Universität Potsdam
Mathematisch
Naturwissenschaftliche Fakultät

Modernisierung von Legacy Beweissystemen

BACHELORARBEIT

im Bachelorstudium
Informatik/Computational Science

Eingereicht von:
Alexander Benedict Behrens

Angefertigt am:
Institut für Informatik und Computational Science
Lehrstuhl für Theoretische Informatik

Themensteller/1. Gutachter:
Prof. Dr. Christoph Kreitz

2. Gutachter:
Dipl.-Inf. Mario Frank

Potsdam, der 30. September 2019

Selbstständigkeitserklärung

Hiermit versichere ich diese Arbeit selbstständig und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen der Arbeit, die ich aus diesen Quellen und Hilfsmitteln dem Wortlaut oder dem Sinn nach entnommen habe, sind kenntlich gemacht und im Literaturverzeichnis aufgeführt. Weiterhin versichere ich, dass weder ich noch andere diese Arbeit weder in der vorliegenden, noch in einer mehr oder weniger abgewandelten Form als Leistungsnachweise in einer anderen Veranstaltung bereits verwendet haben oder noch verwenden werden.

Die „Richtlinie zur Sicherung guter wissenschaftlicher Praxis für Studierende an der Universität Potsdam (Plagiatsrichtlinie) - Vom 20. Oktober 2010“ habe ich zur Kenntnis genommen. [1]

Datum, Ort

Unterschrift

[1] Im Internet unter:
https://www.uni-potsdam.de/am-up/2011/ambek-2011-01-037-039.pdf

Zusammenfassung

Das Ziel der Bachelorarbeit *Modernisierung von Legacy Beweissystemen* ist es, einen bestehenden, mit aktueller Software nicht funktionsfähigen automatischen Theorembeweiser, welcher Arithmetik unterstützt auf einem aktuellem (Linux-)System mit vollem Funktionsumfang *Typed first-order theorems with arithmetic* (im folgenden TFA) lösen zu lassen. Die Anpassungen sind zum einen den *Omega Calculator* (im folgenden OC) aus dem gesamten *The Omega Project* (im Folgenden TOP) zu extrahieren. Dazu werden nicht benötigte Dateien, die nach einer Abhängigkeitsermittlung nicht erfasst wurden, aus dem Verzeichnis entfernt. Zum anderen ist es notwendig Anpassungen am *leanCoP*-Ω (im folgenden LCO) Code vorzunehmen, damit die neuste Version von SWI-Prolog, einem Dialekt der Programmiersprache Prolog, unterstützt wird. Hierfür werden die dokumentierten Veränderungen der SWI-Prolog Releases übernommen. Alle Probleme die der Theorembeweiser, in der alten Form, bei der CASC-J5 gelöst hat wurden auch von der jetzt lauffähigen Variante gelöst. Ein fundierter Vergleich zum *CADE ATP System Competition* (CASC)-Wettbewerbssystem ist nicht möglich aufgrund der möglichen Disparität der Hardwarekonfiguration.

Inhaltsverzeichnis

Abkürzungsverzeichnis

FOL	First Order Logic (Logik erster Stufe)
FOF	First Order Formula
TFA	Typed first-order theorems with arithmetic
LC	LeanCoP
LCO	LeanCoP-Ω
OC	Omega Calculator
TOP	The Omega Project
CASC	CADE ATP System Competition
ML	Meta Language
SML	Standard ML
GUI	Graphical User Interface (engl. für grafische Benutzeroberfläche)
IDE	Integrated Development Environment (engl. für integrierte Entwicklungsumgebung)
FFI	foreign function interface
KNF	konjunktive Normalform

Vorwort

„Wir [Informatiker] sind die wahren Dichter, nur müssen wir das, was unsere Phantasie schafft, noch beweisen."

— Leopold Kronecker

Zuerst möchte ich mich bei meinem Betreuer Dipl.-Inf. Mario Frank für die rat- und tatkräftige Unterstützung bedanken. Ebenfalls bedanken möchte ich mich bei Prof. Dr. Christoph Kreitz, mir die Zeit gegeben zu haben ein Thema zu finden, welches mich wirklich interessiert und meinen Fähigkeiten entspricht.
Das Zitat von Leopold Kronecker ist sehr passend, weil er ein deutscher Mathematiker war der sich unter anderem mit Algebra und Zahlentheorie beschäftigte. Zudem ist die Analogie zu den Systemen, die Theoreme beweisen, stimmig. Er wurde am 7. Dezember 1823 in Liegnitz geboren und starb am 29. Dezember 1891 in Berlin [1].
Im ersten Semester meines Bachelor-Studiums Informatik / Computational Science zeigte uns unserer Professor in der Veranstaltung Theoretischer Informatik I was die möglichen Folgen von nicht verifizierter Software oder Hardware in sicherheitskritischen Bereichen sein können. Es kommt zu Millionen Dollar teuren Schäden oder es sterben Menschen. Das zeigte mir, wie wichtig fehlerfreie Softwareentwicklung und Verifikation ist. Vielmehr wie wichtig gut ausgebildete Informatiker sind. Darum entschied ich mich für eine Bachelorarbeit im Bereich der theoretischen Informatik. Mit dem Aufbaumodul Inferenzmethoden lernte ich alle nötigen Grundlagen. Nach der mündlichen Prüfung zu Inferenzmethoden saß ich mit Prof. Kreitz und Mario Frank noch zusammen und nutzte die Möglichkeit, mein Interesse für eine Bachelorarbeit in diesem Thema anzuzeigen. Nach langem Suchen hatten wir ein passendes und interessantes Thema gefunden. Ich hatte sehr viel Spaß in meinem Bachelorstudium und habe es sehr genossen. Zudem habe ich viele nette Leute kennen gelernt und mir noch mehr Wissen angeeignet. Doch es wird Zeit, das zu einem runden Ende zu bringen und mit voller Energie in das Masterstudium zu gehen.

