{if } \varphi \text{ is not a literal} \tag{3}$$
$$\tau_1(\square(\neg q \vee (\varphi \,\mathcal{U}\, \psi))) = \tau_1(\square(\neg q \vee (\varphi \,\mathcal{U}\, q')) \wedge \tau_1(\square(\neg q' \vee \psi)) \qquad \text{if } \psi \text{ is not a literal} \tag{4}$$
$$\begin{aligned}\tau_1(\square(\neg q \vee (l_1 \,\mathcal{U}\, l_2))) = {} & \square(\neg q_1 \vee \Diamond l_2) \wedge \square(\neg q \vee l_1 \vee l_2) \wedge \square(\neg q \vee q' \vee l_2) \\ & \wedge \square(\neg q \vee \bigcirc l_1 \vee \bigcirc l_2) \wedge \square(\neg q \vee \bigcirc q' \vee \bigcirc l_2)\end{aligned} \tag{5}$$

Figure 1: Partial Definition of Transformation Function τ_1

according to (5), $\tau_1(\varphi_2)$ would produce a conjunction of five clauses. This could be avoided by *simplification* using well-known equivalences among temporal formulae. Using such equivalences we can also extend or reduce the scope of temporal operators. For example, *anti-prenexing* would replace $\Diamond(p \vee q)$ by the equivalent $(\Diamond p \vee \Diamond q)$ while *prenexing* would do the opposite. Since τ_1 only preserves satisfiability, we can go even further for φ_1 and φ_2. In both these formulae the propositional variable p only occurs with positive polarity. In analogy to propositional logic, we can apply *pure literal elimination*, that is, we replace variables that only occur positively (negatively) by $\top$ ($\bot$), and then simply. For φ_1 and φ_2 we obtain $\top$ as result. Finally, a peculiarity of the normal form transformation by τ_0 and τ_1 is that for a formula φ in normal form, $\tau_0(\varphi)$ will not be equal to φ. Say, φ_3 is $(q_2 \wedge \Box(\neg q_2 \vee p))$. Then $\tau_0(\varphi_3) = (q_1 \wedge \tau_1(\Box(\neg q_1 \vee ((q_2 \wedge \Box(\neg q_2 \vee p))))))$. Computing τ_1 will involve (2), creating four additional clauses. We can ameliorate this problem by modifying τ_1 so that it treats $\Box(\neg q_1 \vee \Box\varphi)$ like $\Box(\neg q_1 \vee \varphi)$ for the specific propositional variable q_1 used by τ_0.

3 Implementation and Evaluation

We have implemented τ_0, τ_1 and the techniques described in Section 2 in the tool `pre-trp`. By default, `pre-trp` will just compute τ_0 for a given PLTL-formula. Options allow the user to enable additional techniques: `-simp` for simplification, `-ple` for literal elimination, `-aprenex` for anti-prenexing, `-isnf` for the modified version of τ_1.

To evaluate the effectiveness of each technique and their combinations we have used a set of PLTL-formulae collected by Schuppan and Darwiche [6]. The collection consists of 7450 formulae divided into seven classes (acacia, alaska, anzu, forobots, rozier, schuppan, trp). Half of the formulae are obtained by negating the original formulae. Since we also compared `pre-trp` with an earlier implementation of the SNF transformation, we have only used 6135 of these formulae.

Table 1 shows for particular combinations of `pre-trp` options and for each class the total number of fresh propositional variables introduced in the transformation, the total number of clauses produced, the total size of the normal time and the sum of individual computation times. Overall, the combination of `-isnf`, `-simp`, and `-ple` offers the best result. The option `-isnf` offer the greatest improvement on the trp class, as half its formulae are already in normal form. The option `-ple`, pure literal elimination, shows the greatest improvement on the rozier class as most formulae in that class are randomly generated and contain a large number of pure literals. Option `-aprenex`, anti-prenexing, appears to have a detrimental effect. This is in contrast to the results in [3] for basic modal logic, where anti-prenexing was found to be beneficial.

Overall, the results show that the application of preprocessing techniques significantly reduced the size of the normal form and that, if implemented well, as in `pre-trp`, this application comes at negligible computational cost.

Options	Class	Fresh Prop. Variables	#Clauses	Total Size	Sum Times
	acacia	4,385	8,922	56,149	0.00
	alaska	25,588	69,884	467,918	0.00
	anzu	29,684	73,917	500,610	0.00
	forobots	1,344	4,460	37,096	0.00
	rozier	441,771	977,463	6,662,647	0.25
	schuppan	6,871	14,311	86,963	0.00
	trp	81,357	211,201	1,401,406	0.00
	total	**591,000**	**1,360,158**	**9,212,789**	**0.25**
-isnf	acacia	3,091	6,334	41,915	0.00
-simp	alaska	22,309	63,345	431,659	0.00
	anzu	14,270	33,361	214,015	0.00
	forobots	996	3,745	33,146	0.00
	rozier	370,371	806,835	5,469,589	0.27
	schuppan	6,532	13,612	83,105	0.00
	trp	9,417	65,986	595,355	0.00
	total	**426,986**	**993,218**	**6,868,784**	**0.27**
-isnf	acacia	1,187	2,237	14,170	0.00
-ple	alaska	22,062	62,904	429,018	0.00
-simp	anzu	13,995	32,824	210,659	0.00
	forobots	996	3,745	33,146	0.00
	rozier	201,605	418,405	2,828,619	0.00
	schuppan	6,467	12,687	77,480	0.00
	trp	9,417	65,761	593,229	0.00
	total	**255,729**	**598,563**	**4,186,321**	**0.00**
-aprenex	acacia	2,936	5,317	31,227	0.00
-isnf	alaska	39,142	116,696	786,720	0.00
-ple	anzu	33,909	70,842	424,796	0.00
-simp	forobots	996	3,745	33,146	0.00
	rozier	517,913	965,768	6,476,842	0.25
	schuppan	6,867	13,387	81,386	0.00
	trp	287,470	542,456	3,192,606	0.00
	total	**889,233**	**1,718,211**	**11,026,723**	**0.25**

Table 1: Evaluation of SNF Transformation

References

[1] N. Azmy and C. Weidenbach. Computing tiny clause normal forms. In *Proc. CADE-24*, volume 7898 of *LNCS*, pages 109–125. Springer, 2013.

[2] M. Fisher, C. Dixon, and M. Peim. Clausal temporal resolution. *ACM Transactions on Computational Logic*, 2(1):12–56, 2001.

[3] C. Nalon and C. Dixon. Anti-prenexing and prenexing for modal logics. In *Proc. JELIA 2006*, volume 4160 of *LNCS*, pages 333–345. Springer, 2006.

[4] A. Nonnengart and C. Weidenbach. Computing small clause normal forms. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, chapter 6, pages 335–367. Elsevier, 2001.

[5] G. Reger, M. Suda, and A. Voronkov. New techniques in clausal form generation. In *Proc. GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 11–23. EasyChair, 2016.

[6] V. Schuppan and L. Darmawan. Evaluating ltl satisfiability solvers. In *Proc. ATVA 2011*, volume 6996 of *LNCS*, pages 397–413. Springer, 2011.

Models of Coinductive First-order Horn Clauses

Yue Li

Heriot-Watt University, Edinburgh, UK yl55@hw.ac.uk

Abstract: Proof-theoretic study of coinductive Horn clauses needs to establish soundness of proposed coinductive algorithms. Soundness is with respect to coinductive models of logic programs. In this paper we give examples on coinductive Horn clauses, and we review the two standard coinductive models for logic programs.

1 Introduction

Coinduction in logic programming refers to phenomena of non-terminating SLD-derivations and perhaps computation of infinite data therein. The name "coinduction" alludes to some association with "induction", explained later. Work on coinductive Horn clauses started in the 80s up till now, with significant progress made on model-theoretic semantics of infinite computation [4] and finite modelling of coinduction using loop detection [2, 3], and in the reference literatures there are motivating applications.

There are different ways to categorize coinductive Horn clauses, such as, whether or not the derivation is subject to loop detection, and, whether or not infinite data is computed. We give examples of coinductive Horn clauses below.

Example 1. Given program $P_1 : r(X) \supset r(X)$ (we use $\supset$ for implication), the goal $r(a)$ gives rise to an infinite SLD-derivation $r(a) - r(a) - r(a) - \cdots$ which does not compute infinite data but is subject to loop detection as there exist a sub-goal $r(a)$ that can be unified with an ancestor goal $r(a)$.

Example 2. Given $P_2 : p(s(X)) \supset p(X)$, the goal $p(a)$ gives rise to an infinite SLD-derivation $p(a) - p(s(a)) - p(s(s(a))) - \cdots$ which does not compute infinite data and is not subject to loop detection as there does not exist a sub-goal that can be unified with its ancestor goal.

Example 3. Given $P_3 : p(X) \supset p(s(X))$, the goal $p(X)$ gives rise to an infinite SLD-derivation $p(X) - p(X_1) - p(X_2) - \cdots$. In this case loop detection is applicable, for instance, sub-goal $p(X_1)$ unifies with its ancestor $p(X)$ and produces infinite data $X = s(s(\cdots))$. Meanwhile, the SLD-derivation itself also computes towards the same infinite data as given by loop detection.

Example 4. Given $P_4 : q(s(X), L) \supset q(X, [X|L])$, the goal $q(0, L)$ gives rise to an infinite SLD-derivation $q(0, L) - q(s(0), L_1) - q(s(s(0)), L_2) - \cdots$. The infinite data $L = [0, s(0), s(s(0)), \ldots]$ is computed by the SLD-derivation but loop detection is not applicable.

The challenge is to extract finite patterns of *irregular* non-terminating SLD-derivations, which cannot be done by loop detection, as shown in Examples 2 and 4. Above all, within the wider mathematical and computing community, the concept of *proof by coinduction* itself is much less renowned and appears much more obscure than the concept of *proof by mathematical induction*, despite that a fair amount of work on coinduction has been done in some branches of theoretical computing [5].

Our ongoing work promises to solve both problems mentioned above. Particularly, in our proof-theoretic study of coinduction, we explain about what kind of logical reasoning constitutes coinductive reasoning, and in what sense a coinductively derived conclusion is no less valid than an inductively derived one. For instance, a coinductive proof of a statement H, given a set P of premises, is merely an ordinary logical reasoning (i.e. based on classical or intuitionistic logic) that derives H from the augmented set $P \cup \{H\}$ of premises, such that H itself is asserted as known truth and is used in its own proof. Indeed it seems like a circular proof with self-referencing, but this is done in a *guarded* way so that the asserted H is not used *directly* to prove itself. The consequence of such reasoning is the conclusion that H is *coinductively true* with respect to P, which means that H cannot be refuted based on P.

In a mathematical sense, an inductively proved statement F from P is in the *least fixed point* of P which collects all conclusions of P that has a finite inductive proof, while a coinductively proved statement H from P is in the *greatest fixed point* of P, which additionally has all conclusions of P such that no attempt of inductive proof can terminate. In Section 2 we review these fixed points in order to set up a necessary mathematical context for discussion of logical soundness of our proof-theoretic presentation of coinduction. We conclude and discuss related topics in Section 3.

2 Models for logic programs

We review the two coinductive models of logic programs. We assume readers' familiarity with syntax of first order Horn clause and the definition of terms and atoms as trees. We follow the terminology of [4].

A *first order Horn clause* (Horn clause, in short) K has the form

$$\forall x_1 \ldots x_m \quad A_1 \wedge \ldots \wedge A_n \supset A \quad (m, n \geq 0)$$

where A and variants thereof denote first order atoms. We denote the atom A by *head* K and we denote the set $\{A_1, \ldots, A_n\}$ by *body* K. A *logic program* is a finite set of Horn clauses.

Given a logic program P on signature Σ, we define the following sets. The *Herbrand universe*, denoted $\mathcal{H}$, is the set of all finite ground terms on Σ. The *Herbrand base*, denoted $\mathcal{B}$, is the set of all finite ground atoms on Σ. A *Herbrand interpretation* is any subset of $\mathcal{B}$. The *complete Herbrand universe*, denoted $\mathcal{H}'$, is the set of all finite and infinite ground terms on Σ. The *complete Herbrand base*, denoted $\mathcal{B}'$, is the set of all finite and infinite ground atoms on Σ. A *complete Herbrand interpretation* is any subset of $\mathcal{B}'$.

Given a Horn clause F, its *ground instance* on $\mathcal{H}$ is denoted $\lfloor F \rfloor$, and its ground instance on $\mathcal{H}'$ is denoted $\lfloor F \rfloor'$. Given a set S, its power set is denoted $Pow(S)$. There are two *immediate consequence operators* with respect to a program P, which are $\mathcal{T} : Pow\,(\mathcal{B}) \mapsto Pow\,(\mathcal{B})$ and $\mathcal{T}' : Pow\,(\mathcal{B}') \mapsto Pow\,(\mathcal{B}')$, defined respectively as:

$$\mathcal{T}(I) = \{t \in \mathcal{B} \mid F \in P,\ head\ \lfloor F \rfloor = t,\ body\ \lfloor F \rfloor \subseteq I\ \}$$

$$\mathcal{T}'(J) = \{s \in \mathcal{B}' \mid F \in P,\ head\ \lfloor F \rfloor' = s,\ body\ \lfloor F \rfloor' \subseteq J\ \}$$

Note that $\langle Pow(\mathcal{B}), \subseteq \rangle$ and $\langle Pow(\mathcal{B}'), \subseteq \rangle$ are complete lattices. It is also known that $\mathcal{T}$ and $\mathcal{T}'$ are increasing. Then, based on Knaster-Tarski fixed point theorem, each operator has a greatest fixed point (denoted $gfp(\mathcal{T})$ and $gfp(\mathcal{T}')$ respectively), which we take as *coinductive models* for a logic program P, as follows.

$$\mathcal{M} = gfp(\mathcal{T}) = \bigcup\{I \mid I = \mathcal{T}(I)\} = \bigcup\{I \mid I \subseteq \mathcal{T}(I)\}$$

$$\mathcal{M}' = gfp(\mathcal{T}') = \bigcup\{I \mid I = \mathcal{T}'(I)\} = \bigcup\{I \mid I \subseteq \mathcal{T}'(I)\}$$

We call $\mathcal{M}$ the *finite-term coinductive model*, and $\mathcal{M}'$ the *infinite-term coinductive model*. Note that $\mathcal{M} \subseteq \mathcal{M}'$ by definition, as $\mathcal{M}$ gathers all finite ground atoms that either has a finite proof or has an infinite derivation, while $\mathcal{M}'$ additionally contains all such, but infinite, atoms.

We could verify, regarding Examples 1–4, that results by infinite SLD-derivations and by loop detection (whenever applicable) are true with respect to coinductive models. Regarding Example 1,

$$\mathcal{M}(P_1) = \mathcal{M}'(P_1) = \{r(a)\}$$

Obviously the result $r(a)$, which is from both loop detection and infinite derivation, is in the models. Regarding Example 2,

$$\mathcal{M}(P_2) = \{p(a),\ p(s(a)),\ p(s(s(a))),\ \ldots\}$$
$$\mathcal{M}'(P_2) = \mathcal{M}(P_2) \cup \{p(s(s(\cdots)))\}$$

where there is the result $p(a)$ by infinite derivation. Regarding Example 3,

$$\mathcal{M}(P_3) = \emptyset \qquad \mathcal{M}'(P_3) = \{p(s(s(\cdots)))$$

We find the result of loop detection and infinite derivation in $\mathcal{M}'(P_3)$. Regarding Example 4,

$$\mathcal{M}(P_4) = \emptyset \qquad \mathcal{M}'(P_4) = \{q(t, \geq t) \mid t \in \mathcal{H}'(P_4)\}$$

where we use $\geq t$ as a short hand for the infinite list $[t, s(t), s(s(t)), \ldots]$, and note that $\geq s(s(\cdots))$ is $[s(s(\cdots)), s(s(\cdots)), s(s(\cdots)), \ldots]$. The result $q(0, \geq 0)$ by infinite SLD-derivation is in this model. Note that $\mathcal{M}(P_3)$ and $\mathcal{M}(P_4)$ are empty, since with respect to P_3 and P_4, no finite atom has a finite proof or an infinite derivation whatsoever.

3 Conclusion and discussion

We work with the finite- and infinite-term coinductive models when studying coinductive Horn clauses. A coinductive reasoning scheme (such as infinite SLD-derivation and loop detection, and our current proof-theoretic study) for Horn clauses is *sound* if all statements provable by the scheme are true with respect to coinductive models.

Infinite SLD-derivations given in Examples 2 and 4, corresponding to programs P_2 and P_4 respectively, are not subject to loop detection. However, there still exist goals for these two programs, such that the goals give rise to infinite SLD-derivations to which loop detection is applicable. For instance, providing goal $p(X)$ for P_2, we have an infinite SLD-derivation $p(X) - p(s(X)) - p(s(s(X))) - \cdots$, and loop detection produces infinite data $X = s(s(\cdots))$ by unifying $p(s(X))$ and $p(X)$, and the result $p(s(s(\cdots)))$ is in $\mathcal{M}'(P_2)$; providing goal $q(X, L)$ for P_4, we have an infinite SLD-derivation $q(X, L) - q(s(X), L_1) - q(s(s(X)), L_2) - \cdots$ from which loop detection produces infinite data $X = s(s(\cdots))$ and $L = \geq s(s(\cdots))$ by unifying $q(X, L)$ with $q(s(X), L_1)$, and the result $q(s(s(\cdots)), \geq s(s(\cdots)))$ is in $\mathcal{M}'(P_4)$. Given a program, we define a *Gupta goal* as a goal that gives rise to an infinite SLD-derivation to which loop detection is applicable. We can see that for all four programs from Examples 1–4, there exist a Gupta goal. An interesting question to ask is, *Can we find a program, for which every non-terminating goal is not a Gupta goal* ? So far we did not find such a program.

The two decisive steps in a coinductive soundness proof is the construction of a post-fixed point candidate, and then the validation of the candidate. The paper [1] presents instructive techniques involved in automating the first step.

Acknowledgement

Thanks for Dr. Ekaterina Komendantskaya for commenting the draft paper, and for Prof. Stephen H. Muggleton for raising the question about Gupta goal.

References

[1] L. Dennis et al. Using a generalisation critic to find bisimulations for coinductive proofs. In *Automated Deduction—CADE-14*, pages 276–290. Springer Berlin Heidelberg, 1997.

[2] G. Gupta et al. Coinductive logic programming and its applications. In *ICALP'07*, pages 27–44, 2007.

[3] E. Komendantskaya and Y. Li. Productive corecursion in logic programming. *J. TPLP (ICLP'17 post-proc.)*, 17(5-6):906–923, 2017.

[4] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[5] D. Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

Pattern-Based Reasoning to Investigate the Correctness of Knowledge Graphs

Kemas Wiharja[1] Jeff Z. Pan[1] Martin Kollingbaum[1] Yu Deng[2]

[1] University Of Aberdeen, UK
[2] IBM Research, US

Abstract: Most existing knowledge graph refinement approaches either focus on adding missing knowledge to the graphs, i.e., completion, or on identifying wrong information in the graphs, i.e. error detection. All of these approaches do not guarantee the extended knowledge graphs are consistent with the ontological schema of the input graph. The goal of this paper is to investigate the correctness of knowledge graphs by applying pattern-based reasoning. Experimental result shows that pattern-based reasoning could detect the problematic triplets in knowledge graphs.

1 Introduction

Constructing and maintaining large scale good quality knowledge graphs present many challenges. Most existing knowledge graph refinement approaches either focus on adding missing knowledge to the graph, i.e., completion, or on identifying wrong information in the graph, i.e. error detection. Paulheim [3] observed, in his recent survey on knowledge graph refinement, that there exist almost no approaches which do both completion and error detection at the same time. The ontological schema presents a unique opportunity to detect errors. The goal of this paper is to make use of ontological schema to do pattern based reasoning on knowledge graphs.

2 Background

2.1 Knowledge Graph Reasoning

Given a knowledge graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, as consisting of a set $\mathcal{A}$ (ABox) of interconnected typed entities and their attributes, as well as a set of schema axioms $\mathcal{T}$ (TBox) that defines the vocabulary used in a Knowledge Graph [2]. The size of $\mathcal{T}$ is often much smaller than that of $\mathcal{A}$. There are some knowledge graph reasoning services that are relevant to this paper.

- Consistency checking: $\mathcal{G}$ is consistent if there exists a model that satisfies all statements in $\mathcal{T}$ and $\mathcal{A}$.
- Classification: this service compute all the subsumptions among named types in $\mathcal{T}$.
- Justification: given a reasoning task t over $\mathcal{G}$ and its result r, this service computes the minimal subsets of $\mathcal{G}$ that justify the result r.

3 Inconsistency Detection and Repair

3.1 Why and When to Use Reasoning

Since reasoning can be very costly, we need to balance the gains and costs of using knowledge graph reasoning services. This could help us to decide what kind of knowledge graph reasoning services we need to run and when.

The benefits of using reasoning services include:

1. Reasoning helps detecting errors and identify invalid triples in knowledge graphs and repair them, by making use of consistency checking and related justification services.
2. Reasoning helps to materialize knowledge graphs for further embedding based completion.

Among the above two items, the first one is more costly due to the use of justification services, for calculating the minimal subsets of the input knowledge graph. Essentially, this means that we will need to run reasoning over (parts of) $\mathcal{G}$ many times. In practice, it might be time consuming. For this, we did a small experiment. We ran consistency checking and related justification services over the NELL-995 knowledge graph, which has 154,213 triples. There are a few inconsistencies. Calculating justifications for all of them takes more than 1 hour. Thus, in Sec 3.2, we propose to mainly use pattern based reasoning for error detection and repair.

3.2 Logical Inconsistency Detection via Pattern Based Reasoning

The main difference between our proposed pattern based reasoning and consistency checking with justifications is that the latter will consider all kinds of justification patterns for any inconsistencies, while the former only considers some patterns of justifications for some inconsistencies. Since some recent study [4] suggests type assertions are most often more correct in knowledge graphs than relation assertions, in this paper, we propose to focus on pattern based reasoning related to relation axioms.

In the Inconsistency Justifications (IJ) patterns listed in Table 1, since the ABox subsets of the IJ patterns contain at most two data triples, we need to scan through the data sub-graph of the input knowledge graph at most twice.

Once an IJ pattern is detected, we can repair it by removing the relation assertions in the pattern. Therefore, IJ

Table 1: Inconsistency Justification Patterns

ID	TBox subset of the Pattern	ABox subset of the Pattern
1	Domain(r)=D, D ⊓ A ⊑ ⊥	(e1, r, e2), (e1, rdf:type, A)
2	Range(r)=R, R ⊓ A ⊑ ⊥	(e1, r, e2), (e2, rdf:type, A)
3	Asymmetric(r)	(e1, r, e2), (e2, r, e1)

patterns not only help us to detect logical inconsistencies but also help us to repair logical inconsistencies.

4 Example of Inconsistency Justification Patterns in Knowledge Graphs

We used two knowledged graphs for this research: (i)NELL-995 [1] which is a knowledge graph/dataset from carnegie mellon university containing 142,065 triples, (ii) IBM-HM which is knowledge graph/dataset from IBM Hardware Maintenance containing 9228 triples.

Every dataset contains triplets in this format: (head entity; relation; tail entity). For example, in NELL dataset, we could find the valid triples as follows: (person:molly_moore; worksfor; city:washington_DC. This triple is considered as valid because the domain class of relation worksfor is person and the range class is city. We could also find inconsistency justification patterns in NELL such as these two triplets : (river:narew; emptiesIntoRiver; river:vistula) and (river:vistula; emptiesIntoRiver; river:narew). These two patterns are considered as inconsistency patterns because the relation emptiesIntoRiver is asymmetric relations.

5 Effectiveness of Pattern Based Reasoning and Semantic Inconsistency Checking

Experimental setup. To show the effectiveness of the pattern-based reasoning (in short PBR) and semantic inconsistency checking against global checking (in short GC), we composed two scenarios. We used FACT++ 1.6.5 (run with Protege) as the GC. Following are the scenarios:

- comparing how many invalid triples (in percentage) from original graph that can be covered by both approaches and how long it take to run the process.

Dataset. we used two datasets for this scenario: (i)NELL-995 which is a dataset from carnegie mellon university containing 142,065 triples and IBM Hardware Maintenance which is dataset from IBM containing 9228 triples. [1]

Results. The result of this scenario can be seen in table 2.

From table 2, we could see that PBR detects most of the invalid triples and much more efficient than consistency checking.

Table 2: Comparison between reasoner and pattern-based reasoning

Dataset	Reasoner		PBR	
	CE	RT	CE	RT
IBM-HM	100%	4,620 s	100%	296.68 s
NELL	100%	4,811 s	89.47%	119.9 s

6 Conclusion

We prove that pattern-based reasoning can detect problematic triple/inconsistency justification in knowledge graphs. Not only detecting, but our approach also could help in repairing the logical inconsistencies of the graphs. We are still improving the scalability our approach to handle bigger knowledge graphs.

Acknowledgments

The work presented in this paper has been fully supported by the Indonesia Endowment Fund for Education (LPDP).

References

[1] Tom M Mitchell, William W Cohen, Estevam R Hruschka Jr, Partha Pratim Talukdar, Justin Betteridge, Andrew Carlson, Bhavana Dalvi Mishra, Matthew Gardner, Bryan Kisiel, Jayant Krishnamurthy, et al. Never ending learning. In *AAAI*, pages 2302–2310, 2015.

