Studienarbeitsthemen 2018/2019

Stephan Schulz

stephan.schulz@dhbw-stuttgart.de
http://www.dhbw-stuttgart.de/~sschulz

Inhaltsverzeichnis

1 Einführung

Deduktionssysteme bilden logische Denkprozesse nach, um so aus bekannten Fakten gesicherte neue Erkenntnisse abzuleiten. Automatische Deduktion ist ein Teilgebiet der künstlichen Intelligenz, aber auch der theoretischen Informatik. Wir entwickeln an der DHBW Stuttgart den Theorembeweiser E [11, 10], einen der derzeit leistungsstärksten Beweiser für die Prädikatenlogik erster Stufe, und den stärksten dieser Beweiser, der unter einer Open-Source/Free-Software-Lizenz steht. E nimmt regelmäßig an der *CADE ATP System Competition* teil, und ist in seiner Klasse in den letzten Jahren nur vom Beweiser Vampire der Universität Manchester geschlagen worden.

Konkret sind automatische Theorembeweiser Computerprogramme, die nachweisen, dass bestimmte Aussagen zwingend aus einer gegebenen formalen Be-

schreibung folgen. Ein wichtiges Anwendungsbeispiel ist die Verifikation von Software, z.B. der Nachweis, dass ein Fahrassistenzsystem einen PKW niemals beschleunigt, wenn der Fahrer die Bremse tritt, oder dass eine Banktransaktion immer entweder vollständig oder gar nicht abgewickelt wird. Ein anderes Anwendungsbeispiel ist die Beantwortung von komplexen Fragen über großen formalen Wissensbasen, etwas die Frage "Welche europäischen Großstädte sind von Überflutungen bedroht?" Um diese Fragen zu beantworten, muss ein Beweiser im Raum der möglichen logischen Ableitungen nach einem (oder auch mehreren) Beweisen für die Behauptung bzw. Vermutung suchen.

E ist ein *saturierender* Theorembeweiser für die Prädikatenlogik erster Stufe. Ein solcher Beweiser stellt den Suchzustand als Menge von als wahr angenommenen logischen Formeln (*Klauseln*) dar, und kombiniert nach festen Regeln Formeln, die Aussagen über die selben Objekte machen, um so neues Wissen explizit herzuleiten. Dieser Prozess wird ausgeführt, bis der Beweis offensichtlich wird, oder bis der Beweisversuch aufgegeben wird. In der Praxis sucht das System nach einem *Beweis per Widerspruch*, bei dem dann der explizite Widerspruch zwischen den Axiomen und der negierten Vermutung hergeleitet wird.

Fast immer ist die Menge der ableitbaren Formeln unendlich, und die Größe der Wissensbasis wächst sehr schnell. Das erschwert die Beweissuche stark. Um diesen Effekt zu reduzieren haben moderne Kalküle ein Redundanzkonzept, mit der Formeln entweder vereinfacht oder sogar komplett gestrichen werden können. Moderne Beweiser verbringen einen großen Teil der Beweissuche mit solchen *Simplifikationen*.

Um einen Beweiser möglichst effizient zu machen gibt es zwei grundsätzliche Ansätze. Zum einen kann man die Geschwindigkeit steigern, mit der neue Formeln generiert und vereinfacht werden. Dadurch wird ein größerer Teil des Suchraums bearbeitet, was die Chance erhöht, einen Beweis zu finden. Zum anderen kann man versuchen, aus der Menge aller ableitbaren Formeln die für einen Beweisversuch besonders interessanten Formeln erkennen und so die Suche auf viel versprechende Teile des Suchraums konzentrieren. So kann man gezielt tiefer in den Suchraums vordringen und potentiell auch längere Beweise finden.

Neben diesen Kernfragen gibt es auch interessante Aufgabenstellung in der Vorverarbeitung der Beweisprobleme, der Axiomauswahl, der Bearbeitung von Hintergrundtheorien und der Beweisdarstellung.