1 Einleitung

Motivation

Am 26.06.1988 in Habsheim verweigerte die Software des Air France Airbus A 320 die Not-Kommandos des Piloten - vier Menschen verunglückten tödlich.[1] Am 14.09.1993 in Warschau verweigerte die Software des Lufthansa Airbus A 320 den Umkehrschub - zwei Menschen verunglückten tödlich.[2] Im Oktober 1994 führte die fehlerhafte Umsetzung eines Algorithmus in Prozessortabellen im Pentium I Prozessor zu einem 450 Millionen Dollar Schaden.[3] Diese drei Beispiele stehen stellvertretend für aber tausende Unglücke die durch fehlerhafte Software verursacht wurden und mit richtiger Softwareverifikation hätten verhindert werden können [2].
Es ist praktisch nicht möglich alle erdenklichen Zustände zu testen, somit bleibt immer die Ungewissheit, ob unter den nicht bedachten Möglichkeiten die dabei sind die Menschen verletzen oder sogar töten. Natürlich sind hohe wirtschaftliche Schäden ebenso ein Kriterium. Somit ist Softwareverifikation gerade für sicherheitskritische Bereiche unbedingt notwendig. Da in diesen Bereichen die Richtigkeit oder Korrektheit der Software bewiesen sein sollte, reicht eine Validierung durch Testen der Software alleine nicht aus. Das heißt es muss sichergestellt sein, dass die Software das macht wozu sie geschrieben wurde (Validierung) und das ausnahmslos unter allen Umständen (Verifikation). Bei der Softwareverifikation gibt es gewöhnlich zwei Ansätze. Beim ersten Ansatz wird die Korrektheit einer bestehenden Software verifiziert. Der Zweite und weitaus elegantere Ansatz ist die Programmsynthese. Hier wird aus der Spezifikation und dem dazugehörigen Beweis direkt das Programm erzeugt, wodurch die Korrektheit per Definition bewiesen ist.
Automatische Beweiser sind nicht nur in der Softwareverifikation relevant, es gibt noch viele weitere Anwendungsbereiche, wie zum Beispiel das Beweisen von Theorien. Mit Hinblick auf *LeanCoP*-Ω (LCO) speziell im Bereich der Zahlentheorien oder auch Numerik - also allgemein Arithmetik. Das für die Beweisführung von Problemen Computer verwendet werden, war nicht immer selbstverständlich. Das wohl bekannteste und erste Problem das mit Hilfe eines Computers gelöst wurde ist das vier Farben Problem [3]. Es besagt, dass jede Karte, egal wie komplex sie ist mit maximal vier Farben eingefärbt werden kann, sodass zwei angrenzende Länder immer eine unterschiedliche Farbe haben. Ein weiteres, jedoch weitaus einfacheres Problem ist das Damenproblem,

[1] Zeit Archiv (letzter Zugriff 05.09.2019):
`https://www.zeit.de/1988/27/absturz-eines-ueberfliegers`

[2] Unfallgutachten, University of Bielefeld, Research group of Prof. Peter B. Ladkin, Ph.D. (letzter Zugriff 05.09.2019):
`http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html`

[3] NYTimes Archiv (letzter Zugriff 05.09.2019):
`https://www.nytimes.com/1994/11/24/business/company-news-flaw-undermines-accuracy-of-pentium-chips.html`

welches auch mit Hilfe von Arithmetik gelöst werden kann. Das Problem lautet: wie viele Schach-Damen passen auf ein beliebig großes Schachbrett mit den Maßen $n \cdot n$ Feldern, sodass sich die Damen nicht gegenseitig schlagen. Dieses Damenproblem befindet sich auch unter den im CASC verwendeten Problemen aus der TPTP Problem-Bibliothek [4]. Die Arithmetikbehandlung ist zum einen wichtig, da viele Probleme, wie das Damenproblem, arithmetische Theorien beinhalten die zur Lösung Beweiser mit Arithmetik-Unterstützung benötigen. Zum anderen können sie zum Beispiel in Compiler bei der Schleifenoptimierung unterstützen.
LCO hat mit der Zeit an Aufmerksamkeit verloren, jedoch nicht an Wichtigkeit. Um die Konkurrenzfähigkeit, auch von anderen Softwarelösungen, weiterzuentwickeln ist es wichtig und richtig LCO weiterhin zu erhalten. Ferner ist es wichtig Omega zu erhalten, da es eine Alternative zu anderen Verfahren ist. Obwohl der *Omega Calculator* (OC) im Worst Case eine schlechtere Zeitkomplexität hat als andere Verfahren, konnten die Entwickler zeigen, dass der OC bei der Performance im Average Case eine realistische Alternative bietet.[5]
Abschließend noch eine Suggestiv-Frage: Betrachtet man Tragödien, die durch Hardware- oder Softwarefehler mit verursacht wurden, stellt sich aus ethischer Sicht die Schuldfrage. Trägt der Programmierer, der in der letzten Crunchtime vor der Deadline das Programm fertig geschrieben hat die Schuld, der Arbeitgeber der seine „EDV“-Abteilung am liebsten outsourcen würde oder die Endnutzer die das System nur falsch verwendet haben? Es hat sich im Bereich der System- und Software-Verifikation sowie der Programmsynthese schon sehr viel getan. Aber viele auch aktuelle Beispiele zeigen, dass noch viel mehr in sehr gut ausgebildete Informatiker und gute Software investiert werden muss.

Zielstellung

Die Kernaussage des Titels *Modernisierung von Legacy Beweissystemen* (Modernisierung von alten Systemen) lässt sich auf viele Anwendungsbereiche übertragen. Zum Beispiel ist das auch eine häufig gestellte Frage in der Wirtschaft. Alte Systeme modernisieren oder neu entwickelt? Nun wurde in Motivation schon recht ausführlich das *Warum?* besprochen. Jedoch nicht aus der eben erläuterten Perspektive - modernisieren oder neu? Das System bestehend aus dem OC und der angepassten Version von *LeanCoP* (LC). Es hat in der CASC-J5, der damals neu eingeführten Testdivision *Typed first-order theorems with arithmetic* (TFA) [6], den ersten Platz nach gelösten Problemen belegt.[4] Die CASC ist ein Wettbewerb bei dem sich die automatischen Beweissysteme gegeneinander messen. Die Hauptaufgabe besteht darin, den LCO-Beweiser auf einem aktuellen System in seinem Funktionsumfang wiederherzustellen und anschließend zu evaluieren. Zudem sollte der OC zum größten Teil aus dem *The Omega Project* (TOP) extrahiert werden. Das hätte den Vorteil, dass das Projekt viel übersichtlicher werden würde und Anpassungen einfacher wären. Sollte LCO auf

[4]CASC J5 Ergebnisse Zusammenfassung (letzter Aufruf: 09.09.2019):
`http://www.tptp.org/CASC/J5/WWWFiles/ResultsSummary.html`

dem aktuellen System lauffähig sein ist auf eine aktuelle SWI-Prolog (ein Dialekt der logischen Programmiersprache Prolog [7]) Version zu modernisieren, mit allen nötigen Anpassungen.

