IJCAR 2008

4th International Joint Conference on Automated Reasoning

Sydney, Australia, August 10–15, 2008

Workshop Program

Practical Aspects of Automated Reasoning – PAAR-2008/ESHOL-2008

Boris Konev Renate Schmidt Stephan Schulz
(Chairs)

WS 2 – August 10/11

PAAR-2008/ESHOL-2008
Practical Aspects of Automated Reasoning and Evaluation of Systems for Higher Order Logic

Boris Konev[1], Renate A. Schmidt[2], and Stephan Schulz[3]

[1] University of Liverpool
[2] University of Manchester
[3] TU München

1 Introduction

The first Workshop on Practical Aspects of Automated Reasoning was held on August 10–11, 2008, in Sydney, Australia, in association with the 4th International Joint Conference on Automated Reasoning (IJCAR-2008). It was held jointly with the ESHOL Workshop.

PAAR provides a forum for developers of automated reasoning tools to discuss and compare different implementation techniques, and for users to discuss and communicate their applications and requirements. The workshop brought together different groups to concentrate on practical aspects of the implementation and application of automated reasoning tools. It allowed researchers to present their work in progress, and to discuss new implementation techniques and applications.

Topics included were:

- automated reasoning in classical and non-classical logics, implementation of provers;
- automated reasoning tools for all kinds of practical problems and applications;
- practical experiences, case studies, feasibility studies;
- evaluation of implementation techniques and automated reasoning tools;
- benchmarking approaches;
- non-standard approaches to automated reasoning, non-standard forms of automated reasoning, new applications;
- implementation techniques, optimisation techniques, strategies and heuristics;
- system descriptions and demos.

We were particularly interested in contributions that help the community to understand how to build useful and powerful reasoning systems in practice, and how to apply existing systems to real problems.

2 Workshop Programme

The workshop programme included 3 invited talks, 10 regular papers, 10 ESHOL presentations and demonstrations of higher-order logic systems and a panel discussion on the *Evaluation of Systems of Higher-Order Logic*. The invited talks were:

- *Software Model Checking: New Challenges and Opportunities for Automated Reasoning* by Alessandro Armando,
- *Mechanized Reasoning for Continuous Problem Domains* by Rob Arthan.
- *Constraint Modelling: A Challenge for First Order Automated Reasoning* by John Slaney.

The contributed papers were selected with the help of the programme committee from 13 submissions. The accepted papers were:

- *Bit Inference* by Nachum Dershowitz,
- *Collaborative Programming: Applications of logic and automated reasoning* by Timothy Hinrichs,
- *Towards fully automated axiom extraction for finite-valued logics* by Joao Marcos and Dalmo Mendonca,
- *A Small Framework for Proof Checking* by Hans de Nivelle and Piotr Witkowski,
- *Integration of the TPTPWorld into SigmaKEE* by Steven Trac, Geoff Sutcliffe and Adam Pease,
- *Combining Theorem Proving with Natural Language Processing* by Björn Pelzer and Ingo Glöckner,
- *Presenting TSTP Proofs with Inference Web Tools* by Paulo Pinheiro da Silva, Geoff Sutcliffe, Cynthia Chang, Li Ding, Nick del Rio and Deborah McGuinness,
- *randoCoP: Randomizing the Proof Search Order in the Connection Calculus* by Thomas Raths and Jens Otten,
- *The Annual SUMO Reasoning Prizes at CASC* by Steven Trac, Geoff Sutcliffe and Adam Pease,
- *Contextual Rewriting in SPASS* by Christoph Weidenbach and Patrick Wischnewski.

The ESHOL sessions were organised by Christoph Benzmüller, Florian Rabe, Carsten Schürmann, and Geoff Sutcliffe. See [1] in these proceedings for more details.

3 Publication Details

These proceedings are published as CEUR Workshop Proceedings [2]. The PAAR-2008 webpage is:

`http://www.eprover.org/EVENTS/PAAR-2008/paar-2008.html`.

4 Organization

PAAR-2008 Co-Chairs

Boris Konev, University of Liverpool
Renate Schmidt, University of Manchester
Stephan Schulz, TU München

PAAR-2008 Program Committee

Serge Autexier
Nikolaj Bjorner
Peter Baumgartner
Christoph Benzmüller
Koen Claessen
Bernd Fischer
Ulrich Furbach
Martin Giese
Volker Haarslev
Thomas Hillenbrand
Tommi Junttila
Deepak Kapur
Boris Konev
Konstantin Korovin
Bill McCune
Boris Motik
Flavio LC de Moura
Hans de Nivelle
Albert Oliveras
Brigitte Pientka
Adam Pease
Florian Rabe
Silvio Ranise
Renate Schmidt
Stephan Schulz
Carsten Schürmann
Sanjit Seshia
Geoff Sutcliffe
Cesare Tinelli
Josef Urban
Luca Vigano
Andrei Voronkov
Uwe Waldmann

ESHOL-2008 Co-Chairs

Christoph Benzmüller, Saarland University
Florian Rabe, Jacobs University Bremen
Carsten Schürmann, IT University of Copenhagen
Geoff Sutcliffe, University of Miami

5 Schedule

Sunday, August 10

09.00–10.00 Invited talk: Alessandro Armando (shared with CEDAR)
Software Model Checking: new challenges and opportunities for Automated Reasoning

10.00–10.30 Coffee break

10.30–12.30 Session 1

10.30 Joao Marcos and Dalmo Mendonca
Towards fully automated axiom extraction for finite-valued logics

11.10 Paulo Pinheiro da Silva, Geoff Sutcliffe, Cynthia Chang, Li Ding, Nick del Rio and Deborah McGuinness
Presenting TSTP Proofs with Inference Web Tools

11.50 Steven Trac, Geoff Sutcliffe and Adam Pease
Integration of the TPTPWorld into SigmaKEE

12.30–14.00 Lunch break

14.00–15.00 ESHOL invited talk: Rob Arthan
Mechanized Reasoning for Continuous Problem Domains

15.00–15.30 ESHOL presentations

15.00 Rob Arthan
ProofPower

15.10 Stefan Berghofer
Isabelle

15.20 Lucas Dixon
IsaPlanner

15.30–16.00 Coffee break

16.00–17.00 ESHOL presentations

16.00 Guillaume Melquiond
Coq

16.10 Josef Urban
Mizar

16.20 Joe Hurd
HOL

16.30 Carsten Schürmann
Delphin

16.40 Christoph Benzmüller and Frank Theiss
Omega

16.50 Christoph Benzmüller and Frank Theiss
LEO II

17.00 Mark Kaminski
TPS

17.30– ESHOL demos
running in parallel

Monday, August 11

09.00–10.00 Invited talk: John Slaney
Constraint Modelling: A Challenge for First Order Automated Reasoning

10.00–10.30 Coffee break

10.30–12.30 Session 2

10.30 Christoph Weidenbach and Patrick Wischnewski
Contextual Rewriting in SPASS

11.10 Nachum Dershowitz
Bit Inference

11.50 Hans de Nivelle and Piotr Witkowski
A Small Framework for Proof Checking

12.30–14.00 Lunch break

14.00–15.30 Session 3

14.00 Timothy Hinrichs
Collaborative Programming: Applications of logic and automated reasoning

14.40 Björn Pelzer and Ingo Glöckner
Combining Theorem Proving with Natural Language Processing

15.30–16.00 Coffee break

16.00–17.10 Session 4

16.00 Steven Trac, Geoff Sutcliffe and Adam Pease
The Annual SUMO Reasoning Prizes at CASC

16.40 Thomas Raths and Jens Otten
randoCoP: Randomizing the Proof Search Order in the Connection Calculus

17.20–18.00 ESHOL panel: Rob Arthan, Lucas Dixon, Joe Hurd
Evaluation of Systems for Higher Order Logic

References

1. C. Benzmüller, F. Rabe, C. Schürmann, and G. Sutcliffe. Evaluation of systems for higher-order logic (ESHOL). In B. Konev, R. A. Schmidt, and S. Schulz, editors, *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning (PAAR-2008/ESHOL-2008), Sydney, Australia, August 10–11, 2008*, pages 22–25. CEUR Workshop Proceedings, 2008.
2. B. Konev, R. A. Schmidt, and S. Schulz, editors. *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning (PAAR-2008/ESHOL-2008), Sydney, Australia, August 10–11, 2008*. CEUR Workshop Proceedings, 2008.

Table of Contents

Software Model Checking: new challenges and oppurtunities for Automated Reasoning *(invited talk)* . . . 1
Alessandro Armando

Mechanized Reasoning for Continuous Problem Domains *(invited talk)* . . . 2
Rob Arthan

Constraint Modelling: A Challenge for First Order Automated Reasoning *(invited talk)* . . . 11
John Slaney

Evaluation of Systems for Higher-order Logic *(ESHOL)* . . . 22
Christoph Benzmueller, Florian Rabe, Carsten Schuermann, Geoff Sutcliffe

Bit Inference . . . 26
Nachum Dershowitz

Collaborative Programming: Applications of logic and automated reasoning . . . 36
Timothy Hinrichs

Towards fully automated axiom extraction for finite-valued logics . . . 46
Joao Marcos, Dalmo MendonÃa

A Small Framework for Proof Checking . . . 56
Hans de Nivelle, Piotr Witkowski

The Annual SUMO Reasoning Prizes at CASC . . . 66
Adam Pease, Geoff Sutcliffe, Nick Siegel, Steven Trac

Combining Theorem Proving with Natural Language Processing . . . 71
BjÃrn Pelzer, Ingo GlÃckner

Presenting TSTP Proofs with Inference Web Tools . . . 81
Paulo Pinheiro da Silva, Geoff Sutcliffe, Cynthia Chang, Li Ding, Nick del Rio, Deborah McGuinness

randoCoP: Randomizing the Proof Search Order in the Connection Calculus . . . 94
Thomas Raths, Jens Otten

Integration of the TPTPWorld into SigmaKEE . . . 103
Steven Trac, Geoff Sutcliffe, Adam Pease

Contextual Rewriting in SPASS 115
Christoph Weidenbach, Patrick Wischnewski

Software Model Checking: New Challenges and Opportunities for Automated Reasoning

Alessandro Armando

AI-Lab, DIST, Università di Genova, Italy

Abstract. Software Model Checking is emerging as one of the leading approaches to automatic program analysis. State-of-the-art software model checkers exhibit levels of automation and precision often superior to those provided by traditional software analysis tools. This success is due to a large extent to the use of Satisfiability Modulo Theory (SMT) solvers to support reasoning about complex and even infinite data structures (e.g. bit-vectors, numeric data, arrays) manipulated by the program being analysed. In this talk I will survey the opportunities and challenges posed to Automated Reasoning by this new application domain.

Mechanized Reasoning for Continuous Problem Domains (Extended Abstract)

R.D. Arthan

Lemma 1 Ltd.
2nd Floor, 31A Chain Street, Reading RG1 2HX, UK
& Department of Computer Science,
Queen Mary, University of London, London E1 4NS, UK
rda@lemma-one.com

Abstract. Specification and verification in continuous problem domains are key topics for the practical application of formal methods and mechanized reasoning. I discuss one approach to linear continuous control systems and consider the challenges and opportunities raised for mechanized reasoning. These include practical implementation and integration issues, algorithms in computational real algebraic geometry and hard open questions such as the Schanuel conjecture. I conclude with an overview of some recent new results on decidability and undecidability for vector spaces and related theories.

1 Introduction

For some years, I have been involved with tools used for formally specifying and verifying digital subsystems of avionics control systems [2]. The models used in this work typically have discrete time and continuous data. These discrete models emerge only at the end of a chain of refinements starting from a purely continuous top-level model of the overall system. To apply formal verification techniques earlier in the chain could offer significant benefits in the shape of increased dependability, early detection of defects, and reduction in validation costs. Practical techniques for mechanized reasoning about continuous problem will be a key factor in obtaining these benefits.

To understand the challenges that formal verification of continuous systems offers for mechanized reasoning, it is helpful to do some methodological thinking. In the first part of the talk, I give an overview of an approach to linear continuous systems that builds on the well-known ideas of Hoare logic that have proved so fruitful in program verification. It turns out that linearity makes this approach very tractable given adequate support for reasoning about the mathematical problem domains involved, namely vector spaces, typically with some additional structure such as an inner product or a norm.

However, some significant problems arise from this. While the first order theory of the real field is decidable, in practice, engineers will want to work with

the kind of rich vocabulary supported by typical computer algebra systems; so even to deal with the field of scalars in our vector spaces, we may need to go well beyond the usual first order theory of the real numbers. Moreover, even if we have some solution to this problem or have an application in which a simple language for the scalars is adequate, we need methods for reasoning about vector spaces. In the second part of the talk, I will describe some new results on decidability and undecidability for various theories of inner product spaces and normed vector spaces (including Hilbert spaces and Banach spaces). It turns out that the very uniform geometric and algebraic properties of inner product spaces lead to decidable theories, while, with only trivial exceptions, theories of normed vector spaces are undecidable. Nonetheless, the universal fragment admits a decision procedure. I believe there is plenty of scope for interesting and useful further research in this area.

2 Reasoning about linear systems

Let us consider an approach to reasoning about linear systems proposed in [1]. By reusing some well-known ideas from software specification and verification, this approach is designed to be modular and scalable. It deals with a type of model supported by widely used tools such as Simulink. These tools allow a system to be expressed as a signal flow graph formed by wiring together primitive components.

As an example, Figure 1 represents a mechanical system in which a force f acts on a cart of mass m attached to a wall by a spring with spring constant, k. It is a graphical representation of the following differential equation:

$$m\ddot{x}(t) + kx(t) - f(t) = 0.$$

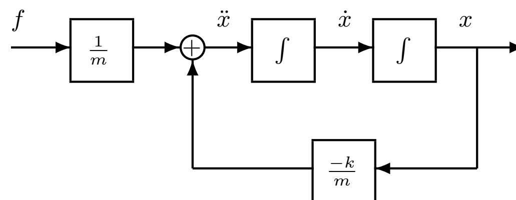

Fig. 1. A Linear Signal Flow Graph

The arrows in the diagram suggest a distinction between inputs and outputs that is missing from the differential equation. They let us view figure 1 as specifying the function mapping the force function f to the position function x. With this intensional viewpoint, the diagram might serve, for example. as a design for an analogue computer that simulates the mechanical system.

The lists of real-valued functions of time that appear as the lists of inputs and outputs to the primitive components in our diagram form vector spaces over the field $\mathbb{R}$ of real numbers. Moreover, the primitive components of the diagram represent linear transformations on those vector spaces (integration, scalar multiplication and addition). Such diagrams are called *linear signal flow graphs* and are very common in engineering practice. From now on we restrict our attention to linear signal flow graphs.

In the example of figure 1 the diagram happens to be a function, but, in general, a differential equation may not have a unique solution for a given initial condition. So in general a diagram denotes an input/output relation that is not necessarily total or single-valued. Rather than trying to ban partial or multi-valued relations, we will deal with them by borrowing some ideas from the world of relational specification of programs. This turns out to work particularly nicely given the algebraic structure we have to hand.

We write $r : X \leftrightarrow Y$ to denote that r is a relation between the sets X and Y, i.e., a subset of $X \times Y$, and use $x \;\underline{r}\; y$ as a shorthand for $(x, y) \in r$. If r and s are relations, $(r; s)$ denotes the relational composition r followed by s, so that $x \;\underline{(r; s)}\; y$ iff there is a z with $x \;\underline{r}\; z$ and $z \;\underline{s}\; y$. If $r : X \leftrightarrow Y$, $r^{-1} : Y \leftrightarrow X$ is the relational inverse of r, defined by taking $x \;\underline{r^{-1}}\; y$ iff $y \;\underline{r}\; x$. We write Ar for the image of a set A under the relation r. So if $r : X \leftrightarrow Y$, then $\mathsf{dom}(r) = Yr^{-1}$ is the domain of r, $\mathsf{ran}(r) = Xr$ is its range and r acts as a relation between any sets A and B such that $\mathsf{dom}(r) \subseteq A$ and $\mathsf{ran}(r) \subseteq B$.

If $r : X \leftrightarrow Y$, $A \subseteq X$ and $B \subseteq Y$, a *Hoare triple*, $\{A\}\, r\, \{B\}$, is the logical judgement which holds whenever $A \subseteq \mathsf{dom}(r)$ and $Ar \subseteq B$. A and B are referred to as the *pre-condition* and *post-condition* respectively. Hoare triples may be characterised in terms of weakest pre-conditions: the *weakest pre-condition*, $\mathsf{wp}(r, B)$, of B through r is the set of all points in the domain of r whose image under r is contained entirely in B. As is easily verified, the Hoare triple $\{A\}\, r\, \{B\}$ holds, iff $A \subseteq \mathsf{wp}(r, B)$.

The weakest pre-condition $\mathsf{wp}(r, B)$ contrasts with the pre-image Br^{-1} of B under r comprising all points whose image under r meets B. In general $\mathsf{wp}(r, B)$ is a proper subset of Br^{-1}. But if r is a function (not necessarily total), one has that $\mathsf{wp}(r, B) = Br^{-1}$. It turns out that something quite similar holds for the input/output relations defined by linear signal flow graphs.

In fact, the input/output relation between vector spaces V and W determined by a a linear signal flow graph is what is called an *additive relation* [8], i.e., a non-empty relation $r : V \leftrightarrow W$ that forms a subspace of $V \times W$. Additive relations generalise linear transformations. Like a linear transformation, an additive relation has a *kernel*, $\mathsf{ker}(r) = \{v : V \mid v \;\underline{r}\; 0\}$, which one can view as a uniform measure of the information lost by r. Dually, r has an *indeterminacy*, $\mathsf{ind}(r) = \{v : V \mid 0 \;\underline{r}\; v\}$, which one can view as a uniform measure of the non-determinism of r: if $v \;\underline{r}\; w$, then the set of elements related to v by r is $w + \mathsf{ind}(r)$. It turns out that to form the weakest pre-condition $\mathsf{wp}(r, B)$, one simply discards from B any element w for which $w + \mathsf{ind}(r)$ is not contained

in B, and then $\mathsf{wp}(r, B)$ is the the pre-image through r of what remains. I.e., putting $B_0 = \{b : B \mid b + \mathsf{ind}(r) \subseteq B\}$ one has that $\mathsf{wp}(r, B) = B_0 r^{-1}$.

In figure 2 we show a set of constructors for forming new signal flow graphs from old. We call a signal flow graph a *structured block diagram* if it is formed from primitive components using these constructors. In [1], we prove that structured block diagrams are complete in the sense that subject to reasonable assumptions on the set of primitive components the input/output relation of an arbitrary signal flow graph can be expressed as the input/output relation of a structured block diagram (cf., the Turing completeness of while programs).

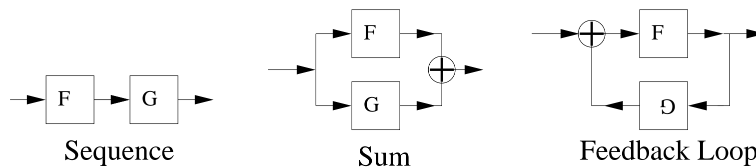

Fig. 2. Structured Block Diagram Constructors

Using the characterisation of the weakest pre-condition given above, we can then derive a Hoare logic for structured block diagrams. For example, we have the **linear combination rule**

$$\frac{\{A\}\, r\, \{B\} \quad \{A\}\, s\, \{B_1\}}{\{A\}\, \beta r + \gamma s\, \{\beta B + \gamma B_1\}}$$

Assuming we have a tractable characterisation of the primitive blocks, the Hoare logic reduces the problem of verifying any structured block diagram against given pre- and post-conditions reduces to a problem in the assertion language we are using to express the pre- and post-conditions.

For example, assume that we are working with finite-dimensional vector spaces $\mathbb{R}^m$, $m \in \mathbb{N}$ and that our primitive blocks are given by matrices with constant rational coefficients. Let us make assertions about vectors $(v_1, \ldots, v_m) \in \mathbb{R}^m$ using first order formulae in the language of the real field with free variables drawn from $v_1, \ldots v_m$. Then our approach automatically reduces any verification problem to a problem in the language of the real field. Thus, in contrast with the situation for programming languages, a large and natural class of signal flow graphs has a decidable verification problem, since, by a classic result of Tarski [11], the first order theory of the reals is decidable.

However, there are practical concerns: the time complexity of the decision procedure for the first order theory of the reals is provably doubly exponential in the number of bound variables in the formula (this theoretical bound being achieved by Collins' method of cylindrical algebraic decomposition [5]). The best known algorithms have the advantage of being at worst doubly exponential in

the number of quantifier alternations [3], and that would be advantageous in the present context, but these have not yet been implemented.

As suggested in [1], if one restricts to so-called linear formulae, i.e., ones in which multiplication is restricted to have at least one operand constant, the more efficient method of Fourier-Motzkin-Hodes applies [7]. However, the restriction to linear formulae *and* rational coefficients would generally be too restrictive for practical use, since even simple properties such as $|v_1| < \sqrt{2}$ would not be expressible. Now Fourier-Motzkin elimination is effective over any subfield of the reals in which one can effectively compute. So one might consider linear formulae over arbitrary real algebraic numbers, but calculation with such numbers is possible but complex to implement [10]. Of course, engineers are also likely to want calculation with transcendental functions as well. Towards this, we have Macintyre and Wilkie's result that Schanuel's conjecture implies the decidability of the real exponential field [9]. So progress on a natural engineering problem may be contingent on a hard unsolved problem in pure mathematics!

3 Decidability for theories of vector spaces

A few years ago, on being asked by John Harrison about decidability for vector spaces, Robert M. Solovay promptly invented quantifier elimination procedures for a range of theories. Some special cases of these have so far been implemented and found very useful in practice [6]. Solovay also demonstrated that the theory of Banach spaces is undecidable. Since then Solovay, Harrison and I have simplified and extended these results and a full exposition is in preparation. Here I sketch some of the main results and methods.

We work in a two-sorted first order language with sorts $\mathcal{R}$ for scalars and $\mathcal{V}$ for vectors. The intended interpretation of the sort $\mathcal{R}$ is the set $\mathbb{R}$ of real numbers. We have function symbols $_ + _, _ \times _ : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ and $-_ : \mathcal{R} \to \mathcal{R}$ which in the intended interpretations are the usual field operations on $\mathbb{R}$. We have function symbols $_ + _ : \mathcal{V} \times \mathcal{V} \to \mathcal{V}$, $-_ : \mathcal{V} \to \mathcal{V}$ and $_ \times _ : \mathcal{R} \times \mathcal{V} \to \mathcal{V}$ which in the intended interpretations make the set denoted by $\mathcal{V}$ into a real vector space. We have scalar constants $m/n : \mathcal{R}$ for each rational number m/n and we have the vector constant $\mathbf{0} : \mathcal{V}$ to be interpreted as the zero vector. The first order theory of real vector spaces is the set of sentences in our language that are valid in all the intended interpretations.

The theory of normed spaces is obtained by adding to the language a function symbol $||_|| : \mathcal{V} \to \mathcal{R}$ whose intended interpretation is a norm on the vector space. A norm defines a metric on the set of vectors via $d(\mathbf{v}, \mathbf{w}) = ||\mathbf{v} - \mathbf{w}||$. Recall that a normed space is called a Banach space if it is complete with respect to this metric (i.e., if every Cauchy sequence converges).

The theory of inner product spaces is obtained by adding a function symbol $\langle _, _ \rangle : \mathcal{V} \times \mathcal{V} \to \mathcal{R}$ whose intended interpretation is an inner product. Recall that an inner product space that is also a Banach space under the norm defined by $||\mathbf{v}|| = \sqrt{\langle \mathbf{v}, \mathbf{v} \rangle}$ is called a Hilbert space.

We consider the theories of vector spaces, normed spaces, Banach spaces etc. with various restrictions on the dimension, e.g., the theory of all finite dimensional inner product spaces. We write IP, resp., $\mathsf{IP}^{\mathbb{F}}$, resp., IP^{∞} for the theories of real inner product spaces where the dimension is unconstrained, resp., constrained to be finite, resp., constrained to be infinite, and HS, $\mathsf{HS}^{\mathbb{F}}$ and HS^{∞} for the theories of Hilbert spaces with the corresponding constraints on the dimension. Completeness is guaranteed if the dimension is finite, so $\mathsf{IP}^{\mathbb{F}} = \mathsf{HS}^{\mathbb{F}}$.

A sentence in any of these languages that contains no vector-valued subexpressions is just a sentence in the first order language of the real field and its truth is independent of the interpretation of the vector sort. If we can eliminate all the vector quantifiers from a formula, then occurrences of the vector constant $\mathbf{0}$ can readily be eliminated to give an equivalent formula in the first order language of the real field.

If B is a basis for a vector space V, then we can define an inner product on V by requiring $\langle \mathbf{b}, \mathbf{b} \rangle = 1$ for $\mathbf{b} \in B$ and $\langle \mathbf{b}, \mathbf{c} \rangle = 0$ for $\mathbf{b}, \mathbf{c} \in B$ with $\mathbf{b} \neq \mathbf{c}$ and extending to V by bilinearity. Thus the theory of inner product spaces is a conservative extension of the theory of vector spaces and a decision procedure for the theory of inner product spaces is also a decision procedure for the theory of vector spaces.

The key to decidability for inner product spaces is the fact that it takes at most k degrees of freedom to decide a sentence containing k vector variables. I.e., a sentence P containing k vector variables is valid in all inner product spaces iff it is valid in $\mathbb{R}^n$ for $0 \leq n \leq k$. This is proved by considering a process that replaces vector quantifiers by blocks of scalar quantifiers. The process transforms a formula containing k vector variables into one which is equivalent in spaces of dimension at least k and in which vector variables only appear within arithmetic constraints on inner products $(\mathbf{v}, \mathbf{w})$, with $\mathbf{v}$, $\mathbf{w}$ free. Applying the process to a sentence P with k vector variables results in a sentence in the language of a real field which is equivalent in dimensions k or higher. From P one can effectively construct a sentence $P|_n$ containing no vector-valued subexpressions which is valid iff P is valid in $\mathbb{R}^n$. Writing D_n (resp. $D_{\leq n}$) for a sentence asserting that the dimension of the space is n (resp. at most n), one finds that P is equivalent to:

$$(D_0 \wedge P|_0) \vee (D_1 \wedge P|_1) \vee \ldots \vee (D_{k-1} \wedge P|_{k-1}) \vee (\neg D_{\leq(k-1)} \wedge P|_k)$$

Applying the quantifier elimination algorithm for the first order theory of the reals to the subformulae $P|_n$, this leads to the following result:

Theorem 1 *The theories* IP, $\mathsf{IP}^{\mathbb{F}}$, IP^{∞}, HS, $\mathsf{HS}^{\mathbb{F}}$ *and* HS^{∞} *are all decidable.*

When we consider decidability for normed spaces we find that even the theory of 2-dimensional spaces is undecidable and actually admits a primitive recursive reduction of second order arithmetic. The proof uses the following fact that is well-known to descriptive set theorists and others, but seems not to have appeared in the literature in quite the form we need.

Theorem 2 *Let K be a (many-sorted) first-order language including a sort $\mathcal{R}$, constants $0 : \mathcal{R}$ and $1 : \mathcal{R}$ and function symbols $_ + _, _ \times _ : \mathcal{R} \times \mathcal{R} \to \mathcal{R}$ whose intended interpretations form the field of the real numbers. Let $\mathcal{M}$ be some class of structures for K in which $\mathcal{R}$ and these symbols have their intended interpretations and let $\mathcal{T}$ be the theory of $\mathcal{M}$, i.e., the set of all sentences valid in every member of $\mathcal{M}$. If there is a formula $N(x)$ of K with one free variable x of sort $\mathcal{R}$ such that in some structure in the class $\mathcal{M}$, $N(x)$ defines the set of natural numbers (i.e., $\{x : \mathbb{R} \mid N(x)\} = \mathbb{N}$), then there is a primitive recursive reduction of second order arithmetic to $\mathcal{T}$.*

Here is a sketch of the proof: one can write down a sentence Nat which asserts that the subset of the reals defined by $N(x)$ satisfies the Peano axioms, and then, in any structure for the language in which the reals have their intended interpretation, Nat holds iff $N(x)$ does indeed define the natural numbers. Now if P is any sentence in the language of Peano arithmetic, we may view P as a sentence in the first order language of the reals and then construct a new sentence P^* by relativizing all quantifiers to $N(x)$ (i.e., $\forall x \cdot Q$ is replaced by $\forall x \cdot N(x) \Rightarrow Q$ and $\exists x \cdot Q$ is replaced by $\exists x \cdot N(x) \land Q$). But then the sentence $\mathsf{Nat} \Rightarrow P^*$ is in $\mathcal{T}$ iff it P is valid in arithmetic. This gives a reduction of first order arithmetic to $\mathcal{T}$. A reduction of second order arithmetic is obtained in a similar way using real numbers to represent sets of natural numbers, e.g., using n-ary expansions.

So, for example, this gives a very simple proof that the first order theory of metric spaces is undecidable: in the metric space Z whose elements are the integers with the distance defined by $d(\mathbf{p}, \mathbf{q}) = |\mathbf{p} - \mathbf{q}|$, we can clearly define the natural numbers by the formula $N(x) := \exists \mathbf{p}\ \mathbf{q} \cdot x = d(\mathbf{p}, \mathbf{q})$. By the above theorem, the theory of metric spaces must therefore admit a primitive recursive reduction of second order arithmetic and hence is undecidable.

Write NS, resp., NS^n, resp., $\mathsf{NS}^{\mathbb{F}}$, resp., NS^{∞} for the theories of normed spaces where the dimension is unconstrained, resp., constrained to be n, resp., constrained to be finite, resp., constrained to be infinite, and write BS, BS^n etc. for the theories of Banach spaces with the corresponding constraints on the dimension. We have the following theorem which implies that with the exception of $\mathsf{NS}^1 = \mathsf{BS}^1$ (which is the same as the the theory of the real field) all of these theories are undecidable.

Theorem 3 *There is a primitive recursive reduction of second order arithmetic to each of the theories BS, BS^{∞}, NS, $\mathsf{NS}^n = \mathsf{BS}^n$, $\mathsf{NS}^{\mathbb{F}} = \mathsf{BS}^{\mathbb{F}}$, and NS^{∞} ($n \geq 2$).*

The proof is based on a construction of a 2-dimensional normed space X in which a certain first order formula defines the natural numbers as a subset of the field of scalars. By theorem 2, this immediately gives the result for $\mathsf{NS}^2 = \mathsf{BS}^2$ and $\mathsf{NS}^{\mathbb{F}} = \mathsf{BS}^{\mathbb{F}}$. The other parts of the result follow by considering the cartesian product of X and a Hilbert space of appropriate dimension. X is constructed by taking the norm whose unit disc is the "infinigon" D shown in figure 3. D is the convex hull of the set comprising the two vectors $\pm\mathbf{e}_1$ together with the unit vectors $\pm\mathbf{v}_i$ on the lines through the origin and the points $(i, 1)$, $i \in \mathbb{Z}$.

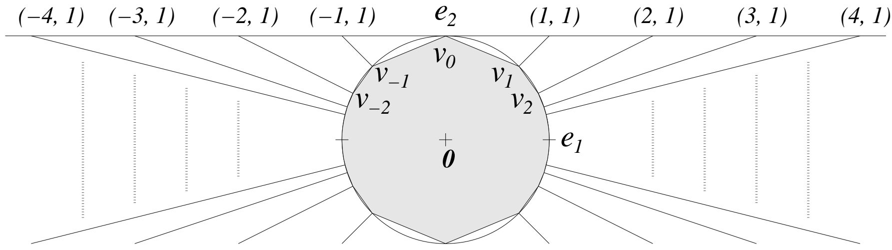

Fig. 3. The unit disc in the space X

Observing that the points $\pm\mathbf{v}_i$ are the isolated extreme points of the unit disc, while the only non-isolated extreme points are the points $\pm\mathbf{e}_1$, one finds that the language of normed spaces is sufficiently expressive for us to characterise the set of points $(i, 1)$ for $i \in \mathbb{Z}$ and then it is easy to give a formula $N(x)$ which defines the natural numbers in X.

We say a formula is *additive* if the left operand of all multiplications in the formula are rational constants. With a little care one can arrange for the formula $N(x)$ above to be additive and then with a little more geometric effort, one can give an additive formula $M(x, y, z)$ that in X defines the graph of the multiplication function $_ \times _ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. A variant of theorem 2 can then be used to show that the that even the purely additive fragments of the various theories of normed spaces and Banach spaces are undecidable.

On the positive side for normed spaces, we have the following result on the existence of norms:

Theorem 4 *Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ be vectors in a real vector space V and $b_1, \ldots, b_n$ be real numbers. Then there exists a norm $||_||$ on V such that $||\mathbf{x}_i|| = b_i$ for all $1 \leq i \leq n$ iff:*

- *For all $1 \leq i \leq n$, $b_i \geq 0$.*
- *For all $1 \leq i \leq n$, if $b_i = 0$ then $\mathbf{x}_i = 0$*
- *For each $1 \leq k \leq n$ there are no real numbers $c_1, \ldots, c_n$ such that some $\mathbf{x}_k = \sum_{i=1}^{n} c_1 \mathbf{x}_i$ with $\sum_{i=1}^{n} |c_i| b_i < b_k$.*

Now a quantifier-free formula P in the language of normed spaces containing k free vector variables has a model iff it has a model of dimension k (since in any model the subspace W spanned by the interpretations of the free vector variables is again a model of dimension $n \leq k$ and then $W \times \mathbb{R}^{k-n}$ gives a model of dimension k). From this observation and theorem 4, one can effectively transform P into a formula in the first order language of the real field that is satisfiable iff P is. This gives a decision procedure for purely universal formulae in the language of normed spaces. For purely additive formulae, there is a more efficient procedure which uses a parametrised linear programming algorithm to reduce the problem to linear real arithmetic. An implementation of the latter procedure in HOL Light has proved to be a useful tool.

4 Concluding Remarks

The approach to specification and verification of linear systems presented in section 2 is simple and natural. But even in the simple case of finite-dimensional inner product spaces, there are difficult issues to be addressed for mechanized proof support. The decision procedures of section 3 give a starting point, but our undecidability results show that there is much to be done in identifying useful tractable fragments of theories and good heuristics. There are many fascinating challenges ahead for mechanized reasoning in continuous problem domains.

Acknowledgments

I thank Colin O'Halloran and Nick Tudor of QinetiQ for encouraging and informing my interest in control systems. The approach to reasoning about linear systems is joint work with Ursula Martin, Erik Arne Mathiesen and Paulo Oliva and was supported by EPSRC grants GR/M98340 and GR/L48256. The work on decidability and undecidability for vector spaces is joint work with Robert M. Solovay and John Harrison.

References

1. R. D. Arthan, U. Martin, E. A. Mathiesen, and P. Oliva. Reasoning About Linear Systems. In *SEFM'07*. IEEE Press, 2007.
2. R.D. Arthan, P. Caseley, C. O'Halloran, and A.Smith. ClawZ: Control Laws in Z. In *3rd International Conference on Formal Engineering Methods (ICFEM 2000)*. IEEE, 2000.
3. Saugata Basu, Richard Pollack, and Marie-Françoise Roy. *Algorithms in Real Algebraic Geometry*, volume 10 of *Algorithms and Computation in Mathematics*. Springer-Verlag, 2006.
4. H. Brakhage, editor. *Second GI Conference on Automata Theory and Formal Languages*, volume 33 of *LNCS*. Springer-Verlag, 1976.
5. G. E. Collins. Quantifier Elimination for Real Closed Fields by Cylindrical Algebraic Decomposition. In Brakhage [4], pages 134–183.
6. John Harrison. A HOL Theory of Euclidean Space. In Joe Hurd and Thomas F. Melham, editors, *Proceedings of TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2005.
7. Louis Hodes. Solving Problems by Formula Manipulation in Logic and Linear Inequalities. *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 553–559, 1971.
8. S. Mac Lane. *Homology*, volume 114 of *Der Grundlehren der mathematischen Wissenschaften*. Springer, 1975.
9. A. J. Macintyre and A. J. Wilkie. On the Decidability of the Real Exponential Field. In Piergiorgio Odifreddi, editor, *Kreiseliana: About and Around Georg Kreisel*, pages 441–467. A. K. Peters, 1996.
10. Renaud Rioboo. Towards Faster Real Algebraic Numbers. *J. Symb. Comput.*, 36(3-4):513–533, 2003.
11. Alfred Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, 1951.

Constraint Modelling: A Challenge for First Order Automated Reasoning (extended abstract)

John Slaney

NICTA and the Australian National University

Abstract. The process of modelling a problem in a form suitable for solution by constraint satisfaction or operations research techniques, as opposed to the process of solving it once formulated, requires a significant amount of reasoning. Contemporary modelling languages separate the first order description or "model" from its grounding instantiation or "data". Properties of the model independent of the data may thus be established by first order reasoning. In this talk, I survey the opportunities arising from this new application direction for automated deduction, and note some of the formidable obstacles in the way of a practically useful implementation.

1 Constraint Programming

A constraint satisfaction problem (CSP) is normally described in the following terms: given a finite set of decision variables $v_1, \ldots, v_n$ with associated domains $D_1, \ldots, D_n$, and a relation $C(v_1, \ldots v_n)$ betwen the variables, a *state* is an assignment to each variable v_i of a value d_i from D_i. A state is a *solution* to the CSP iff $C(d_1, \ldots, d_i)$ holds. In practice, C is the conjunction of a number of constraints each of which relates a small number of variables. It is common to seek not just any solution, but an optimal one in the sense that it minimises the value of a specified *objective function*.

Mathematical programming is the special case in which the domains are numerical (integers or real numbers) and the constraints are equalities or inequalities between functions (usually polynomial, nearly always linear, in fact) of these. The techniques usable for numbers are so different from those for the general case, however, that CP and MP are often seen as contrasting or even opposing approaches.

Logically, C is a theory in a language in which the v_i are proper names ("constants" in the usual terminology of logic). A state is an interpretation of the language over a domain (or several domains, if the language is many-sorted) corresponding to the domains of the variables, and a solution is an interpretation that satisfies C. On this view, CSP reasoning is the dual of theorem proving: it is seeking to establish possibility (satisfiability) rather than necessity (unsatisfiability of the negation).

Techniques used to solve CSPs range from the purely logical, such as SAT solving, through finite domain (FD) reasoning which similarly consists of a backtracking search over assignments, using a range of propagators appropriate to different constraints to force some notion of local consistency after each assignment, to mixed integer programming using a variety of numerical optimisation algorithms. Hybrid solution methods, in which different solvers are applied to sub-problems, include SMT (satisfiability modulo theories), column generation, large neighbourhood search and many more or less *ad hoc* solver combinations for specific purposes. The whole area has been researched intensively over the last half century, generating an extensive literature from the automated reasoning, artificial intelligence and operations research communities. The reader is referred to Dechter's overview [3] for an introduction to the field.