[2] Jeff Z Pan, Diego Calvanese, Thomas Eiter, Ian Horrocks, Michael Kifer, Fangzhen Lin, and Yuting Zhao. *Reasoning Web: Logical Foundation of Knowledge Graph Construction and Query Answering: 12th International Summer School 2016, Aberdeen, Uk, September 5-9, 2016, Tutorial Lectures*, volume 9885. Springer, 2017.

[3] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[4] Heiko Paulheim and Christian Bizer. Improving the quality of linked data using statistical distributions. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):63–86, 2014.

[1] PBR : Pattern-Based Reasoning; CE : Coverage of Errors (in %); RT : Run Time (in second)

A tree-style one-pass tableau for an extension of ECTL$^+$.

Alexander Bolotov[1] Montserrat Hermo[2] Paqui Lucio[2]

[1] University of Westminster, W1W 6UW, London, UK.
[2] University of the Basque Country, 20018-San Sebastián, Spain.

Abstract: Only restricted versions of fairness are expressible in the well-known branching temporal logics ECTL and ECTL$^+$, while the full expressiveness of branching-time logic in CTL* makes this logic extremely challenging for the application of the tableau technique. Tree-shaped one-pass tableaux are well suited for the automation and are amenable for the implementation. We present here a sound and complete method of tree-style one-pass tableau for a sub-logic of CTL* which is more expressive than the logic ECTL$^+$allowing the formulation of some fairness constraints with 'until' operator. The provided example follows by an algorithm for constructing a systematic tableau that enables to prove completeness.

1 Introduction

For the specification of the reactive and distributed systems, or, most recently, autonomous systems, where the modelling of the possibilities 'branching' into the future is essential, the branching-time logics (BTL) give us an appropriate framework. The most used class of formalisms are 'CTL' (Computation Tree Logic) type logics: CTL itself, ECTL (Extended CTL) [2] that was defined to enable simple fairness constraints but not their Boolean combinations and ECTL$^+$ ([3]) which further extends ECTL allowing Boolean combinations of ECTL fairness constraints (but not permitting their nesting). The literature on fairness constraints, even in linear-time setting, lacks the analysis of their formulation with a 'stronger' temporal operator - $\mathcal{U}$ ('until') such as $\Box(A\,\mathcal{U}\,B)$ or $A\,\mathcal{U}\,\Box B$. Here we bridge this gap, providing an analysis of such complex fairness constraints with $\mathcal{U}$ (also allowing the nesting of temporal operators) in the branching-time setting weaker than CTL*. Thus, we consider the logic that extends ECTL$^+$ with the modalities $\Box\mathcal{U}$ and $\mathcal{U}\,\Box$. While the addition of the former does not increase the ECTL$^+$ expressiveness[1], $A\,\mathcal{U}\,(\Box B)$ cannot be expressed in the ECTL$^+$ language. The fairness constraint $\mathsf{A}(p\,\mathcal{U}\,\Box q)$ can be read as 'q is true along all paths of the computation except possibly their finite initial interval, where p is true'.

Lamport Notation / CTL-type name	Formulae expressible here but not above
$\mathcal{B}(\mathcal{U},\circ)$ / CTL	
$\mathcal{B}(\mathcal{U},\circ,\Box\Diamond)$/ ECTL	$\mathsf{E}(\Box\Diamond q)$
$\mathcal{B}^+(\mathcal{U},\circ,\Box\Diamond)$ / ECTL$^+$	$\mathsf{E}(\Box\Diamond q \wedge \Box\Diamond r)$
$\mathcal{B}^+(\mathcal{U},\circ,\mathcal{U}\,\Box)$	$\mathsf{A}((p\,\mathcal{U}\,\Box q) \wedge (s\,\mathcal{U}\,\Box\neg q))$
$\mathcal{B}^\star(\mathcal{U},\circ)$ / CTL*	$\mathsf{A}\Diamond(\circ p \wedge \mathsf{E}\circ\neg p)$

Figure 1: BTL classification.

In Figure 1 we fit our logic into the hierarchy of branching-time logics: '$\mathcal{B}$' is used for 'Branching', followed by the set of only allowed modalities as parameters; $\mathcal{B}^+$ indicates admissible Boolean combinations of the modalities and $\mathcal{B}^\star$ reflects 'no restrictions' in either concatenations of the modalities or Boolean combinations between them. We present a tree-style one-pass tableau for the logic $\mathcal{B}^+(\mathcal{U},\circ,\mathcal{U}\,\Box)$ continuing the analogous developments in linear-time case [4].

$\mathcal{B}^+(\mathcal{U},\circ,\mathcal{U}\,\Box)$ language is defined with linear-time temporal operators $\Box$ (always), $\circ$ (next time), and $\mathcal{U}$ (until), and path quantifiers - A (on all future paths) and E (on some future path). The state (σ) and path (π) formulae are defined below (state formulae are wff).

$\sigma ::= L \mid \sigma_1 \wedge \sigma_2 \mid \sigma_1 \vee \sigma_2 \mid \mathsf{A}\pi \mid \mathsf{E}\pi$
$\pi ::= \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \circ\sigma \mid \sigma\,\mathcal{U}\,\sigma \mid \sigma\,\mathcal{U}\,(\Box\sigma) \mid \Box\sigma$
$\quad \mid \Box(\sigma\,\mathcal{U}\,\sigma)$

2 The tableau method

While our tableaux are AND-OR trees with nodes labelled by sets of state formulae, the only rule which introduces AND-nodes is the next-state rule:

$$\frac{\Sigma, \mathsf{A}\circ\Phi_1, \ldots, \mathsf{A}\circ\Phi_n, \mathsf{E}\circ\Psi_1, \ldots, \mathsf{E}\circ\Psi_m}{\mathsf{A}\Phi_1, \ldots, \mathsf{A}\Phi_n, \mathsf{E}\Psi_1 \;\&\; \ldots \;\&\; \mathsf{A}\Phi_1, \ldots, \mathsf{A}\Phi_n, \mathsf{E}\Psi_m}$$

Figure 2: NEXT-STATE RULE. (Σ is a (possibly empty) set of literals; Φ_i, Ψ_i are non-empty sets of formulae.)

The next-state rule labels a branch that splits into m branches, at the node labelled by the premise of this rule. The conclusion of this rule uses the $\&$ symbol to reflect to generation of m AND-successor nodes.

$$\frac{\Sigma, \mathsf{E}(\Box(\sigma_1\,\mathcal{U}\,\sigma_2) \wedge \Pi)}{\Sigma, \mathsf{E}((\sigma_1\,\mathcal{U}\,\sigma_2) \wedge \circ\Box(\sigma_1\,\mathcal{U}\,\sigma_2) \wedge \Pi)}$$

Figure 3: α-RULE ($\mathsf{E}\Box\mathcal{U}$). (Σ is a (possibly empty) set of state-formulae and Π is a (possibly empty) conjunction of path-formulae.)

We also apply α- and β-rules: an α-rule expands a branch at the node labelled by its premise, with a node labelled by the conclusion; a β-rule splits a branch by two or

[1] $\Box(A\,\mathcal{U}\,B)$ can be expressed in ECTL$^+$ by $\Box(A \vee B) \wedge \Box\Diamond B$.

$$\frac{\Sigma, \mathsf{A}((\Box\sigma) \vee \Pi)}{\Sigma, \sigma, \mathsf{A}((\circ\Box\sigma) \vee \Pi) \mid \Sigma, \mathsf{A}\Pi}$$

Figure 4: β-RULE $(\mathsf{A}\Box\sigma)$. (Σ, σ is a set of state-formulae and Π is a (possibly empty) disjunction of path-formulae.)

three OR-nodes labelled by the formulae in its conclusion (separated by $|$).

We handle inputs in a new, branching-time, setting in 'analytic" way, extending similar construction for linear-time logic [4]. This extension is possible due to the definition of the 'context' in which eventualities are to be fulfilled in this new setting. Our β^+-rules are characteristic (and crucial!) for our construction. They tackle difficult cases of formulae in $\mathcal{B}^+(\mathcal{U}, \circ, \mathcal{U}\Box)$. The β^+-rules, similarly to β-rules, split a branch into two or three branches; these are the only rules that utilise 'context' to force the soonest satisfaction of the eventualities. The context is given by the sets of state (Σ) and path (Π) formulae. While, the outer-context was already used in the linear-time tableaux [4], for branching-time, the new concept of the 'inner-context' is introduced. Figure 5 shows a β^+ rule that handles a disjunction of formulae, including $\mathcal{U}$ in the scope of the A quantifier.

$$\frac{\Sigma, \mathsf{A}((\sigma\,\mathcal{U}\,\sigma') \vee \Pi)}{\Sigma, \sigma' \mid \Sigma, \sigma', \mathsf{A}(\circ((\sigma \wedge (\neg\Sigma \vee \varphi_\Pi))\,\mathcal{U}\,\sigma') \vee \Pi) \mid \Sigma, \mathsf{A}\Pi}$$

Figure 5: β^+-RULE $(\mathsf{A}\mathcal{U}\,\sigma)^+$. (Σ, σ, σ' is a set of state-formulae, Π is a (possibly empty) disjunction of path-formulae, and φ_Π is the state-formula introduced by Definition 1.)

Definition 1 *Let Π be a disjunction of path-formulae of the three forms $\Box\sigma$, $\sigma\,\mathcal{U}\,\Box\sigma'$ and $\Box(\sigma\,\mathcal{U}\,\sigma')$ where σ and σ' are state-formulae. The formula φ_Π to be the following disjunction of state-formulae:*

$$\bigvee_{\Box\sigma\in\Pi} \sigma \;\vee \bigvee_{\sigma\,\mathcal{U}\,\Box\sigma'\in\Pi} \sigma' \;\vee \bigvee_{\Box(\sigma\,\mathcal{U}\,\sigma')\in\Pi} \mathsf{E}(\textsc{t}\,\mathcal{U}\,\sigma').$$

3 Example

Our example of an open tableau illustrates the use of the inner context (see Figure 6). This tableaux is constructed by a systematic algorithm that keeps (along a branch) exactly one –if there is some– marked eventuality forcing its fulfilment. In Fig. 6, the semicolon inside the A-quantifier stands for disjunction, the marked eventuality is in black-boxes and $(\mathsf{Q}\circ)$ is the next-state rule. Note that Π consists of a path-formula $\Box p$, so φ_Π is just p. Since the marked eventuality is $\textsc{t}\,\mathcal{U}\,\neg p$ and the outer-context Σ is empty, the subformula $\sigma \wedge (\neg\Sigma \vee \varphi_\Pi)$ in the conclusion of the rule in Fig. 5 is just p. It is notable that this inner-context p enables the central branch to loop, given a model of the initial formula that –in this branch– does not force the eventuality, but satisfies the other disjunct in the A-quantifier: $\Box p$.

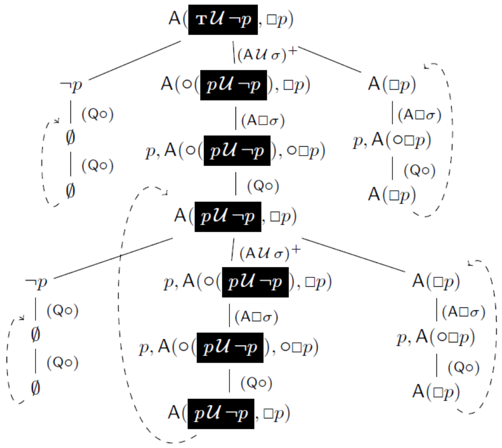


Figure 6: Open Tableau

4 Conclusion

We presented a tree-style one pass tableaux method for a new logic in the family of BTL – $\mathcal{B}^+(\mathcal{U}, \circ, \mathcal{U}\Box)$ – which extends the expressiveness of ECTL^+ fairness by a new class of fairness constraints utilising the $\mathcal{U}$ operator. The full details and the correctness proof are given in [1]. The tableaux rules that invoke the inner-context, enabled us to handle a particularly difficult class of formulae: A-disjunctive formulae with eventualities. The proof of correctness of β^+-rules is based on identifying relevant state-formulae inside specific path-modalities. This opens the prospect to study more expressive logics (eg $\mathsf{CTL}^\star$) by identifying subformulae that do not affect the 'context' which allows to simplify given structures. The presented technique, being the extension of a similar one for the linear-time setting, is amenable for implementation. In the refinement and implementation of our new algorithm we will rely on similar techniques used in the implementation of its linear-time analogue.

References

[1] A. Bolotov, M. Hermo, and P. Lucio. Extending fairness expressibility of ECTL+: a tree-style one-pass tableau approach, http://www.sc.ehu.es/jiwlucap/techreport18.pdf. Technical report, February 2018.

[2] E. A. Emerson and J. Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *Journal of Computer and System Sciences*, 30(1):1 – 24, 1985.

[3] E. A. Emerson and J. Y. Halpern. Sometimes and not never revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, January 1986.

[4] J. Gaintzarain, M. Hermo, P. Lucio, M. Navarro, and F. Orejas. Dual systems of tableaux and sequents for PLTL. *Journal of Logic and Algebraic Programming*, 78(8):701–722, 2009.

Leo-III: A Theorem Prover for Classical and Non-Classical Logics

Alexander Steen

Institute of Computer Science, Freie Universität Berlin, Arnimallee 7, 14195 Berlin,
a.steen@fu-berlin.de

Abstract: The automated theorem prover Leo-III for classical higher-order logic with Henkin semantics and choice is presented. Leo-III is based on extensional higher-order paramodulation and accepts every common TPTP dialect (FOF, TFF, THF), including their recent extensions to rank-1 polymorphism (TF1, TH1). In addition, the prover natively supports reasoning in almost every normal higher-order modal logic.

1 Introduction

Leo-III is an automated theorem prover (ATP) for classical higher-order logic (HOL) with Henkin semantics and choice that is implemented in Scala.[1] It is the successor of the well-known LEO-II prover [3], whose development significantly influenced the build-up of the TPTP THF infrastructure. Leo-III collaborates with external theorem provers, in particular, with first-order (FO) ATPs.

Leo-III supports all common TPTP dialects (CNF, FOF, TFF, THF) as well as the polymorphic variants TF1 and TH1. The prover returns results according to the standardized SZS ontology and additionally produces a TSTP-compatible (refutation) proof certificate, if a proof could be found. Furthermore, Leo-III natively supports reasoning for almost every normal HO modal logic. These hybrid logic competencies make Leo-III, up to the author's knowledge, the most widely applicable ATP available to date.

The presentation of Leo-III is partly adopted from a recent system description [5] which also contains extensive evaluation results on various benchmark sets.

HOL. HOL as addressed here has been proposed by Church, and further studied by Henkin, Andrews and others [1]. It provides lambda-notation, as an elegant and useful means to denote unnamed functions, predicates and sets (by their characteristic functions). In the remainder a notion of HOL with Henkin semantics and choice is assumed.

2 Calculus

Leo-III extends a complete, paramodulation based calculus for HOL with practically motivated, heuristic inference rules. They are grouped as follows:

Clause normalization. Leo-III employs *definitional clausification* to reduce the number of generated clauses. Moreover, *miniscoping* is employed prior to clausification. The remaining rules are straightforward.

Primary inferences. The primary inference rules of Leo-III are *paramodulation*, *equality factoring* and *primitive substitution*. The first two introduce *unification constraints* that are encoded as negative literals. Note that these rules are unordered and produce numerous redundant clauses. Leo-III uses several heuristics to restrict the number of inferences, including a *HO term ordering*. While these restrictions sacrifice completeness in general, recent evaluations confirm practicality of this approach [5]; complete search may be retained though. Primitive substitution instantiates free variables at top-level with approximations of predicate formulas using so-called *general bindings*.

Unification. Unification in Leo-III uses a variant of *Huet's pre-unification rules*. Negative equality literals are interpreted as unification constraints and are attempted to be *solved eagerly* by unification. In contrast to LEO-II, Leo-III uses *pattern unification* whenever possible. In order to ensure termination, the pre-unification *search is limited* to a configurable depth.

Extensionality rules. Dedicated *extensionality rules* are used in order to eliminate the need for extensionality axioms in the search space. The rules are similar to those of LEO-II [3].

Clause contraction. Additionally to standard simplification routines, Leo-III implements are variety of (equational) *simplification procedures*, including *subsumption*, destructive *equality resolution*, heuristic *rewriting* and contextual *unit cutting* (simplify-reflect).

Defined Equalities. Leo-III scans for common definitions of equality predicates and heuristically instantiates (or replaces) them with *primitive equality*.

Choice. Leo-III is designed for HOL with choice. Specialized rules *instantiate choice predicates* for subterms that represent choice operator applications and remove uninstantiated choice axiom schemes.

Function synthesis. If plain unification fails for a set of unification constraints, Leo-III may try to *synthesize function specifications* using dedicated choice instances that simulate suitable if-then-else terms. In general, this rule tremendously increases the search space. However, it also enables Leo-III to solve some hard problems (with TPTP rating 1.0). Also, Leo-III supports improved *reasoning with injective functions* by postulating the existence of left-inverses.

[1] Leo-III is freely available (BSD license) at http://github.com/leoprover/Leo-III.

Heuristic instantiation. Prior to clause normalization, Leo-III might *instantiate universally quantified* variables. This include exhaustive instantiation of finite types as well as partial instantiation for otherwise interesting types.

3 External Cooperation

In the tradition of the cooperative nature of the LEO prover family, Leo-III collaborates during proof search with external reasoning systems, in particular, with first-order ATPs such as E, iProver and Vampire as well as SMT solvers like CVC4. Unlike LEO-II, which translated proof obligations into untyped first-order languages, Leo-III, by default, translates its higher-order clauses to (polymorphic or monomorphic) many-sorted first-order formulas.

Leo-III accumulates higher-order clauses during proof search and repeatedly invokes all cooperating systems on the generated proof obligations. For that purpose, various translation mechanisms from HOL to different variants of first-order logic are implemented. If any external (first-order) reasoning system finds the submitted proof obligation to be unsatisfiable, the original HOL problem is unsatisfiable as well and a proof for the original conjecture is found.

4 Polymorphic HOL Reasoning

Leo-III supports reasoning in first- and higher-order logic with rank-1 polymorphism. The support for polymorphism has been strongly influenced by the recent development of the TH1 format for representing problems in rank-1 polymorphic HOL.

Central to the polymorphic adaption of Leo-III's calculus is the notion of type unification. Intuitively, whenever calculus rules requires two premises to have the same type, in the polymorphic adaption it suffices that the two terms' types are unifiable. For a concrete inference, the type unification is then applied first to the clauses; followed by the standard inference rule itself.

External cooperation for polymorphically typed HO clauses is already subsumed by the existing translation framework.

5 Modal Logic Reasoning

Modal logics have many relevant applications in computer science, artificial intelligence, mathematics and computational linguistics. They also play an important role in many areas of philosophy, including ontology, ethics, philosophy of mind and philosophy of science. Many challenging applications, as recently explored in metaphysics, require FO or HO modal logics (HOMLs). The development of ATPs for these logics, however, is still in its infancy.

Leo-III is addressing this gap. In addition to its role as a classical reasoner, it is the first ATP that natively supports a very wide range of normal HOMLs. To achieve this, Leo-III internally implements the shallow semantical embeddings approach [2, 4]. The key idea in this approach is to provide and exploit faithful mappings for HOML input problems to HOL.

Leo-III supports (but is not limited to) FO and HO extensions of the well known modal logic cube. When taking the different parameter combinations into account (constant/cumulative/varying domain semantics, rigid/non-rigid constants, local/global consequence relation, etc.) this amounts to more than 120 supported HOMLs [4, §2.2]. The exact number of supported logics is in fact much higher, since Leo-III also supports multi-modal logics.

6 Summary and Outlook

Leo-III is a state-of-the-art higher-order reasoning system offering many relevant features and capabilities. Due to its wide range of natively supported classical and non-classical logics, which includes polymorphic HO logic and numerous FO and HO modal logics, the system has many topical applications in computer science, AI, maths and philosophy. Additionally, an evaluation on heterogeneous benchmark sets shows that Leo-III is also one of the most effective HO ATP systems to date [5]. Leo-III complies with existing TPTP/TSTP standards, gives detailed proof certificates and it plays a pivotal role in the ongoing extension of the TPTP library and infrastructure to support modal logic reasoning.

Acknowledgements

The work was supported by the German National Research Foundation (DFG) under grant BE 2501/11-1 (LEO-III).

References

[1] Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In Dov M. Gabbay et al., editors, *Handbook of the History of Logic, Volume 9 — Computational Logic*, pages 215–254. North Holland, Elsevier, 2014.

[2] Christoph Benzmüller and Lawrence Paulson. Multimodal and intuitionistic logics in simple type theory. *The Logic Journal of the IGPL*, 18(6):881–892, 2010.

[3] Christoph Benzmüller, Nik Sultana, Lawrence C. Paulson, and Frank Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[4] Tobias Gleißner, Alexander Steen, and Christoph Benzmüller. Theorem provers for every normal modal logic. In Thomas Eiter and David Sands, editors, *LPAR-21*, volume 46 of *EPiC Series in Computing*, pages 14–30, Maun, Botswana, 2017. EasyChair.

[5] Alexander Steen and Christoph Benzmüller. The higher-order theorem prover Leo-III. CoRR, `https://arxiv.org/abs/1802.02732`, submitted, 2018.

The Elfe Prover

Verifying mathematical proofs of undergraduate students

Maximilian Doré

LMU Munich, m.dore@campus.lmu.de

Abstract: Elfe is an interactive system for teaching basic proof methods in discrete mathematics. The user inputs a mathematical text written in fair English which is converted to a special data-structure of first-order formulas. Certain proof obligations implied by this intermediate representation are checked by automated theorem provers which try to either prove the obligations or find countermodels if an obligation is wrong. Elfe is implemented in Haskell and can be accessed via a reactive web interface or from the command line. Background libraries for sets, relations and functions have been developed.

1 Introduction

Teaching mathematics in university is still a mostly analogous endeavour. Immediate feedback would greatly increase the learning curve – it is often difficult to see when a proof is complete or what steps are missing. Such feedback could be provided by machines. And indeed, many attempts have been made to formalize mathematics. Most prominently, the interactive theorem provers Isabelle and Coq are advanced systems. However, mathematical beginners are overwhelmed by the capabilities of such systems since using them requires a deep understanding of workings of automated theorem provers (ATP).

The goal of this work is to provide users with a system that gives feedback on proofs entered in a fairly natural Mathematical language. Thereby the users are detached from the technicalities of ATPs. Archetype for our tool was the System For Automated Deduction [3], however, we have developed our own input language and internal proof representation.

```
Include functions.

Let A,B,C be set.

Let f: A → B.
Let g: B → C.

Lemma: g∘f is injective implies f is injective.
Proof:
  Assume g∘f is injective.
  Assume x ∈ A and x' ∈ A and (f{x}) = (f{x'}).
  Then ((g∘f){x}) = ((g∘f){x'}).
  Hence x = x'.
  Hence f is injective.
qed.
```

Figure 1: Exemplary Elfe text

Consider the exemplary proof in Figure 1 which is in fact a valid Elfe text. After including a background library and introducing specific sets A, B and C and functions f and g, a lemma is proposed that if the composition of f and g is injective, so the firstly applied f must be injective. This lemma is proven by the reasoning that if f maps two elements x and x' to the same element, the composition of f and g must map them to the same elements. Since this composition is injective, it follows that x and x' are the same elements and f is thus injective. Note that (g∘f){x} denotes the function application of g∘f which is put in brackets to specify the precedence of the symbols.

The Elfe system is implemented in Haskell and can be accessed through a web interface or a command-line interface (CLI) as shown in Figure 2. After the text is entered via one of its interfaces, it will be transformed into a representation in first-order logic, which is introduced in Section 2. Keywords like Then and Hence have special meanings in an Elfe proof and are used to structure a mathematical proof. This structure is captured in an intermediate proof representation which is introduced in Section 3. The internal representation implies certain proof obligations which are checked by ATPs.

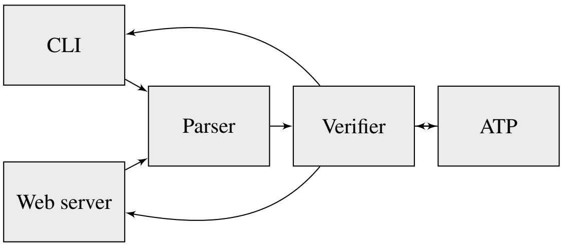


Figure 2: Architecture of the Elfe system

The prover will be presented at CSEDU 2018, a preprint of the conference proceeding can be found in [1]. A full description of the system can be found in [2]. An instance of the system is available online[1], as well as its source code[2].

2 Elfe language

First-order logic is used to encode mathematical statements. Most transformations are straightforward from Elfe to first-order logic, e.g., P implies f is injective is transformed to $P \to injective(f)$. In order to make an Elfe text more legible, additional commands introduce meta-language features: The command Include can be used to include the

[1] https://elfe-prover.org

[2] https://github.com/maxdore/elfe

axioms of a background theory, The command Notation is used to introduce syntactic sugars. The command Let binds a predicate symbol to a variable, effectively assigning a type to a symbol.

3 Statement Sequences

In order to capture the structure of a proof, we propose so-called statement sequences. Intuitively, a statement holds a first-order formula with an identifier and a proof. A proof can consist of other statements in order to represent complex proof objects.

Definition 1. Statement sequences.
A statement S is a tuple ID × GOAL × PROOF where

- ID is an unique alphanumeric string
- GOAL is a formula in first-order logic
- PROOF
  ASSUMED | BYCONTEXT |
  BYSUBCONTEXT $Id_1, ..., Id_n$ |
  BYSEQUENCE $S_1, ..., S_n$ | BYSPLIT $S_1, ..., S_n$

A statement sequence is a finite list of statements $S_1, ..., S_n$.