Aufbau der Arbeit

In der Bachelorarbeit *Modernisierung von Legacy Beweissystemen* werden die Peano-, Presburger- und Induktionsfreie Arithmetik beleuchtet, ebenso wie die Entscheidungsprozeduren Elimination of Quantifiers, Fourier-Motzkin Variablen Elimination, Omega-Test und Sup-Inf. Außerdem werden der Sequenzen- [8, S.110, S.211-212], Tableau- [8, S.110-114] und Konnektionskalkül [8, S.39-55] erklärt und anschließend eine Auswahl von jeweils drei interaktiven und automatischen Beweisern vorgestellt. Die Auswahl der interaktiven Beweiser beschränkt sich auf Isabelle, Coq und Nuprl, da diese drei von großen Wirtschaftskonzernen verwendet werden oder sehr verbreitet sind. So verwendet zum Beispiel IBM Coq[5] und Isabelle und Nuprl ist in der europäischen und amerikanischen Industrie weit verbreitet. Des Weiteren sind sie in der TOP 10 der Beweiser, in denen die meisten der TOP 100 der mathematischen Sätze formalisiert wurden.[6] Die Wahl der automatischen Theorembeweisern umfasst leanCoP-Ω, SPASS+T und Vampire, weil diese drei Arithmetik unterstützen oder entsprechende Module enthalten.
Nach dem Kapitel zur Theorie wird im Kapitel Methodik die Analyse des Omega Projektes [9], der Schnittstelle zu leanCoP und leanCoP selbst durchgeführt. Anschließend wird daraus ein Konzept eruiert, um das Konvolut, als automatischen Theorembeweiser zusammengefasst, zu reaktivieren und zu modernisieren. Das Vorgehen wird im nächsten Kapitel Durchführung der Anpassung als Implementierungsschritt niedergeschrieben. Für die anschließende Evaluation werden die Probleme, gegen die das schon einmal lauffähige Beweissystem bei der CASC-J5 [10] antreten musste, verwendet. Diese 75 getypten arithmetischen Theoreme müssen von dem reaktivierten Beweissystem und dem modernisierten Beweissystem gelöst werden. Diese beiden Systeme unterscheiden sich lediglich in der verwendeten Prolog-Version. Beendet wird die Arbeit mit dem Ausblick, der sowohl weiterführende Gedanken und Ideen enthält, als auch abschließende Worte findet.

[5] Coq für das framework S.40 (letzter Aufruf: 27.09.2019):
`http://www.cse.chalmers.se/research/group/logic/Types/types-activity-report2.pdf`

[6] Formalizing 100 Theorems, Radboud Universiteit Nijmegen (NL), ICIS (letzter Aufruf: 09.09.2019):
`http://www.cs.ru.nl/~freek/100/`

2 Theoretischer Hintergrund

In diesem Kapitel werden die für ein besseres Verständnis benötigten Theorien und Konzepte vorgestellt. Begonnen wird mit den drei arithmetischen Theorien Peano-Arithmetik, Presburger-Arithmetik und Induktionsfreier Arithmetik. Die Presburger-Arithmetik ist die Peano-Arithmetik ohne Multiplikation, die auch im OC verwendet wird. Der OC wird im späteren Verlauf genau untersucht und beschrieben. Nach den arithmetischen Theorien werden vier relevante arithmetische Entscheidungsprozeduren vorgestellt. Dies sind Elimination of Quantifiers, Fourier-Motzkin Variable Elimination, der Omega-Test und Sup-Inf. Das Verfahren Elimination of Quantifiers ist sehr ineffizient, da sich die Formeln dabei stark vergrößern. Die Grundidee findet sich aber in anderen Verfahren wieder. Bei der Fourier-Motzkin Variable Elimination wird die Erweiterung für die natürlichen Zahlen vorgestellt, welche der Omega-Test verwendet. Dieser wiederum wird vom OC verwendet. Als letztes Verfahren wird Sup-Inf vorgestellt. Danach werden der Sequenzenkalkül und der Tableau-Kalkül vorgestellt, außerdem wird der von LC verwendete Konnektionskalkül vorgestellt. Durch die maschinennahe Funktionsweise ist er, genau wie Resolution, gut für Theorembeweiser geeignet. Abschließend werden die drei interaktiven Beweissysteme Isabelle, Coq und Nuprl, ebenso wie die drei automatischen Beweissysteme leanCoP-Ω, SPASS+T und Vampire vorgestellt.

2.1 Arithmetische Theorien

Peano-Arithmetik

Die Peano-Arithmetik ist eine Theorie, welche die Menge der natürlichen Zahlen $\mathbb{N}$ mit Hilfe der fünf folgenden Peano-Axiome beschreibt. Sie wurden erstmals 1889 von Giuseppe Peano formal definiert.[11]

Axiome der Peano-Arithmetik [1] [2] [12, S.218ff]

1. $0 \in \mathbb{N}$ (0 ist hier eine natürliche Zahl) (1)
2. Für jedes $x \in \mathbb{N}$, existiert genau ein $x' \in \mathbb{N}$, genannt Nachfolger von x (2)
3. $x' \neq 0$ (0 ist kein Nachfolger einer natürlichen Zahl) (3)
4. $x = y$ genau dann, wenn $x' = y'$ (4)
5. Wenn $M \subseteq \mathbb{N}$ und $0 \in M$ und $x \in M$ ist $x' \in M$, d.h. $M = \mathbb{N}$. (5)

[1] Peano Arithmetic (letzter Aufruf: 25.06.19):
`https://planetmath.org/peanoarithmetic`
[2] Axiomensystem nach Peano (letzter Aufruf: 01.07.19):
`https://mathepedia.de/Axiomensystem_nach_Peano.html`