Constraint programming is an approach to designing software for CSPs, whereby a library of solvers is used in the same maner as libraries of mathematical function computations. The search is controlled by a program written in some high-level language (sometimes a logic programming language, but in modern systems often C++ or something similar) and specific solvers may be used to evaluate particular predicates or perform propagation steps, or may be passed the entire problem after some preprocessing. CP platforms vary in the degree to which they automate control of the propagation queue and the like, or leave it in the hands of the programmer. The constraint programming paradigm gives a great deal of flexibility, allowing techniques to be tailored to problems, while at the same time accessing the power and efficiency of high-performance CSP solvers.

1.1 Separating Modelling from Solving

Engineering a constraint program for a given problem is traditionally a two-phase process. First the problem must be *modelled*. This is a matter of determining what are the decision variables, what are their domains of possible values and what constraints they must satisfy. Then a program must be written to *evaluate* the model by using some solver or combinaton of solvers to search for solutions. Most of the CP and OR literature concerns the second phase, assuming that "the problem" resulting from the modelling phase is given.

In recent years, there has been a growing realisation of the importance of modelling as part of the overall process, so modern CP or MP platforms feature a carefully designed modelling language such as ILOG's OPL [7] or AMPL from Bell Labs [5]. Contemporary work on modelling languages such as ESRA [4], ESSENCE [6] and Zinc [8] aims to provide a rich representation tool, with primitives for manipulating sets, arrays, records and suchlike data structures and with the full expressive power of (at least) first order quantification. It also aims to make the problem representation independent of the solver(s) so that one and the same conceptual model can be mapped to a form suitable for solution by mixed integer programming, by SAT solving or by local search.

In the present report, the modelling language used will be Zinc, which is part of the G12 platform currently under development by NICTA (Australia).

My involvement is in designing and implementing the user environment for G12, which will incorporate theorem proving technology along with much else. The current theorem proving research is joint work with Peter Baumgartner.[1]

1.2 The G12 Platform

The G12 constraint programming platform provides a series of languages and associated tools. At the base is Mercury [10, 2] , a logic programming language with adaptations for contraint logic programming with a propagation queue architecture (`http://www.cs.mu.oz.au/research/mercury/`). Below that are the solvers, which include commercial ones like CPLEX, third party open source ones like MiniSAT and many of our own. The API for incorporating solvers is quite straihtforward. On the top level is the modelling language Zinc, of which more below. Between Mercury and Zinc (remember your periodic table) is Cadmium, a very declarative programming language based on term rewriting, which is designed for mapping one syntax to another and is used in G12 mainly to convert Zinc specifications into simpler ones. For instance, they may be flattened by unrolling quantifiers, or clausified, or expressed as linear programs or as SAT problems.

G12 clearly separates the conceptual model, written in Zinc, from the design model intended to be realised as a Mercury program. It also draws a distinction between the *model*, which is a first order description of the generic problem, and the *data* which are the facts serving to ground the model in a particular instance. The commonly used distinction between facts and rules or "integrity constraints" in deductive databases is somewhat similar.

G12 programs can be related to Zinc models in a spectrum of ways. It is possible to write a constraint program to solve the problem, treating the Zinc specification just as a guide, as is often done in conventional CP. The program can throw the entire problem onto one solver such as CPLEX, or onto MiniSAT as Paradox does, or can mix solvers in arbitrarily fine-grained ways to produce hybrids tailored to individual problems. Alternatively, the program can take the Zinc specification and data as input, in which case we think of it as a way of evaluating the model over the data. It is even possible to avoid writing *any* program, leaving the G12 system itself with its default mappings to do the evaluation and writing only in Zinc. This last represents the closest approach yet to the Holy Grail of high-level programming: one does not program at all; one tells the computer what the problem is, and it replies with the solution.

[1] We have benefitted greatly from being in a team that has included Michael Norrish, Rajeev Gore, Jeremy Dawson, Jia Meng, Anbulagan and Jinbo Huang, and from the presence in the same laboratory of an AI team including Phil Kilby, Jussi Rintanen, Sylvie Thiébaux and others. The G12 project involves well over 20 researchers, including Peter Stuckey, Kim Marriott, Mark Wallace, Toby Walsh, Michael Maher, Andrew Verden and Abdul Sattar. The details of our indebtedness to these people and their colleagues are too intricate to be spelt out here.

Zinc Zinc is a typed (mostly) first order language. It has as basic types `int`, `float` and `bool`, and user-defined finite enumerated types. To these are applied the `set-of`, `array-of`, `tuple`, `record` and subrange type constructors. These may be nested, with some restrictions mainly to avoid such things as infinite arrays and explicitly higher order types (functions with functional arguments). The type `string` is present, but is only used for formatting output. It also allows a certain amount of functional programming, which is not of present interest. It provides facilities for declaring decision variables of most types and constants (parameters) of all types. Standard mathematical functions such as `+` and `sqrt` are built in. Constraints may be written using the expected comparators such as $=$ and $\leq$ or user-defined predicates to form atoms, and the usual boolean connectives and quantifiers (over finite domains) to build up compounds. Assignments are special constraints whereby parameters are given their values. The values of decision variables are not normally fixed in the Zinc specification, but have to be found by some sort of search.

For details, see `http://users.rsise.anu.edu.au/~jks/zinc-spec.pdf`.

Analysing models It is normal to place the Zinc model in one file, and the data (parameters, assignments and perhaps some enumerations) in another. The model tends to stay the same as the data vary. For example, without changing any definitions or general specifications, a new schedule can be designed for each day as fresh information about orders, jobs, customers and prices becomes available.

The user support tools provided by the G12 development environment should facilitate debugging and other reasoning about models independently of any data. However, since the solvers cannot evaluate a model until at least the domains are specified, it is unclear how this can be done. Some static visualisation of the problem, such as views of the Zinc-level constraint graph, can help a little, but to go much further we need a different sort of reasoning.

2 Deductive Tasks

There is no good reason to expect a theorem prover to be used as one of the solvers for the purposes of a constraint programming platform such as G12. Apart from the fact that typical constraint satisfaction problems are trivially satisfiable—the main issue is optimality, not the existence of solutions—the reasoning required amounts to propagation of constraints over finite domains rather than to chaining together complex inferences. For this purpose SAT solvers and the like are useful, but traditional first order provers are not. However, for analysing the models before they have been grounded by data, first order deduction is the only option. The following tasks are all capable of automation:

1. Proof that the model is inconsistent.
   Inconsistency can indicate a bug, or merely a problem overconstrained by too many requirements. It can arise in "what if" reasoning, where the programmer has added speculative conditions to the basic description or it can

arise where partial problem descriptions from different sources have been combined without ensuring that their background assumptions mesh. Often, inconsistency does not afflict the pure model, but arises only after part of the data has been added, so detecting it can require a certain amount of grounded reasoning as well as first order unification-driven inference.
A traditional debugging move, also useful in the other cases of inconsistency, is to find and present a [near] minimal inconsistent core: that is, a minimally inconsistent subset of the constraints. The problem of "axiom pinpointing" in reasoning about large databases is similar, except that in the constraint programming case the number of possible axioms tends to be comparatively small and the proofs of inconsistency comparatively long. The advantage of finding a first order proof of inconsistency, rather than merely analysing nogoods from a backtracking search, is that there is some hope of presenting the proof to a programmer, thus answering the question of *why* the particular subset of constraints is inconsistent.

2. Proof of symmetry.
   The detection and removal of symmetries is of enormous importance to finite domain search. Where there exist isomorphic solutions, there exist also isomorphic subtrees of the search tree. Moreover, there can be thousands or millions of solutions isomorphic to a given one, meaning that almost all of the search is a waste of time and can be eliminated if the symmetries are detected early enough. A standard technique is to introduce "symmetry breakers", which are extra constraints imposing conditions satisfied by some but not all (preferably exactly one) of the solutions in a symmetry class. Symmetry breakers prevent entry to subtrees of the search tree isomorphic to the canonical one.
   It may be evident to the constraint programmer that some transformation gives rise to a symmetry. Rotating or reflecting the board in the N Queens problem would be an example. However, other cases may be less obvious, especially where there are side constraints that could interfere with symmetry. Moreover, it may be unclear whether the intuitively obvious symmetry has been properly encoded or whether in fact every possible solution can be transformed into one which satisfies all of the imposed symmetry breakers.
   It is therefore important to be able to show that a given transformation defined over the state space of the problem does actually preserve the constraints, and therefore that it transforms solutions into solutions. Since symmetry breakers may be part of the model rather than part of the data, we may wish to prove such a property independently of details such as domain sizes. There is an example in the next section.
3. Redundancy tests.
   A redundant constraint is one that is a logical consequence of the rest. It is common to add redundant constraints to a problem specification, usually in order to increase the effect of propagation at each node of the search tree. Sometimes, however, redundancy may be unintentional: this may indicate a bug—perhaps an intended symmetry-breaker which in fact changes nothing—or just a clumsy encoding. Some constraints which are not redun-

dant in the model may, of course, become redundant when the data are added.

Where redundant constraints are detected, either during analysis of the model or during preprocessing of the problem including data, this might usefully be reported to the constraint programmer who can then decide whether such redundancy is intentional, whether it matters or not, and whether the model should be adjusted in the light of this information. It may also be useful to report irredundancy where a supposedly redundant constraint has been added: the programmer might usefully be able to request a redundancy proof in such a case.

4. Functional dependency.

   The analogue at the level of functions of redundancy at the level of propositions is dependency in the sense that the values of certain functions may completely determine the value of another for all possible arguments. As in the case of constraint redundancy, functional dependence may be intentional or accidental, and either way it ay be useful to the constraint programmer to know whether a function is dependent or not.

   Consider graph colouring as an example. It is obvious that in general (that is, independently of the graph in question) the extensions of all but one of the colours are sufficient to fix the extension of the final one, but that this is not true of any proper subset of the "all but one". In the presence of side constraints, however, and especially of symmetry breakers, this may not be obvious at all. In such cases, theorem proving is the appropriate technology.

5. Equivalence of models.

   It is very common in constraint programming that different approaches to a given problem may result in *very* different encodings, expressing constraints in different forms and even using different signatures and different types. The problem of deciding whether two models are equivalent, even in the weak sense that solutions exist for the same values of some parameters such as domain sizes, is in general hard. Indeed, in the worst case, it is undecidable. However, hardness in that sense is nothing new for theorem proving, so there is reason to hope that equivalence can often enough be established by the means commonly used in automated reasoning about axiomatisations.

6. Simplification

   A special case of redundancy, which in turn is a special case of model equivalence, occurs in circumstances where the full strength of a constraint is not required. A common example is that of a biconditional ($\Leftrightarrow$) where in fact one half of it ($\Rightarrow$) would be sufficient. Naïve translation between problem formulations can easily lead to unnecessarily complicated constraints such as $x < \sup(S)$ which is naturally rendered as $\exists y \in S((\forall z \in S.z \leq y) \wedge x < y)$, while the simpler $\exists y \in S.x < y$ would do just as well. Formal proofs of the correctness of simplifications can usefully be offered to the programmer at the model analysis stage.

```
int: N;
array[1..N] of var int: q;
constraint forall (x in 1..N, y in 1..x-1) (q[x] != q[y]);
constraint forall (x in 1..N, y in 1..x-1) (q[x]+x != q[y]+y);
constraint forall (x in 1..N, y in 1..x-1) (q[x]-x != q[y]-y);
solve satisfy;
```

Fig. 1. Zinc model for the N Queens problem

2.1 Preliminary experiments

Our experiments are very much work in progress, so we are in a position only to report preliminary findings rather than the final word. Here are comments on just two toy examples, more to give a flavour than to present systematic results.

Proving symmetry We consider the N Queens problem, a staple of CSP reasoning. N queens are to be placed on an a chessboard of size $N \times N$ in such a way that no queen attacks any other along any row, column or diagonal. The model is given in Figure 1 and the data consists of one line giving the value of N (e.g. '`N = 8;`'). Suppose that as a result of inspection of this problem for small values of N it is conjectured[2] that the transformation $s[x] = q[n + 1 - x]$ is a symmetry. We wish to prove this for all values of N. That is, we need a first order proof that the constraints with s substituted for q follow from the model as given and the definition of s. Intuitively, the result is obvious, as it corresponds to the operation of reflecting the board, but intuitive obviousness is not proof and we wish to see what a standard theorem prover makes of it.

The prover we took off the shelf for this experiment was Prover9 by McCune [9]. Clearly a certain amount of numerical reasoning is required, for which additional axioms must be supplied. The full theory of the integers is not needed: algebraic properties of addition and subtraction, along with numerical order, seem to be sufficient. All of this is captured in the theory of totally ordered abelian groups, which is quite convenient for first order reasoning in the style of Prover9. We tried two encodings: one in terms of the order relation $\leq$ and the other an equational version in terms of the lattice operations `max` and `min`.

The first three goals:
$(1 \leq x \wedge x \leq n) \Rightarrow 1 \leq s(x)$
$(1 \leq x \wedge x \leq n) \Rightarrow s(x) \leq n$
$s(x) = s(y) \Rightarrow x = y$
are quite easy for Prover9 when $s(x)$ is defined as $q(n+1-x)$. By contrast, the other two
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) + x \neq s(y) + y$
$(1 \leq x \wedge x \leq n) \wedge (1 \leq y \wedge y \leq n) \Rightarrow s(x) - x \neq s(y) - y$
are not provable inside a time limit of 30 minutes, even with numerous helpful lemmas and weight specifications to deflect useless subformulae like $q(q(x))$ and

[2] In fact, there is experimental software to come up with such conjectures automatically.

$q(n)$. It makes little difference to these results whether the abelian l-group axioms are presented in terms of the order relation or as equations.

To push the investigation one more step, we also considered the transformation obtained by setting s to q^{-1}. This is also a symmetry, corresponding to reflection of the board about a diagonal, or rotation through 90° followed by reflection as above. This time, it was necessary to add an axiom to the Queens problem definition, as the all-different constraint on q is not inherited by s. The reason is that for all we can say in the first order vocabulary, N might be infinite—e.g. it could be any infinite number in a nonstandard model of the integers—and in that case a function from $\{1 \ldots N\}$ to $\{1 \ldots N\}$ could be injective without being surjective.

The immediate fix is to add surjectivity of the 'q' function to the problem definition, after which in the relational formulation Prover9 can easily deduce the three small goals and the first of the two diagonal conditions. The second is beyond it, until we add the redundant axiom

$x_1 - y_1 = x_2 - y_2 \Rightarrow x_1 - x_2 = y_1 - y_2$

With this, it finds a proof in a second or so. In the equational formulation, no proofs are found in reasonable time.

The more general issue, however, is that many CSP encodings make implicit use of the fact that domains are finite, as a result of which it may be impossible to deduce important properties by first-order reasoning without fixing bounds on parameters. If theorem proving is to be a useful tool in G12, ways will have to be found to circumvent such difficulties, issuing warnings if necessary.

Another message from the experiments is that a lot of arithmetical reasoning tricks and transformations will have to be identified and coded into the system. The above transformation of equalities between differences (and its counterparts for inequalities) illustrates this.

An encouraging feature is that a considerable amount of the reasoning turns only on algebraic properties of the number systems, and so may be amenable to treatment by standard first order provers.

Proving redundancy A toy example of redundant constraints is found in the following logic puzzle [1]:

> On June 1st, five couples will celebrate their wedding anniversaries. Their surnames are Johnstone, Parker, Watson, Graves and Shearer. The husbands' given names are Russell, Douglas, Charles, Peter and Everett. The wives' given names are Elaine, Joyce, Marcia, Elizabeth and Mildred.
>
> 1. Joyce has not been married as long as Charles or the Parkers, but longer than Douglas and the Johnstones.
> 2. Elizabeth has been married twice as long as the Watsons, but only half as long as Russell.
> 3. The Shearers have been married ten years longer than Peter and ten years less than Marcia.
> 4. Douglas and Mildred have been married for 25 years less than the Graves who, having been married for 30 years, are the couple who have been married the longest.

5. Neither Elaine nor the Johnstones have been married the shortest amount of time.
6. Everett has been married for 25 years

Who is married to whom, and how long have they been married?

Parts of clue 1, that Joyce has been married longer than Douglas and also longer than the Johnstones, are deducible from the other clues. Half of clue 5, that Elaine has not been married the shortest amount of time, is also redundant. The argument is not very difficult: a little arithmetical reasoning establishes that the five numbers of years married are 5, 10, 20, 25 and 30 (three of them are 30, 5 and 25, and the five contain a sequence of the form $x, 2x, 4x$). Mildred has been married for 5 years (clue 4) from which it quickly follows that both Elaine and Joyce have been married for longer than Mildred and therefore than Douglas. That Joyce has been married for longer than the Johnstones is a slightly more obscure consequence of the other clues, but a finite domain constraint solver has no difficulty with it.

Presenting the problem of deriving any of these redundancies to Prover9 is not easy. The small amount of arithmetic involved is enough to require a painful amount of axiomatisation, and even when the addition table for the natural numbers up to 30 is completely spelt out, making use of that to derive simple facts such as those above is beyond the abilities of the prover. Even given an extra clause stating that the five numbers involved are 5, 10, 20, 25 and 30, in ten thousand iterations of the given clause loop it gets nowhere near deducing that Joyce has been married for longer than Douglas.

If the fact that the numbers of years are all in the set $\{5, 10, 15, 20, 25, 30\}$ is given as an axiom, *and* extra arguments are given to all function and relation symbols to prevent unification across sorts, then of course the redundancy proofs become easy for the prover. However, it is unreasonable to expect that so much help will be forthcoming in general. Even requiring just a little of the numerical reasoning to be carried out by the prover takes the problem out of range.

Part of the difficulty is due to the lack of numerical reasoning, but as before, forcing the problem statement into a single-sorted logic causes dramatic inefficiency. It is also worth noting that the proofs of redundancy are long (some hundreds of lines) and involve nearly all of the assumptions, indicating that axiom pinpointing is likely to be useless for explaining overconstrainedness at least in some range of cases.

2.2 Conclusions

While, as noted, the investigation is still preliminary, some conclusions can already be drawn. Notably, work is required on expanding the capacities of conventional automatic theorem provers:

1. Numerical reasoning, both discrete and continuous, is essential. The theorems involved are not deep—showing that a simple transformation like reversing the order $1 \ldots N$ is a homomorphism on a model or restricting

attention to numbers divisible by 5—but are not easy for standard theorem proving technology either. Theorem provers will not succeed in analysing constraint models until this hurdle is cleared.
2. Other features of the rich representation language also call for specialised reasoning. Notably, the vocabulary of set theory is pervasive in CSP models, but normal theorem provers have difficulties with the most elementary of set properties. Some first order reasoning technology akin to SMT, whereby specialist modules return information about sets, arrays, tuples, numbers, etc. which a resolution-based theorem prover can use, is strongly indicated. Theory resolution is the obvious starting point, but is it enough?
3. Many-sorted logic is absolutely required. There are theorem provers able to exploit sorts, but most do not—a telling point is that TPTP still does not incorporate sorts in its notation or its problems.
4. Constraint models sometimes depend on the finiteness of parameters. Simple facts about them may be unprovable without additional constraints to capture the effects of this, as illustrated by the case of the symmetries of the N Queens problem. This is not a challenge for theorem provers as such but rather for the process of preparing constraint models for first order reasoning.
5. In some cases, proofs need to be presented to human programmers who are not working in the vocabulary of theorem proving, who are not logicians, and who are not interested in working out the details of complicated pramodulation inferences. Despite some efforts, the state of the art in proof presentation remains lamentable. This *must* be addressed somehow.

Despite the above challenges, and perhaps in a sense because of them, constraint model analysis offers an exciting range of potential rôles for automated deduction. Constrait-based reasoning has far wider application than most canvassed uses of theorem provers, such as software verification, and certainly connects with practical concerns much more readily than most of [automated] pure mathematics. Reasoning about constraint models without their data is a niche that only first (or higher) order deductive systems can fill. Those of us who are concerned to find practical applications for automated reasoning should be working to help them fill it.

References

1. Anonymous. Anniversaries: Logic puzzle. `http://www.genealogyworldwide.com/genealogy_fun.php`.
2. Ralph Becket, Maria Garcia de la Banda, Kim Marriott, Zoltan Somogyi, Peter Stuckey, and Mark Wallace. Adding constraint solving to Mercury. In *Proceedings of the Eighth International Symposium on Practical Aspects of Declarative languages*. Springer Verlag, 2006.
3. Rina Dechter and David Cohen. *Constraint Processing*. Morgan Kaufmann, 2003.
4. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Logic Based Program Synthesis and Transformation: 13th International Symposium, LOPSTR'03, Revised Selected Papers (LNCS 3018)*, pages 214–232. Springer-Verlag, 2004.

5. Robert Fourer, David Gay, , and Brian Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 2002. `http://www.ampl.com/`.
6. A.M. Frisch, M. Grum, C. Jefferson, B. Martínez Hernández, and I. Miguel. The design of essence: A constraint language for specifying combinatorial problems. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 80–87, 2007.
7. Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, 1999.
8. Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints, Special Issue on Abstraction and Automation in Constraint Modelling*, 13(3), 2008.
9. William McCune. Prover9 and mace4. `http://www.cs.unm.edu/ mccune/mace4/`.
10. Zoltan Somogyi, Fergus Henderson, and Thomas Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Eighteenth Australasian Computer Science Conference*, 1995. `http://www.cs.mu.oz.au/research/mercury/information/papers.html`.

Evaluation of Systems for Higher-order Logic (ESHOL)

Christoph Benzmüller[1], Florian Rabe[2], Carsten Schürmann[3], Geoff Sutcliffe[4]

[1]Saarland University, Germany [2]Jacobs International University, Germany
[3]IT University of Copenhagen, Denmark [4]University of Miami, USA

1 Introduction

The ESHOL sessions of the PAAR workshop focussed on the use of higher-order reasoning systems. A particular focus was on means to evaluate higher-order reasoning systems. The notion of higher-order included, but was not limited to, ramified type theory, simple type theory, intuitionistic and constructive type theory, and logical frameworks. The notion of reasoning systems included automated and semi-automated provers, model generators, as well as proof and model checkers. There were two parts to the ESHOL sessions: (i) higher-order system demonstrations, and (ii) a panel discussion. Additionally, one of the PAAR invited speakers, Rob Arthan, gave a talk in the ESHOL topic area.

2 Higher-order System Demonstrations

The following higher-order systems were demonstrated in the system demonstration sessions. Each presenter gave a 10 minute "talk" slot to present the system to the audience in the traditional laptop+projector mode (giving a brief overview of the system and a demonstration of it running and solving some of the problems in Appendix A). Following the 10 minute presentations there was an open forum during which presenters were all available to give individual and more detailed information and demonstrations.

- Coq, *Guillaume Melquiond*
- Delphin, *Carsten Schürmann*
- HOL, *Joe Hurd*
- Isabelle, *Stefan Berghofer*
- IsaPlanner, *Lucas Dixon*
- LEO-II, *Christoph Benzmüler and Frank Theiss*
- Mizar, *Josef Urban*
- Omega, *Frank Theiss and Christoph Benzmüller*
- ProofPower, *Rob Arthan*
- TPS, *Mark Kaminski*

3 Panel Discussion

The ESHOL panelists were Rob Arthan, Lucas Dixon, and Joe Hurd. The panel discussed ideas, suggestions, and potential problems related to:

- The buildup of an higher-order TPTP infrastructure.
- The development of automated reasoning systems for higher-order logic (or fragments of it).
- Promising application areas for automated higher-order reasoning systems.
- The planned organization of a higher-order CASC at CADE-22 in 2009.

References

1. C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 - The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page Accepted, 2008.

A Sample Problems for System Demonstrations

The two first problems should be simple enough for every system, to provide a starting point for comparisons and discussion. The third example is Cantor's Theorem, which might be more difficult. The problems are presented in the TPTP "THF" language for simple type theory, which was recently developed by the organizers [1]. The language is based on Church's simple type theory, and is a syntactically conservative extension of the untyped first-order TPTP language.

A.1 Puzzle Example

```
%------------------------------------------------------------------------
thf(islander,type,( islander: $i )).
thf(knight,type,( knight: $i )).
thf(knave,type,( knave: $i )).
thf(says,type,( says: $i > $o > $o )).
thf(zoey,type,( zoey: $i )).
thf(mel,type,( mel: $i )).
thf(is_a,type,( is_a: $i > $i > $o )).

thf(kk_6_1,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ islander )
     => ( ( is_a @ X @ knight )
        | ( is_a @ X @ knave ) ) ) )).

thf(kk_6_2,axiom,(
    ! [X: $i] :
      ( ( is_a @ X @ knight )
     => ! [A: $o] :
          ( ( says @ X @ A ) => A ) ) )).

thf(kk_6_3,axiom,
    ! [X: $i] :
      ( ( is_a @ X @ knave )
     => ! [A: $o] :
          ( ( says @ X @ A ) => ~ ( A ) ) )).

thf(kk_6_4,axiom,
    ( ( is_a @ zoey @ islander )
    & ( is_a @ mel @ islander ) )).

thf(kk_6_5,axiom,
    ( says @ zoey @ ( is_a @ mel @ knave ) )).

thf(kk_6_6,axiom,
    ( says @ mel
    @ ~ ( ( is_a @ zoey @ knave )
        | ( is_a @ mel @ knave ) ) )).

thf(query,theorem,(
    ? [Y: $i,Z: $i] :
      ( ( ( Y = knight )
      <~> ( Y = knave ) )
      & ( ( Z = knight )
      <~> ( Z = knave ) )
      & ( is_a @ mel @ Y )
      & ( is_a @ zoey @ Z ) ) )).
%------------------------------------------------------------------------
```

A.2 Set Theory Example

```
%------------------------------------------------------------------------
%----Signatures for basic set theory predicates and functions.
thf(const_in,type,(
    in: $i > ( $i > $o ) > $o  )).

thf(const_intersection,type,(
    intersection: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

thf(const_union,type,(
    union: ( $i > $o ) > ( $i > $o ) > ( $i > $o ) )).

%----Some axioms for basic set theory.
thf(ax_in,axiom,(
    ( in
    = ( ^ [X: $i,S: ( $i > $o )] :
          ( S @ X ) ) ) )).

thf(ax_intersection,axiom,(
    ( intersection
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          & ( in @ U @ S2 ) ) ) ) )).

thf(ax_union,axiom,(
    ( union
    = ( ^ [S1: ( $i > $o ),S2: ( $i > $o ),U: $i] :
          ( ( in @ U @ S1 )
          | ( in @ U @ S2 ) ) ) ) )).

%----The distributivity of union over intersection.
thf(thm_distr,conjecture,(
    ! [A: ( $i > $o ),B: ( $i > $o ),C: ( $i > $o )] :
      ( ( union @ A @ ( intersection @ B @ C ) )
      = ( intersection @ ( union @ A @ B ) @ ( union @ A @ C ) ) ) )).
%------------------------------------------------------------------------
```

A.3 Cantor's Theorem

```
%------------------------------------------------------------------------
thf(surjectiveCantorThm,conjecture,(
   ~ ( ? [G: $i > $i > $o] :
       ! [F: $i > $o] :
       ? [X: $i] :
         ( ( G @ X )
         = F ) ) )).
%------------------------------------------------------------------------
```

Bit Inference

Nachum Dershowitz

16 David Avidan St., Tel Aviv, Israel
nachumd@gmail.com

Abstract. Bit vectors and bit operations are proposed for efficient propositional inference. Bit arithmetic has efficient software and hardware implementations, which can be put to advantage in Boolean satisability procedures. Sets of variables are represented as bit vectors and formulæ as matrices. Symbolic operations are performed by bit arithmetic. As examples of inference done in this fashion, we describe ground resolution and ground completion.

"It does take a little bit of inference."
– Tony Fratto, Deputy Press Secretary, USA

1 Introduction

Boolean satisfiability, though NP-complete, is a problem that is solved on a daily basis with real-life industrial instances comprising millions of variables and clauses. See, for example, [18].

1.1 The Problem

Suppose B is a Boolean formula and $p_1, \ldots, p_v$ are its propositional variables. The *Boolean satisfiability (SAT) problem* is to find an assignment of truth values (0 and 1) to a subset of the variables, such that the formula becomes a tautology, or else to determine that no such assignment exists, in which case the formula is *unsatisfiable*.

Formulæ are often framed in clausal form. A *literal* is any variable p_j or its negation $\overline{p_j}$. A *clause* c is a (multi-) set of positive and negative literals, intending their disjunction. A *(clausal) formula* C is a set or list of clauses, intending their conjunction.

1.2 An Idea

Bit arithmetic enjoys efficient software and hardware implementations. These can be put to great advantage in satisfiability procedures. Sets of variables can be represented as bit vectors, rather than as (linked) lists, or tries. Formulæ would be represented as matrices, rather than as linked lists or binary decision diagrams [5]. Symbolic operations are, accordingly, replaced by bit arithmetic.

1.3 Related Work

There has been considerable work on the use of reconfigurable hardware for SAT solving in general or for individual instances (e.g. [24,22]). In contrast, here we are interested in leveraging the native operations of binary hardware for the problem.

1.4 This Paper

The use of bit operations on large bit arrays for the purpose of large-scale propositional inference, as elaborated here, appears to be novel.

The next section shows how formulæ are encoded as vectors of bits. As examples of the use of bit operations, the following two sections consider two important families of propositional inference, namely, ground resolution and ground completion. *Ground resolution* is the resolution rule for variable-free clauses, as used for SAT in [8]. *Ground completion* is an inference rule for variable-free equations, using equations from left-to-right to replace "equals-by-equals". The final two sections discuss aspects of the practicality of the suggestion.

2 Representation

A clause c can be represented by two bit vectors $c^0[1{:}\,v]$ and $c^1[1{:}\,v]$, where v is the number of bits in the vector, $c^0[j] = 1$ iff the negative literal $\overline{p_j}$ occurs in c, and $c^1[j] = 1$ iff the positive literal p_j occurs therein. Thus, a variable p_k (or literal $\overline{p_k}$) is identified with the vector containing a single 1 in position k (or $k + v$, respectively). Let c also denote the $2v$-bit-long concatenation of c^0 and c^1, symbolized $c^0 \frown c^1$, and c^* the reverse concatenation $c^1 \frown c^0$. To encode a tautological clause "true", one can add a bit in the 0th position, $c[0]$, to clauses c, and use $\top$ to abbreviate the corresponding vector p_0.

The standard set operations will denote the corresponding bit-vector functions. For example, $\cap$ represents logical-and and $\varnothing$ is the zero-vector, which corresponds to the value *false*. So, if $c^0 \cap c^1 \neq \varnothing$, then c is tautological, as it includes both a literal and its negation. Symmetric-difference (exclusive-or) is $\oplus$. Set difference can be obtained in two steps when it is not directly available: $x \setminus y = x \cap \overline{y}$. We will let $\|c\|$ count the number of ones (the "population count") in vector c. Inequalities of bit vectors treat the low-index bits as most significant. It is customary to also use 0 and 1 for *false* and *true*, respectively.

A *binomial* equation $e^L = e^R$ between Boolean monomials (products of propositional variables) can likewise be represented as two bit vectors $e^L[1{:}\,v]$ and $e^R[1{:}\,v]$, where $e^L[j] = 1$ iff the variable p_j occurs in the left side e^L and $e^R[j] = 1$ iff it occurs in the right side e^R. In this case, the most significant bits $e^L[0]$ and $e^R[0]$ can conveniently be set to indicate the monomial 0 (*false*), regardless of the values of other bits (thinking of the zero-bit as indicating a 0-factor). Thus, p_0 represents the truth constant *false*, but so does any vector with its most significant bit on. Accordingly, the truth constant *true* is denoted by the zero-vector (empty monomial) $\varnothing$.

A list C of n clauses $c_1, \ldots, c_n$ may be represented as a pair of $n \times (v+1)$ matrices, C^0 and C^1, where $C^r[i,j] = 1$ iff $c_i^r[j] = 1$ (for $r = 0, 1$). To refer to the whole jth column, one can write $C^r[*, j]$ ($r = 0$, 1, or blank). All or half of the ith row, $C^r[i, *]$, is just c_i^r ($r = 0$, 1, or blank, for left half, right half, or both halves, respectively). Similarly, a list of n Boolean equations may be represented as a pair of matrices, C^L and C^R, for left and right sides of equations.

3 Resolution

A clause is *empty*, and hence unsatisfiable, if $c = \varnothing$ (that is, $\|c\| = 0$). A clause is a *unit* if $\|c\| = 1$, which coerces the truth value of its one literal. A clause is *trivial* (tautological), and may be deleted, if $c^0 \cap c^1 \neq \varnothing$, since it disjoins a literal and its complement. To delete a clause, we will set its (high-order) 0-bit to 1. Two clauses c and d *resolve on* p_k if $c^* \cap d = p_k$, for some (positive or negative) literal p_k, producing a new clause $(c \cup d) \setminus (p_k \cup p_k^*)$. The resultant clause may be empty or a unit, but resolving non-units yields a non-empty clause.

Resolution provers invariably include simplification stages, such as unit propagation and subsumption, which we discuss next.

3.1 Unit Propagation

A unit clause c propagates and simplifies clause d if $c^* \subseteq d$, in which case the result is $d' = d \setminus c^*$. If the result d' is empty ($d = c^*$), the problem is unsatisfiable. If the result is a unit, then d' can be used in the same fashion. *Binary constraint propagation* (*BCP*) is the repeated application of subsumption by units and unit propagation – until no further simplifications are possible. BCP is a central component of the Davis-Putnam-Logemann-Loveland backtracking SAT procedure [7], and its modern incarnations. It is expensive (typically consuming 80–90% of the running time), but is not necessary for completeness (and can significantly degrade proof search; see [11]).

Let n be the number of clauses, and let $u[0\colon 2v]$ be a bit-vector of length $2v + 1$. At the conclusion of the algorithm in Fig. 1, all the units obtained by propagating the clauses of C will be marked in u. The n-step main loop repeats at most v times. An empty clause c_i means the problem is unsatisfiable. To delete a row, we set $c_i := \top$; it would be enough to let $c_i[0] := 1$. The matrix can be compacted by removing the deleted rows (at any juncture) and/or the columns marked in u (after any complete pass).

3.2 Subsumption

Clause c *subsumes* clause d if $c \subseteq d$, in which case d is superfluous. For this to be the case, we must have $c \leq d$, as binary numbers, but this is an insufficient condition. Using standard operations, $c \subseteq d$ iff $c \cup d = d$.

Subsumption is more expensive than unit propagation and should normally be preceded by BCP. It can be implemented like sorting, with the addition of

```
u := ∅
b := true
while b do
    b := false
    for i := 1, ..., n do
        if u ∩ c_i = ∅
          then c_i := c_i \ u*
               if c_i = ∅ then fail
               if ||c_i|| = 1
                 then u := u ∪ c_i
                      b := true
          else c_i := ⊤
```

Fig. 1. Binary constraint propagation

```
for i := 1, ..., n − 1 do
    if c_i ≠ ⊤ then
      for j := i + 1, ..., n do
        if c_i > c_j
          then if c_j ⊆ c_i
                 then c_i := c_j
                      c_j := ⊤
                 else c_j :=: c_i
          else if c_i ⊆ c_j
                 then c_j := ⊤
```

Fig. 2. Subsumption

checking whether the smaller of any pair subsumes the larger, in which case, the larger is deleted – for a cost of $O(n \lg n)$ vector-operations to check all clauses. Deleted rows should be removed. For an n^2 version, à la selection sort, see Fig. 2. Subsumption is often not cost-effective in standard implementations, but might be in this context. (Satellite [3,12], interestingly, does use bit vectors to estimate the applicability of subsumption.)

Other implementations of these algorithms, taking advantage of matrix operations, are conceivable.

3.3 The Davis-Putnam Resolution Procedure

The original Davis-Putnam (DP) procedure resolves clauses, variable by variable [8]. See Fig. 3. There are various heuristics for ordering the variables, such as choosing the one that appears in the most clauses. Columns can be presorted to reflect such policies. BCP can be incorporated, and perhaps subsumption, taking into account that the literals p_k and $\overline{p_k}$ are removed with each iteration on k.

```
m := n
for k := 1, ..., v do
    n := m
    for i := 1, ..., n − 1 do
        for j := i + 1, ..., n do
            if c_i ∩ c*_j ⊂ (p_k ∪ ¬p_k)
              then m := m + 1
                   c_m := (c_i ∪ c_j) \ (p_k ∪ ¬p_k)
                   if c_m = ∅ then fail
```

Fig. 3. Davis-Putnam resolution

```
m := 0
k := n
while k > m do
    n := m
    m := k
    for i := 1, ..., m do
        for j := n + 1, ..., m do
            if e_i^L ⊆ e_j^R
              then e_j^R := e_j^R \ e_i^L ∪ e_i^R
            if e_i^L ⊆ e_j^L
              then e_j^L := e_j^L \ e_i^L ∪ e_i^R
                   if e_j^R > e_j^L then e_j := e*_j
            else if e_i^L ∩ e_j^L ≠ ∅
              then k := k + 1
                   e_k^L := e_j^L \ e_i^L ∪ e_i^R
                   e_k^R := e_i^L \ e_j^L ∪ e_j^R
                   if e_k^R > e_k^L then e_k := e*_k
```

Fig. 4. Knuth-Bendix completion

4 Completion

Knuth-Bendix *completion* [14], and its extensions, repeatedly finds overlaps between equations (using only the larger side of any equation), to infer new equations. (In contrast, paramodulation [23] looks at both sides of equations.) Equational reasoning provides an alternative inference paradigm to propositional reasoning, with equations in completion playing an analogous rôle to clauses in resolution.

We are interested in the ground (variable-free) case of completion, where the operations are associative and commutative [1,16,17]. As examples of completion in the realm of Boolean formulæ, we will consider ground Horn-clause theories and Gaussian elimination over $\mathbb{Z}_2$.

```
b := true
while b do
    b := false
    for i := 1, ..., n − 1 do
        for j := i + 1, ..., n do
            if e_i^L ⊆ e_j^L
              then e_j^L := e_j^L \ e_i^L ∪ e_i^R
                   if e_j^R > e_j^L then e_j^L :=: e_j^R
                   b := true
            if e_i^L ⊆ e_j^R
              then e_j^R := e_j^R \ e_i^L ∪ e_i^R
                   b := true
```

Fig. 5. Inter-reduction

4.1 Horn-Clause Completion

A clause is *Horn* if it has at most one positive literal. A Horn clause $p_0 \vee \neg p_1 \vee \cdots \vee \neg p_n$ is equivalent to the binomial equation $p_0 p_1 \cdots p_n = p_1 \cdots p_n$; a negative Horn clause $\neg p_1 \vee \cdots \vee \neg p_n$ is equivalent to the monomial equation $p_1 \cdots p_n = 0$. See [4] for details regarding such representations.