Consider the example in Figure 3. The statements S_{fun} and S_{inj} are definitions and thus do not to be checked. Their PROOF is therefore ASSUMED. The statement S holds the lemma of our text. In order to verify its validness, we have to construct a more complex proof object. The complete explanation of this can be found in [1]. To give an overview of the other types of PROOF: A proof BYSEQUENCE and BYSPLIT makes it possible to nest more complex derivation sequences. A statement annotated with BYCONTEXT will be checked by the background provers. BYSUBCONTEXT is a special case of this proof type which allows for restricting the context of the statement.

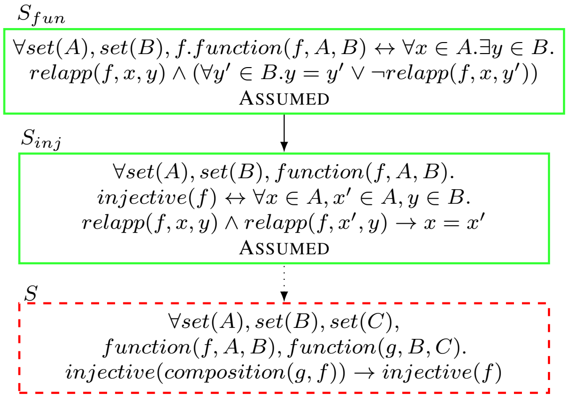

S_{fun}
$\forall set(A), set(B), f.function(f, A, B) \leftrightarrow \forall x \in A.\exists y \in B.$
$relapp(f, x, y) \wedge (\forall y' \in B.y = y' \vee \neg relapp(f, x, y'))$
ASSUMED
S_{inj}
$\forall set(A), set(B), function(f, A, B).$
$injective(f) \leftrightarrow \forall x \in A, x' \in A, y \in B.$
$relapp(f, x, y) \wedge relapp(f, x', y) \rightarrow x = x'$
ASSUMED
S
$\forall set(A), set(B), set(C),$
$function(f, A, B), function(g, B, C).$
$injective(composition(g, f)) \rightarrow injective(f)$

Figure 3: Exemplary statement sequence

3.1 Proved Statements

Since we want to verify that a text is sound, we need to introduce a soundness criteria for statements. Axioms of a text are considered correct, but the lemma needs a more subtle criteria. First we will define which axioms are considered relevant to a statement. Intuitively, the context of a statement in a statement sequence are all statements "above" it.

Definition 2. Context of a statement
Let $S_1, ...S_n$ be a statement sequence. The context of a statement S_k is inductively defined as

- $\Gamma(\text{EMPTY}) = \emptyset$,
- $\Gamma(S_k) = \{S_1.\text{GOAL}, ..., S_{k-1}.\text{GOAL}\}$
  $\cup\, \Gamma(S_k.\text{PARENT})$.

For example, in Figure 3, the context of statement S consists of the respective goals of S_{fun} and S_{inj} (as well as other definitions of the background library which are omitted here). With that, we can define an appropriate soundness criteria for statements.

Definition 3. Proved statement.
Let S be a statement with S.GOAL = ϕ.
We call S proved iff $\Gamma(S) \vDash \phi$.

In other words, a statement is considered proved if it already followed from the theory created by its context. The statements S_{fun} and S_{inj} in Figure 3 are not proved since they build up the axioms of our theory. The statement S however should follow from these axioms, i.e., should be a proved statement. In order to show that S is proved, we will inductively create a more complex proof object such that correctness of the proof object implies that S followed from its context.

That the proof object can be constructed does not necessarily imply the correctness of the text. Instead, the proof object contains proof obligations which need to be checked by ATPs. To give an idea of what a proof obligation can look like: In our proof in Figure 1, we proved injectivity of f by taking two elements of the domain which are mapped to the same element in the codomain, and then showing that these elements must be equal. That this construction indeed implies injectivity is checked by the background provers.

References

[1] M. Doré and K. Broda. The Elfe System - Verifying mathematical proofs of undergraduate students. *ArXiv e-prints*, 2018. 1801.10513.

[2] Maximilian Doré. ELFE – An interactive theorem prover for undergraduate students, 2017. Bachelor thesis.

[3] Konstantin Verchinine, Alexander Lyaletski, and Andrei Paskevich. SYSTEM FOR AUTOMATED DEDUCTION (SAD): a tool for proof verification. In *Proc. CADE-21*, pages 398–403, 2007.

Proving security properties of CHERI-MIPS

Kyndylan Nienhuis Alexandre Joannou Peter Sewell

Computer Laboratory, University of Cambridge
{first.last}@cl.cam.ac.uk

Abstract: CHERI-MIPS is an instruction set architecture that provides hardware support for secure encapsulation and fine-grained memory protection. The guarantees it intends to offer are described in high-level prose, which makes it difficult to understand what they precisely are, whether they are true, and whether they indeed provide memory protection. We describe ongoing work on proposing formal definitions of these guarantees and proving that they are true.

1 THIS VERSION IS FOR METADATA EXTRACTION ONLY

References

[1] CHERI. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/.

[2] The L3 model of CHERI/MIPS. https://github.com/acjf3/l3mips.

[3] A. C. Fox. Directions in ISA specification. In *ITP*, pages 338–344, 2012.

[4] M. Gordon and A. Pitts. The HOL logic and system. In *Real-Time Safety Critical Systems*, volume 2, pages 49–70. Elsevier, 1994.

[5] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2012.

[6] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, A. Joannou, B. Laurie, S. W. Moore, S. J. Murdoch, R. Norton, S. Son, and H. Xia. Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 6). Technical report, University of Cambridge, Computer Laboratory, 2017.

[7] J. Woodruff, R. N. M. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proc. ISCA*, 2014.

Higher-order Reasoning Vampire Style

Ahmed Bhayat Giles Reger

University of Manchester, Manchester, UK

Abstract: Higher-order logic (HOL) is utilised in numerous domains from program verification to the formalisation of mathematics. However, automated reasoning in the higher-order domain lags behind first-order automation. Many higher-order automated provers translate portions of HOL problems to first-order logic (FOL) and pass them to FOL provers. However, FOL provers are not optimised for dealing with these translations. We extend the Vampire automated theorem prover with special inference rules to facilitate efficient reasoning with translated HOL problems. We present these inferences and explore preliminary results on their experimental performance compared to translations using axioms and to an automated HOL prover.

1 Introduction

Most automated theorem provers for higher-order logic (HOL) utilise a first-order (FO) theorem prover to discharge some of the proof burden [2]. However, this can lead to inefficiencies in practice due to FO provers not being optimised for the often unwieldy translations. The goal of this work is to extend the Vampire theorem prover [7] to reason directly with higher-order problems. The general approach is to perform the translation to FOL directly whilst introducing additional rules that are 'translation-aware'. This is based on our previous experience where introducing special inference rules was more effective than relying on introduced axioms. For example, the so-called FOOL paramodulation rule greatly improved Vampire's performance on a superset of many-sorted FOL [5]. Even when the introduction of rules results in the incompleteness, outcomes can be superior to those achieved via a complete axiomatisation [6]. Consider the following classic translation of a HOL problem utilising the venerable Turner combinators [8].

$$\begin{array}{ll} \texttt{axiom:} & add = \lambda xy. + x\ y \\ \texttt{translation:} & add = B + I \\ \texttt{conjecture:} & \forall x. add\ x\ 0 = x \end{array}$$

To reason about this problem a FO prover would either have to be provided with axioms for the combinators or utilise special inference rules. In the first case, the superposition-resolution calculus could combine axioms to form new function definitions. In theory, this process can be hugely explosive as the combinators can combine to form any computable function. For the above example, a set of rewrite rules for the B and I combinators, combined with simple arithmetic reasoning easily leads to a proof. Thus, the use of special inference rules appears to be an elegant solution to combinatorial explosion at the cost of completeness. We do not discuss the syntax and semantics of HOL and FOL and the possible translations from HOL to FOL. Details of these can be found elsewhere [1, 4].

2 Background

Vampire is a saturation-based theorem prover for many-sorted first order logic. Proof search consists of two main parts: (i) translation into clausal form, and (ii) saturation with respect to the resolution and superposition calculus (which is complete for FOL). Proof search is refutational e.g. the conjecture is negated and is established if a contradiction is derived. To enable Vampire to reason about HOL problems, we extend the parser to deal with λ-expressions and application. Translation into clausal form is then extended to translated higher-order features into FOL. This is broadly in line with the approach of Meng and Paulson [8]; Turner combinators are utilised to eliminate λs and a two place *app* function is introduced to avoid partial application. However, note that we are translating to many-sorted FOL rather than unsorted FOL. For each combinator present in the translation, the relevant (sorted) axiom is added. Doing so only for combinators that occur in the translation, leads to a more compact output at the expense of completeness. Further, as logical connectives can be utilised in a curried fashion in HOL, these are translated to FOL constants and relevant axioms are added.

3 Special Inference Rules

An alternative to axiomatising combinators and logical constants is to reason about them natively, similar to how paramodulation avoids the axiomatisation of equality. Three sets of inference rules have been added to Vampire to deal with combinators and HOL constants. Below is an explanation and example of each set.

1. Rules to rewrite connective constants to their FO counterparts when they are fully applied and at the top level e.g. (where vEQ represents logical equivalence):
$$\frac{app(app(vEQ, t_1), t_2) = \$true \vee C}{t_1 = t_2 \vee C}$$

2. Rules to rewrite combinators when they are fully applied e.g. (for the B combinator):
$$\frac{C[app(app(app(B, t_1), t_2), t_3)]}{C[app(t_1, app(t_2, t_3))]}$$

3. Rules to rewrite a fully applied connective constant to true or false (known as short-circuit evaluation) e.g.:
$$\frac{C[app(app(vOR, t_1), t_2)]}{C[\$true]}\ where\ t_1 = \$true\ or\ t_2 = \$true$$

Table 1: Experimental Results on Higher-Order portion of TPTP Library

Solver	Number Proved Unsat or Thm		Unique	Average CPU Time(s)
	Total	in $<$ 30s		
`Satallax 3.2`	2070	1901	593	13.5
`vamp_default`	1229	1222	2	0.9
`vamp_const_off`	1500	1432	190	5.8
`vamp_comb_off`	1308	1264	14	4.1
`vamp_short_circ_elim_off`	1206	1206	0	0.9

4 Preliminary Experimental Results

We evaluate a preliminary implementation of the above translation and inference rules on HOL problems taken from the TPTP library [9]. We select 3077 relevant problems - we focus solely on theorems as our approach is incomplete. We ran Vampire and Satallax 3.2 [3] each for 300 seconds. Vampire is run in portfolio mode extended with new options. We compare four configurations of Vampire: `default` includes all three sets of special inference rules; `const_off` turns off rules for connectives; `comb_off` turns off rules for combinators; `short_circ_elim_off` turns of shortcut elimination rules. Table 1 presents our preliminary results. We report the number of problems that Satallax uniquely solves and for each variant we give the number of problems it can solve uniquely with respect to other variants. Vampire solves 6 problems that Satallax could not. We see that the special inference rule for connectives is significantly less effective than adding the axioms. This can be attributed to the rules being restricted to connectives at the top-level (in this sense it is the 'least complete' of the rules). Even for combinators the special inference rules had a net negative effect, albeit smaller. However, it is interesting to note that the inference rule does solve problems faster (when it solves them at all). These results show that the shortcut elimination rule is always useful.

5 Current and Future Work

The above results demonstrate the trade-off between the completeness of an approach and the effect this has on the efficiency of proof search. It is not (yet) clear that special inference rules cannot be utilised positively in general, but our initial experiments show that a straightforward implementation did not yield the performance boost we had hoped for. One explanation would be that the combinator explosion we are trying to avoid does not occur often in practice for those problems we can solve at all. This may be an effect of our translation being from HOL to many-sorted FOL (not unsorted FOL as in other work). In this case, either polymorphism or an infinite set of combinator axioms is required to be complete even with the axiom translation. Without either of these it is feasible that the search space is restricted enough to not hobble the prover whilst being expansive enough to outperform the non-axiom approach.

Clearly there is still some work to be done to compete with automated HOL provers such as Satallax. So far, we do little to handle what might be called 'true' higher-order reasoning and the gap may reflect the need for such reasoning. In light of this, the challenge is to integrate aspects of higher-order reasoning into Vampire (e.g. limited HO unification) without excessively harming its efficiency.

In the short term, further experimentation is to be carried out on the most effective way to combine axioms and inference rules. In the longer term, the aim is to embed sufficient higher-order reasoning into Vampire to make it effective on a broad range of problems. A proposed first step to this is to embed a λ-calculus within the prover. Other ideas under consideration are using the η-long form of HO terms to remove partial application and thereby avoid the usage of the *app* function and to utilise HO unification pragmatically on 'promising' clauses.

Acknowledgements. The first author would like to thank the family of the late James Elson for facilitating the funding of his PhD. Both authors would like to thank Andrei Voronkov for his help, numerous ideas contained in this abstract have him as their source.

References

[1] Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[2] Christoph Benzmller, Nik Sultana, Lawrence C. Paulson, and Frank Theib. The higher-order prover leo-ii. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[3] Chad E Brown. Satallax: An automatic higher-order prover. In *International Joint Conference on Automated Reasoning*, pages 111–117. Springer, 2012.

[4] Dirk Van. Dalen. *Logic and Structure*. Springer, 2008.

[5] Evgeny Kotelnikov, Laura Kov, and Andrei Voronkov. First class boolean type in first-order theorem proving. page 497701, 2014.

[6] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. *ACM SIGPLAN Notices*, 52(1):260–270, 2017.

[7] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In Natasha Sharygina and Helmut Veith, editors, *CAV 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35, 2013.

[8] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[9] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

Extending the KSP Prover to More Expressive Modal Logics

Fabio Papacchini[1] Cláudia Nalon[2] Ullrich Hustadt[1] Clare Dixon[1]

[1] Department of Computer Science, University of Liverpool, {name.surname}@liverpool.ac.uk
[2] Departament of Computer Science, University of Brasília, nalon@unb.br

Abstract: The abstract discusses how to extend the resolution-based KSP prover for multi-modal logic $\mathbf{K}_n$ to reflexive, serial, symmetric, or transitive logics. Extensions are based on reductions to $\mathbf{K}_n$.

1 Introduction

A modal-layered resolution calculus for the multi-modal logic $\mathbf{K}_n$ is proposed in [3]. The calculus is sound and complete, and its main feature is to reduce the number of required inferences by labelling clauses with the modal level (a notion closely related to modal depth) of their occurrences. This labelling of clauses is performed when computing a normal form called *Separated Normal Form with Modal Levels*, SNF_{ml}. While the original calculus is a decision procedure for both global and local satisfiability, this abstract focuses only on the latter as it is when the main feature of the calculus is utilised fully. Clauses in SNF_{ml} are in one of the following forms.

- Literal clause $ml : \bigvee_{b=1}^{r} l_b$
- Positive a-clause $ml : l' \Rightarrow \boxed{a}\, l$
- Negative a-clause $ml : l' \Rightarrow \langle a \rangle\, l$

where $ml \in \mathbb{N}$ and l, l', l_b are literals. Figure 1 shows the rules of the calculus with σ being a unification of labels defined as $\sigma(\{ml\}) = ml$, and undefined otherwise. The resolution-based prover KSP implements the modal-layered resolution calculus, and [2, 4] show that it often outperforms other state-of-the-art provers.

When extending the calculus to cover more expressive modal logics, two possible approaches come immediately to mind: (1) adding appropriate rules to handle the new features required by the logic under consideration, (2) reducing the logic under consideration to $\mathbf{K}_n$. The former solution is common in many decision procedures for families of modal logics (e.g., [5]). A similar approach is possible also for the modal-layered resolution calculus, but it is non-trivial without requiring the use of global reasoning. This is because information relating newly introduced symbols with clauses occurring at different levels would need to be maintained in order to preserve completeness, a task made easy by allowing global reasoning. As our focus is on local reasoning, this abstract discusses extensions of the calculus by means of reductions of reflexive, serial, symmetric, or transitive extensions of the multi-modal logic $\mathbf{K}_n$ to $\mathbf{K}_n$.

2 Reduction Functions

Relationships between extensions of $\mathbf{K}_n$ to $\mathbf{K}_n$ have already been subject of extensive studies, especially for those extensions where the complexity of the satisfiability problem is known to remain in PSPACE. A thorough study of such relationships is presented in [1] where both global and local reductions are investigated. Complexity preserving reductions for reflexivity, seriality, symmetry, and transitivity are based on the following functions, where $\mathsf{sf}(\varphi)$ represents the set of subformulae of φ, φ is assumed to be in negation normal form (NNF), and a is, respectively, reflexive, serial, symmetric, or transitive.

$$X_{\mathbf{T}}(\varphi) = \{\boxed{a}\varphi' \to \varphi' \mid \boxed{a}\varphi' \in \mathsf{sf}(\varphi)\}$$
$$X_{\mathbf{D}}(\varphi) = \{\langle a \rangle\top \mid \boxed{a}\varphi' \in \mathsf{sf}(\varphi)\}$$
$$X_{\mathbf{B}}(\varphi) = \{\neg\varphi' \to \boxed{a}\,\langle a \rangle\neg\varphi' \mid \boxed{a}\varphi' \in \mathsf{sf}(\varphi)\}$$
$$X_{\mathbf{4}}(\varphi) = \{\boxed{a}\varphi' \to \boxed{a}\,\boxed{a}\varphi' \mid \boxed{a}\varphi' \in \mathsf{sf}(\varphi)\}$$

Those functions, and their composition if the extended language has more than one of the frame properties, are all global reductions to $\mathbf{K}_n$. Local reductions are built upon such functions. The general idea behind local reductions is very simple as they are obtained by repeating the formulae computed by the global reduction functions at all possible levels. This is achieved by repeating each formula by increasing the number of box modalities in front of it as many times as necessary. While such an approach is theoretically sound and it is guaranteed of not increasing the over all complexity, it is not practical. Our aim is to improve the local reductions by allowing the labels in SNF_{ml} to be sets of modal levels.

3 New SNF_{ml} Translation

Without introducing new notation, thereafter we refer to the normalisation presented in this section as SNF_{ml}. The SNF_{ml} of a formula φ is a set S_φ of clauses each of which has an associated set S_{ml} of modal levels with $S_{ml} \subseteq \mathbb{N}$. For presentation purposes clauses in S_φ are represented as $S_{ml} : \varphi'$ where S_{ml} is the set of modal levels associated with φ', and we refer to S_φ as a set of labelled clauses. This implies that if $S_{ml} : \varphi', S'_{ml} : \varphi' \in S_\varphi$, then $S_{ml} = S'_{ml}$. Let S and S' be two sets of labelled clauses, the union of such sets is defined as $S \cup S' = \{S_{ml} \cup S'_{ml} : \varphi \mid S_{ml} : \varphi \in S \text{ and } S'_{ml} : \varphi \in S'\}$ with $S_{ml} = \emptyset$ (resp., $S'_{ml} = \emptyset$) if φ is not a labelled clause in S (resp., S'). For a set S_{ml} of modal levels, we define the two functions $succ(S_{ml}) = \{ml + 1 \mid ml \in S_{ml}\}$, and $pred(S_{ml}) = \{ml - 1 \mid ml \in S_{ml} \text{ and } ml \neq 0\}$. Clauses

$$\text{[LRES]}\quad \begin{array}{rl} ml: & D \vee l \\ ml': & D' \vee \neg l \\ \hline \sigma(\{ml, ml'\}): & D \vee D' \end{array} \qquad \text{[MRES]}\quad \begin{array}{rl} ml: & l_1 \Rightarrow \boxed{a}\, l \\ ml': & l_2 \Rightarrow \Diamond_{a} \neg l \\ \hline \sigma(\{ml, ml'\}): & \neg l_1 \vee \neg l_2 \end{array} \qquad \text{[GEN2]}\quad \begin{array}{rl} ml_1: & l'_1 \Rightarrow \boxed{a}\, l_1 \\ ml_2: & l'_2 \Rightarrow \boxed{a} \neg l_1 \\ ml_3: & l'_3 \Rightarrow \Diamond_{a} l_2 \\ \hline \sigma(\{ml_1, ml_2, ml_3\}): & \neg l'_1 \vee \neg l'_2 \vee \neg l'_3 \end{array}$$

$$\text{[GEN1]}\quad \begin{array}{rl} ml_1: & l'_1 \Rightarrow \boxed{a} \neg l_1 \\ & \vdots \\ ml_m: & l'_m \Rightarrow \boxed{a} \neg l_m \\ ml_{m+1}: & l' \Rightarrow \Diamond_{a} \neg l \\ ml_{m+2}: & l_1 \vee \ldots \vee l_m \vee l \\ \hline ml: & \neg l'_1 \vee \ldots \vee \neg l'_m \vee \neg l' \\ \multicolumn{2}{l}{\text{where } ml = \sigma(\{ml_1, \ldots, ml_{m+1}, ml_{m+2} - 1\})} \end{array} \qquad \text{[GEN3]}\quad \begin{array}{rl} ml_1: & l'_1 \Rightarrow \boxed{a} \neg l_1 \\ & \vdots \\ ml_m: & l'_m \Rightarrow \boxed{a} \neg l_m \\ ml_{m+1}: & l' \Rightarrow \Diamond_{a} l \\ ml_{m+2}: & l_1 \vee \ldots \vee l_m \\ \hline ml: & \neg l'_1 \vee \ldots \vee \neg l'_m \vee \neg l' \\ \multicolumn{2}{l}{\text{where } ml = \sigma(\{ml_1, \ldots, ml_{m+1}, ml_{m+2} - 1\})} \end{array}$$

Figure 1: Rules of the modal-layered resolution calculus.

in SNF_{ml} are in the same forms presented in Section 1, except that the labels are sets $S_{ml} \subseteq \mathbb{N}$.

The transformation of a formula φ into SNF_{ml} is achieved by recursively applying rewriting and renaming. Let φ be a $\mathbf{K}_n$ formula in NNF and t a propositional symbol not occurring in φ, then the translation of φ is given by $\{\{0\} : t\} \cup \rho(\{0\} : t \Rightarrow \varphi)$, where the translation function ρ is defined as follows (with φ and φ' $\mathbf{K}_n$ formulae, and t' a new propositional symbol):

$$\begin{aligned} &\rho(S_{ml} : t \Rightarrow \varphi \wedge \varphi') \\ &\quad = \rho(S_{ml} : t \Rightarrow \varphi) \cup \rho(S_{ml} : t \Rightarrow \varphi') \\ &\rho(S_{ml} : t \Rightarrow \boxed{a}\varphi) \\ &\quad = \{S_{ml} : t \Rightarrow \boxed{a}\varphi\}, \text{ if } \varphi \text{ is a literal} \\ &\quad = \{S_{ml} : t \Rightarrow \boxed{a}t'\} \cup \rho(succ(S_{ml}) : t' \Rightarrow \varphi), \text{ otherwise} \\ &\rho(S_{ml} : t \Rightarrow \Diamond_{a}\varphi) \\ &\quad = \{S_{ml} : t \Rightarrow \Diamond_{a}\varphi\}, \text{ if } \varphi \text{ is a literal} \\ &\quad = \{S_{ml} : t \Rightarrow \Diamond_{a}t'\} \cup \rho(succ(S_{ml}) : t' \Rightarrow \varphi), \text{ otherwise} \\ &\rho(S_{ml} : t \Rightarrow \varphi \vee \varphi') \\ &\quad = \{S_{ml} : \neg t \vee \varphi \vee \varphi'\}, \text{ if } \varphi' \text{ is a disjunction of literals} \\ &\quad = \rho(S_{ml} : t \Rightarrow \varphi \vee t') \cup \rho(S_{ml} : t' \Rightarrow \varphi'), \text{ otherwise} \end{aligned}$$

The resolution calculus using the new SNF_{ml} is as the one in Figure 1 except that (1) labels are sets (e.g., ml is replaced by S_{ml}), (2) σ is defined over a non-empty set S of sets of labels as $\sigma(S) = \bigcap_{S_{ml} \in S} S_{ml}$, (3) inferences are performed only if $\sigma(S) \neq \emptyset$, and (4) occurrences of $ml - 1$ are replaced by $pred(S_{ml})$.

The new SNF_{ml} can be used to compute better local reductions than the one proposed in [1], resulting in an increase in space equivalent to the one of global reductions. The only necessary step is to compute what are the required levels at which formulae resulting from the global reduction functions need to hold. Let us consider the case of reflexive frames. Let φ be a $\mathbf{KT}_n$ formula in NNF and $X_{\mathbf{T}}(\varphi)$ defined as above. Formulae in $X_{\mathbf{T}}(\varphi)$ are not required to hold at all modal levels in order to reduce $\mathbf{KT}_n$ to $\mathbf{K}_n$, but only at the levels where each $\boxed{a}\varphi' \in \mathsf{sf}(\varphi)$ appears. Let $S_{\mathbf{T}}$ be a set of labelled clauses $S_\psi : \psi$ for any $\psi \in X_{\mathbf{T}}(\varphi)$ associated with some $\boxed{a}\varphi' \in \mathsf{sf}(\varphi)$, where S_ψ is the set of modal levels of $\boxed{a}\varphi'$ in φ. The SNF_{ml} of φ is given by $\{\{0\} : t\} \cup \rho(\{0\} : t \Rightarrow \varphi) \cup \bigcup_{S_\psi : \psi \in S_{\mathbf{T}}} (\rho(S_\psi : t_\psi) \cup \rho(S_\psi : t_\psi \Rightarrow \psi))$.