Im ersten Axiom wird festgelegt, dass die Ziffer 0 eine Zahl und Element der beschriebenen Menge ist. Diese definierte Menge ist die Menge der natürlichen Zahlen $\mathbb{N}$. Das nächste Axiome beschreibt die Nachfolgeroperation. Sie definiert, dass jede Zahl aus dieser Menge einen Nachfolger hat. Somit hat auch jede Zahl einen Vorgänger, bis auf die durch das dritte Axiom beschriebene Ausnahme, die Null. Das heißt sie ist das kleinstes Element der Menge oder auch das Startelement. Das vierte Axiom sagt aus, dass zwei Zahlen, die den gleichen Nachfolger haben, identisch sind. Das letzte Axiom ist das Induktionsaxiom, welches aussagt, dass man vom Startelement auf jeden Nachfolger schließen kann.
Mit Hilfe dieser Axiome lassen sich die Addition, die Multiplikation und die "kleinergleich"-Relation definieren. Für die Addition gilt Assoziativität und Kommutativität, die formale Definition sieht wie folgt aus:

$$x+1=x' \qquad \text{für alle } x \in \mathbb{N} \tag{6}$$
$$x+y'=(x+y)' \qquad \text{für alle } x,y \in \mathbb{N} \tag{7}$$

Für die Multiplikation kann ebenfalls Assoziativität und Kommutativität, sowie Distributivität über Addition bewiesen werden. Formal sieht die Multiplikation wie folgt aus:

$$x \cdot 1 = x \qquad \text{für alle } x \in \mathbb{N} \tag{8}$$
$$x \cdot y' = x \cdot y + x \qquad \text{für alle } x,y \in \mathbb{N} \tag{9}$$

Zusammengefasst ist $(\mathbb{N},+)$ eine kommutative Halbgruppe mit neutralem Element. Dies ist das Nullelement des Halbringes $(\mathbb{N},+,\cdot)$. $(\mathbb{N},\cdot)$ ist eine Halbgruppe mit neutralem Element, welches als Einselement des Halbringes $(\mathbb{N},+,\cdot)$ bezeichnet wird. Daraus prägt sich der Halbring $(\mathbb{N},+,0,\cdot,1)$. Unter Hinzunahme der Ordnungsbeziehung lässt sich die initiale oder nach unten Beschränkte lineare Ordnung zeigen. Somit ist der Halbring positiv geordnet und wird durch das Sechstupel $(\mathbb{N},+,0,\cdot,1,\leq)$ beschrieben [12, 13].

Presburger-Arithmetik

Die Presburger-Arithmetik ist kurz gesagt die Peano-Arithmetik ohne Multiplikation, also eine etwas schwächere Arithmetik. Sie umfasst die natürlichen Zahlen mit Null, sowie die drei Funktionen S als unäre Nachfolgerfunktion mit $S(x)=x'$, $+$ als binäre Addition und $<$ als binäre Relation. Die Presburger-Arithmetik ist anders als die Peano-Arithmetik entscheidbar und vollständig, darum ist sie besonders wertvoll für automatische Beweissysteme.

Axiome der Presburger-Arithmetik [3] [14]

1. $x' \neq 0$ (1)
2. $x' = y'$ genau dann, wenn $x = y$ (2)
3. $x + 0 = x$ (3)
4. $x + y' = (x + y)'$ (4)
5. Für jede FOF $P(x), P(0) \wedge \forall x[P(x) \rightarrow P(x+1)] \rightarrow \forall x P(x)$ (5)

Das erste Axiom sagt aus, dass es keine Zahl gibt, dessen Nachfolger 0 ist, somit ist 0 das kleinste Element. Das nächste Axiom bestimmt, dass zwei gleiche Zahlen auch immer den gleichen Nachfolger haben, das heißt, dass die Nachfolgerfunktion bijektiv abbildet. Das dritte Axiom beschreibt 0 als neutrales Element der Addition. Danach wird beschrieben, dass die Reihenfolge, in der man die Nachfolgerfunktion anwendet irrelevant für das Ergebnis ist. Das heißt, das Inkrementieren eines Summanden y im Term $x + y'$ ergibt die selbe Summe wie das Inkrementieren der Summe von $x + y$, also $(x + y)'$. Für jede *First Order Formula* (FOF) gilt das fünfte Axiom, nämlich das Induktionsaxiom.

Induktionsfreie Arithmetik

Die induktionsfreie Arithmetik [15, 16] wurde erstmals im Jahr 1978 von Constable und O'Donnel formal definiert. Wie der Name sagt ist es eine Arithmetik ohne Induktion und unterscheidet sich damit von der Peano-Arithmetik und Presburger-Arithmetik. Sie umfasst die Addition, die Subtraktion, die Multiplikation für ganze Zahlen und die Ordnungsrelation. Die relationalen Operatoren können nur komplett substituiert werden. Daher ist induktionsfreie Arithmetik entscheidbar und ebenfalls gut für das Theorembeweisen geeignet.

2.2 Arithmetische Entscheidungsprozeduren

In diesem Unterkapitel der theoretischen Grundlagen werden kurz vier arithmetische Entscheidungsprozeduren vorgestellt. Dazu gehören *Elimination of Quantifiers*, *Fourier-Motzkin Variable Elimination for Integer Programming Problems*, *Omega-Test* und *Sup-Inf*. Der *Omega-Test* ist das Verfahren hinter dem OC bzw. Teil der Omega Library des TOP und nutzt die *Fourier-Motzkin Variable Elimination*. *Elimination of Quantifiers* ist vergleichbar zur *Fourier-Motzkin Variable Elimination*, jedoch werden die Formeln durch das Verfahren stark aufgebläht. Das *Sup-Inf*-Verfahren nutzt einen komplett anderen Ansatz und wird hier als interessante Alternative vorgestellt.

[3] Presburger-Arithmetic (letzter Aufruf: 25.06.19):
`https://planetmath.org/presburgerarithmetic`

Elimination of Quantifiers

Elimination of Quantifiers [17, 18, 19] (engl. für Elimination von Quantoren) ist eine arithmetische Entscheidungsprozedur, die zum Beispiel schon 1967 von Kreisel and Krivine [20] oder 1968 von Hilbert und Bernays [12] entdeckt wurde. In dem Verfahren werden alle freien Variablen allquantifiziert, um danach die Formel mit Allquantoren in eine äquivalente Formel mit Existenzquantoren zu transformiert. Dann werden alle Quantoren eliminiert bis ein System ohne Variablen entsteht, welches zu wahr oder falsch evaluiert. Also findet eine erfüllbarkeitsäquivalente Transformation statt, das heißt die Formel ohne Variablen evaluiert nur zu wahr/falsch, wenn sie vorher zu wahr/falsch evaluierte.