Two equations e_i and e_j are *critical* iff $e_i^L \cap e_j^L \neq \varnothing$. The *critical equation* (or *critical pair*) is $e^L = e^R$, where $e^L := e_j^L \setminus e_i^L \cup e_i^R$ and $e^R := e_i^L \setminus e_j^L \cup e_j^R$. Critical equations may need to be oriented. Knuth-Bendix (KB) completion (or the analogous Gröbner basis construction [6]) is the repeated generation of critical pairs, interleaved with inter-reduction.

In this manner, completion serves as the inference engine, generating critical pairs from the equational representation of Horn clauses, as shown in Fig. 4.

4.2 Reduction

A major component of completion is *simplification*, akin to *demodulation* [23], by which we mean using equations in one direction to "simplify" other equations (with respect to some measure).

An oriented equation $e^L = e^R$ is *unitary* and can be used to simplify in any of the following three cases:

- *Positive Unit.* If $e^R = \varnothing$, then the equation signifies $e^L = 1$ (since we agreed in Sect. 2 to interpret $\varnothing$ as truth). It follows that $p_i = 1$ for every $p_i \in e^L$. Apply $e^R = \varnothing$ to a monomial m by removing the (superfluous) positive bits: $m := m \setminus e^R$.
- *Negative Unit.* If $\|e^L\| = 1$ and $e^R[0] = 1$, then $p_k = 0$ for the $p_k \in e^L$. Apply $p_k = 0$ by zeroing any monomial in which it appears: **if** $m[k]$ **then** $m[0] := 1$.
- *Unit Equivalence.* If $\|e^L\| = \|e^R\| = 1$ and $e^L[0] = e^R[0] = 0$, then $p_k = p_j$ for the $p_k \in e^L$ and $p_j \in e^R$. Apply $p_k = p_j$ by replacing occurrences of p_k with p_j: **if** $m[k]$ **then** $m[j] := 1$.

```
i := 1
k := 1
while k ≤ v ∧ i ≤ n do
      m := i
      while m ≤ n ∧ ¬c_m[k] do m := m + 1
      if m ≤ n then
         c_m :=: c_i
         for j := 1, . . . , i − 1, m + 1, . . . , n do
               if c_j[k] then c_j := c_j ⊕ c_i
         i := i + 1
      k := k + 1
```

Fig. 6. Gaussian elimination

The results of such unit simplifications can propagate as in resolution.

More generally, an equation $e^L = e^R$ can be used to simplify a monomial m provided all the variables in e^L appear in m, that is, when $e^L \subseteq m$. The *rewrite step* is the assignment $m := m \setminus e^L \cup e^R$. If we use the 0-bit to signify the term 0, as explained above, then reducing products to 0 works as expected.

The lexicographic ordering of monomials is ordinary bit-string inequality. An equation c needs to *reoriented* if $e^R > e^L$, which may transpire after reducing a left side. Other orderings are possible.

To inter-reduce a system C of equations, applying all equations to all equations, as much as possible, first sort C in ascending order according to $\langle \|e^R\| - \|e^L\|, e^L, e_1 \rangle$ and then apply the algorithm in Fig. 5. The idea is that reducing with a "rewrite rule" $\ell \to r$ decreases the binary value of the string it is applied to by $\|\ell\| - \|r\|$, and, long range, one wants to maximize the decreases obtained with each reduction, so as to converge as quickly as possible. This naïve program can presumably still require exponentially many vector operations, but hopefully much better algorithms for inter-reduction can be devised (compare the non-commutative case [13,21]). One may prefer to limit reduction to equations with few variables on the left.

4.3 Gaussian Elimination

A linear equation over $\mathbb{Z}_2$ takes the form $P = 0$, where P is an exclusive disjunction of some of the propositional variables $p_1, \ldots, p_v$. (Since we are using $\oplus$, coefficients are 0 or 1.)

We represent an equation $P = 0$ as a bit vector c of length $v+1$, where $c[k] = 1$ iff p_k is a summand in P and p_0 is the constant 1. Adding (or subtracting) a linear equation c to d is just $d := d \oplus c$. A standard quadratic (vn vector operations) Gaussian elimination procedure is given in Fig. 6.

When (after elimination, say) $\|c\| \leq 2$, the equation c is unitary and is of one of the following three forms: $p_k = 0$, $p_k = 1$, or $p_k = p_j$, for some $k \geq 1$ and $1 \leq j \neq k$.

```
LOAD 0, ci0     |   LOAD 0, ci1
OR 0, cj0       |   OR 0, cj1
LOAD 2, ci0     |   LOAD 2, ci1
AND 2, cj1      |   AND 2, cj0
DIFF 0, 2       |   DIFF 0, 2
STORE 0, c0     |   STORE 0, c0
```

Fig. 7. A resolution step in an assembly language

4.4 Combining the Two

For non-Horn clauses, one needs also to incorporate negation in some form. The BinLin representation of propositional formulæ, proposed in [9,10], uses a combination of equations between monomials and linear equations over $\mathbb{Z}_2$ to represent propositional formulæ in exclusive-or (Boolean ring) normal-form. It provides an alternative to other propositional satisfiability procedures, whether search-based, saturation-based, or hybrid intersection-based methods. In this formalism, variables and equations are added in a satisfiability-preserving fashion, to obtain a set of binomial equations and a set of linear Boolean equations. The binomials undergo inter-reduction and the linear equations undergo Gaussian elimination. Unitary equations are propagated among both sets. This method, too, can be implemented naturally within the framework proposed here.

5 Implementation

Most of the bit-vector operations used in the above sections are readily available on digital computers. Some processors, even way back to the IBM Stretch, provide a hardware instruction for the number of ones in a machine word; in any case, computing $\|c\|$ requires only a few machine instructions [2, No. 169]. Most operations are also available in many software languages (e.g. C). They are all easy to implement in general-purpose or special-purpose hardware.

For example, resolving two single-word (or double-word – for machines with double-word operations) clauses requires approximately 12 machine instructions. Thus v variables require $12\lceil v/w \rceil$ instructions on a w-bit machine. For example, if $w = 64$ and $v = 1000$, fewer than 200 machine instructions are needed. See Fig. 7. This should be contrasted with the large number of machine operations used in a pointer-based implementation.

For large (but presumably sparse) vectors, (iterated) summary bits should prove helpful. (The *summary bit* for a subvector x is 0 iff $x = 0$.) Column operations, such as erasing all occurrences of a true propositional variable, may be sped up by also maintaining transpose matrices [20].

Industrial-strength problems can easily involve hundreds of thousands of variables and millions of clauses. The storage requirements for a bit matrix of that size is in the hundreds-of-gigabyte range. Given enough storage, full-fledged $n \lg n$ subsumption would take a few minutes on a 5000 MIPS 64-bit machine.

6 Discussion

Davis-Putnam resolution is a saturation-based methods for checking propositional satisfiability. The original set of clauses is satisfiable if and only if resolution terminates without having derived the empty clause. Similarly, Knuth-Bendix completion derives the contradiction $1 = 0$ if and only if the input clauses are unsatisfiable. Thus, both methods (Figs. 3 and 4) repeatedly add rows to the matrices of formulæ.

Saturation is often considered too costly in practice. Instead, a backtrack search [7] – based on the clausal representation with unit propagation and subsumption – can easily be built around the above procedures. One simple way to keep track would be to mark rows of the matrix that are added or deleted with the search level. (Instead of changing a row, one would delete and add.) After a significant number of assignments, it may pay to compact the matrix.

Similarly, a recursive-learning intersection-based method [15,19], combining limited saturation, generous simplification, and judicious search can be designed.

The algorithms given here are readily adaptable to highly parallel vector or array architectures. Experiments with simulations are needed to evaluate their practical feasibility.

Acknowledgement

I thank Guan-Shieng Huang for many ideas, discussions and vegetarian meals.

References

1. A. M. Ballantyne and D. S. Lankford. "New decision algorithms for finitely presented commutative semigroups". *J. Computational Mathematics with Applications*, vol. 7, 1981, pp. 159–165
2. M. Beeler, R. W. Gosper, and R. Schroeppel. "HAKMEM", Artificial Intelligence Memo No. 239, Massachusetts Institute of Technology, A. I. Laboratory, Feb. 1972. Available at `http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html`
3. A. Biere, "Resolve and expand, *Proc. of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, Lecture Notes in Computer Science, vol. 3542, Springer-Verlag, Berlin, 2005, pp. 59–60
4. M.-P. Bonacina and N. Dershowitz. "Canonical Inference for Implicational Systems", Proc. of the 4th Intl. Joint Conference on Automated Reasoning, A. Armando, P. Baumgartner, and G. Dowek, eds., Lecture Notes in Computer Science, Springer-Verlag, Berlin, Aug. 2008, pp. 380–395
5. R. E. Bryant. "Symbolic Boolean manipulation with ordered binary-decision diagrams", *ACM Computing Surveys*, vol. 24, 1992, pp. 293–318
6. B. Buchberger. "Gröbner bases: An algorithmic method in polynomial ideal theory". Bose, N. K., ed., *Multidimensional Systems Theory*, Reidel, 1985, pp. 184–232
7. M. Davis, G. Logemann, and D. Loveland. "A machine program for theorem proving", *Communications of the ACM*, vol. 5, 1962, pp. 394–397
8. M. Davis and H. Putnam. "A computing procedure for quantification theory". *J. of the ACM*, vol. 7, no. 3, July 1960, pp. 201–215.

9. N. Dershowitz, J. Hsiang, G.-S. Huang and D. Kaiss. "Boolean ring satisfiability", *Proc. 7th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2004)*, May 2004, pp. 281–286
10. N. Dershowitz, J. Hsiang, G.-S. Huang and D. Kaiss. "Boolean rings for intersection-based satisfiability", *Proc. of the 13th Intl. Conf. on Logic for Programming and Artificial Intelligence and Reasoning*, M. Hermann and A. Voronkov, eds., Lecture Notes in Computer Science, vol. 4246, Springer-Verlag, Berlin, Nov. 2006, pp. 482–496
11. N. Dershowitz and A. Nadel, "From total assignment enumeration to a modern SAT solver", submitted
12. N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination", *Proc. of the 8th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'05)*, Lecture Notes in Computer Science, vol. 3569, Springer-Verlag, Berlin, 2005, pp. 61–75
13. J. Gallier, P. Narendran, D. Plaisted, S. Raatz, and W. Snyder. "An algorithm for finding canonical sets of ground rewrite rules in polynomial time", *Journal of Association for Computing Machinery*, vol. 40, no. 1, 1993, pp. 1–16.
14. D. E. Knuth and P. B. Bendix. "Simple word problems in universal algebras". J. Leech, ed., *Computational Problems in Abstract Algebra.* Oxford: Pergamon Press, 1970, pp. 263–297
15. W. Kunz. "Recursive learning: A new implication technique for efficient solutions to CAD problems — test, verification, and optimization", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, 1994, pp. 1143–1158.
16. C. Marché. "On ground AC-completion". *Proc. of the 4th Intl. Conf. on Rewriting Techniques and Applications*, R. V. Book, ed., Lecture Notes in Computer Science, vol. 488, Springer-Verlag, Berlin, Apr. 1991, 411–422
17. P. Narendran and M. Rusinowitch. "Any ground associative commutative theory has a finite canonical system". *J. Automated Reasoning*, vol. 17, no. 1, Aug. 1996, pp. 131–143
18. M. R. Prasad, A. Biere, and A. Gupta. "A survey of recent advances in SAT-based formal verification". *Software Tools for Technology Transfer*, vol. 7, no. 2, 2005, pp. 156–173
19. M. Sheeran and G. Stålmarck. "A tutorial on Stålmarck's proof procedure for propositional logic". *Proc. of the 2nd Intl. Conference on Formal Methods in Computer-Aided Design*, Lecture Notes in Computer Science, Springer-Verlag, Nov. 1998, pp. 82–99
20. I. Skliarova and A. B. Ferrari. "The design and implementation of a reconfigurable processor for problems of combinatorial computation". *J. Syst. Archit.*, vol. 49, nos. 4–6, Sep. 2003, pp. 211–226
21. W. Snyder. "A fast algorithm for generating reduced ground rewriting systems from a set of ground equations," *J. of Symbolic Computation*, vol. 15, no. 4, 1993, pp. 415–450
22. T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya. "Solving satisfiability problems using reconfigurable computing". *IEEE Trans. VLSI Systems*, vol. 9, no. 1, Feb. 2001, pp. 109–116
23. L. Wos, G. A. Robinson, D. F. Carson, and L. Shalla. "The concept of demodulation in theorem proving". *J. of the ACM*, vol. 14, no. 4, Oct. 1967, pp. 698–709
24. P. Zhong, P. Ashar, S. Malik, and M. Martonosi. "Using reconfigurable computing techniques to accelerate problems in the CAD domain: A case study with Boolean satisfiability". *Proc. of the Design Automation Conf. (DAC)*, 1998, pp. 194–199

Collaborative Programming: Applications of logic and automated reasoning

Timothy L. Hinrichs

University of Chicago
tlh@uchicago.edu

Abstract. Collaborative Programming is characterized by groups of people issuing instructions to computer systems. Collaborative Programming languages differ from traditional programming languages because instruction sets can be incomplete and conflicting, and more of the burden for efficient execution is placed on the computer system. This paper introduces Collaborative Programming and through the discussion of two practical examples argues that tools from logic and automated reasoning form a good foundation for Collaborative Programming technology while at the same time illustrating the need for nonstandard automated reasoning techniques.

1 Introduction

Collaborative Programming comprises all those settings where groups of people issue instructions to computer systems. In contrast to traditional programming languages, Collaborative Programming languages must make combining instruction sets from different parties straightforward and may allow users to express incomplete and conflicting[1] instruction sets. An *incomplete* instruction set may only say what to do some of the time or what actions the system is forbidden from performing. A *conflicting* instruction set may simultaneously instruct the system to perform some action and forbid the system from performing that same action. Technology that supports Collaborative Programming must be able to combine independently authored instruction sets and be tolerant of incompleteness and conflicts.

The notion of Collaborative Programming was developed as a pedagogical device for explaining to researchers in traditional programming languages and systems (e.g. networks, operating systems) the benefits and limitations of logical languages and automated reasoning as compared to more traditional approaches. The word "collaborative" was chosen to capture situations where statements made by independent parties must be combined, a simple operation in logical languages. The word "programming" was chosen to capture situations in which the statements made by independent parties can be construed as instructions to

[1] Here we use the word "conflicting" as opposed to "inconsistent" to differentiate the informal notion of a disagreement and the proof and model theoretic notions of consistency.

a computer system, which is often the case when the statements are made in a formal language. The concept of Collaborative Programming covers situations that leverage the order-irrelevance and formal semantics of logical languages.

The connection between Collaborative Programming and logical languages was forged because of the need and ability to combine instruction sets; however, the connection runs deeper than that. In collaborative settings, it is very natural for users to submit incomplete and conflicting instruction sets. Sometimes people only have opinions on some issues; thus, for a language to reflect a user's true intentions, it must allow users to express incomplete instruction sets, a natural feature of many logical languages. Likewise, when collaborating, people rarely agree on everything; hence, a Collaborative Programming language must allow users to express disagreements, another feature of logical languages. Thus, logical languages are a natural foundation for Collaborative Programming languages, which means that Collaborative Programming language implementations rely on tools from automated reasoning.

Some of the most celebrated tools in automated reasoning, e.g. first-order theorem provers, are designed to detect a particular kind of conflict: a logical inconsistency. More precisely, they determine whether or not an inconsistency exists. As we illustrate in this paper, Collaborative Programming applications sometimes require knowing more than whether or not a conflict exists; they must act based on the type of conflict that occurred. To meet this requirement, theorem provers for Collaborative Programming applications must implement a paraconsistent entailment relation [14]: one that coincides with classical entailment for consistent theories but is more discerning for inconsistent theories.

Paraconsistent theorem provers must overcome an additional computational burden as compared to traditional programming languages. Given a set of instructions issued in a logical language, a computer system must determine which action to perform by analyzing those instructions, resolving conflicts, and filling in gaps. Unlike traditional programming languages, where computing the next action is guaranteed to be fast, computing the next action in a Collaborative Programming setting might require significant computation, which is especially worrisome for real-world applications where efficiency guarantees are important. To alleviate such concerns, we advocate custom-designing a Collaborative Programming language for each application so that it is expressive enough to be useful but no less efficient than is tolerable.

When custom-designing a Collaborative Programming language based on logic, one must choose which style of logic to use. In this paper we consider two specific logics, FHL and DATALOG$^\neg$, that from the perspective of Collaborative Programming represent two interesting language classes: classical logic and logic programming. FHL is a decidable fragment of first-order logic that allows arbitrary quantification (syntactically). DATALOG$^\neg$ is a language for describing and querying relational databases, perhaps the most successful application of logic in computer science. These two languages were chosen because FHL provides the opportunity to confront paraconsistency, while DATALOG$^\neg$ demonstrates that classical logics are not the only option for Collaborative Programming.

This paper examines Collaborative Programming languages for two practical applications: logical spreadsheets (Section 3) and authorization languages (Section 4). In each case, the strengths and weaknesses of FHL and DATALOG$^\neg$ are examined, and in the end one is chosen as the foundation of the language; additionally, issues surrounding conflicts and incompleteness for the chosen language are illustrated and resolved. Finally, we make some closing remarks (Section 5).

2 Preliminaries

The two languages studied in this paper, FHL and DATALOG$^\neg$, are well-known; we use common conventions for their syntax and semantics. FHL, a classical logic, is first-order logic with equality and the following three restrictions: no function constants, a domain closure assumption (DCA), and a unique names assumption (UNA). In this logic, the objects in the universe of every model are exactly the object constants in the language. We call this logic Finite Herbrand Logic (FHL) because the only models of interest are the finite Herbrand models.

The definitions for FHL's syntax are the same as for function-free first-order logic. The definitions for a model and for satisfaction are standard but are simplified to take advantage of the UNA and DCA. A model in FHL is a set of ground atoms from the language. A model satisfies all the ground atoms included in the set. Satisfaction is defined as usual for the Boolean connectives applied to ground sentences. Satisfaction of non-ground sentences reduces to satisfaction of ground sentences. Free variables are implicitly universally quantified. $\forall x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for every object constant a. $\exists x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for some object a.

The other language of interest, DATALOG$^\neg$, is DATALOG with stratified negation. Again, the definitions for its syntax are standard, and we focus on semantics. A model for DATALOG$^\neg$ is the same as that for FHL: a set of ground atoms; however, in contrast to FHL where sentences may be satisfied by more than one model, a set of DATALOG$^\neg$ sentences is always satisfied by exactly one model. Without negation, that model is the smallest one (ordered by subset) that satisfies the sentences under the FHL definition of satisfaction. With negation, the stratified semantics [15] use minimality criteria to choose one model out of all those that satisfy the sentences under the FHL definition.

A set of sentences is satisfiable (or consistent) when there is at least one model that satisfies it. Logical entailment is defined as usual: $\Delta \models \phi$ if and only if every model that satisfies Δ also satisfies ϕ. Entailment for FHL is coNEXPTIME-complete [5], and entailment for DATALOG$^\neg$ is NEXPTIME-complete, e.g. [16].

FHL and DATALOG$^\neg$ are similar because they are both Herbrand-based logics. They are different in that FHL allows a sentence set to be satisfied by multiple models, whereas a DATALOG$^\neg$ sentence set is always satisfied by exactly one model. For the purposes of this paper, the most important consequence of this distinction is that FHL can express true disjunction (entailing/satisfying a disjunction without entailing/satisfying any disjunct) but DATALOG$^\neg$ cannot.

3 Use Case: Logical Spreadsheets

One area of research, popular enough to support a dedicated workshop in 2005 and a DARPA funding opportunity (in the small business sector) in 2004 [9], investigates the application of logic and automated reasoning to bring about the next generation of spreadsheets for the personal computer. These logical spreadsheets remove some of the limitations of traditional spreadsheets. Instead of equations that specify how to compute the value of one cell given the values of other cells, logical spreadsheeets accept arbitrary logical formulae, which allows updates to propagate in any direction and cells to be constrained to obey many-to-many relationships.

For example, using a logical spreadsheet one can require two cells to be assigned the same value; fill in the value of either cell, and the other one updates automatically. In addition, it is possible to constrain one cell to contain a postal code and another cell to contain a city. The postal code is not sufficient to compute the city, nor is the city sufficient to compute the postal code. Nevertheless, choosing a city restricts the possible postal codes, and vice versa.

Logical spreadsheets allow users to specify a set of constraints on the cells in the spreadsheet and then provide visual cues to indicate which values do not satisfy the constraints. Those visual cues include highlighting cells whose values conflict with the constraints and showing a list of values for any given cell that satisfy the constraints given the values of the other cells.

Particularly well-known examples of logical spreadsheets are the HTML forms found on the web. When ordering merchandise from e-commerce web sites, a form that asks for billing information often includes constraints on the combinations of values that can be entered, e.g. the city and postal code must be compatible. Often, web programmers use Javascript to check those constraints as the user enters information. When a constraint violation occurs, an error message appears somewhere on the page.

The difficulty with using Javascript to check constraints is that if the constraints change, the Javascript may require a substantial rewrite. Research into logical spreadsheets has the potential benefit that a web programmer could write down the necessary constraints for the web form elements in a logical language, and the Javascript for checking those constraints would be generated automatically. Small constraint changes that result in large Javascript changes would no longer be problematic because those large changes would be auto-generated.

Different approaches to logical spreadsheets expose different languages for users to express constraints. The language presented here is based on FHL and follows the presentation in [8]. Cells in the spreadsheet correspond to monadic predicates, and a (partial) cell assignment corresponds to a set of ground atoms. Constraints on a spreadsheet are FHL sentences.

For example, to require two cells named $cell_1$ and $cell_2$ to contain the same value, a user could enter the following sentence.

$$cell_1(x) \Leftrightarrow cell_2(x)$$

Likewise, to force the *postal* cell and the *city* cell to contain compatible values, one could write the implication

$$postal(x) \wedge city(y) \Rightarrow compatible(x, y),$$

where *compatible* is appropriately defined. Assigning cell *city* the value *paris* is represented by the atom $city(paris)$.

Conflicts in this language correspond to inconsistent FHL theories. This is problematic because using the traditional notions of satisfaction and entailment, there is no way to differentiate one conflict from another, which is vital information for visually indicating which cells fail to satisfy the constraints.

For example, consider again the web form where two cells are required to contain the same value, and a city cell and a postal code cell are required to contain compatible values. The constraints are the two sentences shown above. Assigning $cell_1$ and $cell_2$ different values causes an inconsistency, i.e. there are no models that satisfy the constraints together with the assignments to $cell_1$ and $cell_2$. This means that every sentence in the language is entailed. Compare this conflict with a conflict that occurs because the city and postal code cells were assigned incompatible values. Again, the theory is inconsistent, which means there are no satisfying models, and all sentences are entailed. Neither satisfaction nor entailment is sufficient for providing the user feedback as to which cells conflict with each other.

Such problems are addressed by work on paraconsistent logics, e.g. [14]. A paraconsistent logic is one in which an inconsistent theory does not entail all logical sentences. The approach described in [8], called *existential entailment* and denoted $\models_E$, combines the traditional notions of satisfaction and entailment in a simple way. In the case of consistent theories, traditional entailment and existential entailment coincide, but in the case of inconsistent theories, existential entailment isolates one conflict from another.

Intuitively, the problem with traditional entailment is that an inconsistent premise set entails every sentence, even if that inconsistency has nothing to do with the sentence in question. For example, the three premises below are inconsistent, which means that both the sentences $q(a)$ and $\neg q(a)$ are entailed.

$$\begin{array}{l} p(a) \\ \neg p(a) \\ q(a) \end{array} \tag{1}$$

However $q(a)$ would be entailed even without the inconsistency, but $\neg q(a)$ is only entailed because of the inconsistency. Existential entailment differentiates these two cases by requiring a satisfiable premise set for proving a conclusion.

Definition 1 (Existential Entailment [8]). *A set of sentences Δ existentially entails a sentence ϕ ($\Delta \models_E \phi$) if and only if there is some satisfiable Δ' that is a subset of Δ such that $\Delta' \models \phi$.*

Existential entailment can be employed as follows to pinpoint those cells in a spreadsheet that conflict with the constraints. Suppose that the constraints are

satisfiable and named Δ and that the assignments of values to cells is Γ. Recall that assigning cell p to value a is represented as $p(a)$. Cell value $p(a)$ conflicts with the constraints and other cell values whenever $\Delta \cup \Gamma$ existentially entails the negation of $p(a)$.

Definition 2 (Logical Spreadsheet Conflict). *Cell p assigned to value a conflicts with the spreadsheet constraints Δ and the partial cell assignment Γ exactly when $\Delta \cup \Gamma \models_E \neg p(a)$.*

We can view Example 1 from above as a set of constraints and cell values for a spreadsheet with a cell p and a cell q. Cell p, having been assigned the value a, should be highlighted as a conflict because $\neg p(a)$ is existentially entailed (by the singleton, satisfiable premise set $\{\neg p(a)\}$). But, cell q assigned a should not be highlighted as a conflict because $\neg q(a)$ is not existentially entailed.

Using this definition of conflict, every time a user changes the value of a cell, the logical spreadsheet must compute existential entailment. Moreover, one cell assignment can cause other cells to violate constraints, meaning that multiple existential entailment queries must be answered for each cell assignment change. Thus, it is important that the computation of existential entailment runs efficiently enough for the spreadsheet to provide real-time visual cues to the user.

Our current implementation focuses on the web form application of logical spreadsheets. It converts a given set of constraints into conjunctive database queries that when evaluated compute existential entailment. Those queries are evaluated by the browser each time a cell value is changed using an in memory database implemented in Javascript. Preliminary testing appears promising both in ease of implementation and performance.

It is noteworthy that the choice to use FHL as the constraint language was not made arbitrarily. When compared to DATALOG$^\neg$, FHL is better suited as the foundation of the constraint language because it can express disjunction[2], whereas DATALOG$^\neg$ cannot. The importance of disjunction for logical spreadsheets can be seen in two ways.

First, FHL semantics is closer in spirit to a natural formalization of logical spreadsheets than is DATALOG$^\neg$. From a mathematical perspective, a logical spreadsheet maps a set of constraints and a partial assignment of cells to the set of all consistent extensions to that assignment. Similarly, FHL semantics maps a set of logical sentences to the set of models that satisfy those sentences. Both map the input to a set of alternatives. In contrast, DATALOG$^\neg$ semantics maps a set of sentences to a single model—to a single alternative.

Second, one of the features logical spreadsheets support that traditional spreadsheets do not, bidirectional update, is intimately tied to disjunction. A simple implication such as $cell_1(a) \Leftarrow cell_2(a)$ represents two possibilities: either the premises are false or the conclusion is true. For bidirectional update to be supported, falsifying the conclusion of the implication requires falsifying the premise, and satisfying the premise requires satisfying the conclusion. These

[2] Here we mean true disjunction: in FHL a theory can entail $p \vee q$ without entailing either p or q.

two equally plausible possibilities are represented succinctly by a disjunction: $cell_1(a) \vee \neg cell_2(a)$.

Logical spreadsheets exemplify Collaborative Programming because the instructions issued by users can conflict, can be incomplete, and can come from multiple sources. Collaboration comes about in a variety of ways. In the case of web forms, the form developers contribute the constraints and the users contribute data. In the case of a standalone application, constraints might originate from different people, each with expertise in different areas of the problem. Even if all of the constraints are created by a single individual, that individual might be collaborating with herself if over time she adds new constraints to the system. Collaboration breeds conflict, and because FHL, the constraint language, is a classical logic, the traditional notion of entailment does not support the functionality promised by the logical spreadsheet paradigm; hence, a paraconsistent entailment relation must be used and implemented efficiently.

4 Use Case: Authorization languages

An active area of research in security centers around logical languages for expressing authorization policies. An authorization policy says, for example, which users can access which resources in which ways, e.g. Alice has permission to write myfile.txt. Such policies are often written by several individuals, each of whom may want to operate independently of the others. The security systems that enforce authorization policies require that every request be either allowed or denied. There is no way to simultaneously allow and deny a request, and there is no way to neither allow nor deny a request. Thus, while authorization policies are defined in collaborative settings, neither conflicting nor incomplete policies can be tolerated by security systems. Formally, an authorization policy maps requests R to either *allow* or *deny*[3].

$$R \rightarrow \{allow, deny\}$$

Despite the fact that an authorization policy is developed for a system that cannot tolerate conflicts or incompleteness, there is no reason to believe that the people collaboratively defining such a policy will disagree less or know more than people in another Collaborative Programming setting. Thus, an authorization language should be able to express conflicts and incompleteness, less people encode instructions they do not intend, yet at the same time should hide conflicts and incompleteness from the security system. Hiding conflicts and incompleteness means that the language should include mechanisms for resolving conflicts and incompleteness when they occur.

For conflicting authorization policies, where a request is both allowed and denied, there are at least two options for resolving that conflict. Deny might

[3] Depending on the setting, a request may contain a number of properties, e.g. the user, the resource, the action to be performed on the resource. For simplicity and generality, we treat a request as an opaque object.

take precedence over allow, or vice versa. It is important that the form of conflict resolution chosen is made known to users so that they can predict the results.

In FHL, it is natural to use a single distinguished predicate *allow*, and whenever an authorization request r is made, it is allowed if $allow(r)$ is entailed and denied if $\neg allow(r)$ is entailed. Conflicts amount to inconsistent theories where $allow(r)$ and $\neg allow(r)$ are both entailed. Conflict resolution is based on existential entailment as described in Section 3.

In DATALOG$^\neg$, a single predicate *allow* is insufficient for expressing conflicts. The language guarantees that if $allow(r)$ is entailed then $\neg allow(r)$ is not entailed. However, by using two distinguished predicates *allow* and *deny*, it is possible to encode conflicts and incompleteness. For any authorization request r, an authorization policy could entail $allow(r)$, $deny(r)$, both, or neither. Again, conflicts can be resolved by giving preference to either *allow* or *deny*.

For incomplete authorization policies, where a request is neither allowed nor denied, there are two separable cases. One form of incompleteness arises because the policy says nothing about a particular request. Similar to the case of conflict resolution, this form of incompleteness can be resolved by choosing either to allow or to deny the request, as the policy makes no commitment whatsoever. The other type of incompleteness, which is only possible in FHL-based languages, occurs when a request appears as a disjunctive consequence of the authorization policy. Resolving this type of incompleteness is more problematic than the first.

For example, consider an authorization policy with two FHL statements:

$$allow(r_1) \vee allow(r_2)$$
$$\neg allow(r_1) \vee \neg allow(r_2).$$

Together the statements say that either r_1 or r_2 must be allowed, and the other must be denied. Arguably, this policy is enforceable: simply make the choice. The problem is that the user may not be able to predict the result. It is imaginable that if the policy were written another way, the opposite choice might be made.

The resolution mechanism for disjunctive incompleteness requires making choices between requests, which is qualitatively different than making a choice between allow and deny. It is far easier to communicate a tie-breaking mechanism about allow and deny than about requests; moreover, it is unnatural to treat some requests differently than others when the authorization policy fails to do so. Thus, authorization languages should not be able to express disjunction.

While there are fragments of FHL that are guaranteed to be nondisjunctive, e.g. Horn clauses, DATALOG$^\neg$ has the benefit that it supports negation and limited recursion, which are difficult to support using nondisjunctive FHL. Thus, DATALOG$^\neg$ is the better choice for authorization languages.

Formalizing and implementing the conflict and incompleteness resolution mechanisms for a DATALOG$^\neg$-based language is straightforward. For example, if the conflict resolution mechanism deems that deny should override allow (a reasonable choice in the context of security), and policy completion allows all unspecified requests, the semantics for the authorization language ($\models'$) would

be defined as follows, where $\models$ is the usual DATALOG$^\neg$ semantics.

$$\Delta \models' deny(r) \text{ iff } \Delta \models deny(r)$$
$$\Delta \models' allow(r) \text{ iff } \Delta \not\models deny(r)$$

This layered approach to language design has two benefits. The core of the language ($\models$) is defined using traditional means and hence can leverage well-known tools. Those tools can be used to analyze a policy according to $\models$, identify conflicts, and inform the authors who contributed the conflicting statements; yet, at the same time, a security system can use $\models'$ to make authorization decisions using a policy without conflicts or incompleteness.

Thus, unlike FHL, which requires nonstandard automated reasoning tools for handling conflicts, conflict resolution in DATALOG$^\neg$ can be built on top of well-known techniques. This could explain the popularity of DATALOG$^\neg$ for authorization languages in the security literature [7, 2, 12, 17, 3, 13, 1, 6]. The drawback is that DATALOG$^\neg$ can only express conflicts in settings where all possible conflicts are known ahead of time. Keywords must be introduced into the language and built into the algorithms for processing that language.

5 Conclusion

Kowalski is famous for illustrating that logic can be used as a programming language, the result of which was the Logic Programming paradigm [10]. Today, the term "Logic Programming" has come to mean a particular type of logic and automated reasoning, syntactically based on implication and semantically concerned with negation as failure. Logic Programming today is consistent with Kowalski's original vision but is more narrowly defined than he intended.

Logic Programming (in Kowalski's original intent) is the right choice for industrial applications only in certain situations. The notion of Collaborative Programming was developed to explain to non-experts what those situations are and to reinvigorate Kowalski's original idea. Other similarly motivated work includes Golog [11], which includes nondeterministic choice operators, and Partial Programs [4], which enable programmers to express incomplete instruction sets.

Collaborative Programming differs from similar initiatives because of its commitment to conflicts. Because instruction sets are issued by multiple people, and people often disagree with one another, a Collaborative Programming language must allow conflicts to be expressed, simply so that the language is capable of capturing peoples' true intentions. Consequently, automated reasoning tools for processing instruction sets must be aware of and tolerate conflicts. In the case of classical logic, this requires automated reasoning tools that implement a paraconsistent entailment relation. In the case of logic programming languages, it requires making ontological commitments within the language and employing algorithms that adhere to those commitments. Each language class has strengths and weaknesses, making the right choice for any particular Collaborative Programming application dependent on the demands of that application.

References

1. Moritz Y. Becker, Cedric Y. Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.
2. Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.
3. John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.
4. Michael R. Genesereth and J. Y. Hsu. Partial programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 238–249, 1991.
5. Timothy L. Hinrichs. *Extensional Reasoning*. PhD thesis, Stanford University, 2007.
6. Timothy L. Hinrichs, Natasha Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Design and implementation of a flow-based security language. Under review, 2008.
7. Sushil Jajodia, Pierangela Samarati, and V S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42. IEEE Press, 1997.
8. Michael Kassoff and Michael R. Genesereth. PrediCalc: A logical spreadsheet management system. *Knowledge Engineering Review*, 22(3):281–295, 2007.
9. Michael Kassoff and Andre Valente. An introduction to logical spreadsheets. *Knowledge Engineering Review*, 22(3):213–219, 2007.
10. Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
11. Hector J. Levesque, Ray Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
12. Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages*, 2003.
13. Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.
14. G. Priest. Paraconsistent logic. In *Handbook of Philosophical Logic*, volume 6, pages 287–293. Kluwer Academic Publishers, 2002.
15. Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.
16. Sergei Vorobyov and Andrei Voronkov. Complexity of nonrecursive logic programs with complex values. In *Proceedings of the ACM SIG for the Management of Data*, pages 244–253, 1998.
17. Marianne Winslett, Charles C. Zhang, and Piero A. Bonatti. Peeraccess: A logic for distributed authorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 168–179, 2005.

Towards fully automated axiom extraction for finite-valued logics

João Marcos and Dalmo Mendonça

DIMAp/CCET, UFRN, Brazil
jmarcos@dimap.ufrn.br,dalmo3@gmail.com

Abstract. We implement an algorithm for extracting appropriate collections of classic-like sound and complete tableaux rules for a large class of finite-valued logics. Its output consists of `Isabelle` theories.[1]

Key words: Many-valued logics, tableaux, automated theorem proving.

1 Introduction

This note will report on the first developments towards the implementation of a fully automated system for the extraction of adequate proof-theoretical counterparts for sufficiently expressive logics characterized by way of a finite set of finite-valued truth-tables. The underlying algorithm was first described in [2]. Surveys on tableaux for many-valued logics can be found in [4, 1]. The implementation has been performed in `ML`, and its application gives rise to an `Isabelle` theory (check [5]) formalizing a given finite-valued logic in terms of two-signed tableau rules.

The survey paper [4] points at a few very good theoretical motivations for studying tableaux for many-valued logics, among them:

- tableau systems are a particularly well-suited starting point for the development of computational insights into many-valued logics;
- a close interplay between model-theoretic and proof-theoretic tools is necessary and fruitful during the development of proof procedures for non-classical logics.

Section 2, right below, recalls the relevant definitions and results concerning many-valued logics as well as their homologous presentation in terms of bivalent semantics defined by clauses of a certain format we call 'gentzenian'. An algorithm for endowing any sufficiently expressive finite-valued logic with an adequate bivalent semantics is exhibited and illustrated for the case of $Ł_3$, the well-known 3-valued logic of Łukasiewicz.

The concepts concerning tableau systems in general and the particular results that allow one to transform any computable gentzenian semantics into a corresponding collection of tableau rules are illustrated in section 3, for the case of $Ł_3$.

[1] A snapshot of the corresponding code can be checked in `http://tinyurl.com/5cakro`.

Section 4 discusses our current implementation, carefully explaining its expected inputs and outputs, and again illustrates its functioning for the case of Ł$_3$. Advantages and shortcomings of our system, in its present state of completion, as well as conclusions and some directions for future work are mentioned in section 5.

2 Many-valued logics

Given a denumerable set At of *atoms* and a finite family $\mathsf{Cct} = \{©^i_j\}_{j\in J}$ of *connectives*, where $\mathsf{arity}(©^i_j) = i$, let $\mathcal{S}$ denote the term algebra freely generated by Cct over At. Here, a *semantics* Sem for the algebra $\mathcal{S}$ will be given by any collection of mappings $\{\S^{\mathcal{V}}_k\}_{k\in K}$ where $\mathsf{dom}(\S^{\mathcal{V}}_k) = \mathcal{S}$ and $\mathsf{codom}(\S^{\mathcal{V}}_k) = \mathcal{V}_k$, and where each collection of *truth-values* $\mathcal{V}_k$ is partitioned into sets of designated values, $\mathcal{D}_k$, and undesignated ones, $\mathcal{U}_k$. The mappings $\S^{\mathcal{V}}_k$ themselves may be called *(ν-valued) valuations*, where $\nu = \mathsf{Card}(\mathcal{V}_k)$. A *bivalent semantics* is any semantics where $\mathcal{D}_k$ and $\mathcal{U}_k$ are singleton sets, for any $k \in K$. For bivalent semantics, valuations are often called *bivaluations*.