1.1 Rahmenbedingungen

Für alle angebotenen Arbeiten gilt:

- Implementierungssprache im Beweiser selbst ist C. Für Programmieraufgaben im Umfeld kommen auch andere Sprachen mit Open-Source-Implementierungen in Frage. Viele Teilprojekte verwenden Python.
- E wird primär für UNIX-artige Betriebssysteme entwickelt, insbesondere Linux und OS-X. Das Versionskontrollsystem ist `git`.

- Wenn Code in den Beweiser integriert wird, so wird er damit implizit wie das System selbst lizenziert (im Moment GPL 2 und LGLP 2). Das heißt auch, dass nur Bibliotheken zum Einsatz kommen können, die ebenfalls unter geeigneten Open-Source-Lizenzen stehen. Wir bemühen uns im allgemeinen, wenig externe Abhängigkeiten in das Kernsystem einzubringen.
- Es wird angestrebt, Ergebnisse der Arbeiten auf Workshops, Konferenzen, oder in Journal-Artikeln zu veröffentlichen. Der Erfolg dieser Bemühungen hängt von der Qualität der Ergebnisse und von der Hartnäckigkeit der Autoren ab.

2 Themen

2.1 *Express yourself*: Compilation von `let` und `ite` in Prädikatenlogik erster Stufe

Automatische Theorembeweiser für die Prädikatenlogik erster Stufe führen die Beweissuche in der Regel nach einer Transformation des Problems in Klauselnormalform (CNF) durch. Der Nutzer spezifiziert seine Anforderung in der Regel aber in einer Sprache mit stärkeren Ausdrucksmitteln - traditional in voller Prädikatenlogik erster Stufe (FOF - *First-Order Form*) oder inzwischen in sortierter First-Order-Logik (TFF - *Typed First-order Form*). Für diese Sprachen existert ein Standard, der sogenannte TPTP-Standard [17, 14, 16], der in Zusammenhang mit der Problembibliothek TPTPT (*Thousands of Problems for Theorem Provers*) [15] entstanden ist.

Um dem Anwender die Formalisierung weiter zu erleichtern, wird dieser Standard weiterentwickelt. Unter den neueren Änderungen sind die Einführung der Konstrukte `let` und `ite`. Mit `let` können lokale Abkürzungen für Terme eingeführt werden. Mit `ite` (*if-then-else*) können bedingte Terme oder Formeln eingeführt werden. Diese Features erhöhen die grundsätzliche Aussagekraft der Logik nicht. Sie können insbesondere bei der Transformation in Klauselnormalform durch konventionelle Klauseln dargestellt werden [3].

Ziel dieser Arbeit ist es, den Parser des Theorembeweisers E zu erweitern, um `let`, `ite`, und Tupel (als Voraussetzung für `let`) verarbeiten zun können, und die Übersetzung dieser Features in in Klauselnormalform zu implementieren.

Literatur

[11, 17, 3, 14, 16, 7]

Team

1 Studierender

2.2 *1+1=2* - Einfache Arithmetik für E

Klassische Theorembeweiser arbeiten rein syntaktisch mit *freien* Funktionssymbolen, d.h. Funktionssymbolen, die für beliebige Funktionen stehen können, und die nur den explizit in den logischen Formeln spezifizierten Einschränkungen genügen müssen. Viele praktische Anwendungen arbeiten aber auf bekannten Domänen und mit bekannten Funktionen. Ein häufiges Beispiel sind die "normalen" Zahlen mit den üblichen Funktionen ($+$, $-$, $*$, ...) und Relationen ($=$, $\geq$, $>$, ...). E unterstützt bereits seit Version 2.0 eine Logik mit verschiedenen Sorten von Objekten, darunter insbesondere die ganzen, rationalen und reellen Zahlen. Ziel dieser Arbeit ist es, dieses Sortensystem mit Leben zu erfüllen, indem die abstrakten Typen mit konkreten numerischen Typen hinterlegt werden und die üblichen arithmetischen Operationen in den Beweiser integriert werden. Die Arbeit eignet sich für 1-3 Studierende, wobei der Umfang der Implementierung mit zunehmender Zahl von Studenten skaliert werden kann,

Literatur

[11, 10, 13, 17]

Team

1–3 Studierende

2.3 *Step by Step* - Detaillierte Beweise für E

Für viele Anwendungen von Theorembeweisern werden explizite Beweisobjekte gefordert. Es reicht also nicht, wenn der Beweiser intern ein Theorem nachweisen kann, der Nutzer erwartet auch eine nachvollziehbare Begründung, *warum* das Theorem gültig ist. Solche Beweise können mit unabhängigen Werkzeugen oder manuell verifiziert werden, und die Struktur des Beweises und interessante Zwischenergebnisse geben einen Einblick in die Anwendungsdomäne und das Suchverhalten.