Fourier-Motzkin Variable Elimination

Im Jahr 1826 entwickelte Jean Baptiste Joseph Fourier [21] ein Verfahren, mit dem man durch wiederholtes Eliminieren von Variablen lineare Ungleichungssysteme lösen kann. 1919 wurde es von Dines [22], 1936 von Motzkin [23] und 1963 von Dantzig [24] wiederentdeckt und aufgegriffen. Bei dem Fourier-Motzkin-Verfahren [25] wird eine Variable, zum Beispiel x, isoliert. Die Ungleichung wird mit ihrem Koeffizienten dividiert, um ein äquivalentes System ohne x zu erhalten. Das Verfahren wurde für lineare Ungleichungssysteme entwickelt und von Williams 1976 für Systeme auf natürliche Zahlen angepasst. In "Fourier-Motzkin Variable Elimination for Integer Programming Problems"[26] beschreibt er alle Anpassungen.

Omega-Test

Der Omega-Test[5] ist ein von W. Pugh 1991 vorgestellter Algorithmus auf ganzen Zahlen. Er kann unter anderem Array-Verweise auf Abhängigkeiten überprüfen und zudem zeigen unter welchen Bedingungen die Abhängigkeiten existieren. Darüber hinaus kann er sowohl Integer-Probleme entscheiden als auch vereinfachen. Die Zeitkomplexität der Verifikation von Presburger-Formeln hat die nichtdeterministische untere Grenze von $O(2^{2^n})$ und die nichtdeterministische obere Grenze von $O(2^{2^{2^n}})$. Jedoch wurde in [5] gezeigt, dass der Omega-Test eine Alternative zu herkömmlichen Näherungsverfahren in Compilern ist.
Das Verfahren lässt sich in sieben Schritte unterteilen. Im ersten Schritt werden alle Gleichungen eliminiert, sodass nur Ungleichungen übrig bleiben. Danach wird getestet, ob sich die resultierenden Ungleichungen widersprechen. Wird einer gefunden gibt es keine Lösung, wenn es keine Widersprüche gibt wird Schritt drei ausgeführt. Dieser formt dichte Ungleichungen in äquivalente Gleichungen um und geht zu Schritt eins. Dichte Ungleichungen sind zum Beispiel $6 \leq 3x+2y$ und $3x+2y \leq 6$, da diese erfüllbarkeitsäquivalent zur Gleichung $6 = 3x+2y$ transformiert werden können. Sind keine dichten Ungleichungen vorhanden werden im vierten Schritt alle redundanten Ungleichungen entfernt. Sind, zum Beispiel, die Ungleichung $x+2y \geq 0$ und $x+2y \geq 5$

gegeben ist die erste Ungleichung redundant. Im fünften Schritt wird getestet, ob das Problem maximal eine Variable enthält. Ist das der Fall, ist das Problem trivial und hat eine Lösung, wenn nicht wird im sechsten Schritt die Fourier-Motzkin Variablen Elimination ausgeführt. Im letzten Schritt werden einige Tests für Spezialfälle, die sonst nicht erfasst werden, durchgeführt. Genauere Informationen zu den "Single Variable Per Constraint"-, "Acyclic"-, "Loop Residue"- und "Delta"-Tests und wann sie verwendet werden gibt es in Pugh [5].

Sup-Inf

Das "Sup-Inf"-Verfahren ist eine Methode, um die Gültigkeit von Formeln in Presburger-Arithmetik ohne Existenzquantoren zu zeigen. Es wurde 1975 von W. W. Bledsoe [27] vorgestellt und 1977 von R. E. Shostak [28] aufgegriffen und weiterentwickelt. Die Weiterentwicklung kann nicht nur auf Gültigkeit prüfen sondern auch auf Ungültigkeit. Erkennt es eine Formel als ungültig, so wird ein Gegenbeispiel gegeben. Der Kern des Verfahrens besteht aus den zwei Algorithmen Sup und Inf. Sup bestimmt das Supremum, also die größte untere Schranke der Variablen und Inf bestimmt das Infimum, also die kleinste obere Schranke der Variablen. Da das "Sup-Inf"-Verfahren einen Widerspruchsbeweis führt wird versucht die negierte Formel F zu widerlegen. Sollte es zu einem Widerspruch kommen ist die Ausgangsformel F gültig. Zu einem Widerspruch kommt es, wenn zwischen den Schranken keine ganzzahlige Lösung vorhanden ist oder das Supremum über dem Infimum liegen.

2.3 Kalkülarten

In diesem Abschnitt beziehe ich mich auf die Lehrbücher von W. Bibel [8, 29] als Quellen. Die Grundbestandteile des formalen Kalküls sind Formeln, Schlussregeln und Axiome. Dabei wird in den allermeisten Fällen der Zusammenhang zwischen Formeln und ihrer Bedeutung durch eine Interpretation hergestellt. Die arabischen Ziffern sind auch nur Symbole die erst durch eine Bildungsvorschrift und eine gelernte Bedeutung als Zahlen interpretiert werden können, vergleichbar mit dem formalen Kalkül. Jedoch gibt es verschiedene Kalküle, von denen im folgenden drei wichtige vorgestellt werden. Als erstes wird der Sequenzenkalkül vorgestellt, der die klassische Grundlage bietet. Der Tableau-Kalkül ist ein verdichteter Sequenzenkalkül, weil er direkt auf dem Syntaxbaum operiert und deshalb weniger Redundanz verursacht. Der Konnektionskalkül arbeitet direkt auf der Matrix von Literalen und befindet sich die Formel in *konjunktive Normalform* (KNF) (engl. CNF für conjunctive normal form) entsteht keine Redundanz. Der Matrixbeweis ist eine Verdichtung des Tableau-Beweises. Daher verwendet LC den Konnektionskalkül, um Redundanz und Speicherplatzverbrauch zu minimieren. Ein weiteres, in Beweisern häufig verwendetes Verfahren, ist die Resolution [30, 31]. Da die Behandlung der Resolution zu weit weg vom Thema führen würde wird sie nicht erläutert.

