



Studienarbeitsthemen 2019/2020

Stephan Schulz

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

Inhaltsverzeichnis

1 Einführung	1
1.1 Rahmenbedingungen	2
2 Themen	3
2.1 $1+1=2$ - Einfache Arithmetik für E	3
2.2 <i>Step by Step</i> - Detaillierte Beweise für E	3
2.3 <i>Can you take a hint?</i> Verbesserte Integration von Hinweisen und Lemmas für die Beweissuche von E	4

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 [7, 6, 5], 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 in der Regel nur vom Beweiser Vampire der Universität Manchester geschlagen worden.

Konkret sind automatische Theorembeweiser Computerprogramme, die nachweisen, dass bestimmte Aussagen zwingend aus einer gegebenen formalen Beschreibung 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 LGPL 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 $1+1=2$ - 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 (=, ≥, >, ...). 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

[6, 5, 8, 9]

Team

2 Studierende

2.2 *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

[6, 5, 4]

Team

1–2 Studierende

2.3 *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 [3] und seinen Nachfolger Prover9 [2] entwickelt, und wurde z.B. für den berühmten Beweis der Robbins-Vermutung [2] verwendet, kommt aber auch in der aktuellen mathematischen Forschung zum Einsatz [1].

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

[10, 1, 6, 5]

Team

1 Studierender

Literatur

- [1] Michael Kinyon, Robert Veroff, and Petr Vojtěchovský. 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.
- [2] William W. McCune. Prover9 and Mace4. <http://www.cs.unm.edu/~mccune/prover9/>, 2005–2010. (accessed 2016-03-29).
- [3] 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.
- [4] S. Schulz. Analyse und Transformation von Gleichheitsbeweisen. Projektarbeit in Informatik, Fachbereich Informatik, Universität Kaiserslautern, 1993. (German Language).
- [5] Stephan Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
- [6] 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.

- [7] 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).
- [8] 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.
- [9] 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.
- [10] Robert Veroff. Using hints to increase the effectiveness of an automated reasoning program: Case studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.