The computation of the set of levels for serial frames is analogous to the one of reflexive frames; for symmetric frames the set of levels is composed of the levels preceding those where $\boxed{a}\varphi' \in \mathsf{sf}(\varphi)$ appears; and for transitive frames the set of levels is composed of all the levels greater than or equal to the smallest level of $\boxed{a}\varphi'$ in φ.

4 Conclusion

The extensions of the KSP prover presented in this abstract are a work in progress, and we do not know yet what is the impact on the prover's performance of moving from single levels as labels to sets of levels. We believe, however, that the new transformation can have a positive impact for two reasons. First, there is no repetition of clauses at different levels, which can result in a reduction of the number of labelled clauses. Second, while with the old transformation the same inference could be performed more than once at different levels, the use of sets and the new unification function allows us to perform it only once. It is part of our future plans to implement and test the proposed reductions.

References

[1] M. Kracht. Notes on the space requirements for checking satisfiability in modal logics. In *AiML'02*, pages 243–264, 2002.

[2] C. Nalon, U. Hustadt, and C. Dixon. KSP: A resolution-based prover for multimodal K. In *IJCAR'16*, volume 9706 of *LNCS*, pages 406–415, 2016.

[3] C. Nalon, U. Hustadt, and C. Dixon. A modal-layered resolution calculus for K. In *TABLEAUX'15*, volume 9323 of *LNCS*, pages 406–415, 2016.

[4] C. Nalon, U. Hustadt, and C. Dixon. KSP: A resolution-based prover for multimodal K, abridged report. In *IJCAI'17*, pages 4919–4923, 2017.

[5] C. Nalon, J. Marcos, and C. Dixon. Clausal resolution for modal logics of confluence. In *IJCAR'14*, volume 8562 of *LNCS*, pages 322–336, 2014.

Detailed Models of Instruction Set Architectures: From Pseudocode to Formal Semantics

Alasdair Armstrong[1] Thomas Bauereiss[1] Brian Campbell[3] Shaked Flur[1]
Kathryn E. Gray[2] Prashanth Mundkur[5] Robert M. Norton[1] Christopher Pulte[1]
Alastair Reid[3] Peter Sewell[1] Ian Stark[3] Mark Wassell[1]

[1] University of Cambridge firstname.lastname@cl.cam.ac.uk
[2] University of Cambridge (while this work was done) kathy_gray@gmx.com
[3] University of Edinburgh firstname.lastname@ed.ac.uk
[4] ARM Ltd, Cambridge, alastair.reid@arm.com
[5] SRI International, Menlo Park, prashanth.mundkur@sri.com

Abstract: Processor instruction set architectures (ISAs) are typically specified using a mixture of prose and pseudocode. We present ongoing work on expressing such specifications rigorously and automatically translating them to interactive theorem prover definitions, making them amenable to mechanised proof. Our ISA descriptions are written in Sail—a custom ISA specification language designed to support idioms from various processor vendor's pseudocode, with lightweight dependent typing for bitvectors, targeting a variety of use cases including sequential and concurrent ISA semantics. From Sail we aim to portably generate usable theorem prover definitions for multiple provers, including Isabelle, HOL4, and Coq. We are focusing on the full ARMv8.3-A specification, CHERI-MIPS, and RISC-V, together with fragments of IBM POWER and x86.

1 Introduction

Instruction Set Architectures are extremely complex, with specifications in manuals containing thousands of pages. In the last decade, there has been significant progress in making ISA specifications amenable to formal reasoning, including a model of a substantial fragment of the ARM ISA, hand-written by Fox in his L3 language [2] and used for formal verification of seL4 [5] and CakeML [9], and the x86 model of Goel et al. [3].

A notable recent industry effort is ARM's public release of its full ARM v8-A specification in machine-readable form, in their internal ASL language [8]. This vendor-provided ISA specification is attractive because it is significantly more detailed, complete, and authoritative than existing models.

To enable theorem proving using this model, ASL has to be translated to the prover of choice. We present a translation to multiple provers, currently Isabelle/HOL and HOL4, via our Sail ISA specification language [4]. Sail aims to support many different uses, including connecting ISA semantics to analysis and exploration tools for relaxed memory models [7]. In ongoing work, we have recently improved several aspects of Sail such as the type system, the generation of efficient emulator code, and the generation of portable theorem prover definitions. We are focusing on the full ARMv8.3-A specification generated from ASL, and are also using Sail for MIPS, CHERI-MIPS, RISC-V, parts of IBM POWER and x86, and a simplified ARM fragment.

2 Structure of an ISA specification in Sail

Sail aims to provide a engineer-friendly, vendor-pseudocode-like language for describing instruction semantics. It is a straightforward imperative language with dependent typing for numeric types and bitvector lengths, checked using Z3, so that ubiquitous bitvector manipulations in ISA specifications can be checked for length and bounds errors. These lengths can be dynamically computed, as in the following example from ARMv8-A:

```
val FPZero : forall 'n , 'n in {16, 32, 64}.
  bits(1) → bits('n)

function FPZero sign = {
  let exponent as 'e =
    (if 'n == 16 then 5 else if 'n == 32 then 8
     else 11) : {|5, 8, 11|};
  F = 'n - exponent - 1;
  exp = Zeros(exponent);
  frac = Zeros(F);
  return sign @ exp @ frac
}
```

This returns either a 16, 32, or 64-bit floating point value, depending on the calling context. The exponent length is dynamically derived from the length of the return bitvector, and given a type variable *'e* for its size. We then create bitvectors involving *'e* and the return length *'n*, and the typechecker can check that they all are of the required length.

A key aim of this typing information is to generate prover code that does not force the user to constantly prove side conditions involving bitvector indexing: for non-dependently-typed provers, we can use our type information to monomorphise definitions as needed.

3 ARM v8.3-A in Sail

We have a complete translation of all the 64-bit instructions in ARM's publicly available v8.3-A specification [8]. ARM's specification is written in their own ASL specification language, and we have developed a tool for converting ASL specifications into Sail automatically. Hand-written specifications tend to focus on small subsets of the architecture, while the ASL-derived Sail specification includes many aspects which are often omitted, such as floating-point support, vector extensions, and system and hypervisor modes. ASL has been used extensively for testing within ARM, giving us confidence that we are accurately modelling the full behaviour allowed by the architecture. Work on validating our translation remains ongoing.

The Sail ARM v8.3-A specification is about 30 000 lines. Despite the ASL specification itself being public, much of the tooling required to easily make use of it is not. By converting it into Sail, we aim to provide open-source tooling for working with the actual v8.3-A specification. A single instruction can often call several hundred auxiliary helper functions, so reasoning about this specification in an interactive theorem prover will be challenging, and a great deal of automation will be needed.

4 Generating Theorem Prover Definitions

We generate theorem prover definitions by first translating Sail specifications to Lem [6], which then provides translations to Isabelle/HOL and HOL4. In principle, Lem also supports translation to Coq, but a direct translation from Sail is likely to produce more idiomatic Coq definitions, allowing us to preserve Sail's dependent types for bitvector lengths. For the translation to Lem, turning these dependent types into the simpler constant-or-parameter form allowed by Lem and theorem provers such as Isabelle/HOL is one of the more intensive transformations we perform. In Lem our example becomes:

```
val FPZero : forall ’N . Size ’N =>
  integer → mword ty1 → M (mword ’N)

let FPZero (N : integer) sign =
  if (eq N 16) then
    let (F : integer) = 16 - 5 - 1 in
    let (exp : bits ty5) =
      Zeros (mk_itself 5 : itself ty5) in
    let (frac : bits ty10) =
      Zeros (mk_itself F : itself ty10) in
    return (bitvector_cast (concat
          (concat sign exp : mword ty6)
          frac) : mword ’N)
  else if (eq N 32) ...
  else fail "FPZero:␣constraints␣unsatisfied"
```

An extra argument N has been added corresponding to *'n*, and an automated dependency analysis has detected that it needs to be a concrete value, generating a case split. Constant propagation fills in concrete values for lengths. Type-level information about lengths has been passed to `Zeros` by changing integer arguments into the singleton `itself` type, giving a function compatible with Lem's type system. While this transformation of dependent types would not be necessary for Coq, a Coq backend would share many of the other parts of the translation pipeline with Lem, such as the translation of imperative code into monadic expressions.

In addition to targeting different provers, we aim to support different use cases. For reasoning in a purely sequential setting, a state monad can be used. In a concurrent setting, we need to be more fine-grained. Modern processors typically execute many instructions simultaneously, reordering their memory and register accesses for increased performance. We support this by using a free monad of an effect datatype. A monadic expression evaluates either to a pure value or to an effect and a continuation (or an exception without continuation). This gives us the fine-grained effect information needed to reason about multiple instructions concurrently; the monad is suitable as an interface to connect the ISA semantics with a relaxed memory model. We recover a purely sequential model using a lifting to the state monad. Isabelle automation for this lifting is provided by simplification rules relating the primitive operations of the monads, allowing us to seamlessly reason about the sequential behaviour of instructions, e.g. using a Hoare logic.

5 Conclusion

We plan to continue improving both Sail, e.g. by adding a Coq backend, and the ISA models. The tool and models are available online [1] under an open-source license. We plan to put the models to actual use in theorem provers, and invite other projects to consider using them as well.

Acknowledgements This work was partially supported by EPSRC grant EP/K008528/1 (REMS), an ARM iCASE award, EPSRC IAA KTF funding, and the CIFV project supported by the United States Air Force under contract FA8750-18-C-7809.

References

[1] The Sail ISA semantics specification language, 2018. `http://www.cl.cam.ac.uk/~pes20/sail/`.

[2] A. C. J. Fox. Directions in ISA specification. In *ITP*, 2012.

[3] S. Goel. The x86isa books: Features, usage, and future plans. In *Proc. 14th ACL2 Workshop*, 2017.

[4] K. E. Gray, G. Kerneis, D. P. Mulligan, C. Pulte, S. Sarkar, and P. Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *MICRO*, pages 635–646, Dec. 2015.

[5] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, 2014.

[6] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: Reusable engineering of real-world semantics. In *ICFP*, pages 175–188. ACM, Sept. 2014.

[7] C. Pulte, S. Flur, W. Deacon, J. French, S. Sarkar, and P. Sewell. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. In *POPL*, Jan. 2018.

[8] A. Reid. Trustworthy specifications of ARM v8-A and v8-M system level architecture. In *FMCAD 2016*, pages 161–168, October 2016.

[9] Y. K. Tan, M. O. Myreen, R. Kumar, A. Fox, S. Owens, and M. Norrish. A new verified compiler backend for CakeML. In *ICFP*, pages 60–73, 2016.

Verifying Strong Eventual Consistency for Conflict-free Replicated Data Types

Victor B. F. Gomes[1] Martin Kleppmann[1] Dominic P. Mulligan[2]
Alastair R. Beresford[1]

[1] Computer Laboratory, University of Cambridge, UK,
{victor.gomes, martin.kleppmann, alastair.beresford}@cl.cam.ac.uk
[2] Security Research Group, Arm Research, Cambridge, UK, dominic.mulligan@arm.com

Abstract: We develop a modular framework in the Isabelle/HOL interactive proof assistant for verifying the correctness of Conflict-free Replicated Data Types (CRDTs), a class of algorithm that provides strong eventual consistency guarantees for replicated data. We avoid correctness issues that have dogged previous mechanised proofs in this area by including a network model in our formalisation, and proving that our theorems hold in all possible network behaviours. Our axiomatic network model is a standard abstraction that accurately reflects the behaviour of real-world computer networks. Moreover, we identify an abstract convergence theorem, a property of order relations, which provides a formal definition of strong eventual consistency.

1 Introduction

A data replication algorithm is executed by a set of nodes in a distributed system, and ensures that all nodes eventually obtain an identical copy of some shared state. Implementing a replication algorithm is a challenging task, as any such algorithm must operate across computer networks that may arbitrarily delay, drop, or reorder messages, experience temporary partitions of the nodes, or even suffer node failure. A number of these replication algorithms exist, each exploring different trade-offs between the strength of data consistency guarantees, scalability and performance. They can be divided into three classes based on the consistency guarantees that they provide: strong consistency, eventual consistency, and strong eventual consistency.

Strong eventual consistency (SEC) is a model that strikes a compromise between strong and eventual consistency [5]. Informally, it guarantees that whenever two nodes have received the same set of updates, possibly in a different order, their view of the shared state is identical, and any conflicting updates are merged automatically. It works in arbitrary network topologies without assuming a central server and is robust to unreliable networks and server failures. However, it suffers from complicated and subtle algorithms. Several of them, published in peer-reviewed venues, were subsequently shown to violate their supposed guarantees. Informal reasoning has repeatedly produced plausible-looking but incorrect algorithms, and there have even been examples of mechanised formal proofs of correctness later being shown to be flawed [4] due to false assumptions about the execution environment.

In this work we use the Isabelle/HOL proof assistant to create a framework for reliably reasoning about the correctness of a particular class of decentralised replication algorithms with SEC, operation-based Conflict-free Replicated Data Types (CRDT). We do this by formalising not only the replication algorithms, but also the network in which they execute, allowing us to prove that the algorithms assumptions hold in all possible network behaviours. We model the network using the axioms of asynchronous unreliable causal broadcast, a well-understood abstraction that is commonly implemented by network protocols. By keeping modules abstract and implementation-independent, we construct a reusable library of specifications and theorems.

We use the framework to provide the first mechanised proof of correctness for the Replicated Growable Array (RGA), the operation-based Observed-Remove Set (ORSet), and a counter datatype. RGA is an especially subtle algorithm, which makes its formal verification of particular interest: "... the reason why RGA actually works has been a bit of a mystery" [1]. Whilst the ORSet is supported as a primitive by the Lasp [3] language for synchronisation-free programming, with an implemention also exported by the Akka framework.

2 Abstract Convergence Theorem

SEC requires *convergence* of all copies of the shared state: whenever two nodes have received the same set of updates, they must be in the same state. To achieve convergence, operation-based CRDTs algorithms usually require that *concurrent* operations commute with each other. Two operations are concurrent if neither *knew about* the other at the time when they were generated. If one operation happened before another then it is reasonable to assume that all nodes will apply the operations in that order.

More abstractly, assume a strict partial order of operations $(\mathcal{O}, \prec)$ and an interpretation function that takes any operation $a \in \mathcal{O}$ to a state transformer $[\![a]\!] : \Sigma \rightarrow \Sigma$. When $a \prec b$ we say that operation a *happens-before* operation b. We write $a \parallel b$ when the operations are incomparable and say that they are concurrent. We lift the interpretation function to a list of operations by composing their interpretations $[\![a_1 a_2 \ldots a_n]\!] = [\![a_1]\!] \circ [\![a_2]\!] \circ \ldots \circ [\![a_n]\!]$. One can further extend these definitions to fallible operations by using an option monad and the Kleisli arrow composition.

Definition 1 *A list ω is said to be* consistent *if ω is the empty list or $\omega = \mu b$ for some operation b and list μ where $\neg\, b \prec a$ for all a in μ.*

As a result, whenever two operations x and y appear in a consistent list, and $x \prec y$, then x must appear before y in the list. However, if $x \parallel y$, the operations can appear in the list in either order.

Definition 2 *A list ω is said to respect commutativity for concurrent operations if for all a and b in ω, if $a \parallel b$ then $[\![a]\!] \circ [\![b]\!] = [\![b]\!] \circ [\![a]\!]$.*

Theorem 1 *If two consistent lists ω and μ have the same set of operations and respect commutativity for concurrent operations then $[\![\omega]\!] = [\![\mu]\!]$.*

Although this theorem may seem *obvious* at first glance—commutativity allows the operation order to be permuted—it is more subtle than it seems. The difficulty arises because operations may succeed when applied to some state, but fail when applied to another state.

3 Axiomatic Asynchronous Network Model

We model a distributed system as an unbounded number of communicating nodes. Our only assumption about the communication pattern of nodes is that each node can be uniquely identified and that the flow of execution at each node consists of a finite totally ordered sequence of execution steps (events). Intuitively, a node can be regarded as a deterministic state machine where each state transition corresponds to a *broadcast* or *deliver* event. We make no assumptions about the reliability or the ordering of messages. If one node broadcasts a message, it may be delivered by other nodes, but we do not state if or when that will happen. Messages may be arbitrarily delayed, reordered, or even lost entirely. It is even acceptable for a node to never deliver any messages besides those it broadcasts itself, modelling a node that is permanently disconnected from the network.

In this setting, one can identify a *happens-before* relation, which captures the causal dependencies between operations. It can be defined in terms of sending and receiving messages on a network. Using vector clocks, this relation can be forced to form a strict partial order.

Theorem 2 *Every list of operations produced by an asynchronous casual network is consistent.*

4 Correctness of CRDTs

The convergence proofs for all of our CRDT implementations follow the same structure. First we define the type of local state at each node, and the types of operations that may be invoked to modify the state. When one node invokes an operation, it is broadcast to other nodes using our network model. An interpretation function is called whenever a message containing an operation is delivered to a node, and it transforms the node's local state to incorporate the operation. To prove convergence, we must show that, subject to certain assumptions, operations commute with each other. Next, we must prove that those assumptions are always satisfied by any concurrent operations in the network. This guarantees that every consistent list (due to Theorem 2) respects commutativity for concurrent operations. When these proof obligations have been met, we obtain convergence for the replicated datatype by Theorem 1. To obtain SEC, a final property needs to be proved for each CRDT: *progress*, i.e. if one node broadcasts a valid operation, and another node applies that operation, then it must not become stuck in an error state.

5 Conclusion

Theorem 1 is independent of any particular network model or replication algorithm. Together with progress theorems, we assert that it constitutes a general but precise definition of strong eventual consistency. As shown by our three CRDT implementation examples, we have not only isolated reusable lemmas and models of networks, but also a proof strategy that algorithm designers can use to obtain a convergence theorem for their operation-based CRDT. We further speculate that our framework is also amenable to formalising other classes of SEC algorithms: Operational Transformation algorithms and state-based CRDTs.

Our Isabelle theory files [2] are open source and included in the Archive of Formal Proofs[1], enabling others to build upon our proof framework.

Acknowledgements

The authors wish to acknowledge the support of the EPSRC "REMS: Rigorous Engineering for Mainstream Systems" grant (EP/K008528), the EPSRC "Interdisciplinary Centre for Finding, Understanding and Countering Crime in the Cloud" grant (EP/M020320), and The Boeing Company.

References

[1] H. Attiya, S. Burckhardt, A. Gotsman, A. Morrison, H. Yang, and M. Zawirski. Specification and complexity of collaborative text editing. In *PODC*, 2016.

[2] V. B. F. Gomes, M. Kleppmann, D. P. Mulligan, and A. R. Beresford. Verifying strong eventual consistency in distributed systems. *PACMPL*, 1(OOPSLA), 2017.

[3] C. Meiklejohn and P. Van Roy. Lasp: a language for distributed, coordination-free programming. In *PPDP*, 2015.

[4] G. Oster, P. Urso, P. Molli, and A. Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, 2005.

[5] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[1] `http://isa-afp.org/entries/CRDT.shtml`

Formalisation of MiniSail in the Isabelle Theorem Prover

Alasdair Armstrong Neel Krishnaswami Peter Sewell Mark Wassell

University of Cambridge {firstname.lastname}@cl.cam.ac.uk

Abstract: Sail is a language used to model instruction set architectures. It has an imperative syntax and a dependent type system. We formalise a core calculus of the language in the Isabelle theorem prover describing the language syntax, substitution, the type system and operational semantics. A number of classic theorems such as preservation and progress are then proved. The purpose of this formalisation is to ensure that the full language is built on sound foundations and to provide a platform for the generation of the implementation of a type checker and evaluator for the language.

1 Introduction

Sail [1, 2] is a language used to model instruction set architectures (ISAs) for CPUs such as ARM, IBM POWER, MIPS, CHERI, RISC-V, and x86. It is an imperative language similar to the vendor pseudocode languages; the semantics of instructions is expressed as imperative code that makes register and memory accesses. Academic ISA models are often for small fragments, but full ISAs typically contain hundreds or thousands of instructions, so Sail models will be large and complex. In order to tame this complexity, a light-weight dependent type system is used. The type system provides integer, boolean, bit vector, register, record and union types. Type level constraints can be specified that constrain the values for integer indexed types such as integers or bit vectors. Function constraints can be used to relate the return value of a functions to values of the function's parameters. To ensure tractablility, constraints are limited to those that are solvable by an external SMT solver, Z3.

It is important to ensure that the Sail language and type system is itself sound and so formalising the language is of benefit. MiniSail is a small subset of Sail intended to capture key aspects of the language making it amenable to formalisation. This paper describes the work in progress to formalise MiniSail in Isabelle.

2 Syntax, Wellformedness and Substitution

Figure 1 shows the grammar of MiniSail. We use let normal form so that complex expressions are unpacked into nested `let` statements. This exposes the types of the subexpressions of complex terms.

The nonterminal τ represents a constrained type (also known as a refinement or liquid type [3]), z ranges over values, b over base types and ϕ over constraints. For example, the type $\{z : int | 0 \leq z \wedge z \leq 32\}$ is the type of integers between 0 and 32 inclusive.

The language grammar is mapped directly into nominal datatypes in Isabelle with binding specifications for the `let` and `case` statements, function definition and liquid types. A context, Γ is an ordered list of (x, b, ϕ) tuples. Program variables (i.e. those introduced in let and function bindings) can be used in types. A set of inductive predicates defines wellformedness with respect to a context ensuring that variables appear in a context before they can be used.

$$\begin{array}{lll}
\text{value, } v & ::= & x \mid n \mid \textbf{T} \mid \textbf{F} \mid \textbf{inl}\ v \mid \textbf{inr}\ v \\
\text{expr, } e & ::= & v \mid v + v \mid v \leq v \mid f\ v \\
\text{stmt, } s & ::= & \textbf{let}\ x = e\ \textbf{in}\ s \mid \\
 & & \textbf{if}\ v\ \textbf{then}\ s\ \textbf{else}\ s \mid \\
 & & \textbf{case}\ v\ \textbf{of inl}\ x_1 \rightarrow s \mid \textbf{inr}\ x_2 \rightarrow s \mid \\
\text{fundef, } fd & ::= & \textbf{fun}\ f\ (x : b[\phi]\ ) : \tau = s \\
\text{prog, } p & ::= & fd_1\ ; .. ; fd_n\ ; s \\
 & & \\
\text{base, } b & ::= & \textbf{int} \mid \textbf{bool} \mid b + b \\
\phi & ::= & \textbf{T} \mid \textbf{F} \mid e = e \mid e \leq e \mid \phi \wedge \phi \mid \\
 & & \phi \vee \phi \mid \neg\phi \\
\tau & ::= & \{z : b \mid \phi\ \}
\end{array}$$

Figure 1: MiniSail Grammar

3 Validity, Subtyping and Typing

SMT solver logic is modelled using an inductive predicate where we define an inductive rule for each property that we expect the solver to have. For example, that $\Gamma \models \phi \implies \phi$ and basic facts about $+$ and $\leq$ operators. The subtyping judgement, $\Gamma \vdash \tau_1 \leq \tau_2$, is key and allows a smaller type to be used where a larger type is expected. Subtyping holds when the base types match and the constraint of the smaller type implies the constaint of the larger; the latter being checked by the SMT solver logic.

The type system of MiniSail is defined using bidirectional typing: we have the type synthesis judgement $\Gamma \vdash e \Rightarrow \tau$ and the type checking judgement $\Gamma \vdash s \Leftarrow \tau$. We define a type checking nominal inductive predicate for statements and both synthesis and checking nominal inductive predicates for values and expressions. Sum types are an interesting case: the type of a sum value cannot be inferred unless there is a type annotation. For example, with $inl\ v$ we can infer the value of the left side of the sum type from v, but we have no information that will give us the right side of the sum type. So we need a type checking judgement for values and expressions and provide a place in the syntax where the programmer can include a type annotation. The usual solution is to include in the grammar

$$\frac{\begin{array}{c}\Gamma \vdash v_1 \Rightarrow \{z_1 : \mathit{int}|\phi_1\}\\ \Gamma \vdash v_2 \Rightarrow \{z_2 : \mathit{int}|\phi_2\}\end{array}}{\Gamma \vdash v_1 + v_2 \Rightarrow \{z_3 : \mathit{int}|z_3 = v_1 + v_2\}}$$

$$\frac{\begin{array}{c}f : (z_1 : b[\phi_1]) : \tau\\ \Gamma \vdash v \Rightarrow \{z_2 : b|\phi_2\}\\ \Gamma \models \phi_2[z_1 ::= v] \Longrightarrow \phi_1[z_1 ::= v]\end{array}}{\Gamma \vdash f\ v \Rightarrow \tau[z_1 ::= v]}$$

Figure 2: Typing Rules

general type annotations on values and expressions. We instead introduce annotations only where it is required - in `let` and `case` statements. We need an hereditary substitution operation for the operational semantics that picks up the type of a term and drops it into the type annotation of the `let` or `case` statement. A small sample of the typing rules is given in Figure 2.

4 Substitution Lemmas and Operational Semantics

With the type system in place, we are in the process of proving in Isabelle a set of lemmas relating the type of term and the type of that term with a value substituted in. A simplified example is: If $\Gamma \vdash v \Rightarrow \{z : b|\phi\}$ and $(x, b, \phi)\#\Gamma \vdash s \Leftarrow \tau$ then $\Gamma \vdash s[x ::= v] \Leftarrow \tau$.

Single step reduction is defined by an inductive predicate. Next, we will prove that if a statement has a type, then the result of reduction has the same type. This is proved using the substitution lemmas. We will also prove the progress lemma: a well typed statement is either a value or has a reduction step.