Sequenzenkalkül

Der Sequenzenkalkül [8] umfasst Schlussregeln, die die möglichen Umformungen der Sequenzen vorgeben. Wobei die Sequenz eine Kette von Zeichen mit folgender Form ist: $\Gamma \vdash \Phi$. Auf beiden Seiten der Ableitbarkeitsrelation stehen endliche Folgen von Formeln der Prädikatenlogik erster Stufe. Es ist ebenfalls möglich, das Γ als Prämisse (*Antezedenz*) oder Φ als Konklusion (*Sukzedenz*) die leere Folge enthält. Denn der Beweis einer Formel F besteht darin mit leerer *Antezedenz* zu starten, damit die Sequenz $\vdash F$ abgeleitet werden kann. Dafür werden Axiome und Regeln benötigt, die als Hilfsmittel dienen. Diese Regeln haben die Form $\frac{\alpha}{\beta}$, wobei α aus einer Sequenz und β aus maximal zwei Sequenzen besteht. In dieser Form ist α die Prämisse und β die Konklusion. Das freie Variablen nicht gleichzeitig in ihrer Formel oder Sequenz gebunden auftreten dürfen und umgekehrt ist die einzige Einschränkung. Γ, Δ, Φ und Ψ stehen für Folgen von Formeln und A, B, ... stehen für einzelne Formeln. A und S in der Benennung der Regeln stehen für *Antezedenz* beziehungsweise für *Sukzedenz*. Da die Konklusion wahr sein kann wenn die Prämisse falsch ist, jedoch wahr sein muss, wenn die Prämisse wahr ist. Beim analytischen Kalkül, zum Beispiel der Refinement Logic werden Axiome aus der Formel abgeleitet (Top-Down Entwicklung) für ein zielorientiertes Vorgehen [32]. Valide Schlussregeln in der Refinement Logic kann man in Gleichung 6.1 und Gleichung 6.2 im Anhang sehen. Die Regeln können ebenfalls zu einem synthetischen Sequenzenkalkül gehören, es würde sich lediglich die Vorgehensweise invertieren. Das heißt, die Formel würde aus den Axiomen abgeleitet (Bottom-Up Entwicklung) werden und α und β würden die Position tauschen.

Tableau-Kalkül

Der Tableau-Kalkül [8, 33] kann für einen Widerspruchsbeweis verwendet werden. Das heißt es wird das Komplement einer Formel angenommen. Sollten alle Zweige des Beweisbaumes widersprüchlich sein ist die Gültigkeit der Originalformel bewiesen. Dabei wird die Redundanz, alle Formeln vom Beweis mitzuführen, aufgegeben. Die Formeln der linken und rechten Seite der Ableitbarkeitsrelation werden mit Polaritäten X^T für links und X^F für rechts gekennzeichnet. Die Regeln werden nach gleichen Eigenschaften gruppiert. Der erste Typ α umfasst die Regeln andL und orR welche nach Auflösung nur ein Teilziel liefern. Der nächste Typ β umfasst die Regeln andR und orL, da bei beiden nach Auflösung der Beweis verzweigt. Der dritte Typ γ umfasst allL und exR, die nach Auflösung die Variable mit einem Term instantiiert. Der letzte Typ δ umfasst allR und exL und diese deklarieren nach Auflösung eine neue Variable. Aus diesem Grund gibt es die Zusatzregel Typ δ vor Typ γ.
In Abbildung 2.1 kann man einen Beispiel-Tableau-Beweis für die Formel $\neg\forall xPx \vee (Pa \wedge Pb)$ sehen. In dem Beispiel sieht man, dass beide Zweige einen Widerspruch enthalten. Im linken Ast ist Pa zugleich wahr und falsch, im rechten Ast Pb zugleich wahr und falsch. Die Formel an der Wurzel wird negiert, gekennzeichnet durch das hoch gesetzte F. Der Hauptjunktor der Formel $(\neg\forall xPx \vee (Pa \wedge Pb))^F$ ist eine Disjunktion. Da die Formel negiert ist ergibt das die Regel orR der Kategorie α. Die resultierenden

Formeln werden untereinander mit einem Strich getrennt aufgeschrieben. Bei der Regel orR bekommen die Teilformeln die Polarität F. Beim Auflösen einer Negation dreht sich dich Polarität um, wie man anhand von $\forall xPx^T$ sehen kann. Bei der Dekomposition von $(Pa \wedge Pb)^F$ wird die β-Regel andR angewendet und beide Prädikate Pa und Pb beginnen einen eigenen Ast. Da wir die Formel $\forall xPx^T$, für alle x gilt das Prädikat P, haben und allL vom Typ γ ist können wir passende Instanzen erzeugen. Wie zum Beispiel Pa für den linken Ast und Pb für den rechten Ast. Somit ist jetzt jeder Ast widersprüchlich und die Ausgangsformel gültig. Die vollständigen Regeln sind in Abschnitt 6.1, Tabelle 6.1, Tabelle 6.1 und Tabelle 6.1 zu finden und aus Bibel [8, S.112] entnommen.