The canonical notion of (single-conclusion) *entailment* $\models_{\mathsf{Sem}} \subseteq \mathsf{Pow}(\mathcal{S}) \times \mathcal{S}$ induced by a semantics Sem is defined by setting $\Gamma \models_{\mathsf{Sem}} \varphi$ iff $\S^{\mathcal{V}}_k(\varphi) \in \mathcal{D}_k$ whenever $\S^{\mathcal{V}}_k(\Gamma) \subseteq \mathcal{D}_k$, for every $\S^{\mathcal{V}}_k \in \mathsf{Sem}$. The pair $\langle \mathcal{S}, \models_{\mathsf{Sem}} \rangle$ may be called a *generic ν-valued logic*, where $\nu = \mathsf{Max}_{k\in K}(\mathsf{Card}(\mathcal{V}_k))$.

If one now fixes the sets of truth-values $\mathcal{V}$, $\mathcal{D}$ and $\mathcal{U}$, and considers, for each connective $©^i_j$ an interpretation $\widehat{©}^i_j : \mathcal{V}^i \longrightarrow \mathcal{V}$, one may immediately build from that an associated algebra of truth-values $\mathcal{TV} = \langle \mathcal{V}, \mathcal{D}, \{\widehat{©}^i_j\}_{j\in J} \rangle$ (in the present paper, whenever there is no risk of confusion, we shall not differentiate notationally between a connective symbol © and its interpretation $\widehat{©}$). A *truth-functional* semantics is then defined by the collection of all homomorphisms of $\mathcal{S}$ into $\mathcal{TV}$. In this paper, the shorter expression *ν-valued logic* will be used to qualify any generic ν-valued truth-functional logic, for some finite ν, where ν is the minimal value for which the mentioned logic can be given a truth-functional semantics characterizing the same associated notion of entailment.

The canonical notion of entailment of any given semantics, and in particular of any given truth-functional semantics, may be emulated by a bivalent semantics. Indeed, consider $\mathcal{V}_2 = \{T, F\}$ and $\mathcal{D}_2 = \{T\}$, and consider the 'binary print' of the algebraic truth-values produced by the total mapping $t : \mathcal{V} \longrightarrow \mathcal{V}_2$, defined by $t(v) = T$ iff $v \in \mathcal{D}$. For any ν-valuation $\S^{\mathcal{V}}$ of a given semantics Sem, consider now the characteristic total function $b_\S = t \circ \S^{\mathcal{V}}$. Now, collect all such bivaluations $b_\S$'s into a new semantics $\mathsf{Sem}(2)$, and note that $\Gamma \models_{\mathsf{Sem}(2)} \varphi$ iff $\Gamma \models_{\mathsf{Sem}} \varphi$. The standard 2-valued notion of inference of Classical Logic is characterized indeed by a bivalent truth-functional semantics. In general, though, a bivalent characterization of a logic with a ν-valued truth-functional semantics explores the trade-off between the 'algebraic perspective' of many-valuedness, with its many 'algebraic truth-values' and its semantic characterization in terms of a set of homomorphisms, on the one hand, and the classic-inclined 'logical perspective', with its emphasis on characterizations based on 2 'logical values',

on the other hand (for more detailed discussions of this issue, check [2, 7]). Our interest in this paper is to probe some of the practical advantages of the bivalent classic-like perspective as applied to the wider domain of finite-valued truth-functional logics.

Our running example in this paper will involve Łukasiewicz's well-known 3-valued logic Ł$_3$, characterized by the algebra of truth-values $\langle\{1, \frac{1}{2}, 0\}, \{1\}, \{\neg, \rightarrow, \vee, \wedge\}\rangle$, where the interpretation of the unary negation connective $\neg$ sets $\neg v_1 = 1 - v_1$ and the interpretation of the binary implication connective $\rightarrow$ sets $(v_1 \rightarrow v_2) = \mathsf{Min}(1, 1 - v_1 + v_2)$. The binary symbols $\vee$ and $\wedge$ can be introduced as primitive interpreting them through $(v_1 \vee v_2) = \mathsf{Max}(v_1, v_2)$ and $(v_1 \wedge v_2) = \mathsf{Min}(v_1, v_2)$, but they can also more simply be introduced by definition just like in Classical Logic, setting $\alpha \vee \beta \stackrel{\text{def}}{=} (\alpha \rightarrow \beta) \rightarrow \beta$ and $\alpha \wedge \beta \stackrel{\text{def}}{=} \neg(\neg\alpha \vee \neg\beta)$. The binary print of an arbitrary atom of Ł$_3$ and of its negation is illustrated in the table below.

v	$t(v)$	$\neg v$	$t(\neg v)$
1	T	0	F
$\frac{1}{2}$	F	$\frac{1}{2}$	F
0	F	1	T

(1)

Given some finite-valued logic $\mathcal{L}$ based on a set of truth-values $\mathcal{V}$, we say that $\mathcal{L}$ is *functionally complete* over $\mathcal{V}$ if any n-valued operation, for $n = \mathsf{Card}(\mathcal{V})$, may be defined with the help of a suitable combination of its primitive operators $\{\widehat{©}^i_j\}_{j \in J}$. When $\mathcal{L}$ is not functionally complete from the start, we may consider $\mathcal{L}^{\mathbf{fc}}$ as any functionally complete n-valued conservative extension of $\mathcal{L}$. Given truth-values $v_1, v_2 \in \mathcal{V}$, we say that they are *separated*, and we write $v_1 \sharp v_2$, in case v_1 and v_2 belong to different classes of truth-values, that is, in case either $v_1 \in \mathcal{D}$ and $v_2 \in \mathcal{U}$, or $v_1 \in \mathcal{U}$ and $v_2 \in \mathcal{D}$. Given a unary, primitive or defined, connective Ⓢ of a given truth-functional logic, with interpretation $\widehat{Ⓢ}$, we say that Ⓢ *separates* v_1 and v_2 in case $\widehat{Ⓢ}(v_1) \sharp \widehat{Ⓢ}(v_2)$. Obviously, for any pair of truth-values of $\mathcal{L}^{\mathbf{fc}}$ it is possible to find or to define in the corresponding term algebra an appropriate *separating connective* Ⓢ. When that separation can be done exclusively with the help of the original language of $\mathcal{L}$, we say that $\mathcal{V}$ is *effectively separable* and the logic $\mathcal{L}$, in that case, will be considered to be *sufficiently expressive* for our purposes. It should be noticed that the vast majority of the most well-known finite-valued logics enjoy this expressivity property.

Notice in particular, from Table 1, how the negation connective of Ł$_3$ separates the two undesignated truth-values. Based on Table 1, one may in fact easily provide a unique identification to each of the 3 initial algebraic truth-values, by way of the following statements:

$$\begin{aligned}
v = 1 &\text{ iff } t(v) = T \\
v = \tfrac{1}{2} &\text{ iff } t(v) = F \text{ and } t(\neg v) = F \\
v = 0 &\text{ iff } t(v) = F \text{ and } t(\neg v) = T
\end{aligned} \tag{I}$$

One can also use this separating connective Ⓢ : $\lambda u.\neg u$ in order to provide a bivalent description of each of the operators of the language. Consider for instance the cases of $A : \lambda vw.(v \to w)$ and $B : \lambda vw.\neg(v \to w)$ (that is, B is ⓈA):

A	1	$\frac{1}{2}$	0
1	1	$\frac{1}{2}$	0
$\frac{1}{2}$	1	1	$\frac{1}{2}$
0	1	1	1

B	1	$\frac{1}{2}$	0
1	0	$\frac{1}{2}$	1
$\frac{1}{2}$	0	0	$\frac{1}{2}$
0	0	0	0

(2)

From those tables it is clear for instance that:

$$§(\neg(\alpha \to \beta)) = 1 \text{ iff } §(\alpha) = 1 \text{ and } §(\beta) = 0 \qquad \text{(II)}$$

Let's write $T : \varphi$ and $F : \varphi$, respectively, as abbreviations for $t(§(\varphi)) = T$ and $t(§(\varphi)) = F$. Then, the statement (II) may be described in bivalent form, with the help of (I), by writing:

$$T : \neg(\alpha \to \beta) \text{ iff } T : \alpha \text{ and } (F : \beta \text{ and } T : \neg\beta) \qquad \text{(III)}$$

In [2] an algorithm that constructively specifies a bivalent semantics for any sufficiently expressive finite-valued logic was proposed. The output of the algorithm is a computable class of clauses governing the behavior of all the bivaluations that together will define a notion of entailment that coincides with the original entailment defined with the help of the algebra of truth-values $\mathcal{TV}$. Moreover, all those clauses are in a specific format we call *gentzenian*, namely, they are conditional expressions of the form $(\Phi \Rightarrow \Psi)$ where both Φ and Ψ are (meta)formulas of the form $\top$ (top), $\bot$ (bottom) or a clause of the form

$$b(\varphi_1^1) = w_1^1 \,\&\ldots\&\, b(\varphi_1^{n_1}) = w_1^{n_1} \mid \ldots \mid b(\varphi_m^1) = w_m^1 \,\&\ldots\&\, b(\varphi_m^{n_m}) = w_m^{n_m}.(G)$$

Here, $w_i^j \in \{T, F\}$, each φ_i^j is a formula of $\mathcal{L}$, the symbol $\Rightarrow$ represents implication (and $\Leftrightarrow$ shall represent bi-implication, abbreviating the conjunction of two clauses of the form (G)), the symbol & represents conjunction, and | represents disjunction. The (meta)logic governing these clauses is FOL, First-Order Classical Logic. One may alternatively represent a clause of the form (G) as $\bigvee_{1\leq k\leq m} \bigwedge_{1\leq s\leq n_m} (b(\varphi_k^s) = w_k^s)$.

With a slight notational change and using FOL, one can see (III) as a description done in an abbreviated gentzenian format:

$$T : \neg(\alpha \to \beta) \Leftrightarrow T : \alpha \,\&\, F : \beta \,\&\, T : \neg\beta \qquad \text{(IV)}$$

Following the above line of reasoning, it is also correct to write, for instance, the clause:

$$\begin{aligned} F : (\alpha \to \beta) \Leftrightarrow\ & T : \alpha \,\&\, F : \beta \,\&\, F : \neg\beta \mid \\ & T : \alpha \,\&\, F : \beta \,\&\, T : \neg\beta \mid \\ & F : \alpha \,\&\, F : \neg\alpha \,\&\, F : \beta \,\&\, T : \neg\beta \end{aligned} \qquad \text{(V)}$$

According to the reductive algorithm described in [2], a sound and complete bivalent version of any sufficiently expressive finite-valued logic $\mathcal{L}$ is obtained if:

(i) the above illustrated procedure is iterated in order to obtain clauses describing exactly in which situation one may assert $T : \copyright_j^i(\alpha_1, \ldots, \alpha_i)$ and $F : \copyright_j^i(\alpha_1, \ldots, \alpha_i)$, as well as $T : \circledS\copyright_j^i(\alpha_1, \ldots, \alpha_i)$ and $F : \circledS\copyright_j^i(\alpha_1, \ldots, \alpha_i)$, for each $\copyright_k^i \in \mathsf{Cct}$ and each one of the separating connectives $\circledS$ of $\mathcal{L}$;

(ii) to all those clauses one adds the following extra axioms governing the behavior of the admissible collection of bivaluations:

(C1) $\top \;\Rightarrow\; T : \alpha \mid F : \alpha$ (C2) $T : \alpha \;\&\; F : \alpha \;\Rightarrow\; \bot$

(C3) $T : \alpha \;\Rightarrow\; \bigvee_{d \in \mathcal{D}} \bigwedge_{1 \leq m < n \leq \mathsf{Card}(\mathcal{D})} w_{mn}^d : \circledS_{mn}(\alpha)$

(C4) $F : \alpha \;\Rightarrow\; \bigvee_{u \in \mathcal{U}} \bigwedge_{1 \leq m < n \leq \mathsf{Card}(\mathcal{U})} w_{mn}^u : \circledS_{mn}(\alpha)$

for every $\alpha \in \mathcal{S}$, where $\circledS_{mn}$ is the unary (primitive or defined) connective that we use to separate the truth-values m and n, and $w_{mn}^v = t(\circledS_{mn}(v))$.

3 Tableaux

Generic tableau systems for finite-valued logics are known at least since [3]. In the corresponding tableaux, however, formulas may receive as many labels as the number of truth-values in $\mathcal{V}$, and that somewhat obstructs the task of comparing for instance the associated notions of proof and of consequence relation to the corresponding classical notions. But with the help of the bivalent semantics illustrated in the previous section it is now straightforward to produce sound and complete collections of classic-like two-signed tableau rules (i.e., each formula appears with exactly one of two labels at the head of each rule).

The basic idea, explained in [2], is to dispose the gentzenian clauses governing the admissible bivaluations in an appropriate way. For that matter, clauses such as (IV) and (V) can be rendered, respectively, as the following tableau rules:

(IV)^{tab}

$T : \neg(\alpha \to \beta)$
|
$T : \alpha$
$F : \beta$
$T : \neg\beta$

(V)^{tab}

$F : (\alpha \to \beta)$

branch 1	branch 2	branch 3
$T : \alpha$	$T : \alpha$	$F : \alpha$
$F : \beta$	$F : \beta$	$F : \neg\alpha$
$F : \neg\beta$	$T : \neg\beta$	$F : \beta$
		$T : \neg\beta$

To those rules corresponding to the truth-tables of the operators and the separating connectives, one should also add rules corresponding to the extra axioms (C1)–(C4). In practical cases, however, axioms (C3) and (C4) can often be proven from the remaining ones. Moreover, axiom (C2) expresses just the usual closure condition on tableau branches. On the other hand, axiom (C1) gives rise in general to the following 'dual-cut' *branching rule*, for arbitrary α:

$T : \alpha$	$F : \alpha$

All other definitions and concepts concerning the construction of tableaux are standard (check [6]).

One might be worried, with good reason, that the unrestrained use of the branching rule may potentially make the corresponding tableaux non-analytic. We will discuss that in the conclusion. The tableau rules originated from the above procedure can naturally be used in order to prove theorems, check conjectures and suggest counter-models, but also, in the meta-theory, to formulate and prove derived rules that can be used to simplify the original presentation of the logic as originated by our algorithm. So, for instance, the above illustrated complex three-branching rule for $F:(\alpha \to \beta)$ can eventually be simplified into one of the following equivalent two-branching rules:

$(\text{V}^{*})^{tab}$

$$F:(\alpha \to \beta)$$
$$\begin{array}{cc} T:\alpha & F:\alpha \\ F:\beta & F:\neg\alpha \\ & F:\beta \\ & T:\neg\beta \end{array}$$

$(\text{V}^{**})^{tab}$

$$F:(\alpha \to \beta)$$
$$\begin{array}{cc} T:\alpha & F:\neg\alpha \\ F:\beta & T:\neg\beta \end{array}$$

4 Implementation

We used the functional programming language `ML` to automate the axiom extraction process. `ML` provides us, among other advantages, with an elegant and suggestive syntax, and a very handy compile-time type checking and type inference that guarantees that we never run into unexpected run-time problems with our program, once it is proved correct with respect to the specification.

The relevant inputs of our program include the detailed definition of a finite-valued logic, such as the logic Ł$_3$ presented in the previous sections, together with an appropriate set of separating connectives for that logic.

Here's an example of a input for the logic Ł$_3$, presented above, where the functions `CSym`, `CAri` and `CTab` take a connective and return its symbol (for printing), arity and truth-table, respectively. A truth-table of a given connective Ⓒ is represented as the list of all pairs `(`$[x_1,\ldots,x_n]$`, y)` such that Ⓒ$(x_1,\ldots,x_n)=y$.

```
(* PROGRAM SIGNATURE *)       (* EXAMPLE OF L3 *)
  signature LOGIC =
sig
    type cct                  datatype cct = Neg | Imp
    val Values                ["0", "1/2", "1"];
    val Designated            ["1"];
    val Connectives           [Neg, Imp];
    val SeparatingD           [];
    val SeparatingND          [Neg];
    val CSym                  fun CSym Neg = "~"
                                | CSym Imp = "-->";
    val CAri                  fun CAri Neg = 1
                                | CAri Imp = 2;
```

```
    val CTab                  fun CTab Neg = [ (["0"], "1"),
                                               (["1/2"], "1/2"),
                                               (["1"], "0") ]
                                | CTab Imp = [ (["0", "0"], "1"),
                                               (["0", "1/2"], "1"),
                                               (["0", "1"], "1"),
                                               (["1/2", "0"], "1/2"),
                                               (...)
                                               (["1", "1"], "1") ]
end;
```

To perform the extraction, our program first generates a list of all necessary rules, as explained in section 2.

```
val rulesList = [T "~(A0)",      T "A0 --> A1",
                 T "~(~(A0))",   T "~(A0 --> A1)",
                 F "~(A0)",      F "A0 --> A1",
                 F "~(~(A0))",   F "~(A0 --> A1)" ];
```

Next, the program converts each connective's truth-table, here given by the function `CTab`, into a table where each value is exchanged by its binary print. The binary print of a value is calculated based on the separating connectives given as input (`SeparatingD` for designated values and `SeparatingND` for undesignated values). For instance, for the case of the connective `Neg`, the clauses are the following:

```
   (* A0 *)               (* ~(A0) *)

   ([F "A0",T "~(A0)"],   [T "~(A0)"])
   ([F "A0",F "~(A0)"],   [F "~(A0)",F "~(~(A0))"])
   ([T "A0"],             [F "~(A0)",T "~(~(A0))"])
```

Now, for each formula in the list of rules, a search is performed through all tables generated in the latter step, and all clauses in which the given formula appears on the right hand side are returned. The left hand side of these clauses represents the branches of the desired tableau rule. For instance, the rules for `T:~A` and `F:~A` are:

```
(* T:~A *)   ([ [F "A0",T "~(A0)"] ],             T "~(A0)")
(* F:~A *)   ([ [F "A0",F "~(A0)"], [T "A0"] ],   F "~(A0)")
```

The next steps include the calculus of axioms (C3) and (C4), and the printing of all definitions, concrete syntax and rules into a text file containing the full theory ready to use in `Isabelle`. `Isabelle`, also written in `ML`, is a generic theorem-proving environment based on a higher-order meta-logic in which it is quite simple to create theories with rules and axioms for various kinds of deductive formalisms, and equally straightforward to define tacticals for the automation of routine tasks and to prove theorems about these systems.

For the case of Ł$_3$, here is the corresponding theory produced as output by our program:

```
theory TL3

imports Sequents
begin
typedecl a
consts
  Trueprop :: "(seq'=>seq') => prop"
  True     :: o
  False    :: o
  TR       :: "a => o"           ("T:_" [20] 20)
  FR       :: "a => o"           ("F:_" [20] 20)
  Neg      :: "a => a"           ("~ _" [40] 40)
  Imp      :: "[a,a] => a"       ("_-->_" [24,25] 25)

 syntax
 "@Trueprop"     ::  "(seq) => prop"  ("[_]" 5)

 ML
{*
fun seqtab_tr  c [s] =  Const(c,dummyT) $ seq_tr s;
fun seqtab_tr' c [s] =  Const(c,dummyT) $ seq_tr' s;
*}
parse_translation {* [("@Trueprop", seqtab_tr "Trueprop")] *}
print_translation {* [("Trueprop", seqtab_tr' "@Trueprop")] *}

local
axioms

axC1:  "[| [ $H, T:A ] ; [ $H, F:A ] |] ==> [$H]"
axC21: "[ $H, T:A, $E, F:A, $G]"
axC22: "[ $H, F:A, $E, T:A, $G]"

axC3:  "[| [ $H, T:A, $G ] |] ==> [ $H, T:A, $G ]"

axC4:  "[| [ $H, T:~(A), $G ] ; [ $H, F:~(A), $G ] |]
                ==> [ $H, F:A, $G ]"

ax0:   "[| [ $H, F:A0, T:~(A0), $G ] |] ==> [ $H, T:~(A0), $G ]"

ax1:   "[| [ $H, F:A0, F:~(A0), $G ] ; [ $H, T:A0, $G ] |]
                ==> [ $H, F:~(A0), $G ]"

ax2:   "[| [ $H, F:A0, F:~(A0), $G ] |] ==> [ $H, F:~(~(A0)), $G ]"

ax3:   "[| [ $H, T:A0, $G ] |] ==> [ $H, T:~(~(A0)), $G ]"

ax4:   "[| [ $H, F:A0, T:~(A0), F:A1, T:~(A1), $G ] ;
           [ $H, T:A0, T:A1, $G ] ;
           [ $H, F:A0, F:~(A0), T:A1, $G ] ;
           [ $H, F:A0, F:~(A0), F:A1, F:~(A1), $G ] ;
           [ $H, F:A0, T:~(A0), T:A1, $G ] ;
           [ $H, F:A0, T:~(A0), F:A1, F:~(A1), $G ] |]
                ==> [ $H, T:A0 --> A1, $G ]"
```

```
ax5:   "[| [ $H, F:A0, F:~(A0), F:A1, T:~(A1), $G ] ;
           [ $H, T:A0, F:A1, F:~(A1), $G ] ;
           [ $H, T:A0, F:A1, T:~(A1), $G ] |]
                ==> [ $H, F:A0 --> A1, $G ]"

ax6:   "[| [ $H, F:A0, F:~(A0), F:A1, T:~(A1), $G ] ;
           [ $H, T:A0, F:A1, F:~(A1), $G ] |]
                ==> [ $H, F:~(A0 --> A1), $G ]"

ax7:   "[| [ $H, T:A0, F:A1, T:~(A1), $G ] |]
                ==> [ $H, T:~(A0 --> A1), $G ]"

 ML {* use_legacy_bindings (the_context ()) *}
end
```

In this theory, `consts` lists the formula constructors. `TR :: "a => o"` means that the constructor `TR` takes a formula (typed `a`) and returns a signed formula (typed `o`). We also extract from the logic received as input each constructor's name to use as syntactic sugar, as well as its associativity rules and priority order.

In the generated axioms corresponding to the tableau rules, `T:X` and `F:X` are (signed) formulas, `$H` and `$E` are sequences of such formulas (contexts) that are not directly involved in the rule, each sequence between square brackets represents a tableau branch, and a collection of branches is delimited by `[|` and `|]`. The symbol `==>` denotes `Isabelle`'s meta-implication. In `Isabelle`, the application of a rule means that is possible to achieve the goal (sequence on the right of the meta-implication) once it's possible to prove the hypotheses (sequences on the left of the meta-implication), which constitute the collection of new subgoals at each step. The branching rule corresponds to axiom `axC1`, and the closure rule for a branch of the tableau corresponds to the axioms `axC21` and `axC22`.

Note for instance that `ax5` corresponds to rule $(\mathrm{V})^{tab}$ from the last section. The simpler rule $(\mathrm{V}^{**})^{tab}$, mentioned in the same section, can of course be written in `Isabelle` as:

```
ax5SS: "[| [ $H, T:A0, F:A1, $E ] ;
           [ $H, F:~(A0), T:~(A1), $E ] |]
                ==> [$H, F:A0 --> A1, $E]"
```

The proof that `ax5` and `ax5S` are indeed equivalent tableau rules, i.e., that one can derive one from the other in the presence of the remaining rules of our theory, can now be done directly with the help of `Isabelle`'s meta-logic. One might notice that, while in [3] the number of rules for each given primitive connective is exponential in its arity, here the number of rules for each such connective is always polynomial both in the arity and in the number of separating connectives of the input logic. The worst-case number of nodes involved in each tableau rule, however, using our algorithm, *is* exponential in the arity of the corresponding connective. Using `Isabelle`'s meta-logic to prove simplifications such as the one illustrated above, for the case of rule (V), the number of nodes involved in each tableau rule may be substantially reduced.

5 Epilogue

The present note has reported on the first concrete implementation of a certain constructive procedure for obtaining adequate two-signed tableau systems for a large number of finite-valued logics. Expressing a variety of logics in the *same* framework is quite useful for the development of comparisons between such logics, including their expressive and deductive powers.

There still remains some room for improvement and extension of both our algorithm (which should still, for instance, be upgraded in order to deal in general with first-order truth-functional logics) and its implementation. By way of an example, we have assumed from the start that the logics received as inputs to our program came together with a suitable collection of separating connectives. This second input, however, could be dispensed with, as the set of all definable unary connectives can in fact be automatically generated in finite time from any given initial set of operators of the input logic. That generation, however, may be costly for logics with a large number of truth-values and is not as yet performed by our system. Another direction that must be better explored, from the theoretical perspective, concerns the conditions for the admissibility or at least for the explicit control of the application of the dual-cut branching rule. On the one hand, the elimination of dual-cut has an obvious favorable effect on the definition of completely automated theorem-proving tacticals for our logics. If that result cannot be obtained in general but if we can at least guarantee, on the other hand, that this branching rule will never be needed, in each case, for more than a finite number of known formulas —say, the ones related to the original goal as constituting its subformulas or being the result of applying the separating connectives to its subformulas— then again this will make it possible to devise tacticals for obtaining fully automated derivations using the above described tableaux for our finite-valued logics.

References

1. Matthias Baaz, Christian G. Fermüller, and Gernot Salzer. Automated deduction for many-valued logics. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 1355–1402. Elsevier and MIT Press, 2001.
2. Carlos Caleiro, Walter Carnielli, Marcelo E. Coniglio, and João Marcos. Two's company: "The humbug of many logical values". In J.-Y. Béziau, editor, *Logica Universalis*, pages 169–189. Birkhäuser Verlag, Basel, Switzerland, 2005. URL = `http://wslc.math.ist.utl.pt/ftp/pub/CaleiroC/05-CCCM-dyadic.pdf`.
3. Walter A. Carnielli. Systematization of the finite many-valued logics through the method of tableaux. *The Journal of Symbolic Logic*, 52(2):473–493, 1987.
4. Reiner Hähnle. Tableaux for many-valued logics. In M. D'Agostino et al., editors, *Handbook of Tableau Methods*, pages 529–580. Springer, 1999.
5. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
6. Raymond M. Smullyan. *First-Order Logic*. Dover, 1995.
7. Heinrich Wansing and Yaroslav Shramko. Suszko's thesis, inferential many-valuedness, and the notion of a logical system. *Studia Logica*, 88(3):405–429, 2008.

A Small Framework for Proof Checking

Hans de Nivelle and Piotr Witkowski

Institute of Computer Science, University of Wrocław, Poland
nivelle|pwit@ii.uni.wroc.pl

Abstract. We describe a framework with which first order theorem provers can be used for checking formal proofs. The main aim of the framework is to take as much advantage as possible from the strength of first order theorem provers in the formalization of realistic formal proofs. In order to obtain this, we restricted the use of higher order constructs to a minimum. In particular, we refrained from λ notation in formulas and from currying.
The first order prover can be freely chosen. All communication with the theorem prover uses TPTP syntax.
The system is intended for teaching, for checking mathematical proofs or correctness proofs of algorithms and also for improving the effectiveness of theorem provers. In its current set up, the system is not intended for building large libraries of checked mathematics.

1 Introduction

We describe a framework with which first order theorem provers can be used for checking formal proofs. The main aim of the framework is to take as much advantage as possible from the strength of first order theorem provers. In order to obtain this, we try to stay as close as possible to first order logic. The only higher order constructs in the logic are second order quantifications. Second order quantification is strong enough to express induction axioms, and set theoretic axioms.
We call the formulas that the framework uses *weak untyped second order* (WUSO) formulas. They are formally defined in Section 1.1. The system stores formulas in *contexts*. A context is essentially a stack of formulas. By specifying operators that modify contexts, the natural deduction rules $\rightarrow$-intro and $\forall$-intro can be defined. The complementary rules $\rightarrow$-elim and $\forall$-elim are obtained by explicitly specifying instances and moduls ponens combinations when a formula is used. These four rules together specify natural deduction for the $(\forall, \rightarrow)$ fragment of WUSO formulas.
All other reasoning is done by delegating reasoning tasks to a first order theorem prover. The theorem prover can be freely chosen by the user. The user can specify with which parameters the theorem prover has to be called, and how it can be recognized when the prover has found a proof.
The main aim of this work is to obtain insight into the question how useful first order theorem provers can be as assistant in the verification of realistic proofs,

and to obtain realistic test data for theorem provers. In addition, we intend to use the system as a tool for teaching logic and verification.

There have been quite a few more attempts to connect first order provers to interactive provers. (See for example [7], [2]) The main difference with these approaches is that we try to adopt the calculus as much as possible towards the theorem prover, instead of plugging the theorem prover into a calculus that is already fixed. Most interactive theorem provers use a variant of higher order logic (with currying) and a rich type system. The standard logic operators are usually defined inside the logic. Translating such formulas into first order logic is a nontrivial task, We hope that we can avoid most of the translation problems by using a logic close to the logic of the theorem prover.

In the literature, a lot of attention has been given to the problem of translating proofs found by a theorem prover back into the calculus of the interactive proof assistant. (See [8], [1], [5], [6]) Using such a translation, it can be avoided that the external theorem prover has to be trusted. In the present implementation, we completely ignore this problem. We acknowledge that this problem is important, but it is not the aspect that we want to study with the present system. We want to study the problem of the effectiveness of first order theorem proving. Our experience (from [1]) is that automated theorem provers are not as effective in solving real life problems as one would hope. Discussions with developers of interactive proof checkers confirm this experience. The problem is also mentioned in [8]. It is our hope that, by taking first order theorem proving into account from the beginning, a system can be obtained in which first order theorem proving can be more effective.

Our approach to proof checking is closely related to Mizar [10], but more basic. Mizar has a rich type system, while we don't have a type system. Mizar internally uses a very weak theorem prover, (somewhat described in [11]), which is able to do some equality reasoning, and some propositional reasoning.

In [9] a proof checking system is described that is in structure somewhat similar to our current system. Both systems use an external theorem prover for proof checking. The main difference is that we want to use our system for checking realistic proofs, while the system of [9] is intended for checking the outputs of theorem provers. In our system, if one wants to increase reliability, one can use multiple theorem provers and have each step checked by different provers.

1.1 Weak Untyped Second Order Logic

We define the fragment of *weak second order logic* used by our system. The fragment is chosen as a compromise between expressibility on one hand, and suitability for first order theorem proving on the other hand. In order to obtain sufficient expressibility, some higher-order features are necessary. In order to remain close to first order logic, we refrained from λ notation and currying in formulas. At present, the fragment is untyped, but this can easily be changed since there are no real obstacles for adding simple types to first order theorem provers.

The fragment is a *second order logic*, because it is allowed to quantify over functions and predicates that work on objects. We call the logic fragment *weak* second order logic, because second order functions cannot be used as arguments of other functions or predicates, and because λ notation is not allowed inside formulas. Although the logic is untyped, we still insist that variables are declared.

Definition 1. *A* declaration *has one of the following two forms:*

- *A function declaration FUNCTION $f{:}n$ declares f as a function symbol of arity n. In case $n = 0$, the function symbol is a constant.*
- *A predicate declaration PREDICATE $p{:}n$ declares p as a predicate with arity n.*

We usually abbreviate FUNCTION $f{:}n$ to FUNC $f{:}n$ and PREDICATE $p{:}n$ to PRED $p{:}n$.

The logic is untyped, and there are no higher-order functions/predicates. Therefore, it is sufficient to specify the arity of a symbol in order to declare it.

Definition 2. *The set of* weakly untyped second order *(WUSO) formulas is recursively defined as follows:*

- *A usual (first order) atom is a WUSO formula.*
- *$\bot$ and $\top$ are WUSO formulas.*
- *If F is a WUSO formula, then $\neg F$ is a WUSO formula.*
- *If F_1 and F_2 are WUSO formulas, then $F_1 \wedge F_2$, $F_1 \vee F_2$, $F_1 \rightarrow F_2$, and $F_1 \leftrightarrow F_2$ are also WUSO formulas.*
- *If F is a WUSO formula, $D_1, \ldots, D_n$, $n > 0$, is a sequence of declarations, then $\forall D_1, \ldots, D_n\ F$ and $\exists D_1, \ldots, D_n\ F$ are WUSO formulas.*

Quantifications of form $\forall\exists$ FUNC $x_1{:}0, \ldots, x_n{:}0\ F$ are called first order. *We usually abbreviate first order quantifications to $\forall\exists\ x_1 \cdots x_n\ F$.*
A WUSO formula is called schematic first order *if it has form $\forall D_1, \ldots, D_n\ F$ or form F, and F contains only first order quantifications.*

In addition to satisfying Definition 2, a formula must be *well formed*, which means that all symbols occurring in it have to be declared.
Our fragment is a bit stronger than the logic used by Mizar [10], which uses schematic first order formulas, but we will probably not make use of this fact in applications. We now give some examples of WUSO formulas:

Example 1. The induction schema for natural numbers:

$$\forall \text{ PRED } p{:}1\ p(0) \wedge [\ \forall \text{ FUNC } n{:}0\ N(n) \rightarrow P(n) \rightarrow P(\text{succ}(n))]$$
$$\rightarrow \forall \text{ FUNC } m{:}0\ N(m) \rightarrow P(m).$$

The axiom of separation (Aussonderungsaxiom):

$$\forall \text{ PRED } p{:}1\ \forall \text{ FUNC } x{:}0\ \exists \text{ FUNC } y{:}0$$
$$(\forall \text{ FUNC } \alpha{:}0\ \alpha \in y \leftrightarrow \alpha \in x \wedge p(\alpha)).$$

1.2 Contexts

The system collects all its assumptions, declarations and proven theorems in a context.

Definition 3. *A* context Γ *is a sequence of form* $\Gamma_1, \ldots, \Gamma_p,\ p \geq 0$. *Each* Γ_i *either has form* C_i *or form* $L_i{:}C_i$. *Each* C_i *in turn must have one of the forms listed below. Each* L_i, *when it is present, is a label. We explain in Definition 4 under which conditions* C_i *can have a label. Here we list the possible forms of the* C_i.

- *A declaration of form* $FUNC\ f{:}n$ *or* $PRED\ p{:}n$.
- *A definition of form* $FUNC\ f := \lambda x_1 \cdots x_n\ t$, *or of form* $PRED\ p := \lambda x_1 \cdots x_n\ F$.
- *An indirect function definition of form* $FUNC\ f := \lambda x_1 \cdots x_n\ y\ F$. *The other definitions are called* direct.
- *An assumption* F.
- *A proven formula* F.

In the list, F *denotes a WUSO formula,* t *a first order term.*

Note that λ abstraction cannot be used in formulas, only in definitions and in substitutions. Because abstraction is possible only on 0-arity function variables, there is no need to include type information in an abstraction.
An indirect function definition defines an n-ary function through an $(n+1)$-ary predicate. In order to be accepted by the system, the user has to provide proofs of $\forall x_1 \cdots x_n \exists y\ F(x_1, \ldots, x_n, y)$ and of $\forall x_1 \cdots x_n \forall y_1 y_2\ F(x_1, \ldots, x_n, y_1) \wedge F(x_1, \ldots, x_n, y_2) \rightarrow y_1 = y_2$.

Definition 4. *Most of the elements* C_i *that can occur in a context have a meaning that can be expressed by a formula. We call this formula* the characteristic formula *of* C_i. *It is defined as follows:*

- *A declaration of form* $FUNC\ f{:}n$ *or* PRED $p{:}n$ *has no characteristic formula.*
- *If* C_i *has form* $FUNC\ f := \lambda x_1 \cdots x_n\ t$, *then the characteristic formula equals*
$$\forall x_1 \cdots x_n\ f(x_1, \ldots, x_n) = t[x_1, \ldots, x_n].$$
The characteristic formula of $PRED\ p := \lambda x_1 \cdots x_n\ F$ *equals*
$$\forall x_1 \cdots x_n\ p(x_1, \ldots, x_n) \leftrightarrow F[x_1, \ldots, x_n].$$
- *The characteristic formula of an indirect function definition* $FUNC\ f := \lambda x_1 \cdots x_n\ y\ F$ *equals*
$$\forall x_1 \cdots x_n\ y\ f(x_1, \ldots, x_n) = y \leftrightarrow F[x_1, \ldots, x_n, y].$$
- *The characteristic formula of a formula assumption* F *equals* F.
- *The characteristic formula of a proven formula* F *equals* F.

A context element C_i *can have a label exactly when it has a characteristic formula. The purpose of the label is to assign a name to the characteristic formula.*

1.3 Forward Reasoning

The calculus has three mechanisms for forward reasoning. These are instantiation, modus ponens and first order reasoning. There is also a mechanism for conditional reasoning, which will be discussed in the next section. The reasoning mechanisms (already without first order theorem proving) cover the usual natural deduction rules for $\forall$ and $\rightarrow$.
Instantiation is the following rule: From $\forall x\ F$ derive $F[x := t]$. Modus ponens is the rule: From A and $A \rightarrow B$ derive B. Instantiation and modus ponens are handled together in *references*. References are used in first order reasoning steps for referring back to formulas that have been proven before. In the references, one can specify which instantiations have to be used, and how modus ponens must be applied.
First-order reasoning is delegated to a first order theorem prover, which can be chosen by the user. Every time a first order reasoning step has to be made, the system prepares an input file in TPTP-syntax, starts the theorem prover, waits for a result, and checks the outputfile for a characteristic string that indicates that a proof was found. At this moment, we do not attempt to check the proof that was found by the theorem prover. The system is designed such a way that it is possible to run each goal on more than one theorem prover, in case one wants to avoid trusting a single theorem prover.
For each first order reasoning step, the user has to indicate its result, and he has to indicate from which premisses he expects the result to be provable. If the proof succeeds, the new formula will be added to the context as a proven formula. The user can assign a label to the formula. The general schema is given in Section 2.1. Although it is a bit more work for the user, listing the premisses avoids that the theorem prover has to be called with large input sets.
In order to prove a new formula using a context Γ, every characteristic formula of an element of Γ can be used. In order to refer to the characteristic formulas, one can make use of the labels. Additionally, indirect references of form '3 formulas after label X' , '2 formulas before label X', 'the last formula', or 'the second last formula' are allowed.

Definition 5. *Given a context Γ, we recursively define the set of references and the formulas that they refer to:*

- *A label L is a reference. In case Γ contains an element with label L, the reference refers to the characteristic formula of L.*
- *An expression of form $L + i$ or $L - i$ is a reference. In case Γ contains an element with label L, the reference refers to the i-th characteristic formula after (or before) L. The references $L+0$ and $L-0$ refer to the same formula as L.*
- *An expression of form $-i$ is a reference. In this case $-i$ refers to the i-th characteristic formula from the end of Γ. The last characteristic formula in Γ can be referred to by -1.*
- *If R is a reference, then $R\Theta$ is a reference. Θ must a be a substitution of form*

$$\{\ FUNC\ f_1 := \lambda \overline{x}_1 t_1, \ldots,\ FUNC\ f_m := \lambda \overline{x}_m t_m,$$

$$PRED\ p_1 := \lambda \overline{y}_1 F_1, \ldots,\ PRED\ p_n := \lambda \overline{y}_n F_n\ \}.$$

If R refers to a formula of form

$$\forall\ FUNC\ f_1 \cdots\ FUNC\ f_m\ \ PRED\ p_1 \cdots\ PRED\ p_n\ F,$$

(possibly after reshuffling top level $\forall$ *quantifiers), then* $R\Theta$ *refers to* $F\Theta$*.*

- *If* R_1 *and* R_2 *are references, then* $MP(R_1, R_2)$ *is also a reference. If* R_1 *refers to a formula* A*, and* R_2 *refers to a formula of form* $A \rightarrow B$*, then* $MP(R_1, R_2)$ *refers to formula* B*.*

The reader may think that MP is superfluous because it is a first order rule. The reason that we added it separately is the fact that, although MP is a first order rule, it can work on formulas that are not first order. When there is no ambiguity, we will omit the type indicators FUNC and PRED in substitutions.