5 Experience

This work has guided Sail development leading to the simplification of the handling of constraints and removal of unification on numeric expressions in types.

Paper formalisations of languages have an underlying convention that terms are worked with up to alpha-equivalence and that, if necessary, renaming of bound variables can occur implicitly. Mechanical formalisations in a theorem prover need to make this convention explicit to the prover. Nominal Isabelle provides the framework for making this easier than encoding the convention explicitly. This makes a nominal formalisation closer to a paper formalisation than a conventional mechanical formalisation however some supporting lemmas are needed and, when defining functions that operate over nominal datatypes, a number of proofs need to be provided. These proofs are sometimes tricky to prove and result in a lot of proof code that looks to to be the same modulo the structure of the binding. However with an intuitive understanding of how Nominal Isabelle works, it is relatively easy to see the approach needed and how to prove lemmas. Looking at prior work and borrowing lemmas has been helpful and sledgehammer as usual is a great assistant. As equality between nominal terms is alpha-equivalence, care is needed when unpacking a term with a binder. For example, if we have $\{z : b|\phi\} = \{z' : b'|\phi'\}$ then it doesn't hold that $\phi = \phi'$ but it is true that swapping any fresh variable with z in ϕ and z' in ϕ' does give equality.

6 Questions and Further Work

Alongside this work, we have developed AST datatypes and inductive rules for typing and reduction that do not use the Nominal package and do not include proofs. From this, the code generation facilities of Isabelle have been used to build an implementation of this language for a larger subset of Sail. The question arises as to whether it is possible to generate code from nominal datatypes, functions and inductive relations. If this is not possible, then one approach is to hand craft an implementation as 'vanilla' functions in Isabelle and then prove this implementation matches the nominal-formalisation.

Ott [4] provides a unified way of specifying the syntax and semantics of a language. A specification can then be exported to LaTex, Ocaml, Coq and Isabelle. With Isabelle, the output is 'vanilla' datatypes for the AST, functions for substitition and free-variables, and inductive predicates for the rules. An extension to Ott would be for the export to target Nominal Isabelle where the datatypes would be nominal ones and the substitition functions defined in the nominal style. Furthermore, supporting lemmas could be automatically generated and this will also reduce the boilerplate.

Acknowledgements This work was partially supported by EPSRC grant EP/K008528/1 (REMS).

References

[1] Sail. `http://www.cl.cam.ac.uk/~pes20/sail/`.

[2] Kathryn E. Gray, Gabriel Kerneis, Dominic Mulligan, Christopher Pulte, Susmit Sarkar, and Peter Sewell. An integrated concurrency and core-ISA architectural envelope definition, and test oracle, for IBM POWER multiprocessors. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 635–646, New York, NY, USA, 2015. ACM.

[3] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. *SIGPLAN Not.*, 43(6):159–169, June 2008.

[4] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 1–12, New York, NY, USA, 2007. ACM.

Real-world formal documentation

Thomas Tuerk

Independent Scholar, Albert-Otto-Str. 8, 65611 Brechen, Germany
thomas@tuerk-brechen.de

Abstract: In recent years there have been tremendous improvements in interactive theorem proving. Nevertheless, it is hardly used during development of even critical, well funded software projects. One reason is the intrinsic difficulty of formal proofs. However, with advancements in automation and user interfaces this reason becomes less and less important. In my opinion, nowadays preconceptions are a more severe hindrance. As soon as terms like *logic*, *proof* or *formal specification* are used, even very clever, highly skilled software engineers tend to think of some kind of *black magic*: way to complicated for mere mortals and while huge gains are luring, you need to sell your soul to get them. Another reason why interactive theorem proving is not commonly used is – in my experience – that often a large initial investment is needed and progress and benefits are hard to measure.

I believe formal methods and in particular interactive theorem proving are vital to deal with the ever increasing complexity of hard- and software. However, before they get more widely used, the issues discussed above need addressing in my opinion. Therefore I started developing a tool called *Advanced Documentation and Testing Tool* (ADATT). Superficially it looks like yet another functional style programming language accompanied with a compiler and other development tools. Specifications written with ADATT can be exported to common theorem provers for reasoning. Moreover they can be exported to common programming languages for execution. In contrast to similar tools like *Lem* it is also possible to write *partial* specifications. Users can start with completely informal documentation in natural language, which ADATT can use to produce production quality documents. Step by step more formal content can be added. To support common development workflows, there should be an immediate return of invested effort and progress should be easily measurable. A special focus is on using partial specifications for advanced testing.

In this abstract, I will present the ideas behind ADATT. The development is still in a very early stage. There is no working prototype yet. However, I hope by presenting the ideas at an early stage it is possible to start discussions and perhaps find collaborators.

1 Motivation

I'm an interactive theorem prover guy. I have been working with HOL 4, Isabelle/HOL and Coq. I mainly used them to improve trust in real-world systems. Thereby I always followed a rather pragmatic approach. My goal was to find bugs and increase the trustworthiness of systems. Especially when reasoning about real-world systems, there is a non-trivial gap between the real system and the model of the system one can reason about. To gain trust into the real system, one needs to show that the model somehow corresponds to the real system (e. g. via careful code review or conformance testing) and additionally show some interesting properties of the model.

I have been working on this kind of verification projects both in academia and industry. Most recently, I worked in industry and tried to harden a microhypervisor using the Coq proof assistant [1]. In my experience, most problems are found while building a formal model of the computer system. This is due to the fact that building a formal model usually involves a detailed review of the existing system and requires the clarification of ambiguous and missing information. In addition, many bugs are found while proving simple properties about isolated parts the model, as this reveals simple implementation mistakes. Proving deep properties tends to reveal comparably few bugs. These are usually bugs in the design of the whole system.

This means that building a formal model is a very worthwhile activity in itself for hunting bugs. However, once you have a formal model – especially if it is executable – it can be easily used for powerful testing, documentation and automated formal methods. Even better, I believe that no formal methods experts are needed to develop basic formal models. If you present formal specifications as high level programs, domain experts are in my experience willing to read and even write formal specifications. This is especially true, if writing such a formal specification has an immediate benefit.

If the development of formal models can partly be done by domain experts while developing and testing a system, the costs for using interactive theorem proving can be lowered. The communication between domain and formal method experts is simplified and the (partial) formal model is a very good basis for formal method experts to extend and reason about. For enabling this vision of having domain experts develop (partial) formal models, good tool support is essential.

2 Lem

The best tool I know for the purposes stated above is Lem [2]. *"Lem is a lightweight tool for writing, managing,* *and publishing large scale semantic definitions"*[1]. It has the look and feel of a functional programming language. Large subsets of Lem specifications can be translated to OCaml as well as definitions for HOL 4, Isabelle/HOL and Coq.

Even before working on Lem, I was fascinated on how much effect the form of presentation of formal methods can have. Programmers are trained to write down precise definitions (that's what a computer program is after all). However, if you ask them to write down a *specification*, the average programmer refuses. It was a revelation to me to see how VeriFast manages to get programmers to specify loops by disguising loop-invariants as programs. This insight grew deeper, while working for Peter Sewell on Lem.

I'm a strong believer in the ideas of Lem. It is vital to bring domain experts and formal verification experts closer together. Moreover, providing an environment that looks like a programming language and supports the normal tools of a programming language is a good choice. It is important to be able to produce human-readable output, executable code and formal specifications from the same input. Even using a functional instead of an imperative language is (while not familiar to many domain experts) a very sensible compromise, since it is comparably straightforward to translate to logic. However Lem does not go far enough in my opinion. Lem is a good tool, if you want to write a complete executable formal specification. It is, however, not suitable to write partial specifications or informal documentation. Moreover, Lem has limited capabilities for testing and measuring progress.

3 ADATT

For these reasons I started developing a tool called *Advanced Documentation and Testing Tool* (ADATT). It is inspired by Lem, but has a different focus. Similar to Lem, ADATT allows writing executable specifications that resemble functional programs and can be translated to interactive theorem provers. However, as the name suggests, ADATT focuses on documentation and testing and considers interactive theorem proving as an extra.

Domain experts should be able to use ADATT to write natural language documentation without any formal content. This should not be much more cumbersome than using other tools. One should be able to produce production-quality documents. Formal content can be added step by step. There should be an immediate benefit for adding formal content. If one – for example – formally declares a function or type, the spelling of it should be checked in the documentation. If you add a type signature, typechecking could take place in the natural language definition.

It is vital that ADATT can deal with *partial* specifications. Partially is supported in multiple ways. Even just declaring a function together with a type signature is a partial specification. You can then add single test cases. These test cases should be easily executable against a real implementation. ADATT aims to provide good support for conformance testing by e. g. providing special code generation. ADATT will support *code contracts* as well as families of executable tests. There is a separate syntactic construct for adding non-executable properties of the function. These cannot be used for testing and are instead intended to be checked by interactive theorem proving. One or more of such non-executable properties can be used as an axiomatic specification for theorem prover backends.

Ideally, however, we would like to end up with executable specifications. I can well imagine that different parts of the specifications are written by different people in different files. A programmer might start with natural language documentation, a function declaration and a few simple test cases. A test engineer might then add code contracts, some more tests and perhaps even a non-executable property. Finally, a formal methods expert might provide an executable specification and add non-executable properties. Different formal method experts might then use theorem provers of their choice to reason about the model, while ADATT keeps track of progress and links developments in various provers.

It is vital to provide some easy measurements of progress in order to integrate ADATT with existing software development processes. This means providing good reports and statistics. (How many functions are declared / specified / executable? Which tests were run when? ...) More interestingly, however, ADATT should be able to measure code-coverage.

4 Conclusion

ADATT is still in its very early stages. There is not even a prototype yet. There is a lot of work still ahead. This is in particular true, since ADATT needs a good user-interface, i. e. integration in commonly used IDEs. However, important design decision have already been made and implementation is well underway. Therefore, I would already value some comments.

References

[1] Hanno Becker, Juan Manuel Crespo, Jacek Galowicz, Ulrich Hensel, Yoichi Hirai, César Kunz, Keiko Nakata, Jorge Luis Sacchini, Hendrik Tews, and Thomas Tuerk. *Combining Mechanized Proofs and Model-Based Testing in the Formal Analysis of a Hypervisor*, pages 69–84. Springer International Publishing, Cham, 2016.

[2] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, 2014. ACM.

[1] citation from `http://www.cl.cam.ac.uk/~pes20/lem`

Tuning Natural Deduction Proof Search by Analytic Methods.

Alexander Bolotov[1] Alexander Gorchakov[2]

[1] University of Westminster, W1W 6UW, London, UK, A.Bolotov@Wmin.ac.uk

[2] Laboratory of information systems in Humanites, Faculty of Philosophy, Lomonosov Moscow State University, 27-4, GSP-1, Moscow, Russian Federation, gorchakov@philos.msu.ru

Abstract: This paper is a result of the analysis of the efficiency of natural deduction proof search and the major weaknesses affecting it. We introduce new analytic strategies based on a new concept ”Auxiliary Truth Set”. We present a combined proof search algorithm for classical propositional logic where a crucially new step is the guidance of the searching procedure by Auxiliary Truth Sets and establish the correctness. We describe the implementation of this new search technique and exemplify its advantages considering the strong version of the Pigeon Hole Principle.

1 Introduction

The natural deduction (ND) proof search we optimise in this paper, was initially formulated for classical setting [2] and then extended to a number of logics – propositional linear-time temporal logic PLTL [1, 3], paracomplete [6] and paraconsistent [5]. Our recent work on the complexity of the method [4] and the implementation of the technique have shown that the method should be tuned to make proofs more efficient. In particular, we were interested in improving the performance of the algorithm on the class of formulae corresponding to the famous pigeon hole principle (PhP) which is often considered as an important ‘testing’ step for theorem provers. In this work we introduce new analytic strategies based on a new concept which we call ”Auxiliary Truth Set (ATS)”. We present a combined proof search algorithm for classical propositional logic where the searching procedure is guided by the ATS and establish the correctness. This technique has been implemented exemplifying its advantages on the strong version of the Pigeon Hole Principle.

2 Natural Deduction System

Figure 1 shows elimination (*el*) and introduction (*in*) rules.

Elimination Rules:

$$\wedge_{el_1}\frac{A \wedge B}{A} \qquad \wedge_{el_2}\frac{A \wedge B}{B}$$

$$\neg_{el}\frac{\neg\neg A}{A} \qquad \Rightarrow_{el}\frac{A \Rightarrow B \quad A}{B} \qquad \vee_{el}\frac{A \vee B \quad \neg A(\neg B)}{B(A)}$$

Introduction Rules:

$$\vee_{in_1}\frac{A}{A \vee B} \qquad \vee_{in_2}\frac{B}{A \vee B}$$

$$\wedge_{in}\frac{A \quad B}{A \wedge B} \qquad \Rightarrow_{in}\frac{[C] \quad B}{C \Rightarrow B} \qquad \neg_{in}\frac{[C] \quad B \quad \neg B}{\neg C}$$

Figure 1: ND Rules

If the conclusion of $\Rightarrow_{in}$ or $\neg_{in}$ is at step n, then $[C]$, where C is the most recent alive assumption, means that C is discharged and all formulae from C up to n are discarded.

An ND-derivation or inference of a formula B from a (possibly empty) set of assumptions Γ is a finite sequence of formulae $A_1, \dots, A_n = B$ such that every A_i $(1 \leqslant i \leqslant n)$ is either an initial assumption or a conclusion of one of the rules applied to some preceding formulae. If a set of initial assumptions Γ is empty then B is a theorem.

3 New Proof Search Algorithm

The proof search algorithm is represented as a sequence of *algo-steps* $\Gamma \vdash \mathcal{G}$, where Γ is an ordered set of formulae in the proof and $\mathcal{G}$ is a stack of goals. The stack control mechanism is based on the LIFO principle. The proof commences with the initial task, of deriving goal g_0 from some given set of formulae $\Gamma = F_1, F_2, \dots, F_m$ $(1 \leq m)$, abbreviated as $F_1, F_2, \dots, F_m \vdash g_0$ (if $\Gamma = 0$, we have a task of proving g_0 as a theorem). Γ can be classified into the following six subsets:
Γ^{init} (initial assumptions in Γ), discarded formulae F^{disc}, formulae F^{el} - premises of the elimination rules, formulae F^{src} that generated new goals, auxiliary assumptions F^{assmp}, all other formulae $F^{poten} = F \backslash (F^{disc} \cup F^{in} \cup F^{el} \cup F^{src})$, where $F^{el} \cap F^{assmp} \neq \emptyset$.
A goal $g_i \in \mathcal{G}$, is reached iff

- if $g_i \neq \bot$ then g_i is *reached* iff $\exists f_i \in \Gamma$ such that $f_i = g_i$ and $f_i \notin F^{disc}$
- if $g_i = \bot$ then g_i is *reached* iff $\exists f_k, f_l$ such that $\{f_k, f_l\} \subset \Gamma$ and $f_l = \neg f_k$ and $f_k \notin F^{disc}$ and $f_l \notin F^{disc}$

If the current goal g_c is reached then it is deleted from $\mathcal{G}$ and the immediately preceding goal becomes our new current goal.

The heuristics are classified depending on the main logical connective of the goal.

(i) ‘implication’: If $g_c = A \Rightarrow B$ then F^{assmp} is updated with A and $\mathcal{G}$ is updated with $g_c = B$.

(ii) ‘conjunction’: If $g_c = A \wedge B$ then we set up goal $g_c = A$ (unless it has been already reached) and A needs to be reached before $g_c = B$ in the same fashion.

(iii) ‘negation’: If $g_c = p$ $(\neg p)$ (for some literal p) then F^{asspm} is updated with $\neg p$ (p) and $\mathcal{G}$ is updated with $\perp$.

(iv) ‘disjunction’: If $g_c = A \vee B$ then $\mathcal{G}$ is updated with $g_c = A$ to be reached by the heuristics unless the ‘negation’ strategy is required. If A is not reachable then all formulae and goals introduced since $g_c = A$, are deleted and $g_c = B$ and the same process applies as for $g_c = A$. If $g_c = B$ is not reached, then, after all deletions, F^{asspm} is updated by $\neg(A \vee B)$, and $\mathcal{G}$ is updated by $g_c = \perp$. For the efficiency, we also add an auxiliary rule $\vee_{el_{aux}} \dfrac{\neg(A \vee B)}{\neg A \wedge \neg B}$

(v) First, we introduce the notion of the ‘proof potential’, which is $\phi = ((F^{assmp} \setminus F^{el}) \cap (F^{assmp} \setminus F^{disc})) \cup F^{poten}$ ‘Auxiliary Truth Set (ATS)’ applies when the potential of the proof $\phi \neq \emptyset$, the current goal, $g_c = \perp$ has been generated by the last assumption, f_n, and none of the rules or other heuristics is applicable. Now we split the set Γ into two sets: $\Gamma_2 = \{f_n\}$ and $\Gamma_1 = \Gamma \setminus \Gamma_2$ and set up the new goal called ‘ATS-goal’=$\neg(\bigwedge_{i=1}^{n} f_i)$, where $f_i \in \Gamma_1 \setminus F^{disc}$. In other words, we set up the goal as the negation of conjunctions of all non-discarded formulae in the proof before f_n, which we now call ‘ATS-assumption’. It is shown that a proof of ‘ATS-goal’ from f_n is necessary and sufficient condition for the presence of a contradiction in $\Gamma_1 \setminus F^{disc}$. Now we generate all sets of truth values that make ‘ATS-goal’ true. Then we check if there is a variable in ‘ATS-goal’ which does not occur in ‘ATS-assumption’ and if there is one we check if this variable is ‘significant’ for the ‘ATS-goal’. This is the process of establishing if this variable takes both values - true and false - under the fixed values of variables of the ‘ATS-assumption’, in which case we eliminate this variable from the consideration. Alternatively, we conclude that ‘ATS-goal’ does not follow from the ‘ATS-assumption’ and terminate proof by claiming that the desired proof from the initial set of assumptions cannot be found. Now let $\phi = \phi^1 \cup \phi^2$, where $\phi^2 = f_n$ and $\phi^1 \cup \phi^2 = \emptyset$. Now we build ATS for ATS-assumption. For each of the groups of formulae from ϕ_2 with common variables we build their truth sets. If a group does not have a variable common with the ATS-goal then we do not consider it. Similarly, we check if a group that has variables common with the ATS-goal does not contain a variable not occurring in ATS-goal and delete this group if this is not true. Comparing ATS for ATS-assumption and ATS-goal, we check if every truth set of ATS-assumption contains at least one element of the truth set of ATS-goal. If we find a combination that violates this then we terminate the proof as we found the evaluation under which the formula given for the proof is false.

4 Experimental results: Pigeon Hole Principle under the new proof search.

To illustrate how the proposed proof search works we present here its performance on the Pigeon Hole Principle comparing with the original proof search for ND [2] and with Buss’s proofs for this important for theorem provers class of testing formulae. Note that Buss’s results are given for the DPLL with clause learning [7].

	ND	ND with ATS	DPLL with clause learning
PHP$_{2}$	93	27	-
PHP$_{3}$	740	60	5
PHP$_{4}$	7883	115	-
PHP$_{5}$	110509	198	-
PHP$_{6}$	1914985	315	129
PHP$_{7}$	-	472	-
PHP$_{8}$	-	675	769
PHP$_{10}$	-	1243	-
PHP$_{12}$	-	2067	20000

References

[1] A. Bolotov, A. Basukoski, O. Grigoriev, and V. Shangin. Natural deduction calculus for linear-time temporal logic. In *Joint European Conference on Artificial Intelligence (JELIA-2006)*, pages 56–68, 2006.

[2] A. Bolotov, V. Bocharov, A. Gorchakov, and V. Shangin. Automated first order natural deduction. In *Proceedings of IICAI*, pages 1292–1311, 2005.

[3] A. Bolotov, O. Grigoriev, and V. Shangin. Automated natural deduction for propositional linear-time temporal logic. In *Temporal Representation and Reasoning, 14th International Symposium on*, pages 47–58, June 2007.

[4] A. Bolotov, D. Kozhemiachenko, and V. Shangin. Paracomplete logic Kl: natural deduction, its automation, complexity and applications. *IfCoLog Journal of Logics and their Applications.*, 5:221–261, 2018.

[5] A. Bolotov and V. Shangin. Natural deduction system in paraconsistent setting: Proof search for PCont. *JJournal of Intelligent Systems*, 21:1–24, 2012.

[6] A. Bolotov and V. Shangin. Tackling incomplete system specifications using natural deduction in the paracomplete setting. In *2014 IEEE 38th Annual Computer Software and Applications Conference*, pages 91–96, July 2014.

[7] S. Buss. Towards NP-P via proof complexity and search. In *Annals of Pure and Applied Logic*, volume 163, pages 906–917, 2012.

Instantiation for Theory Reasoning in Vampire

Giles Reger Martin Riener

University of Manchester, Manchester, UK

Abstract: Reasoning with theories and quantifiers in first-order logic is very hard. Over the past 3 years we have extended the Vampire theorem prover with various techniques for reasoning with problems mixing arithmetic, quantifiers, and uninterpreted functions. In this most recent work we introduce a new method for instantiation that makes use of SMT solvers to find simplifying instances of clauses and a new approach to unification that enables the application of this rule.

1 Introduction

We are interested in extending automated theorem provers for first-order logic to reason effectively with problems containing non-trivial quantification and theories such as arithmetic or datatypes. Such problems arise naturally in, for example, program analysis where quantifiers are required to axiomatise features such as dynamic memory, and arithmetic is central to most real-world programs.

Our work is in the context of the Vampire theorem prover [1]. This is an automated theorem prover (ATP) that is saturation-based and implements the superposition and resolution calculus. In saturation-based theorem provers the approach is to first transform the input problem into clausal form and then saturate the set of clauses with respect to an inference system. Vampire is also a refutational prover; its first step is always to negate the goal, which means that it aims to derive a contradiction. In pure first-order logic this approach can be refutationally complete. This breaks down in the presence of theories such as arithmetic.

Over the past 3 years we have been exploring different approaches for theory reasoning within Vampire. This has included using an SMT solver to guide proof search [2] and heuristics to control the use of theory axioms such as $x + y = y + x$ [3]. This work considers the problem of *instantiation* (for theories) in this context.

2 Background

We consider a many-sorted first-order logic over the signature $\Sigma = (\Xi, \Omega)$. The set Ω contains predicate and function symbols with argument and return values in the set of sorts Ξ (which contains the sort $\mathbb{B}$ of truth values). A *term* is a constant c, a variable x or an application $f(t_1, \ldots, t_n)$ of the n-ary function symbol f to the terms t_1 to t_n. We assume terms are well-sorted. A function symbol p with return sort $\mathbb{B}$ is called a predicate symbol. Its application $p(t_1, \ldots, t_n)$ is called an *atom*. We assume the presence of an equality predicate for each sort. A *literal* is either an atom A or a negated atom $\neg A$. We abbreviate $\neg(c \simeq_s d)$ as $c \not\simeq_s d$. A subterm s of t at position p is written as $t[s]_p$.

A clause is a multiset of literals which is interpreted as a disjunction $L_1 \vee \ldots \vee L_n$. A substitution $\theta = \{x_1 \mapsto t_1, \ldots x_n \mapsto t_n\}$ maps variables to terms; applying θ to a term simultaneously replaces the variables by the corresponding terms. A *unifier* θ of two terms s and t is a substitution such that $(s \simeq t)\theta$ is valid; a *most general unifier* of s and t is a unifier that is not an instance of any other unifier of those terms up to renaming of variables.

A theory defines a class of interpretations. All interpretations in a theory $\mathcal{T}$ agree on the assignment for a set of *theory symbols*. A symbol that does not have a fixed interpretation is called a *non-theory symbol*.

A literal is a *theory literal* if its predicate symbol is a theory symbol. The equality $\simeq_s$ predicate of a sort s is a theory symbol if the sort s is interpreted by a theory. A *pure* literal contains only theory symbols or only non-theory symbols. A clause is *fully abstracted* if it only contains pure literals and *partially abstracted* if non-theory symbols no not appear inside applications of theory symbols. A non-variable term t is a *theory term* (*non-theory term*) if its top function symbol is a theory symbol (non-theory symbol).

Given a clause $L[t] \vee C$, where L is a theory literal and t is a non-theory literal or vice versa, we can separate them by introducing a fresh variable x for t to obtain $L[x] \vee C \vee x \not\simeq t$. Repeating this process leads to an abstracted clause.

2.1 Does Vampire Need Instantiation?

To see why Vampire can benefit from instantiation, consider the first-order clause

$$14x \not\simeq x^2 + 49 \vee p(x) \tag{1}$$

for which there is a single integer value for x that makes the first literal false with respect to the underlying theory of arithmetic, namely $x = 7$. However, if we apply standard superposition rules to the original clause and a sufficiently rich axiomatisation of arithmetic, we will most likely end up with a very large number of logical consequences and never generate $p(7)$, or run out of space before generating it. Indeed, Vampire cannot find a refutation of $14x \not\simeq x^2 + 49$ in reasonable time using our previous approaches [2, 3].

3 What Kind of Instances Do We Want?

Since there are possibly infinitely many instantiations, we only want to create instances with an immediate benefit. The inference rule we consider is of the form

$$\frac{P \vee D}{D\theta} \text{ theory instance} \tag{2}$$

where P contains only pure theory literals and $P\theta$ is unsatisfiable in the given theory. As P contains only pure theory literals we can use a SMT solver to find a model of $\neg P$ and use this to generate θ. In the case of clause 1 above, we pick $P = 14x \simeq x^2 + 49$ to extract $\{x \mapsto 7\}$ from the model generated by the SMT solver. From this we can conclude $p(7)$. If the SMT solver finds $\neg P$ to be unsatisfiable then P is a tautology and $P \vee D$ can instead by removed. Note that we assume that the theory is complete. The result of this approach is that we produce instances that are *shorter*.