E kann interne Beweisobjekte mit minimalem Aufwand erzeugen und diese als Folge von begründeten Schritten ausgeben. Allerdings sind die einzelnen Beweisschritte dieses Beweisobjektes in der Regel weder eindeutig noch atomar. Sie fassen typischerweise eine generierende Inferenz und alle darauf folgenden Vereinfachungen zusammen. Damit sind sie für manche Anwendungen nicht optimal geeignet.

Im Rahmen dieser Studienarbeit soll der vorhandene Mechanismus für die Erzeugung von Beweisobjekten verfeinert werden. Dabei stehen folgende Teilaufgaben an:

1. Genauere Beschreibung der Ableitungen im Beweisobjekt, insbesondere Buchführung darüber, an welchem Positionen in Formeln und Termen Inferenzregeln angewendet werden

2. Erstellung einer Bibliothek und eines Programms, die die Beweise in Einzelschritte zerlegen und alle Zwischenergebnisse explizit macht

3. Update eines Beweisprüfers, der einzelne Beweisschritte mit anderen Theorembeweisern nachvollzieht

Die Arbeit eignet sich für zwei Studenten. Wenn Sie von einem einzelnen Studenten bearbeitet wird, sollten mindesten Punkt 1 und die Bibliothek von Punkt 2 erstellt werden.

Literatur

[11, 10, 9]

Team

1–2 Studierende

2.4 *Can you take a hint?* Verbesserte Integration von Hinweisen und Lemmas für die Beweissuche von E

Der lokale Suchraum, also die Anzahl der ableitbaren Formeln, wächst in der Regel exponentiell mit der Länge der betrachteten Ableitung. Deswegen werden insbesondere lange und komplexe Beweise nur selten direkt gefunden. Ein erfolgreicher Ansatz, trotzdem komplexe mathematische Theoreme zu beweisen, ist es, dem Beweiser bestimmte Zwischenergebnisse, so genannte *hints*, vorzugeben. Dies können entweder von Nutzer als wahr vermutete Lemmata sein, oder auch automatisch aus anderen, einfacheren Beweisen extrahiert werden. Dieser Mechanismus wurde ursprünglich für den Beweiser Otter [6] und seinen Nachfolger Prover9 [5] entwickelt, und wurde z.B. für den berühmten Beweis der Robbins-Vermutung [5] verwendet, kommt aber auch in der aktuellen mathematischen Forschung zum Einsatz [2].

E verwendet eine vergleichsweise einfache Implementierung von Hints. Dabei wird dem Beweiser eine Liste mit Zwischenergebnissen vorgegeben, und jede neu hergeleitete Formel wird mit dieser Liste verglichen. Klauseln, die Hint-Klauseln subsumieren, werden bevorzugt für die weitere Herleitung von neuen Fakten verwendet. Im Rahmen dieser Arbeit soll dieser Mechanismus verbessert werden. Insbesondere fallen dabei folgende Aufgaben an:

1. Passende Hints werden über eine *Index* (im Fall von E über einen *Feature Vector Index)* gefunden. Dieser Index ist für den häufigen Fall der Unit-Klauseln (also Klauseln mit nur einem einzelnen Literal) nicht optimal und sollte um einen *Discrimination Tree Index* für diesen Fall ergänzt werden.

2. Statt auf strikter Subsumption zu bestehen, können auch andere (schwächere oder stärkere) Ähnlichkeitsrelation verwendet werden. Speziell sollte

mindestens eine Version implementiert werden, bei der konstante Funktionssymbole nicht unterschieden werden (d.h. eine Klausel $p(a, X)$ könnte einen Hint $p(b, X)$ "weich subsumieren".