Example 2. In Example 1, the induction scheme can be instantiated by $\{p := \lambda x\ x + 0 = x\}$. The result is

$$0 + 0 = 0 \wedge [\ \forall\ \text{FUNC}\ n{:}0\ N(n) \rightarrow n + 0 = n \rightarrow \text{succ}(n) + 0 = \text{succ}(n)\ ] \rightarrow$$

$$\forall\ \text{FUNC}\ m{:}0\ N(m) \rightarrow m + 0 = m.$$

The separation axiom can be instantiated by $\{\ p := \lambda x\ \text{interesting}(x)\ \}$. The result is

$$\forall\ \text{FUNC}\ x{:}0\ \exists\ \text{FUNC}\ y{:}0$$

$$\forall\ \text{FUNC}\ \alpha{:}0 \quad \alpha \in y \leftrightarrow \alpha \in x \wedge \text{interesting}(\alpha).$$

It is also possible to instantiate with $\{\ p := \lambda x\ \ \neg\text{interesting}(x),\ x := \lambda\ \text{nat}\ \}$. The result is

$$\exists\ \text{FUNC}\ y{:}0\ \forall\ \text{FUNC}\ \alpha{:}0 \quad \alpha \in y \leftrightarrow \alpha \in \text{nat} \wedge \neg\text{interesting}(\alpha).$$

(The last set y can be proven empty by a simple induction argument)

1.4 Conditional Reasoning

Conditional reasoning handles the introduction and dropping of assumptions, and the introduction and dropping of eigenvariables. When an assumption A is dropped, every formula F that was proven in the context of A, has to be replaced by $A \rightarrow F$. When an eigenvariable x is dropped, every formula F that is proven in the context of x, has to be replaced by $\forall x\ A$.
In our system, conditional reasoning is handled by modifications on the context Γ. Suppose that Γ has form Γ_1, C, Γ_2, and that we want to drop C. We specify for each element in Γ_2, how it will be modified. It is not always possible to define a meaningful effect on each element of Γ_2, but we try to be as general as possible. When for some element of Γ_2, no meaningful effect can be defined, it is forbidden to drop C.

- If C is a declaration of form FUNC $c{:}0$, then Γ_2 must consist of definitions and proven formulas only. C is removed by the following procedure: As long as C is not the last element of Γ_2, the complete context can be written in the form Γ_1, C, D, Δ. We will exchange C with D. During the replacement, D is modified, and possibly also Δ. The exchanges are repeated until C reaches the end of Γ_2. Then it can be removed without consequences.
  Write Γ_2 in the form C, D, Δ, and assume that D is a definition with form FUNC $f := \lambda x_1 \cdots x_n\ t$.
  Then D is replaced by FUNC $f' := \lambda c\ x_1 \cdots x_n\ \ t$, and Δ is replaced by $\Delta\{\ f := \lambda x_1 \cdots x_n\ f'(c, x_1, \ldots, x_n)\ \}$.
  If D has form PRED $p := \lambda x_1 \cdots x_n\ F$ then it is treated analogeously. D is replaced by PRED $p' := \lambda c\ x_1 \cdots x_n\ \ F$, and Δ is replaced by $\Delta\{\ p := \lambda x_1 \cdots x_n\ p'(c, x_1, \ldots, x_n)\ \}$.
  Indirect function definitions are dealt with in the same way. We omit the details.
  If D is a proven formula F, then it is replaced by $\forall$FUNC $c{:}0\ \ F$, and Δ is not changed.
- If C is a declaration of form FUNC $f{:}n$ with $n \neq 0$, or of form PRED $p{:}n$, then Γ_2 must consist only of proven formulas. Each proven formula F is replaced by $\forall$ FUNC $f{:}n\ F$. (or by $\forall$ PRED $p{:}n\ F$)
- If C is a direct function definition of form FUNC $f{:}n := \lambda x_1 \cdots x_n\ t$, then Γ_2 is replaced by $\Gamma_2\ \{\ f := \lambda x_1 \cdots x_n\ \ t\ \}$.
  Direct predicate definitions are substituted away in the same way.
- If C is an indirect function definition, it cannot be dropped, because we have no way of substituting it away.
- If C is a formula assumption of form F, then Γ_2 must consist of proven formulas $F_1, \ldots, F_n$ only. Each formula F_i is replaced by $F \rightarrow F_i$.

We think that most of the modifications on Γ_2 are more or less obvious, except for the first case, where a 0-arity function variable is dropped. We give an example of this situation:

Example 3. Consider the context

FUNC $n{:}0$,
PRED $E := N(n) \wedge \exists$ FUNC $m{:}0\ \ N(m) \wedge m + m = n$,
PROVEN $E \rightarrow \exists$ FUNC $m{:}0\ \ d(m) = n$.

The propositional variable E means 'n is even', $N(n)$ denotes 'n is a natural number', and d denotes the doubling function $\lambda x\ x + x$.
Suppose that we want to drop the first assumption FUNC $n{:}0$. Then the definition and the proven formula have to be modified. First, the definition PRED $E := N(n) \wedge \exists$ FUNC $m{:}0\ \ N(m) \wedge m + m = n$ is replaced by PRED $E' := \lambda n\ N(n) \wedge \exists$ FUNC $m{:}0\ N(m) \wedge m + m = n$, and in the proven formula, the substitution $\{\ E := E'(n)\ \}$ is made.
After that, the formula $E'(n) \rightarrow \exists$ FUNC $m{:}0\ \ d(m) = n$ is replaced by
$\forall$ FUNC $n{:}0\ \ E'(n) \rightarrow \exists$ FUNC $m{:}0\ \ d(m) = n$.

The resulting context is

$$\text{PRED } E' := \lambda n \; N(n) \wedge \exists \text{ FUNC } m{:}0 \;\; N(m) \wedge m + m = n,$$
$$\forall \text{ FUNC } n{:}0 \;\; E'(n) \rightarrow \exists \text{ FUNC } m{:}0 \; d(m) = n.$$

A practical implementation will try to reuse the identifier E, instead of replacing E by E'. Note that if one would use the Curry-Howard isomorphism, the two types of modifications, (adding a parameter to a definition, and adding a universal quantifier to a proven formula) would be the same, because under the Curry-Howard isomorphism, definitions and proofs of theorems are the same.

2 Proof Structure

The input to the system consists of a file containing the proof. The system is a batch system. It reads the proof, checks the steps in it, and reports errors. While reading the proof, the system maintains a context Γ, which is updated after every proof step. We list some of the constructions that can occur in proofs. The FROM-rule handles the forward reasoning by the external theorem prover. Most of the other reasoning rules are straightforwardly based on the context modifications that we defined in Section 1.4.

2.1 From

`FROM` is the rule for first order forward reasoning, it is analogeous to the `by` rule of Mizar. It has form:

$$\text{PROVE } L{:}F \text{ FROM } R_1, \ldots, R_n.$$

The $R_1, \ldots, R_n$ must be references that refer to a first order formula. F must be a first order formula. The system calls the external theorem prover which tries to prove F from the formulas denoted by the first order references $R_1, \ldots, R_n$. If it succeeds, F is added to the context as a proven formula. The label L is optional. If a label is present, F will receive label L.

2.2 Permanent Predicate/Function Definitions

A function or predicate definition has one of the following three forms:

$$\text{DEFINE FUNC } D \text{ INDIRECTLY BY } L : E$$

$$\text{EXISTENCE } R_1, \ldots, R_m \text{ UNIQUENESS } S_1, \ldots, S_n.$$

$$\text{DEFINE FUNC } D \text{ BY } L : E, \text{ or}$$

$$\text{DEFINE PRED } D \text{ BY } L : E.$$

D is the identifier being defined. L is a optional label, that will be used for the characteristic formula. E is an expression of form $\lambda x_1 \cdots x_n \ y \ F$, in which F is a formula or a term, dependent on the type of the definition.
In case of an indirect definition, $R_1, \ldots, R_m$ is a list of references from which the theorem prover must be able to prove

$$\forall x_1 \cdots x_n \exists y \ F[x_1, \ldots, x_n, y].$$

$S_1, \ldots, S_n$ is a list of references from which the theorem prover must be able to prove

$$\forall x_1 \cdots x_n \forall y_1 y_2 \ F[x_1, \ldots, x_n, y_1] \wedge F[x_1, \ldots, x_n, y_2] \rightarrow y_1 = y_2.$$

2.3 Local Assumptions

A *local assumption block* has form

$$\text{ASSUME } D_1, \ldots, D_n \text{ IN } P_1, \ldots, P_m \text{ END }.$$

Each D_i has one of the following five forms:

1. PREDICATE $p{:}n$,
2. PREDICATE $p := \lambda x_1 \cdots x_n \ F$,
3. FUNCTION $f{:}n$,
4. FUNCTION $f := \lambda x_1 \cdots x_n \ t$,
5. FORMULA F, in which F is a WUSO formula.

The sequence $P_1, \ldots, P_m$ must be a proof by itself. The system first adds the assumptions $D_1, \ldots, D_n$ to the context. After that, it reads the proof $P_1, \ldots, P_m$, which can make further additions to the context. When reading of $P_1, \ldots, P_m$ is complete, the assumptions $D_1, \ldots, D_n$ are dropped from the context in the order $D_n, D_{n-1}, \ldots, D_1$. The additions, made by the proof $P_1, \ldots, P_m$, are modified according to the rules of Section 1.4.

2.4 Permanent Assumptions

A permanent assumption block has form

$$\text{ASSUME } D_1, \ldots, D_n.$$

Each D_i must have one of the following three forms:

1. PREDICATE $p{:}n$,
2. FUNCTION $f{:}n$,
3. FORMULA F.

3 Conclusions and Future Work

The system is only intended as a first attempt. Probably the most important modification that has to be made, is to add a simple type system. Simple types are very easy to implement in resolution or tableaux. Unfortunately, still none of the existing theorem provers supports simple types. We will extend the next version of Geo with simple types. We also plan to redo the verifications of [3] and of [4] in our system.
The system can be obtained from the homepage of the second author. If the system turns out succesful enough, and stabilizes, we will rewrite it with a trusted code base.

References

1. Marc Bezem, Dimitri Hendriks, and Hans de Nivelle. Automated proof construction in type theory using resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, December 2002.
2. Jean-François Couchot and Stéphane Lescuyer. Handling polymorphism in automated deduction. In Frank Pfenning, editor, *Automated Deduction - CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer Verlag, 2007.
3. Hans de Nivelle and Ruzica Piskac. Verification of an off-line checker for priority-queues. In Peter H. Schmitt, editor, *Proceedings of the 3d IEEE International Conference on Software Engineering and Formal Methods*, pages 210–219, Koblenz, September 2005. IEEE Computer Society Press.
4. Hans de Nivelle and various authors. Verification of the unification algorithm. `www.ii.uni.wroc.pl/~nivelle/teaching/interactive2007/index.html`.
5. Joe Hurd. Integrating gandalf and hol. In *Theorem Proving in Higher Order Logics*, volume 1690 of *LNCS*, pages 311–321. Springer Verlag, 1999.
6. Andreas Meier. TRAMP: Transformation of machine-found proofs into natural deduction proofs at the assertion level. In D. McAllester, editor, *Proceedings of the 17th Conference on Automated Deduction (CADE–17)*, volume 1831 of *LNAI*, pages 460–464, Pittsburgh, USA, 2000. Springer Verlag, Berlin, Germany.
7. Jia Meng and Larry Paulson. Translating higher-order problems to first-order clauses. In Geoff Sutcliffe, Renate Schmidt, and Stephan Schulz, editors, *ESCoR (CEUR Workshop Proceedings)*, volume 192, pages 70–80, 2006.
8. Jia Meng, Claire Quigley, and Lawrence Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.
9. Geoff Sutcliffe. Semantic derivation verification: Techniques and implementation. *International Journal on Artificial Intelligence Tools*, 15(6):1053–1070, 2006.
10. Freek Wiedijk. Writing a Mizar article in nine easy steps. can be obtained from homepage of author.
11. Freek Wiedijk and Andrzej Trybulec. Checker. `http://www.cs.ru.nl/~freek/` .

The Annual SUMO Reasoning Prizes at CASC

Adam Pease[1], Geoff Sutcliffe[2,] Nick Siegel[1], Steven Trac[2]
[1]Articulate Software
apease|nsiegel[at]articulatesoftware.com
[2]University of Miami
geoff|strac[at]cs.miami.edu

Abstract

Previous CASC competitions have focused on proving difficult problems on small numbers of axioms. However, typical reasoning applications for expert systems rely on knowledge bases that have large numbers of axioms of which only a small number may be relevant to any given query. We have created a category in the new LTB division of CASC to test this sort of situation. We present an analysis of performance of last year's entrants in CASC to show how they perform before any opportunity for tuning them to this new competition.

1. Introduction

Previous CASC competitions have focused on proving difficult problems on relatively small numbers of axioms. However, typical reasoning applications for expert systems rely on knowledge bases that have large numbers of axioms, of which only a small number may be relevant to any given query. We have chosen the Suggested Upper Merged Ontology as the basis for a category of the new Large Theory Batch (LTB) division of CASC.

The Suggested Upper Merged Ontology (SUMO) (Niles & Pease, 2001) is a free, formal ontology of about 1000 terms and 4000 definitional statements. It is provided in the SUO-KIF language (Pease, 2003), which is a first order logic with some second-order extensions, and also translated into the OWL semantic web language (which is a necessarily lossy translation, given the limited expressiveness of OWL). In prior work we have described how we transformed SUMO into a strictly first-order form (Pease&Sutcliffe, 2007). SUMO has also been extended with a MId-Level Ontology (MILO), and a number of domain ontologies, which together number some 20,000 terms and 70,000 axioms. SUMO has been mapped to the WordNet lexicon (Fellbaum, 1998) of over 100,000 noun, verb, adjective, and adverb word senses (Niles & Pease, 2003), which not only acts as a check on coverage and completeness, but also provides a basis for work in natural language processing (Pease & Murray, 2003) (Elkateb et al, 2006) (Scheffczyk et al, 2006). SUMO is now in its 75th free version; having undergone five years of development, review by a community of hundreds of people, and application in expert reasoning and linguistics. Various versions of SUMO have been subjected to formal verification with Vampire (Riazanov&Voronkov 2002), which until recently was the only prover we had integrated into our browsing and inference tool suite called Sigma (Pease, 2003). SUMO and all the associated tools and products are available at www.ontologyportal.org.

2.The Competition

The SUMO inference prizes totaling US$3000.00 will be awarded to the best performance on the SMO category of the LTB division of CASC, held at IJCAR 2008. The LTB division has an assurance ranking class and a proof ranking class. In each ranking class the winner will receive $750, the second place $500, and the third place $250 (a system that wins the proof ranking class might also win the assurance ranking class).

We created an additional test to support the participation of model-finders. The SUMO validation prize totaling US$300 will test these systems, and hopefully improve SUMO by finding any problems with the theory. Three subdivisions, each with a $100 prize will be given to those systems which

1. Verify the consistency of, or provide feedback to repair, the base SUMO ontology.
2. Verify the consistency of, or provide feedback to repair, the combined SUMO and MILO ontologies.
3. Verify the consistency of, or provide feedback to repair, the combined SUMO, MILO, and domain ontologies.

The winners of the SUMO challenges will be announced and receive their awards at IJCAR following successful completion of a challenge.

3.Example Test

To give a flavor of what the tests consist of, we present one of them. The question posed to the system can be described as “Can a human perform an intentional action if he or she is dead?”. We create in the test an example instance of an action

```
(instance DoingSomething4-1 IntentionalProcess)
```

then state that an individual is performing the action

```
(agent DoingSomething4-1 Entity4-1)
```

and that the individual is human

```
(instance Entity4-1 Human)
```

The successful theorem prover will then find the following axioms and apply them to prove the conjecture

```
(<=>
   (instance ?X4 Agent)
   (exists (?X5)
     (agent ?X5 ?X4)))

(subclass IntentionalProcess Process)

(=>
  (and
    (subclass ?X403 ?X404)
    (instance ?X405 ?X403))
  (instance ?X405 ?X404))

(=>
  (and
    (agent ?X5 ?X4)
    (instance ?X5 IntentionalProcess))
  (and
    (instance ?X4 CognitiveAgent)
    (not
      (holdsDuring
        (WhenFn ?X5)
        (attribute ?X4 Dead)))))
```

We should note that this proof has the interesting feature that although the form appears to be second order (holdsDuring arg <formula>), the system treats the embedded formula as an uninterpreted list and is able to solve the problem simply by unifying clauses in the list.

While this example is trivial when the necessary axioms are found ahead of time, it becomes very challenging in the context of a large knowledge base, where, in a practical situation, the relevant axioms cannot be known ahead of time. There are hundreds or thousands of axioms involving the term "agent" in SUMO, for example, and the successful theorem prover will have to hunt through those axioms very quickly in order to find just the ones that are relevant to the query being posed.

4.Analysis

In order to test whether the competition was even reasonable, we decided to run it on all the provers in the SystemOnTPTP suite. These were Bliksem 1.12, CARINE 0.734, CiME 2.01, Darwin 1.4.1, DarwinFM 1.4.1, DCTP 1.31, E 0.999, E-KRHyper 1.0, EQP 0.9d, Equinox 1.3, Fampire 1.3, Faust 1.0, FDP 0.9.16, Fiesta 2, Gandalf c-2.6, Geo 2007f, GrAnDe 1.1, iProver 0.2, leanCoP 2.0, LeanTAP 2.3, Mace2 2.2, Mace4 1207, Matita 0.1.0, Metis 2.0, Muscadet 2.7a, Otter 3.3, Paradox 2.3, Prover9 1207, S-SETHEO 0.0, SETHEO 3.3, SNARK 20070805, SOS 2.0, SPASS 3.0, SRASS 0.1, Theo 2006, Vampire 9.0, Waldmeister 806, zChaff 04.11.15, Zenon 0.5.0. We gave each prover 600 seconds on each of 102 problems, generated from 33 distinct queries (possibly with some additional assertions to the knowledge base) each tested with just the ~4000 axioms in SUMO, the ~9000 axioms of SUMO+MILO or the tens of thousands of axioms in SUMO+MILO and all the domain ontologies.

Overall	**SUMO**	**SUMO+MILO**	**All**
Vampire ٩.٠	Vampire ٩.٠	Vampire ٩.٠	Metis ٢.٠
Metis ٢.٠	E ٠.٩٩٩	Metis ٢.٠	Zenon ٠.٥.٠
E ٠.٩٩٩	iProver ٠.٢	SNARK ٢٠٠٧٠٨٠٥	Equinox ١.٣
iProver ٠.٢	leanCoP ٢.٠	Zenon ٠.٥.٠	
leanCoP ٢.٠	Metis ٢.٠	Equinox ١.٣	
Darwin ١.٤.١	Darwin ١.٤.١	Muscadet ٢.٧a	
Zenon ٠.٥.٠	Fampire ١.٣		
Equinox ١.٣	SNARK ٢٠٠٧٠٨٠٥		
Fampire ١.٣	Zenon ٠.٥.٠		
SNARK ٢٠٠٧٠٨٠٥	Equinox ١.٣		
Muscadet ٢.٧a	Muscadet ٢.٧a		
SPASS ٣.٠	SPASS ٣.٠		
Faust ١.٠	Faust ١.٠		

Table 1: Performance ranking

Overall performance is shown in the first column above with Vampire achieving first place. All other provers not listed failed to solve any of the problems. The best performance with SUMO alone is shown then SUMO+MILO and finally performance with all the domain ontologies loaded. The best performing provers still did not solve a majority of the 105 problems in the test set. Vampire solved 31, Fampire 20, E 15 and Metis 14, with the other provers in the single digits or no solutions at all. Prover failing to find solutions were stopped generally because of timeouts, rather than errors in parsing or memory space. Average running times approached the 600 seconds allocated for all provers because of

the low percentage of solved problems.

The differing strengths of several of the provers suggested creating a “meta-prover” combining several systems. The strategy is to give Vampire 400 seconds, then give Metis up to 200 seconds if Vampire failed to find a proof. The combined system gets 33 answers compared to 31 for Vampire alone or 14 for Metis alone, and performance overall is slightly better at 48158 seconds vs. 55419 for Metis and 48599 for Vampire. We might be able to tweak the timeslice allocation to do still better, although further efforts in that regard could be considered overtraining to this particular problem set.

We performed an analysis to determine what set of systems would cover the maximum number of problems (see Figure 1). This is termed a “SOTA” analysis as per (Sutcliffe & Suttner 2001). Vampire solved eight problems solved by no other prover. Metis uniquely solved two, and Fampire 1. This analysis suggests that we should revisit creation of a meta-prover composed of Vampire, Metis and Fampire.

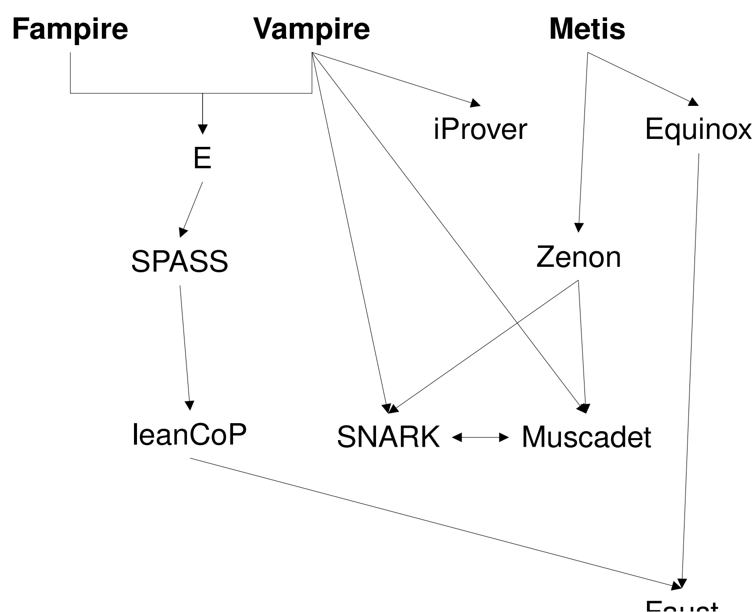

Figure 1: SOTA analysis

5.Conclusions

We have created a category called “SMO” in the new LTB division of CASC to motivate high performance reasoning on practical problems using a broad knowledge base. We believe this will yield some exciting research results, as well as provide the application development community with provers that are more closely optimized to the needs to one sort of practical inference. We have run the tests with existing theorem provers and found the competition to be a reasonable goal for these systems. With tuning, we expect even better performance.

In the future we expect to expand the number of tests in the SMO category. We also anticipate providing a “stratified” set of tests of different expressiveness, in which we extract the horn clause and description logic subsets of SUMO and provide tests on those subsets.

Acknowledgments

This work has been funded by a number of sources, including the Army Research Institute. We are grateful for their investment.

References

Elkateb, S., Black, W., Rodriguez, H, Alkhalifa, M., Vossen, P., Pease, A. and Fellbaum, C., (2006). Building a WordNet for Arabic, in *Proceedings of The fifth international conference on Language Resources and Evaluation (LREC 2006).*

Fellbaum, C. (ed.) WordNet: An Electronic Lexical Database. MIT Press, 1998.

Genesereth, M., (1991). "Knowledge Interchange Format", In Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning, Allen, J., Fikes, R., Sandewall, E. (eds), Morgan Kaufman Publishers, pp 238-249.

Hayes, P., and Menzel, C., (2001). A Semantics for Knowledge Interchange Format, in *Working Notes of the IJCAI-2001 Workshop on the IEEE Standard Upper Ontology*.

Niles, I & Pease A., (2001). "Towards A Standard Upper Ontology." In *Proceedings of Formal Ontology in Information Systems (FOIS 2001)*, October 17-19, Ogunquit, Maine, USA, pp 2-9. See also http://www.ontologyportal.org

Niles, I., and Pease, A. (2003) Linking Lexicons and Ontologies: Mapping WordNet to the Suggested Upper Merged Ontology, *Proceedings of the IEEE International Conference on Information and Knowledge Engineering*, pp 412-416.

Pease, A., (2003). The Sigma Ontology Development Environment, in *Working Notes of the IJCAI-2003 Workshop on Ontology and Distributed Systems*. Volume 71 of CEUR Workshop Proceeding series. See also http://sigmakee.sourceforge.net

Pease, A., (2004). Standard Upper Ontology Knowledge Interchange Format. Unpublished language manual. Available at http://sigmakee.sourceforge.net/

Pease, A., and Murray, W., (2003). An English to Logic Translator for Ontology-based Knowledge Representation Languages. In *Proceedings of the 2003 IEEE International Conference on Natural Language Processing and Knowledge Engineering*, Beijing, China, pp 777-783.

Pease, A., and Sutcliffe, G., (2007) First Order Reasoning on a Large Ontology, in Proceedings of the CADE-21 workshop on Empirically Successful Automated Reasoning on Large Theories (ESARLT).

Ramachandran, D., P. Reagan, K. Goolsbey. First-Orderized ResearchCyc: Expressivity and Efficiency in a Common-Sense Ontology. In *Papers from the AAAI Workshop on Contexts and Ontologies: Theory, Practice and Applications*. Pittsburgh, Pennsylvania, July 2005.

Riazanov A., Voronkov A. (2002). The Design and Implementation of Vampire. *AI Communications*, 15(2-3), pp. 91—110.

Scheffczyk, J., Pease, A., Ellsworth, M., (2006). Linking FrameNet to the Suggested Upper Merged Ontology, in *Proceedings of Formal Ontology in Information Systems (FOIS-2006)*, B. Bennett and C. Fellbaum, eds, IOS Press, pp 289-300.

Sutcliffe G., Suttner C.B. (1998), The TPTP Problem Library: CNF Release v1.2.1, *Journal of Automated Reasoning* 21(2), 177-203.

Sutcliffe G., Suttner C.B. (2001), Evaluating General Purpose Automated Theorem Proving Systems, *Journal of Artificial Intelligence* 131(1-2), 39-54.

Combining Theorem Proving with Natural Language Processing

Björn Pelzer[1] and Ingo Glöckner[2]

[1] Department of Computer Science, Artificial Intelligence Research Group
University of Koblenz-Landau, Universitätsstr. 1, 56070 Koblenz
bpelzer@uni-koblenz.de
[2] Intelligent Information and Communication Systems Group (IICS),
University of Hagen, 59084 Hagen, Germany
ingo.gloeckner@fernuni-hagen.de

Abstract. The LogAnswer system is an application of automated reasoning to the field of open domain question answering, which aims at finding answers to natural language questions regarding arbitrary topics. In our system we have integrated an automated theorem prover in a framework of natural language processing tools to allow for deductive reasoning over an extensive knowledge base derived from textual sources. For this purpose we had to intertwine two opposing approaches: on the one hand formal logic with its precision but brittleness, and on the other hand, machine learning applied to shallow linguistic features, which are robust but less precise. In the paper we present implementation details and discuss obstacles and their proposed solutions.

1 Introduction

Question answering (QA) systems generate natural language (NL) answers in response to NL questions, using a large collection of textual documents.[3] Simple factual questions can be answered using only information retrieval and shallow linguistic methods like named entity recognition [1]. More advanced cases, like questions involving a temporal description, call for deduction based question answering which can provide support for temporal reasoning and other natural language related inferences [2]. Complete QA systems which integrate question answering and logical reasoning are [3,4,5]. The junctures of logic and answer validation are also addressed in research on recognizing textual entailment [6,7].

Our LogAnswer system uses first order logic to represent an extensive knowledge base, and a combination of NL processing tools and an automated theorem prover to derive answers within a few seconds, i.e. in a time frame appropriate for ad-hoc question answering on the web. The fields of automated reasoning and natural language processing (NLP) differ greatly in their methodologies. A

[3] A survey on the progress in question answering technology is provided by the QA track of the TREC conference series (see http://trec.nist.gov) and by the CLEF workshops (see http://www.clef-campaign.org/).

Fig. 1. Screenshot of the LogAnswer prototype. The system is available online at `www.loganswer.de`.

theorem prover generally implements a sound and complete deduction calculus which can produce complex proofs using hundreds of inference steps, and it operates on a set of clauses or formulas where consistency or lack thereof is critical for the result. By contrast, natural language is ambiguous, textual sources may be imperfect, and as a consequence a knowledge base derived from such sources cannot be expected to be consistent. Moreover, it is not feasible to provide a complete formalization of the background knowledge used by persons in understanding natural language. A practical NLP approach must take this into account and employ robustness-enhancing methods to overcome the flaws and deliver useful results.

Merging the reasoning depth of an automated theorem prover with the robustness of NL processing presents a number of difficulties. We will provide a short overview of our LogAnswer system and then address the obstacles and solutions in detail.

2 Description of the LogAnswer System

LogAnswer is a QA-system supporting the German language. A full system description is presented in [8]. LogAnswer operates on a knowledge base consisting of general semantic background rules as well as factual knowledge derived from the German Wikipedia and a corpus of newspaper articles. This knowledge is represented by semantic networks in the MultiNet formalism [9]. The user interacts with LogAnswer via the user interface, where a NL question can be entered into a search box, as shown in Figure 1. The query is then processed in several stages. The WOCADI parser module [10] translates the query into a MultiNet

representation. The formalized text passages from the MultiNet knowledge base are then searched for the query terms, and the matching network fragments are returned as the basis for generating answer candidates. Each retrieved fragment corresponds to a text passage which may contain an answer to the question. Currently, 200 MultiNet passage representations are retrieved for each question. Shallow linguistic methods provide additional filtering, further cutting down the number of candidate passages.

At this stage the deduction-based processing begins. The automated reasoning component of LogAnswer is E-KRHyper [11], an automated theorem prover for first order logic with equality, based on the hyper tableaux calculus [12,13]. E-KRHyper is an in-house system, developed for embedding in knowledge representation applications. It has been equipped with several features, commands and modes of operation for this purpose. Our familiarity with the system allows us to perform deep modifications when required. Because of our experience in adapting this prover to numerous systems in the past, it was a natural choice for a reasoner in LogAnswer. Its performance is roughly comparable to Otter [14], a system which generally serves as a benchmark among automated theorem provers.

The query and the MultiNet candidate passages are converted into first order logic representations. For each answer candidate the theorem prover E-KRHyper attempts to refute the negated query representation in conjunction with the background knowledge rules and the respective candidate. A successful refutation indicates that an answer has been found, and the queried information is extracted from the refutational proof.

If answers are found for multiple candidate passages, then they are ranked according to features extracted from logic processing and from shallow linguistic analysis (e.g. lexical overlap). The five best answers are presented to the user. Depending on user choice, the answers are given only in the form of snippets from the textual sources or as precise answers together with the snippets providing context. The quality score shown with each result is an estimate of the correctness probability of the answer determined by a machine learning approach.

3 Knowledge Representation

MultiNet (**Multi**layered Extended Semantic **Net**works) [9] is a formalism for knowledge representation via semantic networks, which is particularly suited for the meaning representation of natural language. Characteristic of MultiNet is its stable inventory of pre-defined relations (edge labels), which made possible the long-term development of a computational lexicon based on MultiNet [15], and the introduction of so-called layer attributes. These attributes are used in multi-dimensional node descriptions which serve to capture quantification and other aspects of meaning that cannot be expressed using the relational means of a semantic network.

The formalism is generally independent of any particular natural language, although the tools for translating NL into MultiNet are only available for German

(and in a rudimentary form for English) at this time.[4] The knowledge base of LogAnswer was therefore derived from textual sources in German, namely the German Wikipedia and a corpus of newspaper articles. All in all about 12 million sentences have been translated into semantic networks, which are stored using Scheme syntax.

For the deduction-based processing in LogAnswer this knowledge base is further translated into first-order logic, stored in the TPTP format [16], a standard among automated theorem provers. The MultiNet nodes and attributed arcs can be translated in a fairly straightforward manner into relations and constants, and the same holds for the logical MultiNet rules which express the semantics of words and the logical properties of the MultiNet relations. However, it should be noted that MultiNet goes beyond the expressivity of pure first-order logic and TPTP. For example, MultiNet networks can contain generalized quantifiers like 'most' or 'almost all'. These aspects of the MultiNet representation are lost in the translation to logical expressions. In fact, our current translation from MultiNet to logic results in Horn logic (plus arithmetic expressions). It is planned to translate into a more powerful logic including equality in order to better capture the actual meaning of the natural language sentences. In order to allow such an extension, the E-KRHyper prover for full first-order logic with equality was chosen, which also supports arithmetic expressions.

4 Query Representation

For our purposes of logic-based question answering, the NL question must be translated into a conjunctive list of query literals. Synonyms are normalized by replacing all lexical concepts with canonical synset representatives. If the question asks for specific information, then this is represented by a special $FOCUS$ variable. In a successful proof this variable will be instantiated with the desired information from the knowledge base.

For example, *Rudy Giuliani war Bürgermeister welcher US-Stadt?*[5] translates into the following logical query:

$$\begin{aligned}&\mathrm{attr}(X_1, X_2), \mathrm{attr}(X_1, X_3), \mathrm{val}(X_2, \mathit{rudy.0}), \mathrm{sub}(X_2, \mathit{vorname.1.1}),\\ &\mathrm{val}(X_3, \mathit{giuliani.0}), \mathrm{sub}(X_3, \mathit{nachname.1.1}), \mathrm{sub}(FOCUS, \mathit{usstadt.1.1}),\\ &\mathrm{attch}(FOCUS, X_1), \mathrm{sub}(X_1, \mathit{b\ddot{u}rgermeister.1.1})\end{aligned}$$

5 Deduction-based Query Processing

Each of the candidate passages retrieved from the knowledge base may contain an answer to the query, so a separate proof attempt is made for each candidate.

[4] In order to adapt the system to another language, a computational lexicon and a parser for generating MultiNet representations from expressions in that language must be provided. While the logical core of LogAnswer can remain the same, the lexical-semantic relations (synonyms, nominalizations etc.) used by LogAnswer must also be adapted to the language of interest.

[5] *Rudy Giuliani was the mayor of which city in the USA?*

The proof is done by refutation, with the logical query representation being treated as a negated conjecture. E-KRHyper operates on a clause normal form (CNF) representation and converts its first order input accordingly. To continue our example, E-KRHyper uses the following representation of the query:

$$\neg\,\mathrm{attr}(X_1, X_2) \vee \neg\,\mathrm{attr}(X_1, X_3) \vee \neg\,\mathrm{val}(X_2, \mathit{rudy.0}) \vee \neg\,\mathrm{sub}(X_2, \mathit{vorname.1.1}) \vee$$
$$\neg\,\mathrm{val}(X_3, \mathit{giuliani.0}) \vee \neg\,\mathrm{sub}(X_3, \mathit{nachname.1.1}) \vee \neg\,\mathrm{sub}(\mathit{FOCUS}, \mathit{usstadt.1.1}) \vee$$
$$\neg\,\mathrm{attch}(\mathit{FOCUS}, X_1) \vee \neg\,\mathrm{sub}(X_1, \mathit{bürgermeister.1.1})\,.$$

The prover keeps track of the $FOCUS$ variable throughout all clause transformations and inference steps, so that its binding can be extracted from a proof even if it has been renamed.

As its derivation E-KRHyper builds a hyper tableau in the form of a literal tree, using the hyper extension inference: if the negative literals of a clause unify with complementary literals from a tableau branch, then the positive literals of the clause are added as leaves. A branch is closed once it is found to contain a contradiction. In a derivation for LogAnswer this is the case when all the negative literals from the query unify with the branch; given that the query has no positive literals, the branch gets closed. The term bound to the $FOCUS$ variable in the unifying substitution used in the refutational proof then represents the queried information.

The current logical background knowledge consists of Horn formulas only, but with the ongoing translation of the MultiNet knowledge the logical rules will eventually contain non-Horn formulas as well. In a hyper tableaux derivation these can lead to tableau branching, with multiple closed branches and thus multiple closing unifiers. In such a case E-KRHyper extracts all the bindings for the $FOCUS$ variable from the proof and presents them as different answers to the main LogAnswer system.

6 Ensuring Robustness

A knowledge base derived from textual sources is bound to have imperfections. Furthermore, the logical background knowledge provided to the prover will never completely cover the actual background knowledge of a person who reads the text. Finally, the candidate passages are not guaranteed to contain an answer to the query, since they have only been selected by relatively simple filtering. For these reasons it is not certain that E-KRHyper can find a proof within an answer candidate, even if the corresponding NL source actually contains the queried information. Given that LogAnswer is supposed to provide answers in a short amount of time, the maximum time slot dedicated to a single proof attempt is constrained severely to ensure that all answer candidates can be tested. However, while a time limit ensures that the prover will not work indefinitely on one futile candidate when answers would be readily available in others, it does not help against missing an answer due to minor mismatches. When the formulas being reasoned upon have all been derived from imperfect textual sources and by

imperfect tools for linguistic analysis, then formal logic may be too rigorous in demanding a perfect proof for an answer, and small compromises may actually be acceptable.[6]

For this reason E-KRHyper is embedded in a *relaxation loop*: if no proof is found within a time limit, then the query is relaxed by dropping a query literal and restarting the prover with the shortened query.[7] In theory this can be continued until a proof of the simplified query succeeds, but since LogAnswer aims to produce useful answers the loop will be stopped before all literals are skipped.

Also, rather than skipping a random query literal, the derivation progress during the failed refutation attempt can be used to guide the choice of which literal to drop. Given a negated query clause $\neg Q_1 \vee \cdots \vee \neg Q_n$, E-KRHyper tries to unify the query literals with fresh variants $B_1, \ldots, B_n$ of complementary branch literals from the current tableau branch b by testing the query literals from left to right. The unifying substitution is extended in every step such that starting with the empty substitution σ_0, σ_k is a substitution with $Q_i\sigma_k = B_i, \forall i \in \{1 \ldots k\}$, with $k \in \{1 \ldots n\}$. If one query literal $\neg Q_l$ fails to find a matching partner in the branch which would allow an extension of the current substitution σ_{l-1} to σ_l, then the remaining literals $\neg Q_{l+1}, \ldots, \neg Q_n$ are not tested under σ_{l-1}.