4 Instantiation in a Saturation-Based Theorem Prover

For clauses containing inequalities, we would prefer to apply the equality resolution rule

$$\frac{s \not\simeq t \vee C}{C\theta}\ \theta = \mathsf{mgu}(s,t), \text{equality resolution}$$

instead of instantiation. For the clause $x \not\simeq 1 + y \vee p(x,y)$, equality resolution leads to $p(y+1,y)$ which is more general than $p(1,0)$ obtained from instantiating with $\{x \mapsto 0, y \mapsto 0\}$. Moreover, abstraction and instantiation may work against each other. If we consider the clause $p(1,5)$, it will be abstracted to $x \not\simeq 1 \vee y \not\simeq 5 \vee p(x,y)$. But the substitution $\{x \mapsto 1, y \mapsto 5\}$ makes $\neg(x \not\simeq 1 \vee y \not\simeq 5)$ valid. If we use it to instantiate $p(x,y)$, we re-obtain the original clause $p(1,5)$.

To prevent these effects, we introduce a further restriction on P. A literal L is *trivial* in clause C if

- L is of the form $x \not\simeq t$ and x does not occur in t
- L is a pure theory literal
- every occurrence of x in C is either $x \not\simeq t$, in a literal that is not pure or another literal trivial in C

The inference rule (2) then has the restrictions that

- P contains only pure literals
- P contains no literals trivial in $P \vee D$
- $\neg P\theta$ is valid in $\mathcal{T}$

Note that there is no requirement on P to be maximal. The more literals P has, the more precise the instantiation becomes. This comes at the risk of over-specialising, even after the removal of trivial literals.

5 Extending Unification to Help

So far we have left out the role of abstraction. In principle, the rule (2) works on any clause. However, abstracted clauses have more pure theory literals to apply the rule to. For example, the clauses

$$r(14y) \text{ and } \neg r(x^2 + 49) \vee p(x)$$

permit neither the application of resolution nor of theory instantiation. But their abstracted form

$$r(u) \vee u \not\simeq 14y \text{ and } \neg r(v) \vee v \not\simeq x^2 + 49 \vee p(x)$$

can be resolved to $u \not\simeq 14x \vee u \not\simeq x^2 + 49 \vee p(x)$ which becomes $p(7)$ after theory instantiation.

However, fully abstracting every clause has a devastating impact on proof search because it significantly increases the clause length. If we only apply theory instantiation after such a resolution step, we can modify the unification procedure to generate an abstraction on the fly. Unification with abstraction, written $\mathbf{mgu}_{abs}(s,t)$, returns a pair (θ, D), if possible, where D is a disjunction of inequalities and θ is a substitution making $(D \vee s \simeq t)\theta$ valid in a theory $\mathcal{T}$. If we can show that D are unsatisfiable then we have performed unification modulo $\mathcal{T}$. By happy coincidence, the theory instantiation rule can handle such theory constraints.

Unification with abstraction should not be applied to eagerly as it, in the limit, it can be used to make any two terms unify. For example, we would like to prevent abstraction in the case of resolving $r(1)$ with $r(2)$ because the generated constraint $1 \simeq 2$ can never be true. In general, $\mathbf{mgu}_{abs}$will never produce constraints that can not be equal in the underlying theory. We have also experimented with heuristics that decide for which subterms abstractions are generated.

The calculus can be adapted to use unification with abstraction instead of the traditional one. The resolution rule then becomes

$$\frac{A \vee C_1 \quad \neg A' \vee C_2}{(D \vee C_1 \vee C_2)\theta}\ res_{wA}$$

where $(\theta, D) = \mathbf{mgu}_{abs}(A, A')$. The factoring, superposition and equality resolution rules can be similarly adapted.

6 Summary

We have implemented a new approach to reasoning with theories and quantifiers in a saturation-based theorem prover. This approach utilises an SMT solver to find useful instances and extends unification to produce clauses that are likely to have useful instances. We have implemented these approaches in Vampire[4], our experiments indicate that unification with abstraction is beneficial for some cases.

Acknowledgements. We describe work published by the first author, Martin Suda, and Andrei Voronkov [4].

References

[1] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV 2013*, volume 8044 of *LNCS*, pages 1–35, 2013.

[2] Giles Reger, Nikolaj Bjørner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *GCAI 2016*, volume 41 of *EPiC Series in Computing*, pages 39–52. EasyChair, 2016.

[3] Giles Reger and Martin Suda. Set of support for theory reasoning. In *IWIL Workshop and LPAR Short Presentations*, volume 1 of *Kalpa Publications in Computing*, pages 124–134. EasyChair, 2017.

[4] Giles Reger, Martin Suda, and Andrei Voronkov. Unification with abstraction and theory instantiation in saturation-based reasoning. In *TACAS*, 2018.

Towards Polynomial Time Forgetting and Instance Query Rewriting in Ontology Languages

Sen Zheng Renate A.Schmidt

University of Manchester, Manchester, UK

{sen.zheng,renate.schmidt}@manchester.ac.uk

Abstract: Ontologies can present a conceptual view of a relational database. This ontology-based data access (OBDA) can allow a client to query enterprise ontologies directly. The problem of rewriting and optimisation of such queries against ontologies is insufficiently studied in database research. In this paper, we discuss using uniform interpolation to forget some symbols, especially role symbols, to rewrite instance queries against ontologies. In particular, when there is no nesting in an ontology, our forgetting algorithm is guaranteed to terminate in a polynomial time. We introduce Ackermann's lemma-based algorithm to preserve semantic equivalence during query rewriting. We further extend our approach to linear Datalog$^{\pm}$ rules (existential rules with equality) and also the guarded fragment of first-order logic. These two languages can be regarded as generalisations of description logics, which provide bases of ontology languages.

1 Introduction

An ontology is a form of graph-based database management system, and it allows automated processing and reasoning. In the ontology-based data access (OBDA), an ontology is used as a conceptual layer of relational databases, allowing clients to manage and query data more directly. Such querying is called the ontology-based query answering (OBQA). It is now an insufficiently studied problem in database research.

In OBDA, an instance query q is answered against a database D with an ontology Σ such that $D \cup \Sigma \models q$. An instance query is a unary atomic query such as A(x). In this setting, Description Logics (DLs) are used as an ontology language. However, the best known decidable fragments of DLs are 2EXPTIME-complete, which makes querying very hard. Most OBQA systems are based on a lightweight DLs, such as DL-Lite [5] and $\mathcal{EL}$ [2] families. These DLs are designed to guarantee decidability and polynomial time data complexity for the query answering. DLs only allow unary and binary relations, Calì [3] argues that the Datalog$^{\pm}$ language, which have multi-ary and unary relations, is a strong tool for query answering. In Datalog$^{\pm}$, guarded Datalog$^{\pm}$ and its subclass linear Datalog$^{\pm}$ show good decidability results [4]. Having lightweight DLs and linear Datalog$^{\pm}$ ontologies, researchers focus on rewriting and optimising queries to make querying more effective. In [5], authors proposed the *perfect reformulation* algorithm to do rewriting. [7] gives a polynomial rewriting approach for linear Datalog$^{\pm}$.

Since DLs $\mathcal{ALCOI}$ ($\mathcal{ALC}$ with nominals and inverse roles) can be seen as a fragment of first-order logic, its translation in first-order logic is generalised as the Guarded Fragment (GF) [1]. The guards in GF correspond to role symbols in DLs. Moreover, GF is also a superclass of the linear Datalog$^{\pm}$ that follows GF format.

In this paper, we use a *forgetting algorithm* to rewrite queries while preserving semantic equivalence, and we are interested in a new different class from previous ones, GF without nested formulas, to forget guard symbols. In particular, when guards do not occur in non-guard positions, the data complexity and combined complexity of our algorithm is tractable.

2 Forgetting and GF

Forgetting is a non-standard reasoning procedure to remove the forgetting signatures from original formulas, and keep the remaining formulas semantically equivalent to the original formulas. In other words, the result formulas are equivalent to the original formulas up to the forgetting signatures. This work is motivated by [11] and [10] that extends the forgetting algorithm to first-order logic.

GF is robustly decidable [8], but it does not have the Craig Interpolation Property, thus the Uniform Interpolation property (a.k.a the forgetting property) [9]. That means forgetting some predicates in GF does not guarantee that the result still belongs to GF. Recent research use the model theory to show that the forgetting signature can only be non-guard predicates and it fails to forget guards.

In this paper, we show that by introducing $\exists$-guard, guards can be forgotten without losing semantic equivalence.

3 Forgetting Guards in GF

We define a guarded formula without any nested formulas as a flat guarded formula. In particular, for flat guarded formulas with equality and constants, we concern forgetting guards when there is no guard occurring at a non-guard position in other formulas. The input is a set of formulas N combined by formulas mentioned above, and the forgetting signatures are a set of guards G in N. Our algorithm has 4 major steps:

1. **Normalisation** In this step, every input formula in N is formalised as a formula without any free variables.

We add universal quantifications to free variables in N, and then transform these formulas into their negation normal forms N_1.

2. **Structural Transformation** In this step, each formula in N_1 is transformed into its clausal form. We introduce new predicates, also known as definers, to do structural transformation. During structural transformation, each formula in N_1 is transformed differently depending on the root of its formula tree. For some formulas in N_1 that contain constants and equalities, we use the term abstraction rule and the equality elimination rule as follows.

   $$\frac{N \cup \{C(\bar{x}, \bar{a})\}}{N \cup \{C(\bar{x}, \bar{y}) ... \lor y_n \not\approx a_n\}}$$

   where $\bar{y}$ is disjoint with $\bar{x}$.

   $$\frac{C \lor x \not\approx a}{C \lor Q_i(x), \neg Q_i(a)}$$

   where Q_i is a fresh predicate.

   We also introduce some special definers $\approx$-*guard* eqG and $\exists$-*guard* eG. An eqG is used to define equalities such as $x \not\approx a$, and an eG is used as a guard for existential quantified unguarded clause such as $\exists xy(A(x) \land B(y))$. An eG is used to transform it into $\exists xy(eG(x,y) \land A(x) \land B(y))$. After structural transformation, the clausal form of formulas in N_1 is ground or is positive conjunction of atoms or contains a negative literal that has all variables in this clause. We call the set of result clauses N_2.

3. **Forgetting Guards** In this step, the set of guard symbols G in the forgetting signatures are eliminated. We use Ackermann's Lemma to eliminate guards one at a time. The set of result formulas are called N_3.

4. **Eliminating Definers** In this step, the aim is to eliminate definer symbols introduced in step 2. Ackermann's Lemma is also used to eliminate these definers.

Given flat guarded formulas, possibly with equality and constants, and assuming there is no guard occurring at a non-guard position in other formulas. We can claim that:

Claim 3.1 *This forgetting algorithm is sound, terminating and forgetting complete.*

Claim 3.2 *The complexity of our algorithm is polynomial.*

4 Conclusion and Ongoing Work

In this paper, we show that we can forget guards in flat GF in a polynomial time without losing semantic equivalence. Because we consider instance queries in this paper, this approach can be used as a query rewriting algorithm to forget guards in Datalog$^{\pm}$ that follows GF format, and to forget roles in $\mathcal{ALCOI}$. In future, we will focus on forgetting a non-guard predicates in GF and apply our algorithm to other possible applications like abduction reasoning [6].

References

[1] Hajnal Andréka, István Németi, and Johan van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[2] Franz Baader. Terminological cycles in a description logic with existential restrictions. In *IJCAI*, volume 3, pages 325–330, 2003.

[3] Andrea Calì. Ontology querying: datalog strikes back. In *Reasoning Web International Summer School*, pages 64–67. Springer, 2017.

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

[5] Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The dl-lite family. *Journal of Automated reasoning*, 39(3):385–429, 2007.

[6] Diego Calvanese, Magdalena Ortiz, Mantas Simkus, and Giorgio Stefanoni. Reasoning about explanations for negative query answers in dl-lite. *Journal of Artificial Intelligence Research*, 48:635–669, 2013.

[7] Georg Gottlob, Marco Manna, and Andreas Pieris. Polynomial rewritings for linear existential rules. In *IJCAI*, pages 2992–2998, 2015.

[8] Erich Grädel. Why are modal logics so robustly decidable? In *Bulletin EATCS*. Citeseer, 1999.

[9] Eva Hoogland and Maarten Marx. Interpolation and definability in guarded fragments. *Studia Logica*, 70(3):373–409, 2002.

[10] Patrick Koopmann. Practical uniform interpolation for expressive description logics. 2015.

[11] Yizheng Zhao and Renate A Schmidt. Role forgetting for alcoqh (δ)-ontologies using an ackermann-based approach. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1354–1361. AAAI Press, 2017.

Iterative Abduction using Forgetting

Warren Del-Pinto Renate Schmidt

University of Manchester, UK

{warren.del-pinto, renate.schmidt}@manchester.ac.uk

Abstract: Abductive reasoning computes hypotheses to explain new observations, making use of background knowledge. Here, knowledge is represented using description logic (DL) ontologies and forgetting is used to compute consistent, semantically minimal hypotheses in a given signature of abducibles. This paper gives a brief overview of a method for abduction in DLs that uses forgetting. Building on this, the notion of iterative abduction is then proposed and outlined, as this will form the basis for future research in integrating abduction and induction in DLs.

1 Introduction

Abduction was identified as a form of reasoning by C.S. Peirce [7]. The aim of abductive reasoning is to generate hypotheses for new observations, making use of background knowledge. Here, knowledge is represented as a description logic (DL) ontology and the computed hypotheses are restricted to those satisfying the following conditions:

Definition 1: Abduction in Ontologies. *Let $\mathcal{O}$ be an ontology, $\mathcal{S}_A$ a set of abducibles, and ψ a set of axioms such that $\mathcal{O} \not\models \perp$, $\mathcal{O}, \psi \not\models \perp$ and $\mathcal{O} \not\models \psi$. The abduction problem is to find a set of axioms $\mathcal{H}$, consisting only of symbols in $\mathcal{S}_A$, such that:* (i) *$\mathcal{O}, \mathcal{H} \not\models \perp$,* (ii) *$\mathcal{O}, \mathcal{H} \models \psi$ and* (iii) *There is no other hypothesis $\mathcal{H}'$ such that $sig(\mathcal{H}') \subseteq \mathcal{S}_A$, $\mathcal{O}, \mathcal{H}' \models \psi$ and $\mathcal{H} \models \mathcal{H}'$, and $\mathcal{O}, \mathcal{H}' \not\equiv \mathcal{O}, \mathcal{H}$*

This restricts hypotheses to those containing only abducible symbols that are (i) consistent with the background knowledge $\mathcal{O}$, (ii) explain the observation ψ when added to $\mathcal{O}$ and (iii) make the fewest assumptions necessary to explain ψ, i.e. are *semantically minimal*. Such restrictions are needed to reduce the search space of hypotheses and to ensure that hypotheses are informative.

DL ontologies consist of a TBox containing information about general entities known as concepts and an ABox containing information regarding specific constants called individuals. As a result, the abduction problem is divided into two tasks: TBox abduction, where the hypothesis and observation take the form of TBox axioms, and ABox abduction, where both take the form of ABox (ground) assertions.

2 Forgetting for Abduction

A method for performing both TBox and ABox abduction using forgetting, also known as uniform interpolation or second-order quantifier elimination [6, 2], has been developed [1]. Forgetting eliminates symbols from an ontology, while preserving all entailments of the original ontology representable in the restricted signature. The result is a new ontology, known as a uniform interpolant.

Framing abduction in terms of forgetting requires the use of contrapositive reasoning as follows: $\mathcal{O}, \mathcal{H} \models \psi$ *iff* $\mathcal{O}, \neg\psi \models \neg\mathcal{H}$, where $\mathcal{O}$ is an ontology, ψ is an observation and $\mathcal{H}$ is a hypothesis.

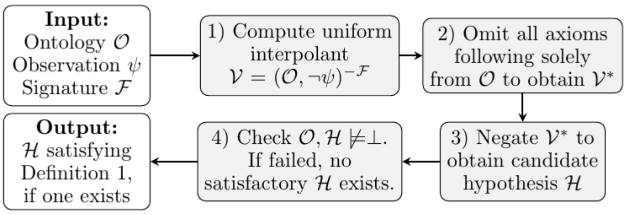

Input:
Ontology $\mathcal{O}$
Observation ψ
Signature $\mathcal{F}$
1) Compute uniform interpolant $\mathcal{V} = (\mathcal{O}, \neg\psi)^{-\mathcal{F}}$
2) Omit all axioms following solely from $\mathcal{O}$ to obtain $\mathcal{V}^*$
3) Negate $\mathcal{V}^*$ to obtain candidate hypothesis $\mathcal{H}$
4) Check $\mathcal{O}, \mathcal{H} \not\models \perp$. If failed, no satisfactory $\mathcal{H}$ exists.
Output:
$\mathcal{H}$ satisfying Definition 1, if one exists

Figure 1: Main steps of the proposed abduction method.

Forgetting is used to compute a uniform interpolant $\mathcal{V}$ of an ontology $\mathcal{O}$ together with the negation of an observation ψ, i.e. $\mathcal{V} = (\mathcal{O}, \neg\psi)^{-\mathcal{F}}$. Since uniform interpolants are sets of entailments of the input, in this case $(\mathcal{O}, \neg\psi)$, the role of forgetting is to compute the set of entailments $\neg H$ required for contrapositive reasoning. By negating the uniform interpolant $\mathcal{V}$ a candidate hypothesis is obtained, where the abducible symbols $\mathcal{S}_A$ are defined by $\mathcal{S}_A = sig(\mathcal{O}, \psi) \setminus \mathcal{F}$: the complement of the forgetting signature $\mathcal{F}$.

However, to satisfy the consistency requirement in Definition 1(i), it is first necessary to apply postprocessing to $\mathcal{V}$. This postprocessing removes all axioms that follow solely from the original ontology $\mathcal{O}$, as these do not contribute to an explanation of ψ and will be inconsistent when negated under contrapositive reasoning. The result is the reduced uniform interpolant $\mathcal{V}^*$, containing only those axioms obtained as the conclusions of inferences between the background knowledge in $\mathcal{O}$ and the observation to be explained ψ. In small theories this can be done by performing entailment checks on each of the axioms $\beta \in \mathcal{V}$: if $\mathcal{O} \models \beta$, then remove β from $\mathcal{V}$. However, in the setting of large DL ontologies this is too computationally expensive to provide a feasible solution. Thus, the current method utilises annotations, which are concepts disjunctively appended to axioms in ψ. This annotation is retained in the results of any inferences with axioms in ψ. The set $\mathcal{V}^*$ is extracted by deleting all non-annotated axioms in $\mathcal{V}$, by checking the signature of these axioms for the presence of the annotation concept [1].

As uniform interpolants are the strongest necessary entailments [5] of the input, in this case $(\mathcal{O}, \neg\psi)$, negating $\mathcal{V}^*$ results in the weakest sufficient or semantically minimal hypotheses satisfying Definition 1, since strongest necessary and weakest sufficient conditions are dual notions [5].

3 Iterative Abduction

The aim of ongoing research is to extend the above abduction framework to enable the iterative computation of sets of increasingly stronger hypotheses for a given observation: which will be referred to as *iterative abduction*.

This can be seen as a form of tree search over the space of possible, semantically minimal hypotheses. Starting with $(\mathcal{O}, \neg\psi)$, the first step is to compute a set of uniform interpolants: one for each unique forgetting signature $\mathcal{F}$ containing a single concept from $sig_c(\psi)$. This results in the set of strongest possible uniform interpolants $\mathcal{V}^l$, i.e. those that preserve the most entailments of $(\mathcal{O}, \neg\psi)$, where l denotes the current depth of the tree. By repeating the steps in Figure 1, a corresponding set of weakest, semantically minimal hypotheses $\mathcal{H}^l$ is obtained. Each pair $\{\mathcal{V}^l_i, \mathcal{H}^l_i\}$ can be seen as a node of the tree, where $1 \leq i \leq N^l$ and N^l is the number of nodes at depth l.

This procedure is repeated for every node. In each case, forgetting is applied to the uniform interpolant $\mathcal{V}^l_i$ for each signature $\mathcal{F}$ containing only one concept in $sig_c(\mathcal{H}^l_i)$. Once this has been applied to each pair $\{\mathcal{V}^l_i, \mathcal{H}^l_i\}$, the result is a new set of pairs $\{\mathcal{V}^{l+1}_j, \mathcal{H}^{l+1}_j\}$ where $1 \leq j \leq N^{l+1}$.

Termination of this procedure could be decided in multiple ways. For small examples, it is feasible to continue computing hypotheses along each branch until no stronger hypotheses remain. For large ontologies it is likely that this would not be practical. Instead, the procedure could be performed step-by-step until the user is satisfied with the informativeness of the hypothesis obtained. Alternatively, induction could be used to judge each hypothesis with respect to a set of examples: stronger hypotheses could be computed until a certain threshold is reached with respect to the proportion of examples entailed by the hypothesis.

The following small example illustrates this notion.

Example 1: *Let* $\mathcal{O} = \{A \sqsubseteq \exists r.C, B \sqsubseteq \exists r.C, C \sqsubseteq D\}$ *and* $\psi = D(a^*)$*. Computing the uniform interpolant using* $\mathcal{F}{=}\{D\}$ *results in* $\mathcal{V}{=}\{A \sqsubseteq \exists r.C, B \sqsubseteq \exists r.C, \neg C(a^*)\}$*. By applying the steps described in Section 2, the hypothesis* $\mathcal{H} = C(a^*)$ *is obtained. Next, computing the uniform interpolant of* $\mathcal{V}$ *using a second forgetting signature* $\mathcal{F}_2 = \{C\}$ *gives* $\mathcal{V}_2 = \{\neg A(a^*), \neg B(a^*)\}$*. Applying the same steps again gives a new hypothesis* $\mathcal{H}_2 = (A \sqcup B)(a^*)$*. This is stronger than* $\mathcal{H}$*, and is also the result that would be obtained by computing the uniform interpolant of* $(\mathcal{O}, \neg\psi)$ *using a forgetting signature* $\mathcal{F} = \{D, C\}$ *in one step.*

4 Outlook and Future Work

The abduction method described in Section 2 has been implemented using the forgetting tool LETHE [3], and can perform TBox [4] and ABox abduction in the DL $\mathcal{ALC}$ over complex, non-Horn axioms excluding role assertions. Experimental results over a corpus of real-world ontologies indicate that this method is practical: for example taking an average of 412.34 and 46.44 seconds for TBox and ABox abduction respectively for an ontology containing over 111,000 axioms, for forgetting signature of size 1.

However, the existing abduction method has several limitations. These include the inability to specify role symbols as non-abducibles and a lack of support for multiple individuals, including role assertions, in both observations and hypotheses. To overcome these limitations, possibilities include extending the tool LETHE to include skolemization to introduce new individuals outside the signature of the background ontology $\mathcal{O}$, or utilising other forgetting tools such as FAME [9] which supports nominals. Improving the expressibility of the current abduction system will naturally extend to the iterative abduction approach proposed in this paper, in particular allowing the computation of stronger hypotheses involving for example role assertions.

Other future work includes the identification and evaluation of suitable methods of inductive learning to be integrated with abduction via the iterative approach. Preliminary starting points for this investigation may include work in abductive logic programming [8] or statistical relational AI. The aim is then to evaluate the practical utility of this approach on a real-world case study.

References

[1] W Del-Pinto and R. A. Schmidt. Forgetting-based abduction in $\mathcal{ALC}$-ontologies (extended abstract). In *Workshop on SOQE and Related Topics*, volume 2013, pages 27–35. CEUR Workshop Proceedings, 2017.

[2] D. M. Gabbay, R. A. Schmidt, and A. Szałas. Second-order quantifier elimination: Foundations, computational aspects and applications. *Studies in Logic: Mathematical Logic and Foundations*, 12, 2008.

[3] P. Koopmann and R. A. Schmidt. Uniform interpolation and forgetting for $\mathcal{ALC}$ ontologies with ABoxes. In *Proc. AAAI'15*, pages 175–181. AAAI Press, 2015.

[4] P. Koopmann and R. A. Schmidt. LETHE: Saturation based reasoning for non-standard reasoning tasks. In *Proc. ORE'15*, volume 1387, pages 23–30. CEUR Workshop Proceedings, 2015.

[5] F. Lin. On strongest necessary and weakest sufficient conditions. *Artificial Intelligence*, 128:143–159, 2001.

[6] C. Lutz and F. Wolter. Foundations for uniform interpolation and forgetting in expressive description logics. In *Proc. IJCAI'11*, pages 989–995. AAAI Press, 2011.

[7] C. S. Peirce. Deduction, induction and hypothesis. *Popular Science Monthly*, 13:470–482, 1878.

[8] O. Ray. Nonmonotonic abductive inductive learning. *J. of Applied Logic*, 7:329–340, 2009.

[9] Y. Zhao and R. Schmidt. Forgetting concept and role symbols in $\mathcal{ALCOIH}\mu^+(\nabla, \sqcap)$-ontologies. In *Proc. IJCAI'16*, pages 1345–1352. AAAI Press, 2016.

Equivariant ZFA with Choice: a position paper

Murdoch J. Gabbay

Heriot-Watt University, Scotland, UK

Abstract: We propose *Equivariant ZFA with Choice* as a foundation for nominal techniques that is stronger than ZFC and weaker than FM, and why this may be particularly helpful in the context of automated reasoning.