3. Hints können einmal oder mehrmals verwendet werden.

Neben der Implementierung sollen die verschiedenen Optionen experimentell erprobt werden.

Literatur

[18, 2, 11, 10]

Team

1 Studierender

2.5 *The Need for Speed* - Redesign und Implementierung des Termdatentyps in E

E ist einer der erfolgreichsten Beweiser für die Prädikatenlogik erster Ordnung. Einer der Gründe für diesen Erfolg ist, dass das Design auf solide entworfenen und implementierten Datentypen beruht. Im Zentrum stehen dabei *shared terms*, ein Datentyp, der sowohl für Terme der Prädikatenlogik erster Stufe als auch für die interne Darstellung von Formeln eingesetzt wird.

Nicht-triviale Terme bestehen aus einem Funktionsymbol und einer passenden Anzahl von Argumenten, die selbst wieder Terme sind. Aus historischen Gründen werden diese Teile in E durch ein `struct` für den Kopf des Terns und einen Pointer auf ein separates Array von Argumenttermen repräsentiert. Seit ISO/IEC 9899:1999 (C99) gibt es die Möglichkeit, sogenannte *flexible array members* zu verwenden, um in Strukturen direkt ein Array unbestimmter Größe zu verwenden. Durch eine Umstellung vom separaten Array auf flexible Arrays würde sowohl Speicherplatz gespart, bei jedem Zugriff eine Indirektion vermieden, und auch das Caching-Verhalten verbessert.

Ziel dieser Arbeit ist es, diese Umstellung im Beweiser E vorzunehmen und das Laufzeitverhalten vor und nach dieser Umstellung zu vergleichen. Die Arbeit erfordert gute Kenntnisse von C und besondere Sorgfalt beim Programmieren, da es sich um die zentrale Komponenete eines aktiv genutzten Systems handelt.

Bei erfolgreicher Durchführung kann das Ergebnis vermutlich auf einem Implementierungs-orientierten Workshop wie z.B. dem *International Workshop on the Implementation of Logics* (IWIL) oder den *Workshop on Practical Aspects of Automated Reasoning* (PAAR) vorgestellt werden.

Literatur

[10, 12, 4]

Team

1 Studierender

2.6 *Need to know* - Prämissenauswahl mit der *alternating path method*

Für den Erfolg einer Beweissuche ist es kritisch, den Beweiser nicht mit überflüssigen Fakten zu überladen. Historisch wurden viele Beweisprobleme von Mathematikern so minimal wie möglich formalisiert. Heute werden Deduktionssysteme aber oft eingesetzt, um viele verschiedene Probleme auf Grundlage einer großen Wissensbasis oder Axiomatisierung zu lösen. Das kann z.B. eine mathematische Domäne mit tausenden von Axiomen und Lemmata sein, oder auch eine *Common-Sense*-Wissensbasis mit Millionen von Fakten. Für jedes konkrete Problem wird dabei in der Regel nur eine sehr kleine Teilmenge aller bekannten Fakten und Regeln gebraucht. Die Identifikation einer möglichst kleinen, aber ausreichend großen Teilmenge macht die Suche nach einem Beweis sehr viel einfacher und oft erst plausibel. Unser Beweiser **E** [10, 12] implementiert z.B. eine Axiomauswahl nach den SInE-Verfahren [1].

Professor David Plaisted von der Duke University hat ein neues Verfahren für die Prämissenauswahl entwickelt, dass in Abhängigkeit von dem Beweisziel eine Menge von Axiomen identifiziert, in dem so genannte *alternierende Pfade* betrachtet werden [8]. Dadurch werden (im Fall von Klausellogik) nur solche Axiome ausgewählt, die theoretisch an einem Resolutions-Widerspruchsbeweis mit dem Beweisziel vorkommen können.

Im Rahmen dieser Arbeit soll das Verfahren entweder auf Grundlage des vorhandenen Axiomfilters für den Beweiser **E** in C oder auf Grundlage unseres pädagogischen Beweisers PyRes in Python implementiert und experimentell evaluiert werden. Wenn zwei Studenten das Thema bearbeiten, dann ist auch eine Implementierung in beiden Sprachen/Umgebungen eine Option.