Instead E-KRHyper generates a *partial result*. A partial result is represented by a triple $(\{Q_1, \ldots, Q_{l-1}\}, \sigma_{l-1}, \{Q_l, \ldots, Q_n\})$, consisting of the list of successfully unified query literals, the unifying substitution, and the list of query literals that were not unified, the first of which being the one that failed, whereas the remaining were not tested at all. If E-KRHyper fails to find a proof within the time limit, then all partial results generated so far are returned to the main LogAnswer system. One of the best partial results is selected (i.e. one of the partial results with the highest number of refuted query literals), the failed literal $\neg Q_l$ is removed from the negated query clause and E-KRHyper restarts its derivation with the new query clause $\neg Q_1 \vee \cdots \vee \neg Q_{l-1} \vee \neg Q_{l+1} \vee \cdots \vee \neg Q_n$. When two partial results have the same number of failed literals, then additional criteria are used for selecting the best partial result: (a) partial results which provide a binding to the $FOCUS$ variable are considered better than partial results which do not bind the queried variable (this criterion is important since the system can only generate answers when the $FOCUS$ variable has been bound); and (b) a binding of the $FOCUS$ variable to a constant is preferable to a binding of the $FOCUS$ variable to a complex term (at the moment, the system is unable to generate answers for skolem terms, so answer extraction will only succeed when the queried variable is bound to a constant which directly represents a discourse entity).

[6] A training set is used to learn a useful interpretation of results for failed proofs [17].
[7] See Sect. 8 for an example. Another system which uses relaxation for achieving more robustness in logic-based QA is COGEX [3].

7 A Theorem Prover as a Reasoning Server

At the time of this writing LogAnswer is deployed as a web-based QA-system with an interface analogous to that of a typical search engine. This usage places certain demands on the performance of LogAnswer which must be able to respond to numerous queries from multiple users in a short time. The reasoning stage within E-KRHyper can easily be the most time-consuming phase in the processing of a query, even with relaxation. Cutting back on operations here is crucial for maintaining responsiveness. Fortunately there are several opportunities for such measures.

To summarize, the clause input for a single proof attempt consists of:

- **background knowledge:** 10,200 CNF clauses,
- **query:** the negated query clause, on average 8 literals before relaxation,
- **answer candidate:** the logic representation of the candidate passage, circa 230 unit clauses.[8]

E-KRHyper maintains this clause set with the help of several dicrimination-tree indexes [18]. The time required for the construction of these extensive tree data structures can exceed the allotment for the reasoning phase. Repeating this for each proof attempt would be prohibitive, in particular when considering that there may be m answer candidates to check for a single query, with n relaxation steps for each candidate, resulting in $m \times n$ proof attempts for each query.

However, the clause sets and their indexing trees differ only very slightly between any two reasoning tasks. Only the current query and answer candidate can change between two proof attempts. The background knowledge, which comprises approximately 97% of every clause set, remains the same. Therefore LogAnswer does not restart E-KRHyper for each reasoning task. Instead the prover is started once and provided with the background knowledge, so that the construction of the bulk of the indexing structures is only done once during this initialization. The task-specific clauses are then added and retracted again as needed. This is done using the mechanism of *layered* discrimination-trees: the task-specific clauses are not actually added to the same tree structures as the background knowledge. Instead additional trees (the *layers*) will be created to store the new clauses. Index reading operations access all layers, whereas writing operations only use the latest layer intended for new clauses. Once a proof attempt is completed, the index layers with the now unnecessary clauses are simply discarded. This avoids a lengthy extraction of these clauses from a single, shared tree.

There is even less difference between the clause sets used in two consecutive relaxation steps. The only change here is the removal of one query literal. This allows us to reuse even more clauses. Given an (interrupted) hyper tableaux derivation, all derived literals which were added to the initial branch before the

[8] The problem set used for determining these numbers consists of 1806 query/candidate passage combinations for the CLEF 2007 questions for German.

first branch split can be treated as lemmas. Since the current logic representation of the knowledge base is Horn only, the set of lemmas actually corresponds to all derived branch literals, although this is bound to change in the future. The lemmas are then kept for the next relaxation step. Thus a repetition of inferences is avoided, and even if a relaxation step is not successful in finding an answer, it still makes use of its limited time slot to add new lemmas. This way we combine relaxation with incremental reasoning [19].

8 Prover Result Evaluation

The input for E-KRHyper is selected and updated by the LogAnswer main system, and the operation of the prover is also directed and controlled as described above. Given that many supporting text passages will be analysed for a query, the results of E-KRHyper must be subjected to further evaluation as well in order to select those answers that are most suitable for the user.

For this we apply a machine learning approach which combines both logic-based and shallow syntactic features [17]: whenever E-KRHyper terminates with a refutation, then the respective text passage is regarded as containing an answer to the query. All such passages are ranked by an aggregated score, computed from logic-based criteria regarding the respective proofs, like the number of relaxation steps required, as well as from shallow syntactic features, like the relative proportion of lexical concepts and numerals in the question which find a match in the candidate passage. If the user has opted to receive answers in the form of text snippets, then the five best passages are selected for presentation.

On the other hand, if precise answers are desired, then further processing is necessary. The queried information is extracted directly from a refutational proof as the binding of the $FOCUS$ variable. The context of this instantiating answer value within the semantic network underlying the candidate passage is used to phrase the actual answer that can be presented to the user.

Continuing our running example we take a look at one of the candidate passages for which E-KRHyper will terminate: *Hinter der Anklage stand der spätere Bürgermeister von New York, Rudolph Giuliani.*[9] The logical representation of the passage is shown here (actually, only the fragment of the representation which models the relational structure):

$$\begin{aligned}&\mathrm{hinter}(c221, c210) \land \mathrm{sub}(c220, \mathit{nachname.1.1}) \land \mathrm{val}(c220, \mathit{giuliani.0}) \land\\ &\mathrm{sub}(c219, \mathit{vorname.1.1}) \land \mathrm{val}(c219, \mathit{rudolph.0}) \land \mathrm{prop}(c218, \mathit{sp\ddot{a}t.1.1}) \land \mathrm{attr}(c218, c220) \land\\ &\mathrm{attr}(c218, c219) \land \mathrm{sub}(c218, \mathit{b\ddot{u}rgermeister.1.1}) \land \mathrm{val}(c216, \mathit{new_york.0}) \land\\ &\mathrm{sub}(c216, \mathit{name.1.1}) \land \mathrm{sub}(c215, \mathit{stadt.1.1}) \land \mathrm{attch}(c215, c218) \land \mathrm{attr}(c215, c216) \land\\ &\mathrm{subs}(c211, \mathit{stehen.1.1}) \land \mathrm{loc}(c211, c221) \land \mathrm{scar}(c211, c218) \land \mathrm{sub}(c210, \mathit{anklage.1.1})\end{aligned}$$

A full proof of the query fails since the system is lacking knowledge that *Rudy* is a short form of *Rudolph*. Moreover, the fact that New York is a US city is not known to the system. Therefore two query literals cannot be proved, viz $\mathrm{val}(X_2, \mathit{rudy.0})$

[9] *Responsible for the charges was the future mayor of New York, Rudolph Giuliani.*

and sub(*FOCUS*, *usstadt.1.1*). After two relaxation steps, which skip these two literals, a proof is found with a constant *c215* bound to the *FOCUS* variable, which corresponds to an entity mentioned in the text. The information about the position of the entity in the text string is then used to extract the answer *New York*.

Some answers generated this way are discarded immediately. This includes trivial answers which return the term from the query (*Who is Virginia Kelley?* - *Virginia Kelley*) and non-informative answers (*the mother* instead of *the mother of Bill Clinton*). Of those answers passing these sanity checks the five best in the aforementioned ranking are selected and then displayed together with the supporting passages.

9 Conclusions

In this paper we have explored an application of automated reasoning to open domain question answering. With the ever growing amounts and availability of digitally stored information the utilization of this data is becoming an important but difficult task. Question answering systems strive to find specific information in a significantly more goal-directed manner than search engines. This requires deep reasoning over the semantics of stored data. Our approaches bridge the gaps between the precision yet brittleness of deduction and the robustness but limited accuracy of shallow linguistic techniques. This is achieved by combining the results of both levels using machine learning. We have shown how the difficulties of a theorem prover dealing with imperfect NL-derived data can be solved by embedding the deduction component in a robust knowledge processing framework, which uses as feedback loop to gradually relax the logical query.

The proposed approach to ensuring robustness has been evaluated in [17]. The task was that of identifying the correct answer passages in a set of 12,337 retrieved candidate passages. A baseline system using shallow features but no logical reasoning achieved an F-score of 42.7% in this experiment (other criteria for filtering quality were also studied). When combining shallow features and strict proofs, the F-score increased by 7.3%. Adding relaxation achieved another 7.9% improvement of the F-score. The best results were obtained in a configuration which allowed three relaxation steps. This experiment demonstrates that the performance of our QA system profits from the automated reasoning capabilities of the logic prover and from the proposed relaxation technique. Another benefit of the logic-based approach is that the answer bindings determined by the prover provide the basis for answer extraction.

In the future we intend to further improve the translation of the MultiNet formalism into first-order logic, enabling us to fully exploit the expressivity of the semantic networks in combination with automated reasoning.

References

1. Prager, J., Brown, E., Coden, A., Radev, D.: Question-answering by predictive annotation. In: SIGIR '00: Proceedings of the 23rd Annual International ACM

SIGIR Conference on Research and Development in Information Retrieval, New York, NY, ACM Press (2000) 184–191
2. Moldovan, D., Bowden, M., Tatu, M.: A temporally-enhanced PowerAnswer in TREC 2006. In: Proc. of TREC-2006, Gaithersburg, MD (2006)
3. Moldovan, D., Clark, C., Harabagiu, S., Maiorano, S.: COGEX: A logic prover for question answering. In: Proc. of NAACL-HLT 2003. Volume 1., Morristown, NJ (2003) 87–93
4. Saias, J., Quaresma, P.: The Senso question answering approach to Portuguese QA@CLEF-2007. In: Working Notes for the CLEF 2007 Workshop, Budapest, Hungary (2007)
5. Glöckner, I., Hartrumpf, S., Leveling, J.: Logical validation, answer merging and witness selection: A study in multi-stream question answering. In: Proc. of RIAO-07, Pittsburgh (2007)
6. Bos, J., Markert, K.: When logical inference helps determining textual entailment (and when it doesn't). In: Proc. of 2nd PASCAL RTE Challenge Workshop. (2006)
7. Bobrow, D., Condoravdi, C., Crouch, R., de Paiva, V., Kaplan, R., Karttunen, L., King, T., Zaenen, A.: A basic logic for textual inference. In: Proceedings of the AAAI Workshop on Inference for Textual Question Answering, Pittsburgh, PA (Jul 2005)
8. Furbach, U., Glöckner, I., Helbig, H., Pelzer, B.: LogAnswer - A Deduction-Based Question Answering System. In: IJCAR 2008 - 4th International Joint Conference on Automated Reasoning, Sydney, Australia, 10th - 15th August, 2008, Proceedings, to appear. Lecture Notes in Computer Science, Springer (2008)
9. Helbig, H.: Knowledge Representation and the Semantics of Natural Language. Springer (2006)
10. Hartrumpf, S.: Hybrid Disambiguation in Natural Language Analysis. Der Andere Verlag, Osnabrück, Germany (2003)
11. Pelzer, B., Wernhard, C.: System Description: E-KRHyper. In: Automated Deduction - CADE-21, Proceedings. (2007) 508–513
12. Baumgartner, P., Furbach, U., Niemelä, I.: Hyper Tableaux. In: JELIA'96, Proceedings. (1996) 1–17
13. Baumgartner, P., Furbach, U., Pelzer, B.: Hyper Tableaux with Equality. In: Automated Deduction - CADE-21, Proceedings. (2007)
14. McCune, W.: OTTER 3.3 Reference Manual. Argonne National Laboratory, Argonne, Illinois (2003)
15. Hartrumpf, S., Helbig, H., Osswald, R.: The semantically based computer lexicon HaGenLex. Traitement automatique des langues **44**(2) (2003) 81–105
16. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning **21**(2) (1998) 177–203
17. Glöckner, I., Pelzer, B.: Exploring robustness enhancements for logic-based passage filtering. In: KES2008, Proceedings, to appear. Lecture Notes in Computer Science, Springer (2008)
18. McCune, W.: Experiments with Discrimination-Tree Indexing and Path Indexing for Term Retrieval. Journal of Automated Reasoning **9**(2) (1992) 147–167
19. Beckert, B., Pape, C.: Incremental theory reasoning methods for semantic tableaux. In: Proceedings, 5th International Workshop on Theorem Proving with Analytic Tableaux and Related Methods. Lecture Notes in Computer Science, Springer (1996)

Presenting TSTP Proofs with Inference Web Tools

Paolo Pinheiro da Silva[1], Geoff Sutcliffe[2], Cynthia Chang[3], Li Ding[3], Nick del Rio[1], and Deborah McGuinness[3]

[1]University of Texas at El Paso, [2]University of Miami, [3]Rensselaer Polytechnic Institute

Abstract. This paper describes the translation of proofs in the Thousands of Solutions from Theorem Provers (TSTP) solution library to the Proof Markup Language (PML), and the subsequent use of Inference Web (IW) tools to provide new presentations of the proofs. The translation enriches the TSTP proofs with proof provenance meta-data, and provides new possibilities for proof processing.

1 Introduction

The Thousands of Problems for Theorem Provers (TPTP)[1] problem library [12] and the Thousands of Solutions from Theorem Provers (TSTP)[2] solution library are large corpora of data for and produced by Automated Theorem Proving (ATP) systems and tools. In particular, the TSTP provides solutions to the TPTP problems, so that the main computation linking the two libraries is the execution of ATP systems on the TPTP problems to produce the TSTP solutions. Additionally, there are many other tools, mostly from the TPTPWorld [10], that are used on the corpora for other tasks such as problem analysis, problem transformation, proof analysis, proof verification, and proof presentation. The TPTP and the TSTP files are written using the TPTP language [11]. The common modus operandi of the tools is to work on one file (problem or solution) at a time, focussing on the first-order logical data therein.

The Inference Web (IW)[3] [5] is a semantic web based knowledge provenance infrastructure that supports interoperable explanations of sources, assumptions, learned information, and answers, as an enabler for trust. IW includes two components that are important for this work, the Proof Markup Language (PML) ontology [7] and the IW toolkit. PML is a semantic web based representation for exchanging explanations, including provenance information - annotating the sources of knowledge, justification information - annotating the steps for deriving the conclusions or executing workflows, and trust information - annotating trustworthiness assertions about knowledge and sources. The IW toolkit provides web-based and standalone tools that facilitate human users to browse, debug,

[1] http://www.tptp.org
[2] http://www.tptp.org/TSTP/
[3] http://inference-web.org/

explain, and abstract knowledge encoded in PML. In contrast to the TPTP, there is less focus on the logical data and the fine-grained reasoning processes - PML supports arbitrary logical data and inference steps including, e.g., extraction of data from non-logical sources, conversion to logical forms, clausification and first-order inferences, etc.

There are obvious parallels between the TPTP language/TPTPWorld and the PML language/IW toolkit. While the scope of the IW is broader than the logic-focussed TPTP/TSTP, there are obvious benefits to building bridges between the two. Principally, the TSTP offers a large corpora of data for testing and developing the IW, and the IW offers alternative views of the proofs in the TSTP. This paper describes the translation of TSTP files to PML format, and the presentation of the proofs using IW tools. The contribution of this work is to add value to TPTP proofs, by translation to PML and viewing the translated proofs with IW tools. Specific benefits include an XML proof format for TPTP proofs, links to provenance information maintained in the IW (e.g., information about ATP systems), structural search tools (rather than `grep`ing over the text form of TPTP proofs), and new views on TPTP proofs and proof nodes.

This paper is organized as follows: Section 2 provides the necessary background about the TPTP, TSTP, and PML. Section 3 describes the translation of TSTP files into PML format. Section 4 describes four IW tools' presentations of the proofs, demonstrating the value of these different views.

2 Background

2.1 About the TPTP and TSTP

The top level building blocks of TPTP and TSTP files are *annotated formulae*, *include directives*, and *comments*. An annotated formula has the form:

language`(`*name*`,` *role*`,` *formula*`,` *source*`,` *useful_info*`).`

The *language*s currently supported are `fof` - formulae in full first order form, and `cnf` - formulae in clause normal form. The *role* gives the user semantics of the *formula*, e.g., `axiom`, `definition`, `lemma`, `conjecture`, which guides the use of the formula in an ATP system. The logical *formula*, in either FOF or CNF, uses a consistent and easily understood notation [13]. The forms of identifiers for uninterpreted functions, predicates, and variables follow Prolog conventions, i.e., functions and predicates start with a lowercase letter, variables start with an uppercase letter, and all contain only alphanumeric characters and underscore. The TPTP language also supports interpreted symbols, which either start with a `$`, e.g., `$true` and `$false`, or are composed of non-alphanumeric characters, e.g., `=` and `!=` for equality and inequality. The basic logical connectives are `!`, `?`,`~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for $\forall$, $\exists$,$\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Quantified variables follow the quantifier in square brackets, with a colon to separate the quantification from the logical formula. The *source* of an annotated formula describes where the formula came from, most commonly a `file` record or an `inference` record. A `file` record stores the name of the file from which the annotated formula was read, and optionally the name of the annotated formula

as it appears in the file. An `inference` record stores three items of information about an inferred formula: the name of the inference rule provided by the ATP system; a list of useful information items, e.g., the semantic `status` of the formula as an SZS ontology value [13]; and a list of the parents. The *useful_info* is a list of arbitrary useful information, as required for user applications. An example of a FOF formula, supplied from a file, is:

```
fof(formula_27,axiom,
    ! [X,Y] :
      ( subclass(X,Y) <=>
      ! [U] : ( member(U,X) => member(U,Y) )),
    file('SET005+0.ax',subclass_defn),
    [description('Definition of subclass'), relevance(0.9)]).
```

An example of an inferred CNF formula is:

```
cnf(175,lemma,
    ( rsymProp(ib,sk_c3) | sk_c4 = sk_c3 ),
    inference(factor_simp,[status(thm)],[
        inference(para_into,[status(thm)],[96,78,theory(equality)])]),
    [iquote('para_into,96.2.1,78.1.1,factor_simp')]).
```

Each problem file in the TPTP has a header section and a list of the annotated formulae that describe the problem. The header section contains information fields that provide context for the problem, including: the name and domain of the problem, short and long English descriptions of the problem, information about the source of the problem, the status of the problem in terms of the SZS ontology, and statistics about the problem. Each file in the TSTP has a header section and a list of the annotated formulae that describe the solution. The header section contains information fields that provide context for the solution, including: the name of the ATP system that produced the derivation, the name of the TPTP problem, the command line issued to run the ATP system, information about hardware and software resources used, the date and time the solution was produced, the result and output status in terms of the SZS ontology, and statistics about the solution.

At the time of writing this paper, the TPTP contains 11279 problems in 35 domains, and the TSTP contains the results of running 43 ATP systems and system variants on all the problems in the TPTP. The solution files are classified according to the TPTP problem domains, then by TPTP problem, and finally by the ATP systems – this information is visible in the directory hierarchy and solution file name.

2.2 About PML

PML is an interlingua for representing and sharing explanations generated by various intelligent systems such as hybrid web-based question answering systems, text analytic components, theorem provers, task processors, web services, rule engines, and machine learning components. PML is split into three modules – provenance, justification, and trust relations.

- The provenance ontology provides primitives for recording properties of entities that have been used or processed. Properties such as name, description, date and time of creation, authors, and owner, can be recorded. The IW Registry provides a public repository that allows users to register meta-data about entities.
- The justification ontology provides primitives for encoding justifications for derived conclusions. Some details are provided below.
- The trust relation ontology provides primitives for explaining belief assertions associated with information and trust assertions associated with sources.

PML classes are OWL [6] classes (they are subclasses of `owl:Class`), and PML data is therefore expressible in the RDF/XML syntax. PML is used to build OWL documents representing both proofs and proof provenance information. For this work, the representation of proofs is of primary interest. The two main constructs of proofs in PML are NodeSets and InferenceSteps. A NodeSet is used to host a set of alternative justifications for one conclusion. A NodeSet contains:

- A URI that is its unique identifier.
- The conclusion of the proof step.
- The expression language in which the conclusion is written.
- Any number of InferenceSteps, each of which represents an application of an inference rule that justifies the conclusion.

An InferenceStep contains:

- The inference rule that was applied to produce the conclusion.
- The antecedent NodeSets of the inference step.
- Bindings for variables in the inference.
- Any number of discharged assumptions.
- The original sources upon which the conclusion depends.
- The inference engine that performed the inference step.
- A time stamp recording when the inference step was performed.

A proof consists of a collection of NodeSets, with a root NodeSet as the final goal, linked recursively to its antecedent NodeSets.

3 TPTP to PML Translation

The translation of a TSTP proof into PML is done by parsing the TSTP file using the TPTP-parser[4], and extracting the necessary information into PML object instances. The proof is translated into a PML NodeSet collection, with each formula in the solution being translated as singleton member of the collection (but see Section 5 for hints about future work which will aggregate proofs, so

[4] A parser for the TPTP language written in Java by Andrei Tchaltsev at ITC-irst, available from http://www.freewebs.com/andrei_ch/

that NodeSets may have multiple elements). Additionally, the conjecture of the corresponding TPTP problem is translated into a PML Query, and the `English` header field of the problem into a PML Question. The Query contains a pointer to the Question and to all NodeSet collections (from different ATP systems) that provide a solution. The Query thus provides a starting point for accessing all the proofs for that problem.

To translate a TPTP formula into a PML NodeSet, the translator needs to determine the following:

– The language of the formula, either `fof` or `cnf`. Both `fof` and `cnf` have corresponding PML provenance elements registered in the IW registry.
– The TPTP role. This is used to help determine the inference rule of the formula.
– The logical formula. The formula text is used as the NodeSet conclusion string.
– The inference engine (ATP system) that produced the proof. The translator looks in the header of the TSTP file to find the engine name. Each engine is registered in the IW registry. For example, EP has an URI of http://inference-web.org/registry/IE/EP.owl#EP.
– The inference rule used to derive the formula. Leaves of proofs that have an `axiom` role are considered to have been derived by "direct assertion". Leaves of proofs that have an `assumption` role are considered to have been derived by "assumption". For internal nodes that have an `inference` record, the translator extracts the inference rule from the record.
– The antecedent list (for inferred formulae). The members of the parent list in the `inference` record are used to form the antecedent list of the InferenceStep.
– The external source. The source is used to form the source usage of a NodeSet's inference step to describe where the conclusion originated from.
– Date and time. The translator obtains the date and time from the header of the TSTP file, to record when the proof was created.

When all information is gathered from a TSTP formula, the translator creates a NodeSet instance, and adds it to the collection forming the proof. For example, the following node from EP 0.999's proof of `PUZ001+1` ...

```
cnf(57,plain,
    ( hates(butler,X1)
    | ~ killed(X1,agatha) ),
    inference(spm,[status(thm)],[36,45,theory(equality)])).
```

... is represented by the following PML ...

```
<rdf:RDF
    xmlns:pmlp="http://inference-web.org/2.0/pml-provenance.owl#"
    xmlns:ds="http://inference-web.org/2.0/ds.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://inference-web.org/2.0/pml-justification.owl#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#">
```

```
  <NodeSet rdf:about="http://inference-web.org/proofs/tptp/Solutions/
PUZ/PUZ001+1/EP---0.999/answer.owl#ns_57">
    <pmlp:hasCreationDateTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">
2008-05-01T17:11:39-04:00</pmlp:hasCreationDateTime>
    <hasConclusion>
      <pmlp:Information>
        <pmlp:hasRawString rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
hates(butler,X1) | ~ killed(X1,agatha)</pmlp:hasRawString>
        <pmlp:hasLanguage rdf:resource="http://inference-web.org/registry/LG/
TPTPCNF.owl#TPTPCNF"/>
        <pmlp:hasDescription>
          <pmlp:Information>
            <pmlp:hasRawString rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
cnf(57,plain,
    ( hates(butler,X1)
    | ~ killed(X1,agatha) ),
    inference(spm,[status(thm)],[36,45,theory(equality)])).
</pmlp:hasRawString>
            <pmlp:hasLanguage rdf:resource="http://inference-web.org/registry/LG/
TPTPCNF.owl#TPTPCNF"/>
            <pmlp:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
TPTP Formula</pmlp:hasName>
            <pmlp:hasURL rdf:datatype="http://www.w3.org/2001/XMLSchema#anyURI">
http://www.cs.miami.edu/~tptp/cgi-bin/DVTPTP2WWW/view_file.pl?Category=Solutions&amp;
Domain=PUZ&amp;File=PUZ001+1&amp;System=EP---0.999.THM-CRf.s#57</pmlp:hasURL>
          </pmlp:Information>
        </pmlp:hasDescription>
      </pmlp:Information>
    </hasConclusion>
    <isConsequentOf>
      <InferenceStep>
        <hasIndex rdf:datatype="http://www.w3.org/2001/XMLSchema#int">0</hasIndex>
        <fromAnswer rdf:resource="http://inference-web.org/proofs/tptp/Solutions/PUZ/
PUZ001+1/EP---0.999/answer.owl#answer"/>
        <hasInferenceRule rdf:resource="http://inference-web.org/registry/DPR/
EP0.999Spm.owl#EP0.999Spm"/>
        <hasAntecedentList>
          <NodeSetList>
            <ds:first rdf:resource="http://inference-web.org/proofs/tptp/Solutions/PUZ/
PUZ001+1/EP---0.999/answer.owl#ns_36"/>
            <ds:rest>
              <NodeSetList>
                <ds:first rdf:resource="http://inference-web.org/proofs/tptp/Solutions/PUZ/
PUZ001+1/EP---0.999/answer.owl#ns_45"/>
              </NodeSetList>
            </ds:rest>
          </NodeSetList>
        </hasAntecedentList>
        <hasInferenceEngine rdf:resource="http://inference-web.org/registry/IE/EP.owl#EP"/>
      </InferenceStep>
    </isConsequentOf>
  </NodeSet>
</rdf:RDF>
```


As an aside, the reverse translation from PML to TPTP is trivially possible for proofs translated from TPTP to PML, because the `hasConclusion` element of a NodeSet contains the original TPTP node as plain text. However, reconstruction of the TPTP node from the other NodeSet elements is not always completely possible because some minor items of information are not captured in the PML form. For example, the fact that an inference used the theory of equality, recorded by the `theory(equality)` parent of the TPTP node, is not captured in the PML form. NodeSets that come from sources other than translation from the TPTP are unlikely to be translatable to TPTP form, due to

different items of data being recorded, and different data formats being used. In particular, the PML form records the logical formula of a proof node as a text string in the `hasConclusion` element, and does not parse the formula into a representative structure. Thus if the logical formula is in a non-TPTP language, e.g., KIF [3] or DFG [4], there is no capability within IW to convert that to TPTP form.

4 Presentations

Given the PML encoded proofs from the TSTP, it becomes possible to use IW tools to process the proofs. Four examples are described in this section.

4.1 IW NodeSet Browser

The IW NodeSet browser allows the user to traverse the NodeSets of a proof. The presentation of a NodeSet provides:

- the conclusion, with a control to display its metadata (which contains provenance information);
- the antecedent formula and links to the NodeSets that justify (by inference) this conclusion;
- links to the leaf (evidence) nodes upon which this node depends;
- links to information about the ATP system and the inference rule used;
- the inferred formulae and links to the NodeSets that this conclusion is used to infer, with an option to show the sibling formulae used in each case;
- the formula and a link to the NodeSet finally concluded with the help of this conclusion;
- the query and question answered.

Figure 1 shows a NodeSet from EP's [8] proof for the TPTP problem `PUZ001+1`. The conclusion of the NodeSet is

```
hates(butler,X1) | ~ killed(X1,agatha)
```

The two justifying antecedents are

```
~ richer(X1,X2) | ~ killed(X1,X2)
hates(butler,X1) | richer(X1,agatha)
```

The single inferred formula is

```
hates(butler,esk1_0)
```

and the final conclusion is

```
$false
```

corresponding to the end of the proof by refutation. Figure 2 shows the provenance information obtained for the conclusion by expanding its `show metadata` control, and also the provenance information for EP 0.999 obtained by clicking on its link in the display.

URI: http://inference-web.org/proofs/tptp/Solutions/PUZ/PUZ001+1/EP---0.999/answer.owl#ns_57 (browse) [PML] [tabulator]

Conclusion

```
hates(butler,X1)
 | ~ killed(X1,agatha)
```

show metadata

Justified by

1. Inferred by **inference engine** EP 0.999 with **declarative rule** EP 0.999 spm from the **parents**:

   1. ```
      ~ richer(X1,X2)
       | ~ killed(X1,X2)
      ```
   2. ```
      hates(butler,X1)
       | richer(X1,agatha)
      ```

   show metadata

   Descended from the assertions: show

Used to infer

1. ```
   hates(butler,esk1_0)
   ```
   with the help of: show

Used to finally conclude

1. ```
   $false
   ```

   that answers the query:

   ```
   fof(pel55,conjecture,(
       killed(agatha,agatha) )).
   ```

   that is a formal representation of the question:

   ```
   Someone who lives in Dreadbury Mansion killed Aunt Agatha.
   Agatha, the butler, and Charles live in Dreadbury Mansion,
   and are the only people who live therein. A killer always
   hates his victim, and is never richer than his victim.
   Charles hates no one that Aunt Agatha hates. Agatha hates
   everyone except the butler. The butler hates everyone not
   richer than Aunt Agatha. The butler hates everyone Aunt
   Agatha hates. No one hates everyone. Agatha is not the
   butler. Therefore : Agatha killed herself.
   ```

Fig. 1. IW NodeSet browser presentation from EP's proof of PUZ001+1

Fig. 2. Provenance information in the IW NodeSet browser

4.2 IWBrowser

The Inference Web Browser (IWBrowser)[5] provides a graphical rendering of a PML proof, with links to the underlying provenance information stored in the PML. The presentation also provides options to focus in on the current path to the root of the proof, and to hide nodes in the proof. Figure 3 shows an extract from EP's proof for the TPTP problem PUZ001+1, including the example from Section 4.1. The various boxed links provide the access to provenance information and rendering options.

4.3 Probe-It!

Probe-It![6] [1] is a browser suited to graphically rendering PML based provenance associated with results derived from inference engines and workflows. Probe-It! consists of three primary views to accommodate the different kinds of provenance information: results, justifications, and provenance, which refer to final and intermediate data, descriptions of the generation process (i.e., execution traces) and information about the sources respectively. Figure 4 shows the Probe-It! rendering of SNARK's [9] proof for the TPTP problem GEO053-2. Each node of

[5] http://iw.stanford.edu/iwbrowser/. http://browser.inference-web.org/tptppml/ provides access to the PML translations of the TSTP files.

[6] http://trust.cs.utep.edu/probeit/

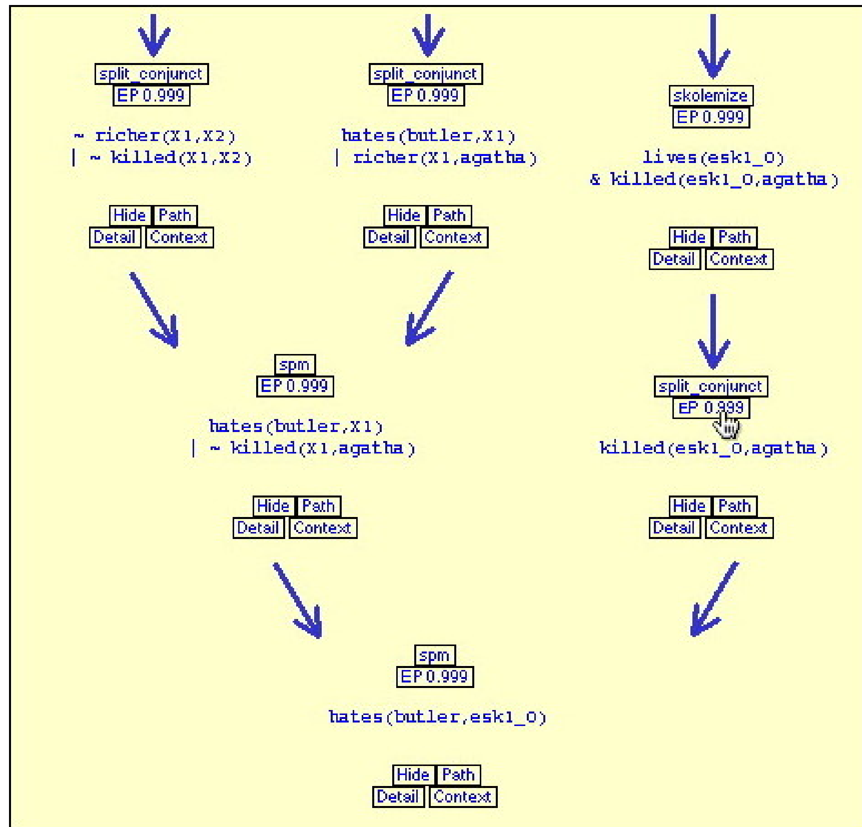

Fig. 3. IWBrowser presentation of an extract from EP's proof of PUZ001+1

the proof is drawn as a square, with orange squares being leaves of the proof and blue squares derived. Provenance information - the inference rule and ATP system name - is given in the upper pane of each square. The logical formula is given in the lower content pane of the square. The "panner" window in the lower left allows the user to move around the proof, while the zoom buttons provide more and less detailed views.

4.4 IWSearch

IWSearch[7] is a service in inference web architecture. It aims to discover, index, and search for PML objects available on the web. IWSearch consists of three groups of services: (i) the discovery services, which utilize Swoogle [2] search results and a focused crawler to discover URLs of PML documents on the web; (ii) the index services, which use an indexer to parse the submitted PML documents and prepare meta-data about PML objects for future search, and use a searcher

[7] http://onto.rpi.edu/iwsearch/

Fig. 4. Probe-It! presentation of SNARK's proof of `GEO053-2`

1 - **100** of total **456** results for **agatha** in **0.9380000233650208** seconds

label	type	more	source
agatha != butler	NodeSet	browse	http://inference-we
agatha != butler	NodeSet	browse	http://inference-we
aunt_agatha != butler	NodeSet	browse	http://inference-we
agatha != butler	NodeSet	browse	http://inference-we
! [X] : (hates(agatha,X) => hates(butler,X))	NodeSet	browse	http://inference-we
agatha != butler	NodeSet	browse	http://inference-we
! [X] : (~ richer(X,agatha) => hates(butler,X))	NodeSet	browse	http://inference-we
! [X] : (hates(agatha,X) => hates(butler,X))	NodeSet	browse	http://inference-we
! [X1] : (lives(X1) => (X1 = agatha | X1 = butler | X1 = charles))	NodeSet	browse	http://inference-we
agatha != butler	NodeSet	browse	http://inference-we
agatha = butler | hates(agatha,agatha)	NodeSet	browse	http://inference-we
agatha_hates_agatha	Document	browse	http://inference-we
! [X0] : (lives(X0) => (X0 = agatha | X0 = butler | X0 = charles))	NodeSet	browse	http://inference-we
agatha_hates_charles	Document	browse	http://inference-we
! [X0] : (~ lives(X0) | X0 = charles | X0 = butler | X0 = agatha)	NodeSet	browse	http://inference-we
butler != agatha	NodeSet	browse	http://inference-we
agatha_hates_agatha	Document	browse	http://inference-we
aunt_agatha != butler	NodeSet	browse	http://inference-we
! [X] : (hates(agatha,X) => ~ hates(charles,X))	NodeSet	browse	http://inference-we

Fig. 5. IWSearch results for the query `agatha`

to serve query calls invoked by users; (iii) the user interface services, which offer keyword search and a categorical browse interface for human or machine users. Figure 5 shows the first results returned from the query "`agatha`", after indexing the PML translations of the TSTP files. The `label` gives the raw string content of the object, the `type` is the class in the PML ontology, the `more` link provides access to that node in the IW NodeSet browser, and the `source` is the URL of the PML document containing this node.

5 Conclusion

The translation of TSTP proofs into PML, and their presentation using IW tools, changes the strict focus on logical aspects of the proof to one that encompasses proof provenance. This type of presentation is necessary for applications that demand justification or explanation of the reasoning performed. This work therefore adds value to the proofs produced by ATP systems, and makes the ATP system more suitable as tools in hybrid reasoning applications.

Work on the translation of TSTP files to PML is ongoing. Improvements and new features will be made in the near future. For example, an IW tool for combining proofs will be used to aggregate proofs from different ATP systems proofs for a given problem. This in turn will make it possible to produce new proofs with preferred characteristics, e.g., with minimal use of certain types of reasoning. The TPTP language has recently been extended to include typed higher-order logic formulae (the "THF" format), and proofs that use this language will automatically be accomodated by the translation to PML, due to the text format used for the logic formulae.

References

1. N. Del Rio and P. Pinheiro da Silva. Probe-it! Visualization Support for Provenance. In G. Bebis, R. Boyle, B. Parvin, D. Koracin, N. Paragios, T. Syeda-Mahmood, T. Ju, Z. Liu, S. Coquillart, C. Cruz-Neira, T. Muller, and T. Malzbender, editors, *Proceedings of the 2nd International Symposium on Visual Computing*, number 4842 in Lecture Notes in Computer Science, pages 732–741, 2007.
2. L. Ding, T. Finin, A. Joshi, R. Pan, R.S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle: A Search and Metadata Engine for the Semantic Web. In L. Gravano, C. Zhai, O. Herzog, and D. Evans, editors, *Proceedings of the 13th ACM Conference on Information and Knowledge Management*, pages 652–659. ACM Press, 2004.
3. M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.
4. R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.
5. D. McGuinness and P. Pinheiro da Silva. Explaining Answers from the Semantic Web: The Inference Web Approach. *Journal of Web Semantics*, 1(4):397–413, 2004.

6. D. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, 2004. World Wide Web Consortium (W3C) Recommendation.
7. P. Pinheiro da Silva, D.L. McGuinness, and R. Fikes. A Proof Markup Language for Semantic Web Services. *Information Systems*, 31(4-5):381–395, 2006.
8. S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
9. M.E. Stickel. SNARK - SRI's New Automated Reasoning Kit. http://www.ai.sri.com/ stickel/snark.html.
10. G. Sutcliffe. TPTP, TSTP, CASC, etc. In V. Diekert, M. Volkov, and A. Voronkov, editors, *Proceedings of the 2nd International Computer Science Symposium in Russia*, number 4649 in Lecture Notes in Computer Science, pages 7–23. Springer-Verlag, 2007.
11. G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.
12. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
13. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.

randoCoP: Randomizing the Proof Search Order in the Connection Calculus

Thomas Raths Jens Otten

Institut für Informatik, University of Potsdam
August-Bebel-Str. 89, 14482 Potsdam-Babelsberg, Germany
{traths,jeotten}@cs.uni-potsdam.de

Abstract. We present randoCoP, a theorem prover for classical first-order logic, which integrates randomized search techniques into the connection prover leanCoP 2.0. By randomly reordering the axioms of the problem and the literals within its clausal form, the incomplete search variants of leanCoP 2.0 can be improved significantly. We introduce details of the implementation and present comprehensive practical results by comparing the performance of randoCoP with leanCoP and other theorem provers on the TPTP library and problems involving large theories.