1 Introduction

Nominal techniques assume a set $a, b, c, \ldots \in \mathbb{A}$ of *atoms*; elements that can be compared for equality but which have few if any other properties. This deceptively simple foundational assumption has many applications—nominal abstract syntax (syntax-with-binding) [9, 11]; as implemented in Isabelle [12]; an open consistency problem [6]; duality results [5, 8]; generalised finiteness for automata and regular languages [10, 1]; rewriting with binding [3]; and more.

So what is a foundation for nominal techniques?

Where this question is addressed in the nominal literature, the answer given is *Fraenkel-Mostowski set theory* (**FM**). In this position paper I will somewhat provocatively outline why this may have been a mistake, or at least a sub-optimal choice. I will propose *Equivariant ZFA set theory with Choice* (**EZFAC**) instead, and suggest why EZFAC may be especially suited to applications in automated reasoning and implementation. One standout point is that FM is inconsistent with the Axiom of Choice, whereas EZFA plus Choice (EZFAC) is consistent.

An expanded discussion of EZFAC is in [7].

$(\mathbf{AtmEmp})$	$t \in s \Rightarrow s \notin \mathbb{A}$
$(\mathbf{EmptySet})$	$t \notin \varnothing$
$(\mathbf{Ext})$	$s, s' \notin \mathbb{A} \Rightarrow (\forall \mathsf{b}.(\mathsf{b} \in s \Leftrightarrow \mathsf{b} \in s')) \Rightarrow s = s'$
$(\mathbf{Pair})$	$t \in \{s, s'\} \Leftrightarrow (t = s \vee t = s')$
$(\mathbf{Union})$	$t \in \bigcup s \Leftrightarrow \exists \mathsf{a}.(t \in \mathsf{a} \wedge \mathsf{a} \in s)$
$(\mathbf{Pow})$	$t \in \mathit{pset}(s) \Leftrightarrow t \subseteq s$
$(\mathbf{Ind})$	$(\forall \mathsf{a}.(\forall \mathsf{b}{\in}\mathsf{a}.\phi[\mathsf{a}{:=}\mathsf{b}]) \Rightarrow \phi) \Rightarrow \forall \mathsf{a}.\phi\, \mathit{fv}(\phi) = \{\mathsf{a}\}$
$(\mathbf{Inf})$	$\exists \mathsf{c}.\varnothing \in \mathsf{c} \wedge \forall \mathsf{a}.\mathsf{a} \in \mathsf{c} \Rightarrow \mathsf{a}{\cup}\{\mathsf{a}\} \in \mathsf{c}$
$(\mathbf{AtmInf})$	$\neg(\mathbb{A} \subseteq_{\mathit{fin}} \mathbb{A})$
$(\mathbf{Replace})$	$\exists \mathsf{b}.\forall \mathsf{a}.\mathsf{a} \in \mathsf{b} \Leftrightarrow \exists \mathsf{a}'.\mathsf{a}' \in u \wedge \mathsf{a} = F(\mathsf{a}')$
$(\mathbf{Choice})$	$\varnothing \neq (\mathit{pset}^*(s) \to s)$ $\quad$ pset^* nonempty powerset
$(\mathbf{Equivar})$	$\forall \mathsf{a}{\in}\mathit{Perm}.(\phi \Leftrightarrow \mathsf{a}{\cdot\cdot}\phi)$.

Figure 1: Axioms of EZFAC

2 Equivariant ZFA with Choice

DEFINITION 2.1. The language of EZFAC is the language of *sets with atoms*—first-order logic with a binary predicate $\in$, and a single constant symbol $\mathbb{A}$ for the **set of atoms**.[1] Axioms are in Figure 1; notation is defined below.

REMARK 2.2. Axioms $(\mathbf{AtmEmp})$ to $(\mathbf{Choice})$ are standard ZFAC (ZF with atoms and Choice). In rule $(\mathbf{Choice})$, pset^* is the *nonempty powerset* operator, since we cannot choose an element of the empty set.

REMARK 2.3. ZF and ZFA are equally expressive: a model of ZFA embed in one of ZF,[2] and vice-versa, and a predicate in ZFA can be translated (quite easily) to one in ZF.

Yet if the translation from ZFA to ZF leads to a quadratic increase in proof-size, or if ZFA is an environment which naturally lets us express *native ZFA concepts*;[3] then the gain from ZFA can be useful, as we will consider.

DEFINITION 2.4. A **permutation** π is a bijection on $\mathbb{A}$. Define a **permutation action** $\pi{\cdot}\mathsf{a}$ by: $\pi{\cdot}\mathsf{a} = \pi(\mathsf{a})$ if $\mathsf{a} \in \mathbb{A}$ and $\pi{\cdot}\mathsf{a} = \{\pi{\cdot}\mathsf{b} \mid \mathsf{b} \in \mathsf{a}\}$ if $\mathsf{a} \notin \mathbb{A}$.

Then given a predicate ϕ in the language of set theory with atoms, define $\pi{\cdot\cdot}\phi$ to be that predicate obtained by replacing every free variable a with $\pi{\cdot}\mathsf{a}$.[4]

REMARK 2.5. $(\mathbf{Equivar})$ asserts that validity is preserved by permuting atoms in all parameters of a predicate. Thus: *atoms are distinguishable, but interchangable.*

For instance if we have proved $\phi(a, b, c)$ for atoms $a, b, c \in \mathbb{A}$ then taking $\pi = (a\, c)$ we have $\phi(c, b, a)$ and taking $\pi = (a\, a')(b\, b')(c\, c')$ we have $\phi(a', b', c')$. We do not have $\phi(a, b, a)$; this may still hold, but not by $(\mathbf{Equivar})$ because no permutation takes (a, b, c) to (a, b, a).

This gives atoms a dual nature. Individually atoms point to themselves,[5] but collectively atoms have the flavour of variables ranging permutatively over $\mathbb{A}$.[6]

REMARK 2.6. **Equivariance is a native ZFA concept**. If our intuitions are ZF-shaped, then equivariance seems counterintuitive: "We can't just permute elements. Suppose atoms are numbers: then are you claiming $1 < 2$ if and only if $2 < 1$?". ZFA makes clear what is going on: the premise "Suppose atoms are numbers" makes no sense, because atoms are *not* numbers!

3 Equivariance, choice, and freshness

π acts bijectively …

LEMMA 3.1. *Suppose $\mathfrak{M}$ is a model of ZFA(C). Then $\mathfrak{M} \vDash \pi{\cdot}y \in \pi{\cdot}x$ if and only if $\mathfrak{M} \vDash y \in x$.*

[1] An Equivariant Higher-Order Logic with Atoms and Choice would be equally feasible. I discuss sets rather than simple types only for convenience.

[2] …by modelling atoms as $\mathbb{N}$, or $\mathit{pset}(\mathbb{N})$, and so forth.

[3] …meaning concepts that are hard to address in full generality in ZF, where we do not have atoms, but easy to see in ZFA, where we do.

[4] Our syntax has just one constant $\mathbb{A}$. If we want more constants, we must define $\pi{\cdot\cdot}\phi$ sensibly. More on this in the full paper [7].

[5] In the Isabelle implementation of FM in my PhD thesis [4] this was literally so: I used *Quine atoms* such that $a = \{a\}$. This removes the condition $a, b \notin \mathbb{A}$ in $(\mathbf{Ext})$, at a cost of some extremely mild non-wellfoundedness.

[6] To see this made precise see Subsection 2.6 and Lemma 4.17 of [2].

. . . so a model of ZFA or ZFAC *already* satisfies $(\mathbf{Equivar})$ [4, Theorem 8.1.10]:

THEOREM 3.2. *If $\mathfrak{M}$ is a model of ZFA(C) then $\mathfrak{M}$ is also a model of EZFA(C).*

Proof. We prove $\mathfrak{M} \vDash \phi \Leftrightarrow \pi\cdot\cdot\phi$ by induction on ϕ. The case of $t \in s$ is Lemma 3.1. The cases of $\forall a.\phi$ and $\mathbb{A}$ use the fact that π is bijective. Other cases are no harder. □

REMARK 3.3. **Why do we need $(\mathbf{Equivar})$?** While Theorem 3.2 shows how instances $(\mathbf{Equivar})$ can be derived in ZFA, in practice the cost of proving them from first principles à la Theorem 3.2 scales with the complexity of the predicate ϕ. Instances of Theorem 3.2 can quadratically dominate development effort in a theorem-prover.[7] This is the problem of α-equivalence, come back to bite us. In contrast, axiom $(\mathbf{Equivar})$ costs *constant* effort: namely, the cost of invoking the axiom.[8] In this sense, equivariance is a *natural axiom*.

REMARK 3.4. **Choice compatible with equivariance.** Surely arbitrary choices are inherently non-equivariant? Not if they are made inside $\mathfrak{M}$. Consider some choice-function $f \in pset^*(x) \to x$. By Theorem 3.2 we immediately obtain $\pi\cdot f \in pset^*(\pi\cdot x) \to \pi\cdot x$. In words: if f is a choice function for x in $\mathfrak{M}$ then by equivariance $\pi\cdot f$ is a choice function for $\pi\cdot x$ in $\mathfrak{M}$. We just permute atoms pointwise in the choice functions.

So the following are consistent with EZFA and derivable in EZFAC: "There exists a total ordering on $\mathbb{A}$"; "Every set can be well-ordered (even if the set mention atoms)".

REMARK 3.5. FM set theory has a *finite support* property that for every x there exists $A \subseteq_{fin} \mathbb{A}$ such that if $\forall a{\in}A.\pi(a) = a$ then $\pi\cdot x = x$.

I argue that it is better to present freshness as a *well-behavedness* property in the larger EZFA(C) universe. This is for several reasons:

1. Presenting $(\mathbf{Fresh})$ as a well-behavedness property instead of an axiom eliminates the 'But FM is inconsistent with Choice' objection to nominal techniques. Anything we can do in FM, we can do in EZFAC by imposing finite support. Choice functions need not have finite support, but (unlike is the case for FM) they still exist in the same EZFAC universe.

2. We sometimes specifically want non-supported elements; for example two recent papers [5, 8] are concerned with sets that have a notion of nominal support, but whose elements do not.

3. Support is not a hereditary property; e.g. 'the set of all well-orderings of atoms' is supported by $\varnothing$, but no well-ordering of $\mathbb{A}$ has finite support; or put another way, the FM universe is a proper subclass of the universe of finitely-supported elements.

4. Even if the reader's next paper uses FM sets, it may be helpful for exposition to observe the natural embedding of the FM universe inside the EZFAC universe. Of course this embedding is obvious, but only *once it is pointed out*.

4 Conclusions

Nominal techniques have developed considerably since the original work [9] yet their foundations have not been critically revisited. Authors and implementors have generally used FM or ZF(C), if foundations are explicitly considered. Yet there is a sense in which FM is too strong, and ZF(C) is too weak. Though these theories are all biinterpretable, that is not enough. We need foundations and implementations that allow us express ourselves precisely, naturally, and without annoying, even crippling, proof-obligations to do with renamings.

In this respect EZFAC seems to have advantages. We have Choice, invoking equivariance has constant cost—and just the clear statement of EZFAC itself will I hope be conceptually useful.

References

[1] Mikołaj Bojańczyk, Bartek Klin, and Sławomir Lasota. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10, 2014.

[2] Gilles Dowek and Murdoch J. Gabbay. PNL to HOL: from the logic of nominal sets to the logic of higher-order functions. *Theoretical Computer Science*, 451:38–69, 2012.

[3] Maribel Fernández and Murdoch J. Gabbay. Nominal rewriting (journal version). *Information and Computation*, 205(6):917–965, June 2007.

[4] Murdoch J. Gabbay. *A Theory of Inductive Definitions with alpha-Equivalence*. PhD thesis, University of Cambridge, UK, March 2001.

[5] Murdoch J. Gabbay. Stone duality for First-Order Logic: a nominal approach. In *HOWARD-60. A Festschrift on the Occasion of Howard Barringer's 60th Birthday*, pages 178–209. Easychair books, 2014.

[6] Murdoch J. Gabbay. Consistency of Quine's New Foundations using nominal techniques. Submitted. See arXiv preprint arxiv.org/abs/1406.4060, 2016.

[7] Murdoch J. Gabbay. Equivariant ZFA and the foundations of nominal techniques. See arXiv preprint arxiv.org/abs/1801.09443, 2017.

[8] Murdoch J. Gabbay and Michael J. Gabbay. Representation and duality of the untyped lambda-calculus in nominal lattice and topological semantics, with a proof of topological completeness. *Annals of Pure and Applied Logic*, 168:501–621, March 2017.

[9] Murdoch J. Gabbay and Andrew M. Pitts. A New Approach to Abstract Syntax with Variable Binding. *Formal Aspects of Computing*, 13(3–5):341–363, July 2001.

[10] Dexter Kozen, Konstantinos Mamouras, and Alexandra Silva. Completeness and incompleteness in nominal Kleene algebra. *Journal of Logical and Algebraic Methods in Programming*, 2017.

[11] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, May 2013.

[12] Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 40(4):327–356, 2008.

[7] This happened in the Isabelle/FM implementation in my thesis [4], and it was crippling. After my PhD I initiated a mark 2 development with an $(\mathbf{Equivar})$ axiom-scheme (actually an Isabelle Oracle).

[8] The modern nominal Isabelle implementation uses automated tactics to prove Theorem 3.2 for certain classes of ϕ, specialised to an application to nominal inductive datatypes. Nominal Isabelle is good at what it does, but it is not (and never claimed to be) a universal nominal foundation.

BREU with Connections

Peter Backeman[1]

Uppsala University, Sweden

peter.backeman@it.uu.se

Abstract: Bounded Rigid *E*-Unification (BREU) is a unification method which has been used with sequent-style theorem proving. In comparison to more ordinary unification methods in this context, BREU works over finite domains of substitutions, thus reducing the complexity of the unification procedure. Earlier work introduced a sequent calculus which was shown to be sound and complete. Here, a tableau calculus is explored which works with *pseudo-clauses*. The connection restriction is added and investigations is made into effects on performance, as well as implications for completeness.

1 Introduction

When finding proofs using sequent/tableau calculus reasoning with rigid variables ("free" variables), one of the hardest steps is to find a substitution such that each branch of the proof tree is closed. Usually, a form of Simultaneous Rigid *E*-Unification (SREU) is required, where all branches are closed at the same time. However, the SREU-problem was famously found to be undecidable [3] and there is no decision procedure to be incorporated in a theorem prover. Several approaches to avoid using SREU exists, e.g., paramodulation. A modification to SREU has been proposed, Bounded Rigid *E*-Unification, which limits the domains of each free variable to a finite domain. In [2], it was shown that even though the substitutions were limited, a sound and complete proof calculus could nonetheless be constructed. In [1] efficient methods of solving the BREU-problem was described and a solver, *ePrincess*, using these in conjunction with a sequent proof calculus was shown to be potentially competitive.

Here the previous work is extended and a tableau calculus, based on BREU, is introduced using the connection restriction. Since this calculus does not use regular unification, previous results regarding the introduction of the connection restriction do not necessarily carry over, e.g., connection tableau using BREU is shown to be incomplete..

2 Bounded Rigid *E*-Unification

One method of handling universal quantifiers in tableaux proof search is by introducing so called "free variables". Already in the middle of the previous century this idea was introduced by Prawitz [5] (who called them "dummy variables"). Free variables acts as placeholders which are later to be substituted by terms such that a valid (closed) proof is obtained. When dealing with first-order logic without equality, one can perform syntactic unification to find substitutions such that complementary literals are unified. However, if equality literals are also in the tree, then unification modulo equality needs to be performed.

Traditionally, one would perform several unification problems in parallel, one *E*-unification for each branch of the proof tree which is to be closed. This is what is known as Simultaneous Rigid *E*-Unification (where the variables are rigid, i.e., they must be replaced by the same term in all branches). SREU was famously shown to be undecidable in the general case [3]. Therefore a modified version of SREU was introduced which limits the possible substitutions of a free variables, by only allowing it to be substituted terms appearing "above" in the proof tree. This was claimed to be sufficient for a complete and sound proof calculus already by Kanger [4] and also proved in [2].

A sequent calculus was described in [2] which uses BREU for closing proof trees and was shown to be sound and complete. Its performance was also shown in [1] to have potential.

3 Connection Tableaux

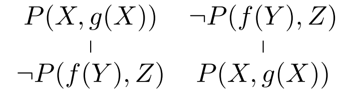
In this work a tableau calculus which utilizes BREU is investigated. Many restrictions of the search space is possible for tableau methods. One popular, and empirically successful, such restriction is by using connection tableaux. Connection tableaux are proof tableaux with the connection restriction, which can be formulated in a tableaux calculus as only allowing proof expansion steps where one branch is immediately closed.

There is much research within the connection tableau field and it is here applied to a tableau calculus with BREU. Some of the properties of connection tableaux using regular unification translates over to the bounded case (e.g., soundness) while other do not. For example, regular connection tableaux are complete, i.e., even though only a proof search is made over connection tableaux; there is always at least one such tableaux if the original formula is indeed valid. This is not the case for this "bounded" tableaux.

Example Consider the the following two clauses:

$$\{\forall x.P(x,g(x)), \forall y,z.\neg P(f(y),z)\}$$

There are only two possible connection tableaux:



(a) *(b)*

The substitution to close the left (a) tableau is $\sigma_a = \{X \mapsto f(Y), Z \mapsto g(X)\}$. However, the therm $f(Y)$ only occurs after X has been introduced to the tableau and thus $X \mapsto f(Y)$ will not be found by a BREU procedure. For the right (b) tableau, it is the same case with $Z \mapsto g(X)$.

This and other properties are investigated to see if any relaxations can be made to still have completeness, or if it is possible to find heuristics for when the connection restriction is useful and when not.

4 Pseudo-Clausal Tableaux

Traditionally, when working with connection tableaux, *clausal tableaux* is used. That is, the starting formula is given on CNF and there is one expansion rule which consists of taking one conjunct and appending each literal as a new branch to a leaf, in such a way that at least one of the new branches are immediately closed. This means that each node will have multiple children, except when it has been expanded by a unit clause.

However, when using BREU in conjunction with the connection restriction, *Pseudo-Clausal Tableaux* is used instead. Briefly, a *pseudo-literal* is a conjunction of n literals, where $n-1$ of the literals are equational literals and one literal is a predicate or equational literal. A *pseudo-clause* is a disjunction of pseudo-literals, possibly with a leading quantifier prefix. Finally, a formula is on *pseudo-clausal normal form* (PCNF) if it is a conjunction of pseudo-clauses. In tableau expansion steps, each pseudo-literal is treated as a single literal. This allows us to rewrite equations to a normal form without explosion of proof size.

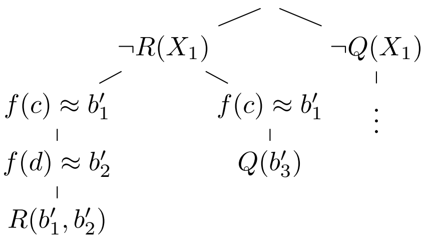
Example Consider the set of clauses:

$$\{\forall x(\neg R(x) \lor \neg Q(x)), R(f(c), f(d)) \lor Q(f(c))\}$$

It is represented by the following set of pseudo-clauses:

$$\{\forall x(\neg R(x) \lor \neg Q(x)),$$
$$\exists b_1, b_2(f(c) \approx b_1 \land f(d) \approx b_2 \land R(b_1, b_2))\lor$$
$$(f(c) \approx b_1 \land Q(b_1))\}$$

If the first and then the second clause would be used in a tableau expansion step, the results would look as follows:



5 Proof Search

One important aspect of proof search over (pseudo-)clausal tableaux is how to enumerate all the possible proofs, i.e., both how to apply the next rule but also when to backtrack. Usually, for the latter some kind of bound is employed, stating how deep the search procedure goes before going back and consider other choices. This bound is then iteratively increased in some manner, thus the search space is systematically explored. It is not clear how this is to be implemented when using bounded unification, since when the unification is restricted, there might be a requirement for longer branches in some certain scenarios, and perhaps this can be exploited when setting the bounds.

6 Other Improvements

There are other interesting improvements to add to the tableau calculus. Normally, a call to a unification procedure yields either a substitution or returns that no substitution exists. An interesting extension is to also return auxiliary information found during the unification procedure.

6.1 Propagating Unit Clauses

In the lazy method for solving BREU described in [1], blocking clauses are generated to prohibit the generation of candidate solution in a lazy manner. These blocking clauses will also hold for certain branches of the current proof tree. Especially unit clauses are of interest since those corresponds to literals which can be appended to branches in the proof tree and facilitate the proof search.

7 Conclusions

In this work new tableau based calculus utilizing BREU is introduced. It is seen to be incomplete but there are possibilities for interesting improvements. This is a work in progress and no interesting experimental data exists yet.

References

[1] P Backeman and P Rümmer. Efficient algorithms for bounded rigid e-unification. In *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer International Publishing, 2015.

[2] P Backeman and P Rümmer. Theorem proving with bounded rigid e-unification. In *Automated Deduction - CADE-25*. Springer International Publishing, 2015.

[3] Anatoli Degtyarev and Andrei Voronkov. Simultaneous rigid E-Unification is undecidable. In *CSL*, 1995.

[4] S. Kanger. A simplified proof method for elementary logic. In *Automation of Reasoning 1: Classical Papers on Computational Logic 1957-1966*. Springer, Berlin, Heidelberg, 1983. Originally appeared in 1963.

[5] Dag Prawtiz. An improved proof procedure. *Theoria*, 26(2), 1960.

Designing a proof calculus for the application of learned search heuristics

Michael Rawson Giles Reger

University of Manchester, UK

Abstract: Machine learning techniques may prove capable of efficiently guiding search for proofs of human theorems. Many modern automated theorem provers (e.g. E [10], SPASS [13], iProver [4], Vampire [5]) make use of proof calculi such as saturation or instance generation which, while efficient, are a bad fit for current machine-learning technology. We propose the use of a type-theoretic calculus for a future theorem prover which attempts to leverage machine-learning techniques.

1 Problems inherent to state-of-the-art calculi

Machine-learning techniques have been most successful in applications where a human operator finds a task easy or intuitive, but algorithms are inefficient or ad-hoc — see recent progress on image classification [6], sentiment analysis [9], and board games [11]. Proof search could also fall into this category: while modern provers are feats of engineering and efficiency, they may still choke on problems which an intelligent agent (i.e. a mathematician) could solve easily by intuition, and the field as a whole is dominated by approaches and heuristics which aim to solve a few new problems. However, integrating learned heuristics to directly guide proof search into these provers is non-trivial. Consider the archetypal saturation-based prover for first-order logic. We identify several challenges facing an intrepid machine-learning practitioner when implementing such a feature.

- Efficiency: modern provers on modern hardware process thousands or millions of (possibly inter-dependent) search steps per second, and as such are extremely sensitive to latency. It would be impractical to add even a few milliseconds of processor time to each step.
- Size of the input space: saturation-based provers may at times generate sufficient clauses to fill the entire memory of a machine. Such a dataset could not be processed by a heuristic in reasonable time.
- Size of the output space: in a saturation step, a clause is selected for resolution with other clauses in an algorithm such as DISCOUNT [1] or OTTER [7]. Current machine-learning techniques work most efficiently with a smaller, fixed output space.
- Heuristic performance: while human provers can gain intuition for human-readable proofs, the artificial processes of negation elimination, skolemisation, clausification and subsequent resolution steps can make the next step unclear. There is therefore no reason to expect a machine to do any better.

We assert that these problems are inherent to most (if not all) of the various calculi employed by modern systems, with the possible exception of tableaux-based provers.

2 Desired properties of a new calculus

With this in mind, a freshly-designed calculus would ideally satisfy the following requirements (in increasing order of importance) in order to be more suitable for machine-learning methods.

1. The calculus should maintain as much of the human intuition behind the (sub-)goal as possible (e.g. the structure of the goal) to allow learning to take place.
2. The calculus should admit proofs of hard problems in a reasonable number of steps. Otherwise, the inevitable performance impact of running the heuristic will mean that a proof is not found quickly.
3. At a given step in proof search, the calculus should have a small input state to consider, and a finite number of next steps which is as small as possible.
4. The calculus should have terms which are amenable to representation in machine-learning algorithms. Representations such as fixed-size real-valued vectors, character sequences, or graphs have all been used for other tasks with differing degrees of success.
5. A calculus should come with an associated method to quantify, in some way, which of the possible next directions is preferable. This might involve a way of scoring “progress” toward a proof, or a means of filtering for relevant premises.

3 Proposal for a calculus that is steerable using machine learning

There is a well-known [12] isomorphism between type theories and logic. In particular, a logical proposition may be seen as a type, with proof search then the problem of finding a term which has that type (*term synthesis*). For example, the simply-typed λ-calculus is sufficient to show the principle of modus ponens

$$P \implies (P \implies Q) \implies Q$$

with the term

$$\lambda x : (P).\lambda f : (P \to Q).\ (f\ x)$$

A classical first-order logic can be produced in the type theory by generalising the function arrow $\to$ to a dependent product, as in the LF system [3] and adding an operator for classical contradiction. To produce an iterative system for finding proofs one can introduce "holes" (as found in e.g. Agda [8]) to the calculus to represent sub-goals. Proof terms can then be produced iteratively by filling holes in the term such that they satisfy the target type at all times. Search is complete when the term has no holes remaining:

Term	Type
$?$	$?$
$\lambda x : (P).?$	$P \to ?$
$\lambda x : (P).\lambda f : (P \to Q).?$	$P \to (P \to Q) \to ?$
$\lambda x : (P).\lambda f : (P \to Q).\,(?\ ?)$	$P \to (P \to Q) \to ?$
$\lambda x : (P).\lambda f : (P \to Q).\,(f\ ?)$	$P \to (P \to Q) \to\ Q$
$\lambda x : (P).\lambda f : (P \to Q).\,(f\ x)$	$P \to (P \to Q) \to\ Q$

Using such a type theory as a logical calculus has several of the properties presented earlier, without many disadvantages. Experience in the interactive theorem-proving community suggest that dependent type theories work reasonably well for human users, as seen with large developments in Agda, and hence satisfy requirements 1 and 2. Additionally, types act as a form of restriction, and also direction to an extent: only some variables can be used to fill a hole, since otherwise the term would be ill-typed. This partially satisfies requirements 3 and 5.