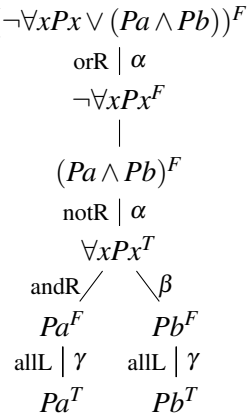

Abbildung 2.1: Beispiel Tableau-Beweis für $\neg\forall xPx \vee (Pa \wedge Pb)$

Konnektionskalkül

Das Konnektionskalkül [8] beschränkt sich auf die Konnektionen in einer Matrix zwischen den Klauseln. Das heißt komplementäre Literale schaffen eine Konnektion, welche alle Pfade durch diese beiden Literale abschließt, da diese nicht mehr für eine gültige Belegung infrage kämen. Wichtig hierbei ist zu erwähnen, dass sich nur auf die Konnektionen konzentriert wird. Würde man alle Pfade betrachten wären das bei einer Matrix mit m Literale pro Klausel und n Klauseln pro Matrix $m \cdot n$ mögliche Pfade. Da nicht alle Klauseln gleich viele Literale haben ist die genauere Darstellung $\prod_{i=0}^{n} m_i$. Dennoch ist dieser Ansatz tunlichst zu vermeiden, denn man hätte eine exponentielle Laufzeit und zusätzlich einen exponentiellen Speicherverbrauch.
Die Hauptbestandteile des Konnektionskalküls sind der Extensionsschritt, der Bereinigungsschritt und backtracking (engl. für Rücksetzschritt oder Rücksetztechnik). Zusätzlich gibt es noch die Reduktion und Restricted Backtracking (engl. für eingeschränktes Rücksetzen), welches von LC 2.0+ [34] verwendet wird. Beim Extensionsschritt wird ein Literal aus der gewählten Startklausel mit seinem Komplement verbunden. Aus der Klausel des Komplements wird wiederum ein Literal gewählt welches wie gehabt mit

seinem Komplement verbunden wird. Sollte kein Extensionsschritt mehr möglich sein, wird ein Bereinigungsschritt vorgenommen. Bei diesem geht man zur letzten Klausel mit unbenutzten Literalen zurück und macht mit diesen Literalen weitere Extensionsschritte, falls das möglich ist. Sollte nach dem letzten Bereinigungsschritt noch offene Literale übrig sein ist die Ausgangsformel nicht gültig. Die aktuelle Klausel oder die aktuelle Position in der Matrix wird mit einem senkrechten Pfeil nach oben gekennzeichnet. Das ausgewählte Literal wird durch eine Box um das Literal hervorgehoben und das verbundene Komplement mit einem Punkt markiert. Die Verbindung wird durch eine Linie zwischen den Beiden visualisiert. Noch nicht fertige Klauseln, also Klauseln mit noch nicht verwendeten Literalen werden durch einen diagonalen Pfeil nach rechts-oben gekennzeichnet, zu denen im Falle eines Bereinigungsschrittes zurück gegangen wird. Diese beiden Schritte bilden das Extensionsverfahren. Sollte in einer Situation eine geeignete Konnektion fehlen, kann das Verfahren durch einen Bereinigungsschritt zu einer der übrigen gekennzeichneten Klauseln zurück springen. Gibt es keine alternative Klausel so terminiert das Verfahren und die negierte Formel ist nicht gültig. Ein wichtiges Lemma besagt, *eine gültige (und nichtleere) Matrix enthält mindestens eine Klausel mit nur unnegierten Atomen und mindestens eine Klausel mit nur negierten Atomen* [8, S.42].

2.4 Interaktive Beweissysteme

Ein interaktives Beweissystem ist ein Programm mit dem man in einer formalen Sprache definierte Probleme manuell (Schritt für Schritt) oder halb-automatisch umformen beziehungsweise lösen kann. Die computerunterstützte Beweisführung wurde lange Zeit sehr kritisch betrachtet und wurde von einigen Wissenschaftlern abgelehnt [35, S.254-255]. Das erste bekannte Problem welches mit Hilfe von einem Computer gelöst wurde war das vier Farben Problem.[3]

Isabelle

Isabelle [36] ist allgemein ein Beweisassistent, der in Scala und der funktionalen Programmiersprache *Standard ML* (SML) geschrieben ist. SML unterstützt auch imperative Konstrukte und stammt von der Programmiersprache *Meta Language* (ML) ab. Ursprünglich wurde er an der University of Cambridge und der Technischen Universität München entwickelt. In Isabelle kann man mathematische Formeln in einer formalen Sprache definieren und mit vorhandenen Werkzeugen umformen und beweisen. Die Hauptanwendung ist die Formalisierung von mathematischen Beweisen und insbesondere die formale Verifikation, die den Nachweis der Richtigkeit von Computerhardware oder -software und den Nachweis der Eigenschaften von Computersprachen und -protokollen umfasst. Eine Vorschau des Standard-*Graphical User Interface (engl. für grafische Benutzeroberfläche)* (GUI) von Isabelle/jEdit ist in Abbildung 2.2 zu sehen. Sie dient als Texteditor mit *Integrated Development Environment (engl. für integrierte*

Entwicklungsumgebung) (IDE) für die Beweisführung.[4] Es ist lediglich die Fourier-Motzkin Variable Elimination in Isabelle implementiert worden [37], also die Ansätze zu Omega sind gegeben. Von anderen genannten Verfahren ist dem Autor keine Implementation bekannt.