1 Introduction

Connection calculi, such as the connection calculus [1, 2], the connection tableau calculus [8], or the model elimination calculus [9], are in contrast to standard tableau or saturation-based calculi not proof confluent. Therefore a large amount of backtracking is required during the proof search. By restricting this backtracking the performance of connection-based proof search procedures can be improved significantly [13]. leanCoP 2.0 [15, 14] is a theorem prover for classical first-order logic based on the connection calculus. A shell script consecutively runs different variants of the core prover with different options, which control the proof search. The most successful variants use restricted backtracking.

The downside of restricted backtracking is the loss of completeness. Whereas proofs for some formulae can be found very quickly, it might be impossible to find proofs for other formulae anymore. Since restricted backtracking cuts off alternative connections, the benefit of this approach strongly depends on the proof search order. The proof search order, in turn, usually depends on the order of clauses and literals in the given formula. Whereas the proof search procedure quickly finds a proof for one order, another order of the same clauses might result in an incomplete proof search order, i.e. no proof is found at all. By reordering the clauses, the downside of restricted backtracking can be minimized.

randoCoP extends the leanCoP 2.0 implementation by repeatedly reordering the axioms and literals of a given problem at random. This increases the chance to find a proof, in particular for the incomplete prover variants. In Section 2 we present experimental results of several reordering techniques and details of the implemented reordering strategy. In Section 3 the performance of randoCoP is compared with current state-of-the-art theorem provers.

2 Randomizing the Proof Search Order

We first describe the basic motivation behind the randomized reordering technique. Afterwards we present a detailed practical analysis, which determines the specific reordering strategy that is used within randoCoP.

2.1 Motivation

leanCoP searches for connections in the order of the given input clauses. Connections to clauses at the beginning of the input clauses are considered first, before (on backtracking) clauses at the end are examined. Therefore reordering clauses is a simple way of modifying the proof search order. But the effect of reordering clauses is limited for complete search strategies (see Section 2.2), since every clause is considered sooner or later anyway.

leanCoP 2.0 has the option to restrict backtracking by cutting off alternative connections in branches that have already been closed before [13]. With this incomplete search technique some clauses might not be considered anymore and the reordering of clauses has a significant effect. It makes it possible to find proofs for problems that could not be solved before. Figure 1 illustrates this fact. The triangle represents the search space, the crosses mark the solutions, and the grey shaded area is the search space that can be traversed within a certain time limit and is roughly the same for all three triangles.

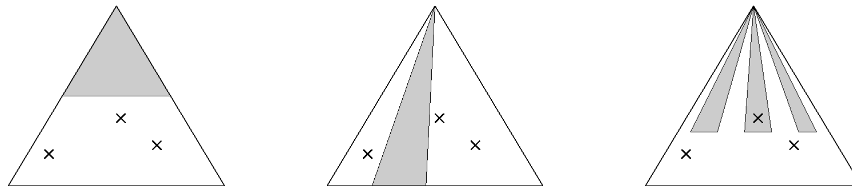

Fig. 1. Search strategies: complete, restricted backtracking without/with reordering

The complete search strategy (left hand side), does not reach the proof depth required for a solution. The search with restricted backtracking (in the middle), reaches the depth of the solutions but not the required breadth. Only the search strategy with restricted backtracking and repeatedly reordering the clauses (right hand side) is able to find a proof.

Clauses can be reordered in several ways, e.g. by rotating or shifting clauses. leanCoP 2.0 already contains an option for reordering clauses that uses a simple perfect shuffle algorithm. But the effect on the proof search is small, since the generated clause orders are not sufficiently diverse. A random reordering mixes the order of clauses more thoroughly. Therefore instead of the built-in reordering technique a randomized reordering should be used. Since the outcome of the proof search needs to be reproducible, a deterministic pseudo-random reordering needs to be implemented.

2.2 Evaluation and Analysis

Our practical experiments have shown that a reordering of the axioms of a given problem is more effective than the reordering of clauses after the clausal form translation has been applied. These experiments also showed that reordering the literals within the clauses has a positive effect on the proof search as well.

The proof search order is randomly modified in the following way. Let $A_1 \wedge A_2 \wedge \ldots A_n \wedge \Rightarrow C$ be the given formula where $A_1, A_2, \ldots, A_n$ are the axioms and C is the conjecture of the problem. Let $\pi^m : \{1, \ldots, m\} \rightarrow \{1, \ldots, m\}$ be a (pseudo-)random permutation.

1. *Reordering of axioms*: The following formula is generated: $A_{\pi^n(1)} \wedge A_{\pi^n(2)} \wedge \ldots \wedge A_{\pi^n(n)} \Rightarrow C$. Let $C_1, C_2, \ldots, C_m$ be this formula in clausal form.
2. *Reordering of literals*: For each clause $C_i{=}\{L_1, \ldots, L_k\}$ the reordered clause $C_i'{=}\{L_{\pi^k(1)}, \ldots, L_{\pi^k(k)}\}$ is generated. $C_1', \ldots, C_k'$ is the final set of clauses.

We first want to find the options of the leanCoP 2.0 core prover that are most suited for our randomized reordering approach. A *variant* of the leanCoP 2.0 Prolog core prover is determined by a set of options that control the proof search [14]. The following options can be used:

1. `nodef`/`def`: The standard/definitional translation into clausal form is done. If none of these options is given the standard translation is used for the axioms whereas the definitional translation is used for the conjecture [13].
2. `conj`: The conjecture clauses are used as start clauses. If this option is not given the positive clauses are used as start clauses.
3. `scut`: Backtracking is restricted for alternative start clauses [13].
4. `cut`: Backtracking is restricted for alternative reduction/extension steps [13].
5. `comp(I)`: Restricted backtracking is switched off when iterative deepening exceeds the active path length `I`.

The option `conj` is complete only for formulae with a provable conjecture, and the options `scut` and `cut` are only complete if used in combination with the option `comp(I)`.

The following tests were performed on all 3644 non-clausal problems (FOF division) of the TPTP library [18] version 3.3.0. The left side of Table 1 shows the number of proved problems without reordering using a time limit of 180 seconds. The right side displays the number of proved problems with repeated reordering. The allotted time limit for each order is 3 seconds with an overall time limit of 180 seconds allowing around 60 reorderings for each problem.

The prover variants using restricted backtracking (options `cut` and `scut`) benefit most from the randomized reordering technique. The greatest advance is made for the variant using the default clausal form translation and both of the options `cut` and `scut`. The variants using a full definitional (`def`) or the standard (`nodef`) clausal form translation and no restricted backtracking show a lower performance. The most successful variants using reordering use the options {`cut`}, {`cut`,`scut`}, {`conj`,`cut`}, and {`conj`,`cut`,`scut`}.

Table 1. Results without and with repeated reordering for different prover variants

options	-	cut	cut,scut
-	1217	1346	1216
def	1111	1321	1204
nodef	1166	1257	1152
conj	1208	1398	1336
conj,def	1208	1407	1349
conj,nodef	1227	1319	1058

options	-	cut	cut,scut
-	1342	1691	1684
def	1044	1402	1428
nodef	1077	1432	1436
conj	1422	1658	1597
conj,def	1095	1452	1370
conj,nodef	1152	1365	1239

We have tested the most successful variants again using different time limits for each order of the axioms/literals. Table 2 shows the results using an overall time limit of 180 seconds and a time limit of 2 seconds, 3 seconds and 4 seconds for proving each order allowing around 90, 60 and 45 reorderings, respectively. The difference in the number of proved problems is rather small with a slight advantage for a time limit of 3 seconds.

Table 2. Results of different time limits per axiom/literal order

options \ time	$2s$	$3s$	$4s$
cut	1678	1691	1688
cut,scut	1670	1684	1650
conj,cut	1636	1658	1665
conj,cut,scut	1569	1597	1607

Further evaluations have shown that the most number of problems are proved by repeatedly running the two variants {cut,scut} and {def,conj,cut} each for a time limit of 3 seconds and 2 seconds on each axiom/literal order, respectively.

In order to prove and refute a large number of problems within the first few seconds, the proof process is started by running the most successful complete prover variant using the options cut and comp(7) once for 5 seconds. A complete prover variant using the option def only is invoked at the end of the proof process for at least 10 seconds.

2.3 The Implementation

randoCoP uses the random library of ECLiPSe Prolog, which contains the predicate rand_perm/2 that randomly permutes a list, and randomise/1 to seed and initialize the random generator, which allows the same sequence of permutations to be generated, and thus makes the result reproducible.

randoCoP consists of a shell script that calls the leanCoP 2.0 core prover extended by a few predicates realizing the reordering of axioms and literals. This shell script is called with an argument that determines the overall time limit and invokes the prover variants according to Section 2.2.

Let $TotalTime$ be this given time limit in seconds. Then the shell script invokes the following variants of the core prover in the following order:

1. Prover variant {`cut`,`comp(7)`} for 5 seconds
2. Repeated reordering step, $(TotalTime - 15)/5$ times:
   (a) Reordering of axioms and literals
   (b) Prover variant {`cut`,`scut`} for 3 seconds
   (c) Prover variant {`def`,`conj`,`cut`} for 2 seconds
3. Prover variant {`def`} for 10 seconds

If, for example, $TotalTime$ is set to 600 seconds, then (600-15)/5=117 reordering steps are done and after each reordering the two prover variants are run for 3 seconds and 2 seconds, respectively.

3 Performance

To evaluate the performance we tested randoCoP on all non-clausal problems in the TPTP library, and the problems of the MPTP challenge. All tests were done on a system with a 3 GHz Xeon processor and 4 Gbyte of memory running Linux and ECLiPSe Prolog 5.8. We compare the performance of randoCoP with the current state-of-the-art provers.

3.1 The TPTP Library

For the tests all 3644 problems of version 3.3.0 of the TPTP library [18] that are in non-clausal form (FOF division) are considered. In order to solve satisfiable or unsatisfiable problems — i.e., those problems without a conjecture — these problems are negated. Equality is dealt with by adding the equality axioms. The time limit for all problems is 600 seconds.

In Table 3 the performance of randoCoP is compared with the following theorem provers: Otter 3.3 [10], version "20070805r009" of SNARK [19], leanCoP 2.0 [14], version "Dec-2007" of Prover9[1] [11], iProver 0.2 [5], Equinox 1.2 [3], SPASS 3.0 [23], E 0.999 [17], and Vampire 9.0 [16]. The rating and the percentage of proved problems for some rating intervals are given. FNE, FEQ and PEQ are problems without, with and containing only equality, respectively. Furthermore, the number of proved problems for each domain (see [18]) that contains at least 10 problems is shown.

randoCoP proves more problems than Equinox, iProver, Prover9, leanCoP 2.0, SNARK and Otter. It solves more problems in the SEU domain than any other prover. The SEU domain contains problems from set theory taken from the MPTP [20] (see also Section 3.2). Including the equality axioms, these problems contain up to 1100 axioms, of which not all are required to prove the conjecture. For the SEU domain randoCoP also shows the biggest improvement compared to leanCoP 2.0, with 491 proved problems compared to 296 problems proved by leanCoP 2.0. A notable improvement is also made for the domains NUM and SWC. 96 of all proved problems have the highest rating of 1.0.

[1] The most recent version "2008-04A" of Prover9 has a significant lower performance.

Table 3. Benchmark results on the TPTP v3.3.0 library

	Otter 3.3	SNARK 08/07	leanCoP 2.0	Prover9 12/07	iProver 0.2	Equinox 1.2	randoCoP 1.0	SPASS 3.0	E 0.999	Vampire 9.0
proved	1310	1565	1638	1677	1858	1876	**1879**	2127	2250	2377
[%]	36%	43%	45%	46%	51%	52%	**52%**	58%	62%	66%
0s to 1s	987	1259	1124	1281	1224	1376	**1161**	1627	1760	1530
1s to 10s	183	130	123	197	370	239	**229**	248	229	283
10s to 100s	106	117	193	141	179	151	**375**	170	192	394
100s to 600s	34	59	198	58	85	110	**114**	82	69	170
rating 0.0	455	435	450	464	473	500	**452**	500	503	488
rating >0.0	855	1130	1188	1213	1385	1376	**1427**	1627	1747	1889
rating 1.0	7	8	13	8	9	18	**96**	19	12	30
0.00...0.24	72%	72%	72%	73%	76%	76%	**73%**	78%	79%	78%
0.25...0.49	40%	62%	48%	70%	73%	72%	**52%**	84%	85%	86%
0.50...0.74	3%	20%	36%	28%	43%	40%	**43%**	58%	71%	80%
0.75...1.00	1%	2%	11%	3%	7%	9%	**27%**	15%	19%	27%
FNE	476	437	492	526	562	541	**496**	555	561	566
FEQ	834	1128	1146	1151	1296	1335	**1383**	1572	1689	1811
PEQ	47	83	30	71	76	196	**31**	191	165	133
AGT	16	17	29	17	19	10	**32**	18	19	52
ALG	60	84	32	83	80	186	**38**	196	162	130
CSR	3	16	2	27	10	15	**3**	2	25	27
GEO	160	121	171	171	172	167	**169**	168	174	177
GRA	5	9	6	1	8	9	**6**	15	15	19
KRS	106	110	105	103	112	111	**105**	107	112	112
LCL	18	56	24	48	33	10	**25**	51	78	101
MED	5	2	7	1	9	5	**5**	9	8	9
MGT	54	58	45	62	65	67	**50**	56	67	67
NLP	6	11	13	11	22	22	**15**	22	22	22
NUM	31	45	70	49	43	45	**87**	48	62	105
PUZ	6	6	6	6	6	6	**6**	6	6	6
SET	229	262	339	276	316	245	**335**	290	339	369
SEU	149	229	296	259	291	305	**491**	353	316	347
SWC	84	131	87	101	170	175	**99**	256	326	297
SWV	157	181	177	183	210	221	**181**	235	225	236
SYN	210	214	217	267	277	265	**218**	279	276	281
refuted	0	113	33	0	329	0	**33**	362	366	0
time out	700	1596	1949	668	1455	1395	**1732**	1125	1023	985
gave up	1181	213	0	320	0	373	**0**	0	0	280
errors	453	157	24	979	2	0	**0**	30	5	2

3.2 The MPTP Challenge

The MPTP challenge is a set of problems from the Mizar library translated into first-order logic [20]. There are two divisions, bushy and chainy, each containing 252 problems. Whereas the bushy division contains only the relevant axioms and lemmata required to prove the main theorem, the chainy division contains all axioms and lemmata that were available at the time of proving the main theorem of the challenge. The time limit for each problem is 300 seconds.

The result of randoCoP on the bushy and chainy division is shown in Table 4 and Table 5, respectively. Again, equality axioms are added, which results in formulae with a total of up to 1700 axioms. The performance is compared with the theorem provers mentioned in Section 3.1.

Table 4. Benchmark results for the bushy division of the MPTP challenge

	Otter 3.3	Prover9 12/07	SNARK 08/07	leanCoP 2.0	iProver 0.2	Equinox 1.2	E 0.999	SPASS 3.0	Vampire 9.0	randoCoP 1.0
proved	68	119	122	128	128	131	141	160	166	**189**
[%]	27%	47%	48%	51%	51%	52%	56%	64%	66%	**75%**
0s to 1s	56	86	91	93	71	86	120	121	104	**93**
1s to 10s	4	19	17	7	33	23	12	16	22	**23**
10s to 100s	7	9	12	20	20	17	8	17	31	**60**
100s to 300s	1	5	2	8	4	5	1	6	9	**13**
time out	46	82	130	124	124	121	111	90	86	**63**
gave up	137	3	0	0	0	0	0	0	0	**0**
errors	1	48	0	0	0	0	0	2	0	**0**

Table 5. Benchmark results for the chainy division of the MPTP challenge

	Otter 3.3	Prover9 12/07	SNARK 08/07	Equinox 1.2	iProver 0.2	Vampire 9.0	SPASS 3.0	leanCoP 2.0	E 0.999	randoCoP 1.0
proved	29	52	58	79	79	81	82	88	91	**128**
[%]	12%	21%	23%	31%	31%	32%	33%	35%	36%	**51%**
0s to 1s	17	33	22	41	29	29	45	38	47	**38**
1s to 10s	7	8	14	14	37	31	18	21	18	**29**
10s to 100s	5	6	14	17	10	14	15	23	20	**34**
100s to 300s	0	5	8	7	3	7	4	6	6	**27**
time out	150	101	194	173	173	171	170	164	161	**124**
gave up	73	0	0	0	0	0	0	0	0	**0**
errors	0	99	0	0	0	0	0	0	0	**0**

randoCoP shows a decent performance in both division. The time complexity is better compared to the other provers, as many problems are (still) proved after 10 seconds. We have not tested the provers MaLARea 0.1, SRASS 0.1, and Fampire 1.3, which prove 187/142, 171/127, and 191/126 of the problems in the bushy/chainy division, respectively (according to the MPTP challenge web site).

4 Conclusion and Related Work

We have presented randoCoP, a theorem prover for classical first-order logic, which integrates a random proof search strategy into the connection prover leanCoP 2.0. Repeatedly reordering the axioms of the problem and the literals within its clausal form improves performance of leanCoP 2.0 significantly.

Some incomplete strategies of leanCoP 2.0 effectively restrict backtracking and increase the depth of the search space that can be investigated within a certain amount of time. But they might cut off specific proof search orders required to find a proof. randoCoP partly compensates for this disadvantage and the loss of completeness by increasing again the breadth of the explored search space. The combination of restricted backtracking and randomized reordering is highly effective, in particular for hard problems containing many axioms.

The core prover of randoCoP and leanCoP 2.0 consists only of a few lines of Prolog code. This indicates that tens to hundreds of thousands of lines in, e.g., C and low-level optimizations are not needed to succeed in automated theorem proving. Instead it shows that good heuristics for traversing the vast search are important in automated reasoning research.

The RCTHEO system [4] randomly reorders clause instances. It is a an OR-parallel version of SETHEO [7], where each node executes one instance of the sequential prover SETHEO. The performance is similar to PARTHEO, a parallel version of SETHEO.

The SETHEO system offers a dynamic subgoal reordering option. The reordering is not at random but prefers subgoals with the highest probability to fail, in order to reduce the search space. Syntactic criteria, such as the number of variables, are used to determine the specific order. The subgoal order is then determined dynamically whenever the next subgoal is selected.

We have tested SETHEO without and with subgoal reordering on all non-clausal problems of the TPTP library (see Section 3.1). *Without* subgoal reordering SETHEO proves 1192 out of the 3644 problems. *With* subgoal reordering (using the option -dynsgreord 2) it only solves 1185 problems, i.e. the performance does not really improve. This confirms our own testing with reordering on several variants of the leanCoP 2.0 core prover (see Section 2.2). The effect of reordering axioms (or clauses) and literals is limited when a complete search in connection calculi is done. In this case the performance can even get worse. On the other hand the incomplete variants of the leanCoP 2.0 core prover benefit significantly from the randomized reordering technique.

Further research includes the adaption of randoCoP to other (non-classical) logics — such as intuitionistic logic [12, 14] or modal logic [6] — for which matrix characterisations exist (see also [21, 22]). It is also worth investigating approaches that randomize, e.g., the order of the conjecture clauses, or dynamically reorders clauses and/or literals *during* the proof search. And finally we plan to (slightly) extend the leanCoP 2.0 core prover so that a compact connection proof is returned. A readable proof is then output by a separate prover component.

The source code of randoCoP can be obtained at the leanCoP website at `http://www.leancop.de`.

Acknowledgements. The authors would like to thank the referees for their useful comments, which have helped to improve this paper.

References

1. W. Bibel. Matings in matrices. *Communications of the ACM*, 26:844–852, 1983.
2. W. Bibel. *Automated Theorem Proving*. Vieweg, second edition, 1987.
3. K. Classen. Equinox, a new theorem prover for full first-order logic with equality. In *Dagstuhl Seminar 05431 on Deduction and Applications*, 2005.
4. W. Ertel. OR-parallel theorem proving with random competition. In A. Voronkov, Ed., *LPAR'92*, LNAI 624, pp. 226–237. Springer, 1992.
5. K. Korovin. Implementing an instantiation-based theorem prover for first-order logic. In C. Benzmueller, B. Fischer, G. Sutcliffe *IWIL-6*, 2006.
6. C. Kreitz, J. Otten. Connection-based theorem proving in classical and non-classical logics. *Journal of Universal Computer Science*, 5:88–112, Springer, 1999.
7. R. Letz, J. Schumann, S. Bayerl, W. Bibel. SETHEO: a high-performance theorem prover. *Journal of Automated Reasoning*, 8:183–212, 1992.
8. R. Letz, G. Stenz Model elimination and connection tableau procedures. *Handbook of Automated Reasoning*, pp. 2015–2114, Elsevier, 2001.
9. D. Loveland. Mechanical theorem proving by model elimination. *JACM*, 15:236–251, 1968.
10. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94/6, Argonne National Laboratory, 1994.
11. W. McCune. Release of Prover9. *Mile High Conference on Quasigroups, Loops and Nonassociative Systems*, Denver, Colorado, 2005.
12. J. Otten. Clausal connection-based theorem proving in intuitionistic first-order logic. *TABLEAUX 2005*, LNAI 3702, pages 245–261, Springer, 2005.
13. J. Otten. Restricting backtracking in connection calculi. Technical report, Institut für Informatik, University of Potsdam, 2008.
14. J. Otten. leanCoP 2.0 and ileanCoP 1.2: high performance lean theorem proving in classical and intuitionistic logic. *IJCAR 2008*. LNCS, Springer, 2008.
15. J. Otten, W. Bibel. leanCoP: lean connection-based theorem proving. *Journal of Symbolic Computation*, 36:139–161, 2003.
16. A. Riazanov, A. Voronkov. The design and implementation of Vampire. *AI Communications* 15(2-3): 91–110, 2002.
17. S. Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2):111–126, 2002.
18. G. Sutcliffe, C. Suttner. The TPTP problem library - CNF release v1.2.1. *Journal of Automated Reasoning*, 21: 177–203, 1998.
19. M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, Ed., *CADE-12*, LNCS 814, pp. 341–355, Springer, 1994.
20. J. Urban. MPTP 0.2: design, implementation, and initial experiments. *Journal of Automated Reasoning*, 37:21–43, 2006.
21. A. Waaler. Connections in nonclassical logics. *Handbook of Automated Reasoning*, pp. 1487–1578, Elsevier, 2001.
22. L. Wallen. *Automated deduction in nonclassical logic*. MIT Press, 1990.
23. C. Weidenbach, R. A. Schmidt, T. Hillenbrand, R. Rusev, D. Topic. System description: SPASS version 3.0. In F. Pfenning, Ed., *CADE-21*, LNCS 4603, pp. 514–520. Springer, 2007.

Integration of the TPTPWorld into SigmaKEE

Steven Trac[1], Geoff Sutcliffe[1], and Adam Pease[2]

[1]University of Miami, USA [2]Articulate Software, USA

Abstract. This paper describes the integration of the ATP support of the TPTPWorld into the Sigma Knowledge Engineering Environment. The result is an interactive knowledge based reasoning environment, with strong knowledge management features, and access to modern state of the art ATP systems for reasoning over knowledge bases.

1 Introduction

The Knowledge Based Reasoning (KBR) community within the field of Artificial Intelligence has long conducted logical reasoning for decision support, planning and many similar applications [8]. Much of this has been done in the context of specialized logics and knowledge representation schemes, e.g., [10, 6]. The Automated Theorem Proving (ATP) community has grown more out of mathematical disciplines, and its applications have tended to be in that realm using classical first-order logic, e.g., [4, 1]. While the various uses of SNARK [20], e.g., [21, 31], are notable exceptions, and several reasoning frameworks have been designed and implemented with a focus on large KBR, e.g., [17, 11, 29], there has not been a lot of use of general purpose classical ATP in KBR. This work brings together the KBR tool SigmaKEE and the ATP support of the TPTPWorld. The Sigma Knowledge Engineering Environment (SigmaKEE) [14] provides a mature platform for browsing and querying a knowledge base, often the Suggested Upper Merged Ontology (SUMO) [13]. The TPTPWorld provides well established standards, systems, and tools for first-order reasoning, stemming from the Thousands of Problems for Theorem Provers (TPTP) problem library [25]. While SigmaKEE has strong knowledge management features, it lacks the reasoning capabilities found in state of the art ATP systems. Conversely, while modern ATP systems are capable of proving hard theorems, they have limited features for interfacing with users of large knowledge bases. The integration of the TPTPWorld into SigmaKEE forms an interactive KBR environment, with strong knowledge management features, and access to modern state of the art ATP systems for reasoning over knowledge bases.

This paper is organized as follows: Sections 2 and 3 provide the necessary background about SigmaKEE and the TPTPWorld. Section 4 describes their integration, including extensions added to meet the needs of SigmaKEE users. Section 5 shows a sample use of the integrated system.

2 SigmaKEE

The Sigma Knowledge Engineering Environment (SigmaKEE) is a KBR environment for developing and using logical theories. It was created to support the Suggested Upper Merged Ontology (SUMO), which is written in a variant of the Knowledge Interchange Format (KIF) language [7] called Standard Upper Ontology Knowledge Interchange Format (SUO-KIF) [14]. SigmaKEE runs as an Apache Tomcat service, providing a browser interface to users. The main components are written in Java, and the user interface is generated by JSP. Users can upload a knowledge base for browsing and querying. An uploaded knowledge base is indexed for high performance browsing and searching. For ontology-like knowledge bases, which have a tree structure, a graph browser is provided. Figure 1 shows the graph browser interface for the top layers of the SUMO. Results from queries are presented in a hyperlinked KIF format that provides linkages back into the knowledge base, as shown in the example in Section 5.

The existing version of SigmaKEE includes a customized version of Vampire [18]. Among state of the art first order logical theorem provers available at the time of SigmaKEE's original development, only this version of Vampire, which is now 5 years old, had all the features required for theorem proving applications in SigmaKEE:

- The ability to extract an answer to a query as bindings of outermost existentially quantified variables in a conjecture.
- The ability to generate a detailed proof that explains how an answer to a query was derived.
- The ability to ask successive queries without reloading the knowledge base.
- The ability to perform basic arithmetic.

In addition, that version of Vampire was released under an approved open source license, and could therefore be tightly integrated with the open source SigmaKEE system.

3 TPTPWorld

The TPTPWorld is a package of TPTP data and software, including the TPTP problem library, a selection of ATP systems, and a suite of tools for processing TPTP format data. Although the TPTPWorld was developed and is primarily used for inhouse maintenance of the TPTP problem library, various components have become publically available and used in applications, e.g., [22, 28].

One of the keys to the success of the TPTP and related projects is their consistent use of the TPTP language [23]. The TPTP language was designed to be suitable for writing both ATP problems and ATP solutions, to be flexible and extensible, and easily processed by both humans and computers. The TPTP language BNF is easy to translate into parser-generator (`lex/yacc`, `antlr`, etc.) input [30]. The SZS ontology [26] provides a fine grained ontology of result and output forms for ATP systems, so that their results can be precisely understood

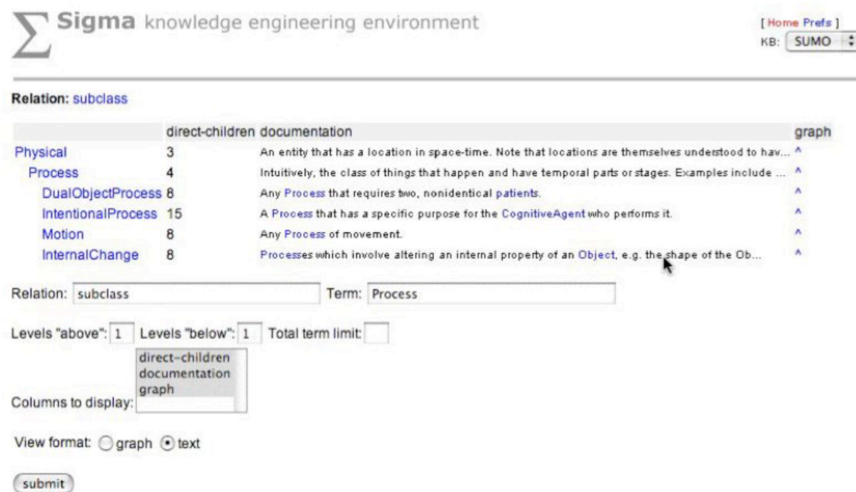

Fig. 1. SigmaKee Graph Browser

when used as input to other tools. The ontology also recommends the way that ontology values should be reported in the output from systems and tools. Figure 2 shows an extract from the top of the result ontology (the full ontology is available as part of the TPTP distribution).

The SystemOnTPTP utility is a harness that allows a problem written in the TPTP language to be easily and quickly submitted to a range of ATP systems and other tools. The implementation of SystemOnTPTP uses several subsidiary tools to prepare the input for the ATP system, control the execution of the chosen ATP system, and postprocess the output to produce an SZS result value and a TPTP format derivation. SystemOnTPTP runs in a UNIX environment, and is also available as an online service via `http POST` requests.[1]

The Interactive Derivation Viewer (IDV) [27] is a tool for graphical rendering and interaction with TPTP format derivations. The rendering uses shape and color to provide visual information about a derivation. The user can interact with the rendering in various ways – zooming, hiding, and displaying parts of the DAG according to various criteria, access to verification of the derivation, and an ability to provide a synopsis of a derivation by identifying interesting lemmas using AGInT [16]. Figure 3 shows the renderings of the derivation and synopsis for the proof output by EP [19] for the TPTP problem `PUZ001+1`.

The One Answer Extraction System (OAESys) and Multiple ANSwer EXtraction framework (MANSEX) are tools for question answering using ATP. Their development was motivated by the limited availability of modern ATP systems that are able to return answers – examples of systems that do are Otter [12], the

1 Hosted at the University of Miami. A browser interface to the service is available at `http://www.tptp.org/cgi-bin/SystemOnTPTP`.

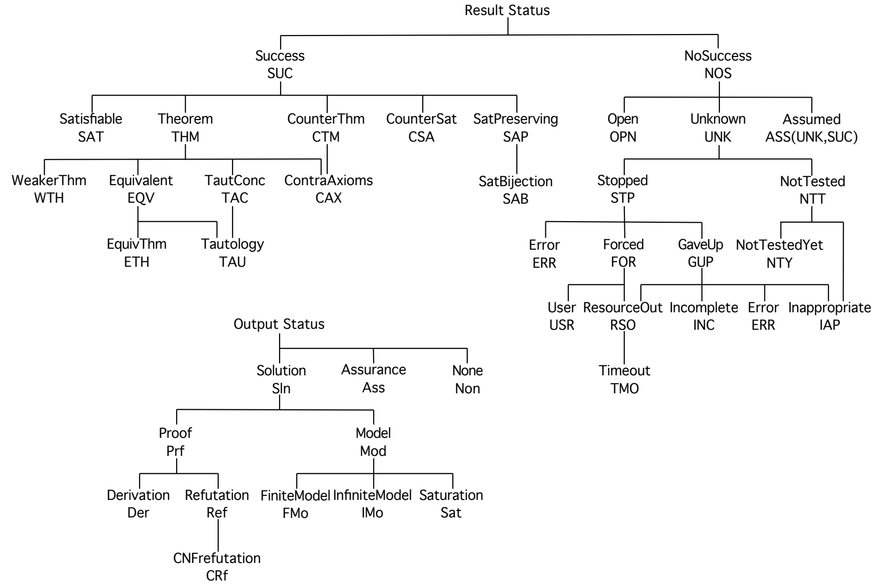

Fig. 2. SZS Ontology

customized version of Vampire used in SigmaKEE, and SNARK.[2] OAESys is a tool for extracting the bindings for outermost existentially quantified variables of a conjecture, from a TPTP format proof of the conjecture. This is done by reproving the conjecture using the Metis system [9], from only the axioms used in the original proof. The variable bindings that Metis reports for each inference step of its proof are analyzed to extract the required bindings (Metis is the only system that we know of that outputs TPTP format proofs and variable bindings for each inference step). The restriction to the axioms used in the original proof aims to make it unlikely for Metis to find a proof with different variable bindings from the original proof. If the axioms used are a subset of the axioms that were originally available, the problem given to Metis could be significantly easier than the original problem. MANSEX is a framework for interpreting a conjecture with outermost existentially quantified variables as a question, and extracting multiple answers to the question by repetitive calls to a system that can report the bindings for the variables in one proof of the conjecture.[3] Suitable systems for reporting bindings are an ATP system that outputs answers, e.g., SNARK, or a combination of an ATP system that outputs TPTP format proofs, e.g., EP, with OAESys. At each iteration of MANSEX, the conjecture is augmented by conjoining either inequalities or negative atoms that deny previously extracted answers. The ATP system is then called again to find a proof of the modified

[2] There is scant hope that more ATP system developers will add question answering to their systems without significant financial or other inducement.

[3] **Acknowledgement:** The original multiple answer extraction system was developed outside the SigmaKEE project by Aparna Yerikalapudi, at the University of Miami.

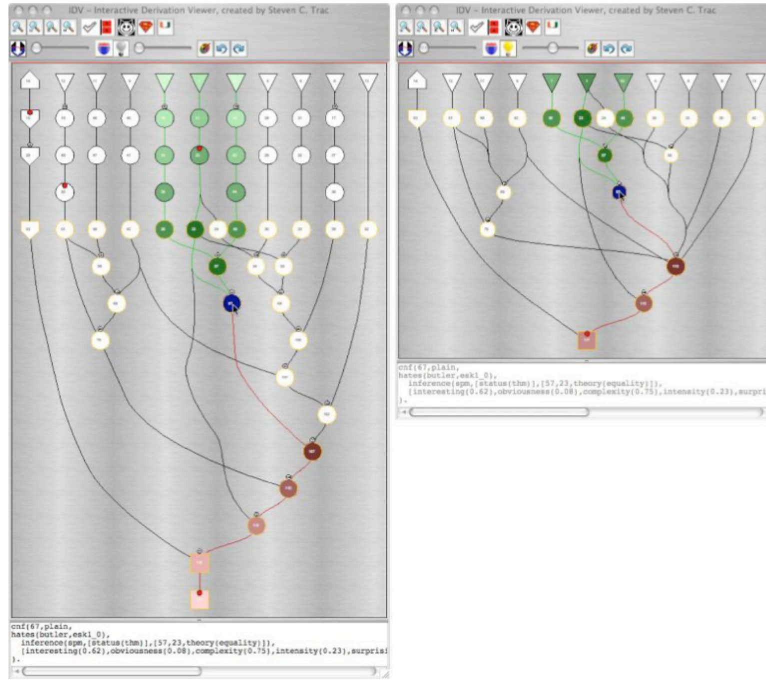

Fig. 3. EP's Proof by Refutation of PUZ001+1

conjecture. In the SigmaKEE context the process has been extended to hide the conjecture modifications from the user - details are provided in Section 4. OAESys and MANSEX both use the proposed TPTP standards for question answering in ATP [24], and SNARK has also been adpated by its developer to use those standards.

The TPTP-parser is a highly reusable Java parser for TPTP data, built using the `antlr` parser-generator.[4] The parser can easily be used without modifications in practically any application. This universality is achieved by isolating the parser code by an interface layer that allows creation of and access to abstract syntax representations of various TPTP elements. A simple but reasonably efficient default implementation of the interface layer is provided with the package.

4 Integration of the TPTPWorld into SigmaKEE

The integration of the TPTPWorld provides SigmaKEE with new capabilities:

- Internal support for TPTP format problems and derivations, using a SUO-KIF to TPTP translation and the TPTP-parser.
- Access to ATP systems for reasoning tasks, using SystemOnTPTP.

[4] **Acknowledgement:** The TPTP-parser was written primarily by Andrei Tchaltsev at ITC-irst. It is available from `http://www.freewebs.com/andrei_ch/`

– Question answering using OAESys, with the ability to provide multiple answers through use of the MANSEX framework.
– Presentation of TPTP format proofs using IDV, or using the existing hyperlinked KIF format extended with SZS ontology status values.
– An extended browser interface for access to these capabilities.

The integration has been implemented by adding external TPTPWorld tools to the SigmaKEE distribution, and embedding Java implementations of TPTPWorld tools directly into SigmaKEE. Figure 4 shows the relevant system components and architecture.

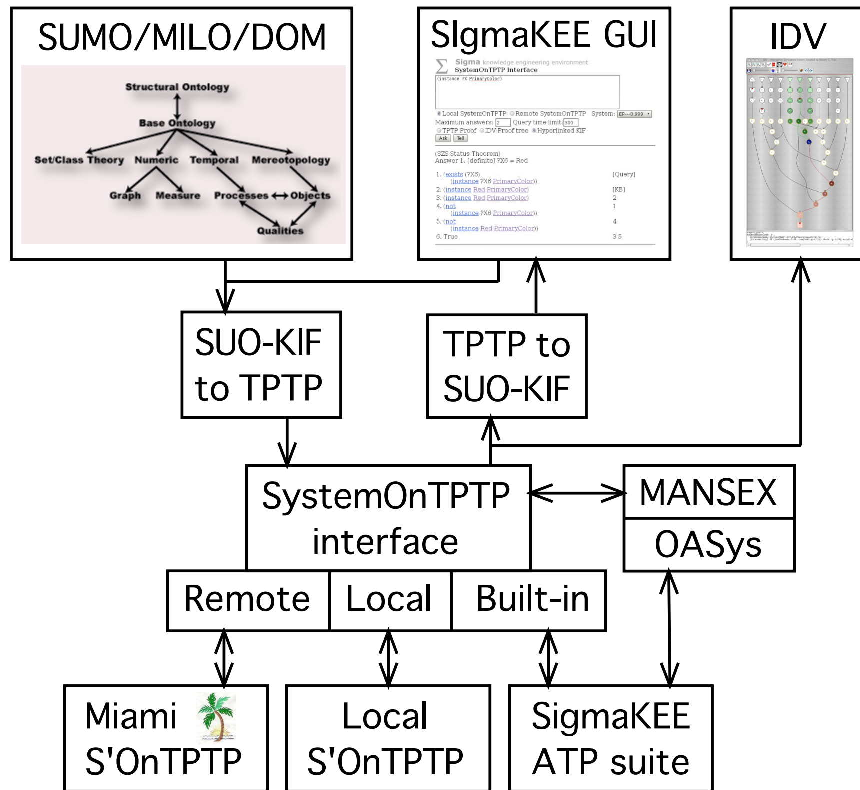

Fig. 4. Architecture of the Integrated Components

SigmaKEE was developed to support knowledge bases written in SUO-KIF, e.g., SUMO. In order to make a large suite of ATP systems available for reasoning over such knowledge bases through use of the SystemOnTPTP utility, knowledge bases are translated to the TPTP language when they are loaded. While much of the translation is syntactic, there are some constructs in SUMO that require

special processing [15]. These include use of sort signatures, sequence variables, variable predicates and functions, and embedded (higher-order) formulae.