Professor Plaisted hat sich angeboten, bei der Betreuung der Arbeit mitzuwirken. Bei Erfolg der Arbeit ist davon auszugehen, dass die Ergebnisse in einem gemeinsamen Papier veröffentlicht werden. Aus diesem Grund sollte die Arbeit idealerweise auf Englisch verfasst werden.

Literatur

[10, 12, 1, 8]

Team

1-2 Studierende

Literatur

[1] Kryštof Hoder and Andrei Voronkov. Sine Qua Non for Large Theory Reasoning. In Nikolaj Bjørner and Viorica Sofronie-Stokkermans, editors, *Proc. of the 23rd CADE, Wroclav*, volume 6803 of *LNAI*, pages 299–314. Springer, 2011.

[2] Michael Kinyon, Robert Veroff, and Petr Vojtěchovskỳ. Loops with abelian inner mapping groups: An application of automated deduction. In Maria Paola Bonacina and Mark E. Stickel, editors, *Automated Reasoning and Mathematics: Essays in Memory of William W. McCune*, volume 7788 of *LNAI*, pages 151–164. Springer, 2013.

[3] Evgenii Kotelnikov, Laura Kovács, Martin Suda, and Andrei Voronkov. A Clausal Normal Form Translation for FOOL. In Christoph Benzmüller, Geoff Sutcliffe, and Raul Rojas, editors, *GCAI 2016. 2nd Global Conference on Artificial Intelligence*, volume 41 of *EPiC Series in Computing*, pages 53–71. EasyChair, 2016.

[4] B. Löchner and S. Schulz. An Evaluation of Shared Rewriting. In H. de Nivelle and S. Schulz, editors, *Proc. of the 2nd International Workshop on the Implementation of Logics*, MPI Preprint, pages 33–48, Saarbrücken, 2001. Max-Planck-Institut für Informatik.

[5] William W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010. (acccessed 2016-03-29).

[6] W.W. McCune and L. Wos. Otter: The CADE-13 Competition Incarnations. *Journal of Automated Reasoning*, 18(2):211–220, 1997. Special Issue on the CADE 13 ATP System Competition.

[7] A. Nonnengart and C. Weidenbach. Computing Small Clause Normal Forms. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 5, pages 335–367. Elsevier Science and MIT Press, 2001.

[8] David Plaisted. Properties and extensions of alternating path relevance - I. arXiv preprint 1905.08842, 2019. `https://arxiv.org/abs/1905.08842`.

[9] S. Schulz. Analyse und Transformation von Gleichheitsbeweisen. Projektarbeit in Informatik, Fachbereich Informatik, Universität Kaiserslautern, 1993. (German Language).

[10] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.

[11] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, pages 735–743. Springer, 2013.

[12] Stephan Schulz, Simon Cruanes, and Petar Vukmirović. Faster, higher, stronger: E 2.3. In Pacal Fontaine, editor, *Proc. of the 27th CADE, Natal, Brasil*, LNAI. Springer, 2019. (to appear).

[13] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A Cross-Community Infrastructure for Logic Solving. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Proc. of the 7th IJCAR, Vienna*, volume 8562 of *LNCS*, pages 367–373. Springer, 2014.

[14] Geoff Sutcliffe. The TPTP Syntax BNF. `http://www.cs.miami.edu/~tptp/TPTP/SyntaxBNF.html`, 2016. (acccessed 2018-07-19).

[15] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[16] Geoff Sutcliffe and Evgeny Kotelnikov. Extended typed first-order form (TFX). http://www.cs.miami.edu/ tptp/TPTP/Proposals/TFXTHX.html, 2018. (acccessed 2018-07-19).

[17] Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Peter Baumgartner. The TPTP Typed First-order Form with Arithmetic. In Nikolaj Bjørner and Andrei Voronkov, editors, *Proc. of the 18th LPAR, Merida*, volume 7180 of *LNAI*, pages 406–419. Springer, 2012.

[18] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.