Unfortunately, there is no evidence to suggest that this technique improves on other calculi in solving the problem of representation (i.e. requirement 4). However, it does reduce the complexity of terms by cutting the number of logical connectives. It may also be possible to view types as a directed graph, with function arrows as edges — we leave this as future work.

4 Possible implementation techniques

Several type-based ITP systems already contain automation for small theorems, such as the *Agsy* term search implementation provided with Agda [8]. These algorithms could be extended or directed with a learned heuristic. Another promising direction involves a tree-search technique used by AlphaGo [11] (and more recently LeanCOP [2]), which balances exploration and exploitation: Monte-Carlo Tree Search. MCTS is particularly appealing as it is both highly parallel and resumable.

5 Summary

We identify problems with state-of-the-art proof calculi in the context of machine guidance, establish a set of desired requirements for a replacement calculus, and propose a type-theoretic calculus for this rôle. Our next step will be to implement a proof-of-example solver for this calculus (this is already in-progress) and use this to explore various machine learning methods for steering proof search.

References

[1] Jürgen Avenhaus, Jörg Denzinger, and Matthias Fuchs. DISCOUNT: A system for distributed equational deduction. In *RTA*, pages 397–402. Springer, 1995.

[2] Michael Färber, Cezary Kaliszyk, and Josef Urban. Monte Carlo tableau proof search. In *CADE*, pages 563–579. Springer, 2017.

[3] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[4] K. Korovin. iProver – an instantiation-based theorem prover for first-order logic. In *IJCAR 2008*, volume 5195, pages 292–298. Springer, 2008.

[5] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV*, pages 1–35. Springer, 2013.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[7] William W. McCune. Otter 3.0 reference manual and guide. Technical report, Argonne National Laboratory, 1994.

[8] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[9] Soujanya Poria, Erik Cambria, and Alexander Gelbukh. Aspect extraction for opinion mining with a deep convolutional neural network. *Knowledge-Based Systems*, 108:42–49, 2016.

[10] Stephan Schulz. E — a brainiac theorem prover. *AI Communications*, 15(2, 3):111–126, 2002.

[11] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[12] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard isomorphism*, volume 149. Elsevier, 2006.

[13] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *CADE*, volume 5663, pages 140–145. Springer, 2009.

Contradiction Separation Based Dynamic Multi-Clause Synergized Automated Deduction

Yang XU^1, Jun LIU^{2,1}, Shuwei CHEN^1, Xiaomei ZHONG^1, Xingxing HE^1

^{1} National-Local Joint Engineering Laboratory of System Credibility Automatic Verification, Southwest Jiaotong University, Chengdu 610031, China
xuyang@home.swjtu.edu.cn

^{2} School of Computing, Ulster University, Northern Ireland, UK
j.liu@ulster.ac.uk

Abstract: This extend abstract briefly introduces our recent research work, i.e., a novel sound and complete contradiction separation (CS) based dynamic multi-clause synergized automated deduction theory, targeted for dynamic and multiple (two or more) clauses handling in a synergized way, while binary resolution is its special case. It includes some basic concepts, the key idea illustration, and a summary about what we have achieved based on this new theory in terms of implementation and experimental studies.

1 Introduction

Resolution [1] has played a key role in automated reasoning for over five decades. Our recent work aims at addressing the following questions: although the simple and elegant binary resolution has been successful, has it been too restrictive? Instead of treating a contradiction as a complementary pair based on two clauses, can we extend it into a contradiction consisting of more than two clauses? Accordingly, can we make a flexible and dynamic selection of the number of clauses involved in each deduction to get better efficiency and capability?

Based on some existing work in the literature plus our previous research work on resolution-based automated deduction based on many-valued logic [2], we recently proposed a CS based dynamic multi-clause synergized automated deduction [3, 4], with the aim at addressing the above questions and achieving the following distinctive features: 1) *multi-clause deduction*: multiple clauses from a clause set are involved in each deduction process; 2) *dynamic and flexible deduction*: the number of clauses involved in the CS in each deduction process is dynamic and flexible; 3) *synergized deduction*: joint handling of multiple clauses and achieving the synergized effects of all the clauses on the deduction result; 4) *guided deduction*: the guided search path to reduce the search space; 5) *robust deduction*: deleting or adding some literals in the clause set will not affect the contradiction and the deduction results; 6) *paralleled deduction*: each clause involved in the contradiction has the equal status and the ordering of each clause appearing in the clause set will not affect if the clause set is a contradiction or not. The sections below give some brief introduction about this theory.

2 CS Based Deduction in Propositional Logic

Definition 2.1 (Contradiction) Let $F = \{C_1, \cdots, C_m\}$ be a clause set. If $\forall (p_1, \cdots, p_m) \in \prod_{i=1}^m C_i$ (the set of all ordered tuples $(p_1, \cdots, p_m)$ such that $p_i \in C_i$), there exists at least one complementary pair among $\{p_1, \cdots, p_m\}$, then $F = \bigwedge_{i=1}^m C_i$ is called a standard contradiction (in short, SC). If $\bigwedge_{i=1}^m C_i$ is unsatisfiable, then $F = \bigwedge_{i=1}^m C_i$ is called a *quasi-contradiction* (in short, QC). In propositional logic, these two concepts are equivalent, but not in the first-order logic case.

Definition 2.2 (CS rule) Assume a clause set $F = \{C_1, \cdots, C_m\}$. The following inference rule that produces a new clause from F is called a *contradiction separation rule*, in short, a CS rule: For each $C_i (i = 1, \cdots, m)$, separate it into two subclauses C_i^- and C_i^+ such that: 1) $C_i = C_i^- \bigvee C_i^+$, where C_i^- and C_i^+ have no common literals; 2) C_i^+ can be an empty clause itself, but C_i^- cannot be an empty clause; 3) $\bigwedge_{i=1}^m C_i^-$ is a standard contradiction. The resulting clause $\bigvee C_i^+$, denoted as $\mathcal{C}_m = (C_1, C_2, \cdots, C_m)$, is called a *contradiction separation clause* (CSC) of $C_1, \cdots, C_m$, and $\bigwedge_{i=1}^m C_i^-$ is called a *separated contradiction* (SC).

Remark 2.1 binary resolution rule is actually a special case of the CS rule when only two clauses are involved in the CS process.

Definition 2.2 Suppose a clause set $F = \{C_1, \cdots, C_m\}$ in propositional logic. $\Phi_1, \cdots, \Phi_t$ is called a a CS based *dynamic deduction sequence* from F to a clause Φ_t, denoted as $\mathcal{D}$, if (1) $\Phi_i \in F$, $i = 1, \cdots, t$; or (2) there exist $r_1, r_2, \cdots, r_{k_i} < i$, $\Phi_i = \mathcal{C}_{k_i}(\Phi_{r_1}, \cdots, \Phi_{r_{k_i}})$.

Remark 2.2 the k_i in (2) varies with the deduction process, this reflects “dynamic deduction”.

Example 2.1 Suppose a clause set $F = \{C_1, C_2, \cdots, C_{13}\}$ in propositional logic with: $C_1 :\sim p_4 \vee p_6$, $C_2 : p_6 \vee \sim p_7$, $C_3 :\sim p_6 \vee p_7$, $C_4 :\sim p_6 \vee \sim p_7$, $C_5 : p_1 \vee p_2 \vee p_3$, $C_6 : p_1 \vee p_2 \vee \sim p_3$, $C_7 :\sim p_1 \vee p_2 \vee p_3$, $C_8 :\sim p_1 \vee \sim p_2 \vee p_3$, $C_9 :\sim p_1 \vee \sim p_2 \vee \sim p_3$, $C_{10} : p_4 \vee \sim p_5 \vee p_7$, $C_{11} : p_1 \vee \sim p_2 \vee p_3 \vee p_4$, $C_{12} : p_1 \vee \sim p_2 \vee \sim p_3 \vee p_5$, $C_{13} :\sim p_1 \vee p_2 \vee \sim p_3 \vee p_6$. Using the CS rule for the clauses $C_5, C_6, C_7, C_8, C_9, C_{11}, C_{12}, C_{13}$, we obtain a CSC involving 8 clauses: $C_{14} = \mathcal{C}_8(C_5, C_6, C_7, C_8, C_9, C_{11}, C_{12}, C_{13}) = p_4 \vee p_5 \vee p_6$. Furthermore, using the CS rule for 3 clauses C_1, C_{10} and C_{14}, we obtain another CSC involving 3 clauses: $C_{15} = \mathcal{C}_3(C_1, C_{10}, C_{14}) = p_6 \vee p_7$. Finally, we have $C_{16} = \mathcal{C}_4(C_2, C_3, C_4, C_{15}) = \emptyset$. This process illustrates a CS based dynamic deduction from F to an empty clause $\emptyset$ using 3 steps of CS deduction.

More examples can be found in [4], and it has also been proved in [4] that the CS-Based Dynamic Deduction in Propositional Logic is sound and complete.

3 CS Based Dynamic Deduction in First-Order Logic

Definition 3.1 (S-CS Rule) Suppose a clause set $F = \{C_1, \cdots, C_m\}$ in first-order logic. Without loss of generality, assume that there does not exist the same variables among $C_1, \cdots, C_m$ (if the same variables appear, there exists a rename substitution which makes them different). The following inference rule that produces a new clause from F is called a *standard* *contradiction separation rule*, in short, an S-CS rule:

For each $C_i(i = 1, \cdots, m)$, firstly apply a substitution σ_i to C_i (σ_i could be an empty substitution but not necessary the most general unifier), denoted as $C_i^{\sigma_i}$; then separate $C_i^{\sigma_i}$ into two sub-clauses $C_i^{\sigma_i^-}$ and $C_i^{\sigma_i^+}$ such that i) $C_i^{\sigma_i} = C_i^{\sigma_i^-} \vee C_i^{\sigma_i^+}$, where $C_i^{\sigma_i^-}$ and $C_i^{\sigma_i^+}$ have no common literals; ii) $C_i^{\sigma_i^+}$ can be an empty clause itself, but $C_i^{\sigma_i^-}$ cannot be an empty clause; iii) $\bigwedge_{i=1}^{m} C_i^{\sigma_i^-}$ is a standard contradiction, that is $\forall(x_1, \cdots, x_m) \in \prod_{i=1}^{m} C_i^{\sigma_i^-}$, there exists at least one complementary pair among $\{x_1, \cdots, x_m\}$. The resulting clause $\bigvee_{i=1}^{m} C_i^{\sigma_i^+}$, denoted as $\mathcal{C}_m^{s\sigma}(C_1, \cdots, C_m)$ (here "s" means "standard", $\sigma = \bigcup_{i=1}^{m} \sigma_i$, σ_i is a substitution to $C_i, i = 1, \cdots, m$), is called a *standard contradiction separation clause* (S-CSC) of $C_1, \cdots, C_m$, and $\bigwedge_{i=1}^{m} C_i^{\sigma_i^-}$ is called a *separated standard contradiction* (S-SC).

The CS based dynamic deduction in first-order logic is defined similarly. It has been proved in [4] that the CS-Based Dynamic Deduction in first-order Logic is also sound and complete.

4 Graphical Illustration of the Key Ideas

This section provides a graphical illustration on the essential features of the CS-based dynamic deduction, as well as the essential difference from binary resolution deduction (Figs 1 and 2). We use the funnel as an intuitive figure to show the automated deduction process from the input clause set. The one coming out from the exit of the funnel is the final output.

Fig. 1 actually also illustrates some insights why the pre-processing and simplification steps are essential in the binary resolution deduction, even take the majority of the steps and time; and also why lots of work have been focused on splitting and simplifying the clause set into the simpler ones just because the exit is too narrow. Fig. 2 illustrates the dynamic and flexible nature of the CS-based dynamic deduction, which reflects the reflects non-determinism, essentially opens multiple paths by which the outcome may be discovered.

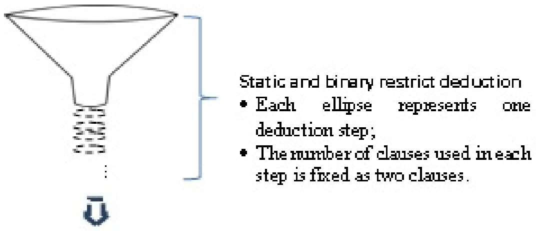

Static and binary restrict deduction
• Each ellipse represents one deduction step;
• The number of clauses used in each step is fixed as two clauses.

Figure 1: The graphical funnel view of binary resolution deduction

5 Summary of Work Done Based on the CS theory

This established theory is just a first step towards an effective CS-based theorem prover, which will need specific algorithms and strategies (including indexing techniques) making the "right" single CS step including the "suitable selection" of the number of clauses to be involved in each deduction process. Although it is challenging, the authors team has done a lot of work already in

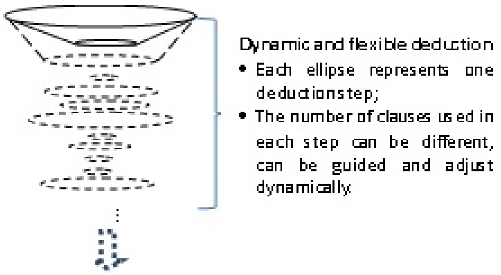

Dynamic and flexible deduction
• Each ellipse represents one deductionstep;
• The number of clauses used in each step can be different, can be guided and adjust dynamically.

Figure 2: The graphical funnel view of CS-based dynamic deduction

this direction along with some SAT solvers and first-order automated deduction systems based on the CS deduction, e.g., [5]. So far, over 2.2 billion SAT problems has been solved, the maximal scale includes 2.5 million clauses, and 3.13 trillion literals. In addition, over 250,000 problems from TPTP [6] and Mizar [7] database have been proved. MC-SCS does not include the superposition calculus yet, have then coupled with other provers, such as Prover 9, E prover and Vampire. So far, 130 Rating-1 problems and 14 unknown problems from TPTP have been proved by extending E/Prover9/Vampire) based on the CS based automated deduction. Currently we are working on extensive and deeper experimental studies and comparative analysis with the state of art based on the benchmark problem; new and better CS-based search algorithms and strategies, forward and backward deduction, complexity analysis as well as the real application etc.

References

[1] J.A. Robinson, A machine oriented logic based on the resolution principle, J. ACM, 12(1)(1965), pp. 23-41.

[2] Y. Xu, J. Liu, X.M. Zhong, and S.W. Chen, Multi-ary alpha-resolution principle for a lattice-valued logic, IEEE Trans. on Fuzzy Systems, 21(5) 2013, pp. 898-912.

[3] Y. Xu, J. Liu, S.W. Chen, X.M. Zhong, A novel generalization of resolution principle for automated deduction, FLINS2016, Aug. 24-26, 2016, Roubaix, France.

[4] Y. Xu, J. Liu, S.W. Chen, X.M. Zhong, X.X. He, Contradiction separation based dynamic multi-clause synergized automated deduction, Submitted to Information Science under review.

[5] J. Zhong, Y. Xu, J. Liu, X.X. He, Q.H. Liu, S.W. Chen, X.M. Zhong, G.F. Wu, X.R. Ning, X.D. Guan, Q.S. Chen, Z.M. Song, Multi-clause synergized CS based first-order theorem prover MC-SCS, ISKE2017, Nanjing, China, Nov. 24-26, 2017, pp. 453-458.

[6] G. Sutcliffe. The TPTP problem library and associated infrastructure: the FOF and CNF parts, v3.5.0. Journal of AR, 43(4)(2009), pp. 337-362.

[7] C. Kaliszyk and J. Urban, MizAR 40 for Mizar 40. CoRR, abs/1310.2805 (2013).

Reasoning with Large Theories Using Over-Approximation Abstraction-Refinement

Julio Cesar Lopez Hernandez Konstantin Korovin

The University of Manchester, School of Computer Science
{lopezhej,korovin}@cs.man.ac.uk

Abstract: We present the results obtained from experimenting with our implementation of the abstraction-refinement framework for reasoning with large theories. Particularly the results obtained from the over-approximation process.

1 Introduction

Efficient reasoning with large theories is one of the main challenges in automated theorem proving (ATP). This problem arises because of the enormous number of superfluous premises in the theories and usually a few of them are needed to prove a conjecture. Therefore it is desirable to select the most relevant axioms when proving a conjecture.

Currently, we are working on defining and formalising a framework based on abstraction-refinement [2, 1, 5, 8] for reasoning with large theories. Our approach encompasses two approximations: the under and over approximations [4]. We present in this abstract our current results of implementing the over-approximation part. We experimented with different over-approximating abstractions, which have been defined in the framework.

2 Over-Approximation

2.1 Preliminaries

Let us consider a set of formulas $\mathcal{F}$ which we call a *concrete domain* and a set of formulas $\hat{\mathcal{F}}$ which we will call an *abstract domain*.

An *abstraction function* is a mapping $\alpha : \mathcal{F} \mapsto \hat{\mathcal{F}}$. When there is no ambiguity we will call an abstraction function just an abstraction of $\mathcal{F}$. The identity function is an abstraction which will be called the *identity abstraction* α_{id}.

A *concretisation function* for α is a mapping $\gamma : \hat{\mathcal{F}} \mapsto 2^{\mathcal{F}}$ such that $F \in \gamma(\alpha(F))$ for all $F \in \mathcal{F}$.

An abstraction α is called *over-approximation abstraction* (wrt. refutation) if for every $F \in \mathcal{F}$, $F \models \bot$ implies $\alpha(F) \models \bot$. We can compose abstractions as mappings. In particular, if $\alpha_1 : \mathcal{F} \mapsto \mathcal{F}_1$ and $\alpha_2 : \mathcal{F}_1 \mapsto \mathcal{F}_2$ then $\alpha_1\alpha_2$ is an abstraction of $\mathcal{F}$. The composition of over-approximating abstractions is an over-approximating abstraction. In further sections, we will define several atomic abstractions which can be composed to obtain a large range of combined abstractions.

We define an ordering on abstractions $\sqsubseteq$ called *abstraction refinement ordering* as follows: $\alpha \sqsubseteq \alpha'$ if for all $F \in \mathcal{F}$, $\alpha(F) \models \bot$ implies $\alpha'(F) \models \bot$. We have that all over-approximating abstractions are above the identity abstraction wrt. the abstraction refinement ordering.

Weakening abstraction refinement of an over-approximating abstraction α is an abstraction α' which is below α and above the identity abstraction wrt. to the abstraction refinement ordering, i.e, $\alpha_{id} \sqsubseteq \alpha' \sqsubseteq \alpha$.

An *over-approximation abstraction-refinement process* is a possibly infinite sequence of weakening abstraction refinements $\alpha_0, \ldots, \alpha_n, \ldots$ such that $\alpha_{id} \sqsubseteq \cdots \sqsubseteq \alpha_n \sqsubseteq \cdots \sqsubseteq \alpha_0$.

2.2 Over-Approximation Procedure

We consider a theory A which is a collection of axioms which we call *concrete axioms* and a set of formulas $\hat{A}^s$ called *abstract axioms*. We will assume that the negation of the conjecture is included in A, so proving the conjecture corresponds to proving unsatisfiability of A.

The *over-approximating procedure* starts with the following steps: it takes a set of concrete axioms A and then applies an over-approximating abstraction function α_s to A, in order to obtain their abstract representation $\hat{A}^s$, $\hat{A}^s = \alpha_s(A)$; we assume that reasoning with abstract axioms simplifies the reasoning process. The over-approximating procedure uses an ATP to try to prove that $\hat{A}^s$ is unsatisfiable. First, if ATP shows satisfiability of $\hat{A}^s$ then we can conclude that A is satisfiable, i.e., the conjecture is disproved. On the other hand, if the ATP proves the unsatisfiability of $\hat{A}^s$, then the procedure extracts and concretises the unsat core $\hat{A}^s_{uc}$ from $\hat{A}^s$, $A_{uc} = \gamma_s(\hat{A}^s_{uc})$. Next, the procedure tries to prove the unsatisfiability of A_{uc}. If A_{uc} is unsatisfiable, the process stops as this proves unsatisfiability of A. Otherwise, if A_{uc} is shown to be satisfiable, the set of axioms A is abstracted using a new abstraction α'_s obtained by weakening abstraction refinement of α_s. The procedure is repeated utilising the refined set of abstract axioms. This loop finishes when the conjecture is proved or disproved or the time limit of the whole procedure is reached.

In the next sections, we present several concrete over-approximating abstractions and their refinements.

2.3 Subsumption-Based Abstraction

The subsumption-based abstraction works by partitioning the set of concrete axioms based on joint literals occurrences and then it assigns an abstract clause that represents each partition. This abstract clause subsumes all clauses in the collection. The refinement process of this abstraction subpartitions one of the previous collections by selecting a new joint literal occurrence. Then the process adds this

Table 1: Were **subs** stands for subsumption-based, **sig** for grouping signature, **arg-fil** for argument filter, **FP** for grouping functions and predicates, **SC** for grouping Skolem functors and constants, and **SS** for argument filter restricted to Skolem and split symbols in iProver.

Depth	**Tolerance**	**Abstractions**	**Signature**	**Arg-filter**	**Until SAT**	**Solutions**
1	1.0	subs, sig, arg-fil	FP	SS	false	957
1	1.0	subs, sig, arg-fil	SC	default	false	38
2	1.0	subs		default	true	27
1	1.0	subs, sig, arg-fil	FP	default	true	11
1	1.0	subs, sig, arg-fil	FP	default	false	8
2	1.0	subs		default	false	2
1	1.0	subs, sig, arg-fil	SC	default	true	1
					Total	1044

Table 2: CASC-26 LTB comparison (solutions out of 1500 problems)

Vampire LTB-4.0	**Vampire LTB-4.2**	**MaLARea**	**iProver-v2.7**	**iProver LTB-2.6**	**E LTB**
1156	1144	1131	1070	777	683

literal to the abstract clause. Thus, each subpartition is represented by an abstract clause which extends the previous abstract clause.

2.4 Argument Filter Abstraction

The argument filter abstraction is based on removing certain arguments in the signature symbols and its refinement consists on restoring arguments of abstract symbols occurring in the abstract proof. This abstraction can be used to abstract variable dependencies by restricting it to the split predicates, which represents variable dependencies between different subclauses. In the same way, we can target formula definitions introduced during clausification and Skolem functions.

2.5 Grouping Signature Abstraction

The grouping signature abstraction abstracts the signature by grouping symbols of the same type and then for each group it assigns an abstract symbol. The refinement process of this abstraction concretises the abstract symbols which are present in the abstract proof. Our current implementation can group symbols in two different ways: i) one is based on grouping Skolem functors and constants, ii) the other one is based on grouping functions and grouping predicates with matching arities.

3 Experiments

We implemented the above over-approximating abstractions with their refinements and integrated them into iProver v2.7. These over-approximations are combined with SInE algorithm [3], which is currently used as an under-approximation.

We evaluated our implementation on the standard benchmark for first-order theorems provers: the TPTP library [7] with the set of problems from the LTB category in CASC-26 [6]. All experiments described in this section were performed using a cluster of computers with the following characteristics: Linux v3.13, cpu 3.1GHz and memory 125GB. We used a time limit of 240s for each attempt to solve a problem.

We experimented with different strategies, which encompass: the over-approximating abstractions defined above, their combinations, parameters for SInE algorithm (depth and tolerance) and iProver option to refine the abstraction function until the abstract unsat core becomes satisfiable. The total number obtained of solutions was 1070 out of 1500. In table 1, we show the number of solutions obtained from the most effective strategies which in total solved 1044 problems. In this table, the column ’Solutions’ shows the number of solutions found by certain strategy but excluding the problems solved by the previous strategies.

4 Conclusion

In table 2, we compared the results from CASC-26 and our current implementation with iProver v2.7. From these results, we can conclude that our proposed approach for reasoning with large theories considerably improves the performance of iProver, getting close to the top systems Vampire and MaLARea.

References

[1] M. P. Bonacina, C. A. Lynch, and L. de Moura. On Deciding Satisfiability by Theorem Proving with Speculative Inferences. *Journal of Automated Reasoning*, 47(2):161–189, 2011.

[2] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5):1512–1542, 1994.

[3] L. Kovács and A. Voronkov. First-order theorem proving and VAMPIRE. *LNCS*, 8044 LNCS:1–35, 2013.

[4] J. C. Lopez Hernandez and K. Korovin. Towards an Abstraction-Refinement Framework for Reasoning with Large Theories. In *IWIL Workshop*, volume 1, pages 119–123, 2017.

[5] D. A. Plaisted. Theorem proving with abstraction. *Artificial Intelligence*, 16(1):47–108, 1981.

[6] G. Sutcliffe. The CADE ATP System Competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[7] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[8] A. Teucke and C. Weidenbach. First-Order Logic Theorem Proving and Model Building via Approximation and Instantiation. In *Frontiers of Combining Systems*, pages 85–100, Cham, 2015.