Once a knowledge base has been loaded into SigmaKEE, queries can be submitted. A query is translated to a TPTP format conjecture, and the previously translated knowledge base provides the axioms. These are submitted to an ATP system through the SystemOnTPTP utility. Queries with outermost existentially quantified variables are treated as questions whose answers are the values bound to those variables in proofs. Three versions of SystemOnTPTP are available: remote access to the online SystemOnTPTP service via `http POST` requests, execution of a locally installed SystemOnTPTP, and a limited internal implementation of SystemOnTPTP. The ATP systems supported by the internal implementation are required to be TPTP-compliant in terms of both input and output, and have licensing that allows them to be added to SigmaKEE. At this stage E/EP, Metis, SNARK, and Paradox [5] are being used.

The advantage of using the local installation or internal implementation of SystemOnTPTP is that they do not rely on an online connection to the remote server. The advantage of the internal implementation is that it is portable to operating systems that do not have the UNIX environment required for SystemOnTPTP, e.g., Windows XP. The user chooses whether to use the remote SystemOnTPTP or a local one, and which ATP system to use. In the remote case the online system is queried to get a list of the available ATP systems. In the local case the ATP systems available in the local SystemOnTPTP installation (if any) and the internal implementation are available. If an ATP system is supported by the internal implementation and is also available through a local SystemOnTPTP installation, the internally supported one is used.

Answers to "question" conjectures are extracted from proofs using an embedding of OAESys into SigmaKEE. As Metis is one of the internally supported systems, it is available for use in OAESys. When the user requests more than one answer, an embedding of the MANSEX framework is used. In the SigmaKEE context the MANSEX process has been extended to displace the conjecture modifications from the user. This extension is done for the second and subsequent proofs found, as follows. After each answer has been extracted by OAESys, the existentially quantified variables in the original conjecture, i.e., the conjecture without the augmentations, are instantiated with the answer values. This instantiated conjecture and just the axioms used in the proof found by the chosen ATP system are passed to that ATP system. This additional ATP system run finds a proof of the (instantiated form of the) original conjecture, rather than of the augmented conjecture. Using MANSEX to get multiple answers is somewhat different, and can produce different answers, from using the customized version of Vampire mentioned in Section 2. MANSEX with OAESys requires multiple ATP system runs: two for the first answer (one to get a proof using the chosen ATP system and another to Metis within OAESys), and three for each successive answer (additionally the final call to the chosen ATP system). In contrast, the customized Vampire backtracks in its proof space to find multiple answers. As a side-effect, the customized Vampire can return the same answer multiple

times if there are multiple proofs that produce the same variable bindings, while MANSEX does not.

The integration of the TPTPWorld provides SigmaKEE with three options for displaying results: TPTP format derivations in plain text format, IDV for displaying TPTP format derivations graphically, and the hyperlinked KIF format. IDV has been embedded into SigmaKEE so that it is directly available. The hyperlinked KIF format has been implemented by translation of TPTP format derivations into SUO-KIF using an augmentation of the TPTP-parser code, and then calling SigmaKEE's hyperlinked KIF presentation feature. The hyperlinked KIF format has been mildly extended to provide more precise information about the formulae in a proof, and to provide SZS status values. An example with a hyperlinked KIF format proof is given in Section 5.

The top part of Figure 5 shows the GUI interface. The interface allows the user to submit a query or to add to the current knowledge base. The interface has the following components (top to bottom, left to right):

- Formula text box - The query or additions are put into this text box in SUO-KIF format.
- Local or Remote SystemOnTPTP, System - Choose which SystemOnTPTP to use, and which ATP system.
- Maximum answers - Desired number of answers for the query.
- Query time limit - CPU time limit for the query.
- Output format - TPTP, IDV, or hyperlinked KIF
- Ask button - Execute the ATP system on the query.
- Tell button - Add the data to the knowledge base.

5 Sample Use

As an example, EP's proof of the following SUO-KIF format query to the SUMO knowledge base is considered: `(instance ?X PrimaryColor)`. The query asks for an instance of a primary color in the SUMO knowledge base. In SUMO the following are considered primary colors: Black, Blue, Red, White, and Yellow. The query was run using the internal implementation of SystemOnTPTP, asking for two answers, with a CPU limit of 300s, and hyperlinked KIF output. Figure 5 shows the result.

EP returns the first proof shown in the output, with SZS status Theorem. OAESys is used to extract the first answer - Red. The proof and answer are translated to the hyperlinked KIF format by SigmaKEE. MANSEX then augments the query to deny the answer Red, and EP returns another TPTP proof behind the scenes. OAESys is used to extract the second answer - Blue, which is used to instantiate the existentially quantified variable of the conjecture. EP returns the second proof shown in the output. The left column of the hyperlinked KIF is labeled SUO-KIF format formulae, with embedded HTML hyperlinks back to terms in the SUMO knowledge base. The right column describes the source of the formula: the parent formulae, the knowledge base (KB), or the query.

Fig. 5. Sample hyperlinked KIF format proofs

6 Conclusion

While KBR and ATP are both mature fields, there has not been significant cross-fertilization between the two communities. Both communities would benefit from a greater degree of interaction. The integration of the TPTPWorld into Sigma-KEE brings together tools that support both communities, which should make collaboration easier, and drive further cross-disciplinary research. The use of the remote SystemOnTPTP illustrates how KBR and other application systems can absolve themselves of the responsibility of maintaining up-to-date ATP systems in-house, and instead use an online service for discharging reasoning requests. The experience with SigmaKEE can be leveraged by others to this end.

Future work includes translation of SUMO and other knowledge bases to the new typed higher-order format (THF) of the TPTP language [3], and use of higher-order ATP systems such as LEO II [2] to answer higher-order queries over the knowledge bases.

Acknowledgement: Thanks to Nick Siegal for his technical advice and contributions to the development of this software.

References

1. ANL. A Summary of New Results in Mathematics Obtained with Argonne's Automated Deduction Software. http://www-unix.mcs.anl.gov/AR/new_results/, 1995.
2. C. Benzmüller and L. Paulson. Exploring Properties of Normal Multimodal Logics in Simple Type Theory with LEO-II. In C. Benzmüller, C. Brown, J. Siekmann, and R. Statman, editors, *Festschrift in Honour of Peter B. Andrews on his 70th Birthday*, page To appear. IfCoLog, 2007.
3. C. Benzmüller, F. Rabe, and G. Sutcliffe. THF0 - The Core TPTP Language for Classical Higher-Order Logic. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, page Accepted, 2008.
4. A. Bundy. *The Computer Modelling of Mathematical Reasoning.* Academic Press, 1983.
5. K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
6. T. Eiter, G. Iannis, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
7. M.R. Genesereth and R.E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, 1992.
8. J. Halpern, R. Fagin, Y. Moses, and M. Vardi. *Reasoning About Knowledge.* MIT Press, 1995.
9. J. Hurd. First-Order Proof Tactics in Higher-Order Logic Theorem Provers. In M. Archer, B. Di Vito, and C. Munoz, editors, *Proceedings of the 1st International Workshop on Design and Application of Strategies/Tactics in Higher Order Logics*, number NASA/CP-2003-212448 in NASA Technical Reports, pages 56–68, 2003.
10. C. Lutz. Description Logics with Concrete Domains - A Survey. In Balbiani P., Suzuki N., Wolter F., and Zakharyaschev M., editors, *Advances in Modal Logics, Volume 4*, pages 265–296. King's College Publications, 2003.
11. B. MacCartney, S. McIlraith, E. Amir, and T. Uribe. Practical Partition-Based Theorem Proving for Large Knowledge Bases. In Gottlob G. and T. Walsh, editors, *Proceedings of the 18th International Joint Conference on Artificial Intelligence* , pages 89–96, 2003.
12. W.W. McCune. Otter 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory, Argonne, USA, 1994.
13. I. Niles and A. Pease. Towards A Standard Upper Ontology. In C. Welty and B. Smith, editors, *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems*, pages 2–9, 2001.
14. A. Pease. The Sigma Ontology Development Environment. In F. Giunchiglia, A. Gomez-Perez, A. Pease, H. Stuckenschmidt, Y. Sure, and S. Willmott, editors, *Proceedings of the IJCAI-03 Workshop on Ontologies and Distributed Systems*, volume 71 of *CEUR Workshop Proceedings*, 2003.
15. A. Pease and G. Sutcliffe. First Order Reasoning on a Large Ontology. In J. Urban, G. Sutcliffe, and S. Schulz, editors, *Proceedings of the CADE-21 Workshop on Empirically Successful Automated Reasoning in Large Theories*, volume 257 of *CEUR Workshop Proceedings*, pages 59–69, 2007.

16. Y. Puzis, Y. Gao, and G. Sutcliffe. Automated Generation of Interesting Theorems. In G. Sutcliffe and R. Goebel, editors, *Proceedings of the 19th International FLAIRS Conference*, pages 49–54. AAAI Press, 2006.
17. W. Reif and G. Schellhorn. Theorem Proving in Large Theories. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction, A Basis for Applications*, volume III Applications of *Applied Logic Series*, pages 225–241. Kluwer Academic Publishers, 1998.
18. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
19. S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
20. M.E. Stickel. SNARK - SRI's New Automated Reasoning Kit. http://www.ai.sri.com/ stickel/snark.html.
21. M.E. Stickel. The Deductive Composition of Astronomical Software from Subroutine Libraries. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, pages 341–355. Springer-Verlag, 1994.
22. G. Sutcliffe, E. Denney, and B. Fischer. Practical Proof Checking for Program Certification. In G. Sutcliffe, B. Fischer, and S. Schulz, editors, *Proceedings of the Workshop on Empirically Successful Classical Automated Reasoning, 20th International Conference on Automated Deduction*, 2005.
23. G. Sutcliffe, S. Schulz, K. Claessen, and A. Van Gelder. Using the TPTP Language for Writing Derivations and Finite Interpretations. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning*, number 4130 in Lecture Notes in Artificial Intelligence, pages 67–81, 2006.
24. G. Sutcliffe, M. Stickel, S. Schulz, and J. Urban. Answer Extraction for TPTP. http://www.tptp.org/TPTP/Proposals/AnswerExtraction.html.
25. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
26. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
27. S. Trac, Y. Puzis, and G. Sutcliffe. An Interactive Derivation Viewer. In S. Autexier and C. Benzmüller, editors, *Proceedings of the 7th Workshop on User Interfaces for Theorem Provers, 3rd International Joint Conference on Automated Reasoning*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 109–123, 2006.
28. J. Urban and G. Sutcliffe. ATP Cross-verification of the Mizar MPTP Challenge Problems. In N. Dershowitz and A. Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 4790 in Lecture Notes in Artificial Intelligence, pages 546–560, 2007.
29. J. Urban, G. Sutcliffe, P. Pudlak, and J. Vyskocil. MaLARea SG1: Machine Learner for Automated Reasoning with Semantic Guidance. In P. Baumgartner, A. Armando, and D. Gilles, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning*, Lecture Notes in Artificial Intelligence, 2008.
30. A. Van Gelder and G. Sutcliffe. Extending the TPTP Language to Higher-Order Logic with Automated Parser Generation. In U. Furbach and N. Shankar, editors,

Proceedings of the 3rd International Joint Conference on Automated Reasoning, number 4130 in Lecture Notes in Artificial Intelligence, pages 156–161. Springer-Verlag, 2006.

31. R. Waldinger. Whatever Happened to Deuctive Question Answering? In N. Dershowitz and A. Voronkov, editors, *Proceedings of the 14th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, number 4790 in Lecture Notes in Artificial Intelligence, pages 15–16, 2007.

Contextual Rewriting in SPASS

Christoph Weidenbach and Patrick Wischnewski

Max-Planck-Institut für Informatik,
Campus E 1.4, Saarbrücken, Germany,
{weidenb,wischnew}@mpi-inf.mpg.de

Abstract. Sophisticated reductions are an important means to achieve progress in automated theorem proving. We consider the powerful reduction rule *Contextual Rewriting* in the context of the superposition calculus. If the rule is considered in its abstract, most general form, the applicability of contextual rewriting is not decidable. We develop a decidable instance of the general contextual rewriting rule and implement it in SPASS. An experimental evaluation on the TPTP gives first insights into the application potential of the rule instance.

1 Introduction

In the superposition context, first-order theorem proving with equality deals with the problem of showing unsatisfiability of a (finite) set N of clauses. This problem is well-known to be undecidable, in general. It is semi-decidable in the sense that superposition is refutationally complete. The superposition calculus is composed of inference and reduction rules. Inference rules generate new clauses from N whereas reduction rules delete clauses from N or transform them into simpler ones. If, in particular, powerful reduction rules are available, decidability of certain subclasses of first-order logic can be shown and explored in practice [1–4]. Hence, sophisticated reductions are an important means for progress in automated theorem proving. In this paper the reduction rule *Contextual Rewriting* is considered in the context of the superposition calculus [5]. Contextual rewriting extends rewriting with unit equations to rewriting with full clauses containing a positive orientable equation. In order to apply such a clause for rewriting, all other literals of that clause have to be entailed by the context of the clause to be rewritten and potentially further clauses from a given clause set. Hence, the name contextual rewriting.

For a first, simplified example consider the two clauses

$$P(x) \rightarrow f(x) \approx x \qquad S(g(a)), a \approx b, P(b) \rightarrow R(f(a))$$

where we write clauses in implication form [6]. Now in order to rewrite $R(f(a))$ in the second clause to $R(a)$ using the equation $f(x) \approx x$ of the first clause with matcher $\sigma = \{x \mapsto a\}$, we have to show that $P(x)\sigma$ holds in the context of the second clause $S(g(a)), a \approx b, P(b)$, i.e., $\models S(g(a)), a \approx b, P(b) \rightarrow P(x)\sigma$. This obviously holds, so we can replace $S(g(a)), a \approx b, P(b) \rightarrow R(f(a))$ by

$S(g(a)), a \approx b, P(b) \rightarrow R(a)$ via a contextual rewriting application of $P(x) \rightarrow f(x) \approx x$.

More general, contextual rewriting is the following rule:

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C = (\Gamma_2 \rightarrow \Delta_2)[u[s\sigma] \approx v]}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t \\ (\Gamma_2 \rightarrow \Delta_2)[u[t\sigma] \approx v]\end{array}}$$

where $(\Gamma_2 \rightarrow \Delta_2)[u[s\sigma] \approx v]$ expresses that $u[s\sigma] \approx v$ is an atom occurring in Γ_2 or Δ_2 and u contains the subterm $s\sigma$. Contextual rewriting reduces the subterm $s\sigma$ of u to $t\sigma$ if, among others, the following conditions are satisfied

$$N_C \models \Gamma_2 \rightarrow A \text{ for all } A \text{ in } \Gamma_1\sigma$$
$$N_C \models A \rightarrow \Delta_2 \text{ for all } A \text{ in } \Delta_1\sigma$$

where N is the current clause set, $C, D \in N$, and N_C denotes the set of clauses from N smaller than C with respect to a reduction ordering $\prec$, total on ground terms. Reduction rules are labeled with an $\mathcal{R}$ and are meant to replace the clauses above the bar by the clauses below the bar. Both side conditions are undecidable, in general. Therefore, in order to make the rule applicable in practice, it must be instantiated such that eventually these two conditions become effective. This is the topic of this paper.

For a more sophisticated, further motivating example, consider the following clause set. It can be finitely saturated using contextual rewriting but not solely with less sophisticated reduction mechanisms such as unit rewriting or subsumption.

Let i, q, r, c be functions, a, b be constants and x_1, x_2, x_3, x_4, y_1 be variables and let $r \succ c \succ q \succ i \succ b \succ a \succ nil$ using the KBO with weight 1 for all function symbols and variables.

1: $\rightarrow q(nil) \approx b$
2: $i(x_1) \approx b, q(y_1) \approx b \rightarrow q(r(x_1, y_1)) \approx b$
3: $i(x_1) \approx b, q(y_1) \approx b \rightarrow q(c(x_1, y_1)) \approx a$
4: $i(x_1) \approx b, q(y_1) \approx b, i(x_3) \approx b \rightarrow$
$r(x_3, c(x_1, y_1)) \approx c(x_1, r(x_3, y_1))$
5: $i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow$
$y_1 \approx nil, q(c(x_1, c(x_2, r(x_3, y_1)))) \approx b$

If we apply superposition right between clause 4 and clause 5 on the term $q(c(x_1, c(x_2, r(x_3, y_1))))$ we obtain the clause
6: $i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, i(x_4) \approx b, q(y_1) \approx b, q(c(x_4, y_1)) \approx b, b \approx a \rightarrow$
$c(x_4, y_1) \approx nil,\ q(c(x_1, c(x_2, c(x_4, r(x_3, y_1))))) \approx b$

which is larger (both in the ordering and the number of symbols) than clause 5. Applying superposition between clause 4 and clause 6 yields an even larger clause. Repeating the superposition inference between clause 4 and these clauses creates larger and larger clauses. Hence, the exhaustive application of the superposition calculus does not terminate on this clause set. Furthermore, none of the reductions which have been implemented so far in Spass and in any

other system we are aware of, can reduce clause 5.[1] However, with contextual rewriting we can reduce clause 5 using clause 3 to

$$7\colon i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow y_1 \approx nil, a \approx b.$$

Clause 7 is a tautology and can be reduced to true. Then the set is saturated since no further superposition inference is possible. In order to apply contextual rewriting we have to verify the side conditions

$$N_C \models i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow i(x_1) \approx b$$

and

$$N_C \models i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow q(c(x_2, r(x_3, y_1))) \approx b.$$

The first condition holds trivially and the latter follows from clause 3 and clause 2.

In this work we presents an instance of contextual rewriting that reduces the above clause set, is decidable and feasible for practical problem instances. We tested our implementation on all problems of the *TPTP* library version 3.2.0 [7]. In Section 2 we develop a practically useful instance of contextual rewriting called *Approximated Contextual Rewriting*. Section 3 presents the implementation of approximated contextual rewriting in Spass. The final section discusses experimental results.

2 Contextual Rewriting

We consider first-order logic with equality using notation from [6]. We write clauses in the form $\Gamma \rightarrow \Delta$ where Γ and Δ are multi-sets of atoms. The atoms of Γ denote negative literals while the atoms of Δ denote the positive literals. A substitution σ is a mapping from the set of variables to the set of terms such that $x\sigma \neq x$ for only finitely many variables x. The reduction rules, in particular the contextual rewriting rule, are defined with respect to a well-founded reduction ordering $\prec$ on terms that is total on ground terms. This ordering is then lifted to literals and clauses in the usual way [6]. A term s is called *strictly maximal* in $\Gamma \rightarrow \Delta$ if there is no different occurrence of a term in $\Gamma \rightarrow \Delta$ that is greater or equal than s with respect to $\prec$.

Contextual rewriting is a sophisticated reduction rule originally introduced in [5] that generalizes unit rewriting and non-unit rewriting [6]. It is an instance of the standard redundancy notion of superposition. A clause C is called *redundant* in a clause set N if there exist clauses $C_1, \ldots, C_n \in N$ with $C_i \prec C$ for $i \in \{1, \ldots, n\}$, written $C_i \in N_C$, such that $C_1, \ldots, C_n \models C$. The clause C is implied by smaller clauses from N. This condition can actually be refined to grounding substitutions: C is redundant if for all grounding substitutions σ for C there are ground instances $C_i\sigma_i$ of clauses $C_i \in N$ such that $C_i\sigma_i \prec C\sigma$, written $C_i\sigma_i \in N_{C\sigma}$, and $C_1\sigma_1, \ldots, C_n\sigma_n \models C\sigma$. Reduction rules are marked with an $\mathcal{R}$ and their application replaces the clauses above the bar with the clauses below the bar.

[1] Actually, the SATURATE system contained the first implementation of contextual rewriting, but it never matured to be widely usable.

Definition 1 (Contextual Rewriting [5]). *Let N be a clause set, $C, D \in N$, σ be a substitution then the reductions*

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C = \Gamma_2, u[s\sigma] \approx v \rightarrow \Delta_2}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t \\ C'' = \Gamma_2, u[t\sigma] \approx v \rightarrow \Delta_2\end{array}}$$

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C = \Gamma_2 \rightarrow \Delta_2, u[s\sigma] \approx v}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t \\ C'' = \Gamma_2 \rightarrow \Delta_2, u[t\sigma] \approx v\end{array}}$$

where the following conditions are satisfied

1. $s\sigma \succ t\sigma$
2. $C \succ D\sigma$
3. $N_C \models \Gamma_2 \rightarrow A$ *for all* A *in* $\Gamma_1\sigma$
4. $N_C \models A \rightarrow \Delta_2$ *for all* A *in* $\Delta_1\sigma$

are called contextual rewriting.

Due to condition 1-1 and condition 1-2 we have $C'' \prec C$ and $D\sigma \prec C$. Then from condition 1-3 and condition 1-4 we obtain that there exist clauses $C_1, \ldots, C_n \in N_C$ and $C_1, \ldots, C_n, C'', D\sigma \models C$. Therefore, the clause C is redundant in $N \cup \{C''\}$ and can be eliminated. The rule is an instance of the abstract superposition redundancy notion.

The side conditions 1-3 and 1-4 having both the form $N_C \models \Gamma \rightarrow \Delta$ are undecidable, in general. There are two sources for the undecidability. First, there are infinitely possible grounding substitutions σ' for the clause $\Gamma \rightarrow \Delta$ and C. Second, for a given σ' there may be infinitely many δ with $C_i\delta \prec C\sigma'$, $C_i \in N$. Therefore, in the following in order to effectively decide the side conditions, we will fix one σ' and restrict the number of considered substitutions δ yielding a decidable instance of contextual rewriting.

First, $N_C \models \Gamma \rightarrow \Delta$ is equivalent to $N_C \cup \{\exists \boldsymbol{x}.\neg(\Gamma \rightarrow \Delta)\} \models \bot$. The existential quantifier can be eliminated by Skolemization yielding a Skolem substitution τ that maps each variable out of $\boldsymbol{x}$ to a new Skolem constant. Consequently, setting σ' to τ yields the instance $N_C \models (\Gamma \rightarrow \Delta)\tau$, where $(\Gamma \rightarrow \Delta)\tau$ is ground. Still there may exist infinitely many δ with $C_i\delta \prec C\tau$, $C_i \in N$ and $C\tau$ may still contain variables.

Therefore, we restrict δ to those grounding substitutions that map variables to terms only occurring in $C\tau$ or $D\sigma\tau$ where we additionally assume that τ is also grounding for C and $D\sigma$, i.e., it maps any variable occurring in C or $D\sigma$ to an arbitrary fresh Skolem constant. Let $N_{C\tau}^{D\sigma\tau}$ be the set of all ground instances of clauses from N smaller than $C\tau$ obtained by instantiation with ground terms from $D\sigma\tau, C\tau$. Then $N_{C\tau}^{D\sigma\tau}$ is finite and $N_{C\tau}^{D\sigma\tau} \subseteq N_{C\tau}$. Consequently, $N_{C\tau}^{D\sigma\tau} \models (\Gamma \rightarrow \Delta)\tau$ is a sufficient ground approximation of $N_C \models \Gamma \rightarrow \Delta$. Even

though this is a decidable approximation of the original problem the set $N_{C\tau}^{D\sigma\tau}$ is exponentially larger than N, in general. Therefore, we represent $N_{C\tau}^{D\sigma\tau}$ implicitly by approximating $N_{C\tau}^{D\sigma\tau} \models (\Gamma \rightarrow \Delta)\tau$ by the application of a reduction calculus $(\Gamma \rightarrow \Delta)\tau \vdash_{Red} \top$. The reduction calculus $\vdash_{Red}$ is composed of a set of reduction rules containing *tautology reduction*, *forward subsumption*, *obvious reduction* and a particular instance of contextual rewriting called *recursive contextual ground rewriting* defined below. Tautology reduction reduces syntactic and semantic tautologies to true whereas forward subsumption reduces subsumed clauses to true. Obvious reduction eliminates trivial literals [6].

The reduction calculus $\vdash_{Red}$ only needs to reduce ground clauses. Therefore, the following definition introduces an instance of contextual rewriting only working on ground clauses. Further, it adapts contextual rewriting such that it implicitly considers clauses from $N_{C\tau}^{D\sigma\tau}$.

Definition 2 (Recursive Contextual Ground Rewriting). *If N is a clause set, $D \in N$, C' ground, σ a substitution then the reduction*

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C' = \Gamma_2, u[s\sigma] \approx v \rightarrow \Delta_2}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t\\ \Gamma_2, u[t\sigma] \approx v \rightarrow \Delta_2\end{array}}$$

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C' = \Gamma_2 \rightarrow \Delta_2, u[s\sigma] \approx v}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t\\ \Gamma_2 \rightarrow \Delta_2, u[t\sigma] \approx v\end{array}}$$

where the following conditions are satisfied

1. *$s\sigma$ is a strictly maximal term in $D\sigma$*
2. *$u[s\sigma] \approx v \succ s\sigma \approx t\sigma$*
3. *$vars(s) \supset vars(D)$*
4. *$(\Gamma_2 \rightarrow A) \vdash_{Red} \top$ for all A in $\Gamma_1\sigma$*
5. *$(A \rightarrow \Delta_2) \vdash_{Red} \top$ for all A in $\Delta_1\sigma$*

is called recursive contextual ground rewriting.

Condition 2-1 and condition 2-2 ensure the ordering restrictions required by contextual rewriting. Condition 2-3 implies that $D\sigma$ is ground. A clause D meeting condition 2-1 and condition 2-3 is called *strongly universally reductive*. Condition 2-4 and condition 2-5 recursively apply the reduction calculus.

The reduction calculus $\vdash_{Red}$ is terminating since C' is reduced to a smaller ground clause. As a consequence, also the reduction relation $\vdash_{Red}$ is terminating. The approximated contextual rewriting rule eventually becomes the below rule.

Definition 3 (Approximated Contextual Rewriting). *Let N be a clause set, $C, D \in N$, σ be a substitution then the reductions*

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C = \Gamma_2, u[s\sigma] \approx v \rightarrow \Delta_2}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t \\ C'' = \Gamma_2, u[t\sigma] \approx v \rightarrow \Delta_2\end{array}}$$

$$\mathcal{R}\frac{D = \Gamma_1 \rightarrow \Delta_1, s \approx t \qquad C = \Gamma_2 \rightarrow \Delta_2, u[s\sigma] \approx v}{\begin{array}{c}\Gamma_1 \rightarrow \Delta_1, s \approx t \\ C'' = \Gamma_2 \rightarrow \Delta_2, u[t\sigma] \approx v\end{array}}$$

where the following conditions are satisfied

1. $s\sigma \succ t\sigma$
2. $C \succ D\sigma$
3. τ *maps all variables from* $C, D\sigma$ *to fresh Skolem constants*
4. $(\Gamma_2 \rightarrow A)\tau \vdash_{Red} \top$ *for all* A *in* $\Gamma_1\sigma$
5. $(A \rightarrow \Delta_2)\tau \vdash_{Red} \top$ *for all* A *in* $\Delta_1\sigma$

are called approximated contextual rewriting.

Note that unit rewriting and non-unit rewriting [6] are also instances of the approximated contextual rewriting rule. Note further that the conditions for the approximated contextual rewriting rule are weaker compared to the recursive contextual ground rewriting rule: the right premise needs not to be ground and the equation $s \approx t$ needs not to be maximal in the first premise. Approximated contextual rewriting uses recursive contextual ground rewriting to effectively decide the side conditions.

3 Implementation

The implementation of SPASS [6] focuses on a sophisticated reduction machinery. This machinery completely interreduces all clauses as it performs *forward reduction* and *backward reduction* whenever a clause is newly generated or modified. Forward reduction reduces the newly generated clause using the previously generated clauses and backward rewriting reduces the previously generated clauses with the new one.

The integration of contextual rewriting into this machinery consists of two steps. First, the search for appropriate contextual rewrite application candidates is analogous to the case of unit rewriting and non-unit rewriting. In addition, the side conditions of contextual rewriting have to be checked. Finding appropriate contextual rewrite application candidates can be solved by standard term indexing [8]. This functionality has already been implemented into SPASS via substitution trees [6, 9].

Second, validating the side conditions of contextual rewriting requires an effective implementation of the reduction calculus $\vdash_{Red}$. First of all, it is too costly to explicitly compute the Skolem substitution τ for each clause $\Gamma \rightarrow$

```
1  RecursiveValidityCheck(CLAUSE C, CLAUSE SET N);
2  Rewritten=TRUE;
3  while Rewritten do
4  |  Rewritten=FALSE;
5  |  if IsEmpty(C) then return FALSE;
6  |  if IsTautology(C) then  return TRUE ;
7  |  if ForwardSubsumption(C, N) then return TRUE;
8  |  if ObviousReduction(C) then Rewritten=TRUE;
9  |  if RecursiveContextualGroundRewriting(C, N) then
   |      Rewritten=TRUE ;
10 end
11 return FALSE
```

Algorithm 1: RecursiveValidityCheck

Δ which the reduction calculus considers for contextual rewriting. Applying τ explicitly requires to allocate memory for the new constants and the new clause and it requires additional computations to build the clause. Because of the recursive structure of the reduction calculus this is not feasible. Therefore, the idea is to simply treat variables as constants for this case. We adapt the ordering modules (KBO, RPOS) such that they can treat variables as constants. Since variables are not contained in the precedence we have to define an ordering on them. In Spass variables are represented by integers which implicitly gives an ordering on variables. Whenever we consider variables to be constants we assume them to have a lower precedence than any other symbol of the signature. Comparisons between variables are performed on the bases of their integer value. As a result, the implementation provides a method for applying τ to a clause $\Gamma \rightarrow \Delta$ without any computation or memory allocation.

The composition of the reduction calculus $\vdash_{Red}$ to an actual decision procedure is depicted in Algorithm 1. The algorithm uses tautology deletion, forward subsumption and obvious reductions from the reduction procedure of Spass. The procedures have to work with respect to the modified, above explained, orderings. Besides of this the implementation of these reductions remains unchanged.

Algorithm 1 expects as input a clause C and a clause set N and reduces C with respect to N in the main loop. IsEmpty(C) checks whether the given clause is the empty clause, IsTautology(C) checks whether C is either a syntactic or a semantic tautology. ForwardSubsumption(C, N) checks whether C is already subsumed by clauses from N and ObviousReduction(C) removes duplicated literals and trivial equations from a clause. Further details can be found in the Spass Handbook [10].

RecursiveContextualGroundRewriting(C, N), depicted in Algorithm 2, subsumes unit and non-unit rewriting and implements recursive contextual ground rewriting. The variables occurring in C are interpreted as constants in the above explained sense. The call to $general_{SDT}(N, u')$ returns the set of generalizations G from N of u' and the respective matcher σ. Then the procedure computes for each of the generalizations the literals and the clauses where they occur resulting

```
1  RecursiveContextualGroundRewriting(CLAUSE C[u[u'] ≈ v],
                                       CLAUSE SET N);
2  G = general_SDT(N, u');
3  foreach (s, σ) ∈ G do
4  |  Lits = LiteralsContainingTerm(s);
5  |  foreach (s ≈ t) ∈ Lits do
6  |  |  D = LiteralOwningClause(s ≈ t);
7  |  |  if (vars(s) ⊃ vars(D) ∧
8  |  |        u[sσ] ≈ v ≻ sσ ≈ tσ
9  |  |        sσ strictly maximal term in Dσ ∧
10 |  |        ∀A ∈ Ante(Dσ) RecursiveValidityCheck(Γ → A) ∧
11 |  |        ∀A ∈ Succ(Dσ) RecursiveValidityCheck(A → Δ) ) then
12 |  |  |  return C[u[tσ] ≈ v];
13 |  |  end
14 |  end
15 end
```

Algorithm 2: RecursiveContextualGroundRewriting

in the contextual rewrite candidates. The candidate clauses are then checked for the non-recursive side conditions of contextual rewriting. If these hold, RecursiveContextualGroundRewriting builds the subproblems and recursively calls RecursiveValidityCheck.

The implementation of approximated contextual rewriting is analogous to the implementation of recursive contextual ground rewriting. The difference is that the input clause C is not interpreted as ground and the local side conditions (line 7 - line 9) are changed with respect to the definition of approximated contextual rewriting.

4 Results

4.1 Results on the TPTP

The *TPTP 3.2.0* [7] is a library consisting of 8984 problems for automated theorem proving systems. We compared the Spass version 3.1 that is version 3.0 extended by some bug fixes, containing our implementation of contextual rewriting to Spass version 3.1 without contextual rewriting.

Table 1 depicts the results. We compare two runs of Spass with a reference run. All runs were performed with Spass options set to *-RFRew=4 -RBRew=2 -RTaut=2* on Opteron nodes running at speed of 2.4 GHz equipped with 4 GB RAM for each node. For the reference run we let Spass run on the TPTP with contextual rewriting turned off and a time limit of 300 seconds. The first run of Spass with contextual rewriting activated and a 300 seconds time bound found 77 additional proofs and lost 151 proofs compared to the run with contextual rewriting disabled. There were no significant differences among the versions on satisfiable problems. The second run of Spass with contextual

Time	Won	Lost	Deceleration Coefficient	Solved Open Problems
300	77	151	1.46	2
600	122	133	1.62	5

Table 1. Forward contextual rewriting

rewriting and a 600 seconds time bound found 122 additional proofs and lost 133 proofs compared to the run with contextual rewriting disabled. Additionally, we computed the average running time of the reference run and each of the other two runs. We considered only those cases where both the reference run and the respective test run terminated and the version with contextual rewriting took at least 10 seconds. The running time slows down on the average at a coefficient of 1.46 for the 300 seconds run and at a coefficient of 1.62 for the 600 seconds run.

The problems where SPASS with contextual rewriting found a proof and the reference run did not were mostly problems with a high TPTP difficulty rating. In the run with 300 seconds our instances even terminated on two problems with rating 1.00 and with 600 seconds on five problems with rating 1.00 meaning that no system has been able to solve these problems so far. The problems are SWC308+1, SWC335+1, in the 300 seconds time bound and additionally in the 600 seconds time bound SWC329+1, SWC342+1, SWC345+1. All proofs were checked by the SPASS proof checker.

It was both a surprise to us that the contextual rewriting version did not improve on satisfiable problems and that it lost so many unsatisfiable problems. For the satisfiable problems this is simply due to the fact that the TPTP does not contain suitable problems. For the unsatisfiable ones we inspected some in detail and figured out that the actual proof found by the standard version got lost through a contextual rewriting application. This is not a new phenomenon, however, it is surprising that it occurs so often on the TPTP. We will further dig into this hoping to find further insight.

4.2 Application to the Example from the Introduction

In this part we depict the application of approximated recursive contextual rewriting on the introductory example in detail. Therewith, we show that superposition together with our instance of contextual rewriting terminates on this example. SPASS with contextual rewriting is able to saturate the clause set from section 1 whereas SPASS without contextual rewriting is not. Recall that we can reduce clause 5 with clause 3 using contextual rewriting if the side conditions are fulfilled. The ground clauses

$8 : i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow i(x_1) \approx b$

$9 : i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, b \approx a \rightarrow q(c(x_2, r(x_3, y_1))) \approx b$

must be entailed by clauses from N_C. Clause 8 is a tautology whereas clause 9 can be rewritten with clause 3 to

10: $i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, a \approx b \rightarrow a \approx b$

using contextual rewriting if in addition the clauses

11: $i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, a \approx b \rightarrow i(x_2) \approx b$

12: $i(x_1) \approx b, i(x_3) \approx b, i(x_2) \approx b, q(y_1) \approx b, a \approx b \rightarrow q(r(x_3, y_1)))) \approx b$

are also entailed by clauses from N_C. Clause 11 is a syntactic tautology and clause 12 is subsumed by clause 2.

5 Summary

In summary, contextual rewriting is costly but it helps solving difficult problems. Our current implementation offers reasonable room for improvement. For example, in SPASS, contextual rewriting is implemented independently from unit and non-unit rewriting such that in case of non-applicability of the rule, the same checks and indexing queries are repeated.

For subsumption nice filters are known to detect non-applicability. It would be worthwhile to search for such filters for contextual rewriting. Eventually, we need to better understand why we lost surprisingly many problems from the TPTP. One possible answer could be that the standard SPASS heuristics for inference selection don't work well anymore with contextual rewriting. The SPASS version described in this paper and used for the experiments can be obtained from the SPASS homepage (`http://spass-prover.org/`) following the prototype link.

References

1. Bachmair, L., Ganzinger, H., Waldmann, U.: Superposition with simplification as a decision procedure for the monadic class with equality. In: Computational Logic and Proof Theory. Volume 713 of LNCS. (1993) 83–96
2. Hustadt, U., Schmidt, R.A., Georgieva, L.: A survey of decidable first-order fragments and description logics. Journal of Relational Methods in Computer Science **1** (2004) 251–276
3. Jacquemard, F., Meyer, C., Weidenbach, C.: Unification in extensions of shallow equational theories. In: Proceedings of RTA-98. Volume 1379 of LNCS., Springer (1998) 76–90
4. Ganzinger, H., de Nivelle, H.: A superposition decision procedure for the guarded fragment with equality. In: LICS. (1999) 295–304
5. Bachmair, L., Ganzinger, H.: Rewrite-based equational theorem proving with selection and simplification. Journal of Logic and Computation **4**(3) (1994) 217–247
6. Weidenbach, C.: Combining superposition, sorts and splitting. In Robinson, A., Voronkov, A., eds.: Handbook of Automated Reasoning. Volume 2. Elsevier (2001) 1965–2012
7. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning **21**(2) (1998) 177–203
8. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In Robinson, J.A., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 1853–1964
9. Graf, P.: Term Indexing. Volume 1053 of LNAI. Springer-Verlag (1995)
10. Weidenbach, C., Schmidt, R., Keen, E.: Spass handbook version 3.0. Contained in the documentation of SPASS Version 3.0 (2007)

Author Index

Armando, Alessandro 1
Arthan, Rob 2

Benzmüller, Christoph 22

Chang, Cynthia 81

de Nivelle, Hans 56
del Rio, Nick 81
Dershowitz, Nachum 26
Ding, Li 81

Glöckner, Ingo 71

Hinrichs, Timothy 36

Marcos, Joao 46
McGuinness, Deborah 81
Mendonca, Dalmo 46

Otten, Jens 94

Pease, Adam 66, 103
Pelzer, Björn 71
Pinheiro da Silva, Paulo 81

Rabe, Florian 22
Raths, Thomas 94

Schürmann, Carsten 22
Siegel, Nick 66
Slaney, John 11
Sutcliffe, Geoff 22, 66, 81, 103

Trac, Steven 66, 103

Weidenbach, Christoph 115
Wischnewski, Patrick 115
Witkowski, Piotr 56