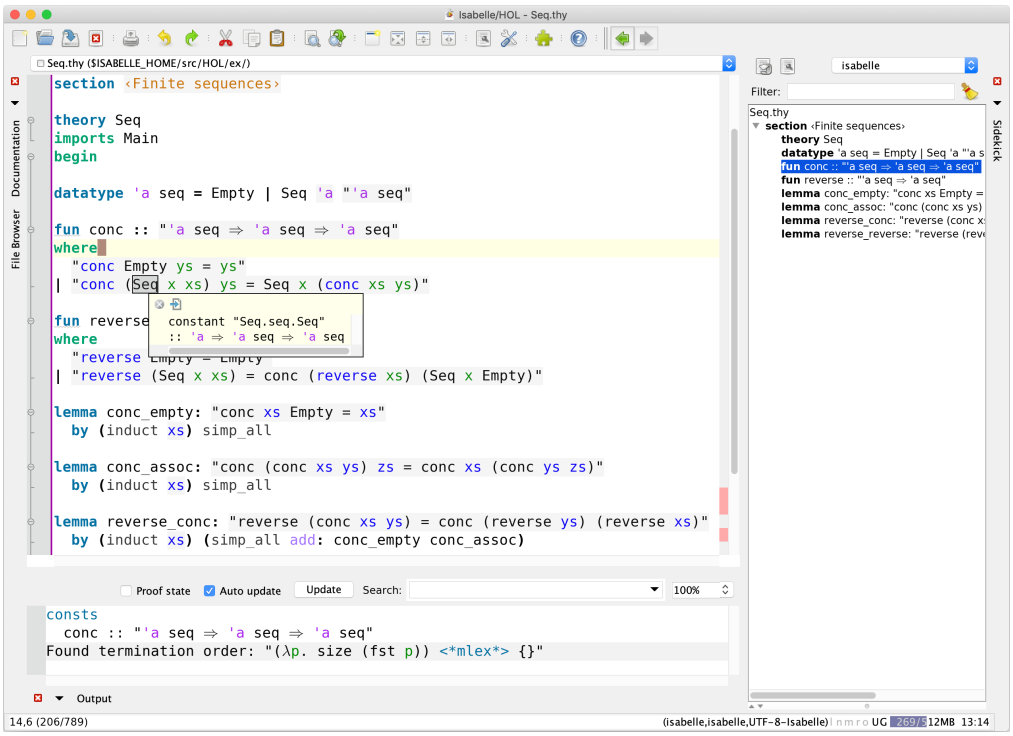

Abbildung 2.2: Standard-GUI von Isabelle/jEdit

Coq

Coq [38] ist ein Beweisassistent der bei der Handhabung formaler Beweisführungen hilft. Mathematische Formeln können in der von Coq zur Verfügung gestellten formalen Sprache Gallina definiert werden. Wie Isabelle bietet auch Coq einige Werkzeuge, wie ausführbare Algorithmen und Theoreme für die interaktive IDE zum erstellen von Beweisen. Coq ist das Ergebnis von 30 Jahren Forschung und Entwicklung. Es begann 1984 mit einer Implementierung des Calculus of Constructions in INRIA-Rocquencourt durch Thierry Coquand und Gérard Huet. Sieben Jahre später entwickelte Christine Paulin die Erweiterung Calculus of Inductive Constructions. Bei der Entwicklung der Coq-Funktionen waren etwa 50 Personen im Laufe der Zeit beteiligt. Die aktuellste Coq Version ist die Version 8.10, Stand September 2019.[5] Die aktuelle

[4] Offizielle Webseite von Isabelle (letzter Aufruf: 28.08.19):
`http://isabelle.in.tum.de/`

[5] Offizielle Webseite von Coq(letzter Aufruf: 28.08.19):
`https://coq.inria.fr/`

Coq Version hat den Omega-Test [39] teilweise implementiert. Was zum Großteil die Fourier-Motzkin Variable Elimination [37] ist.

Nuprl

Die erste Version von Nuprl wurde im Jahr 1984 veröffentlicht [40]. Seitdem wurde es einer dramatischen Entwicklung unterzogen und mehrfach signifikant modifiziert. 16 Jahre nach der ersten Veröffentlichung wurde das Programm sogar von Grund auf neu designt mit einer verbesserten Architektur [41]. Diese sollte den wachsenden Ansprüchen an das Programm gerecht werden. Es basiert auf der intuitionistischen Theorie von Martin-Löf, welches Formalisierung fundamentaler Konzepte der Mathematik, Datentypen und Programmierung unterstützt [42]. NUPRL 5, zu sehen in Abbildung 2.3, ist jetzt die neuste Version und ist allgemein ein System zum Erstellen von Beweisen. Dabei dient es als Framework (engl. für Gerüst oder auch System) für die Entwicklung von formalisiertem mathematischen Wissen, aber auch für die Synthese, Verifikation und Optimierung von Software. Der Beweiser ist wie Coq und Isabelle ein interaktiver, jedoch sind verschiedene Taktiken möglich, um katalysierend auf die Beweisführung zu wirken. Außer der taktisch orientierten Beweisführung unterstützt NUPRL ebenfalls Entscheidungsprozeduren, Evaluation von Programmen, von Benutzer erstellte Spracherweiterungen und eine erweiterbare Bibliothek aus verifiziertem Wissen aus vielen Anwendungsbereichen [43].[6] Die Sup-Inf Prozedur ist in Nuprl implementiert [37].

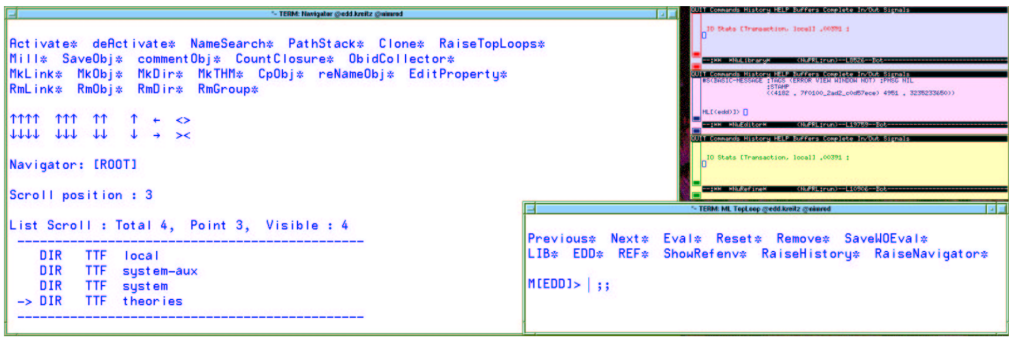

Abbildung 2.3: GUI von NuPRL5

2.5 Automatische Beweissysteme

leanCoP-Ω

LeanCoP-Ω ist ein automatischer Theorembeweiser mit Arithmetik-Unterstützung. Er setzt sich aus dem von Jens Otten und Wolfgang Bibel entwickelten LC-Prover[7] [44],

[6] Offizielle Webseite von Nuprl(letzter Aufruf: 28.08.19):
`http://www.nuprl.org/html/NuprlSystem.html`

[7] leanCoP Internetseite (letzter Aufruf: 11.09.2019):
`http://www.leancop.de/`

der Anbindung an den OC von Jens Otten, Holger Troelenberg und Thomas Raths und dem OC selbst von William Pugh et al. [45] zusammen. Der OC ist Teil des TOP, welches unter anderem auch die Omega Library [46] umfasst. LC ist ein auf dem bereits in Abschnitt 2.3 erläuterten Konnektionskalkül basierender Theorembeweiser, der ebenfalls restricted Backtracking unterstützt[47]. LCO hat keine anderen Entscheidungsprozeduren als dem Omega-Test implementiert.

SPASS+T

SPASS+T[8] ist eine Erweiterung des auf Superposition [48] basierenden Beweisers SPASS [49], welcher nun Teil der Toolsammlung SPASS Workbench[9] ist. Die Erweiterung umfasst Regeln für die Vereinfachung von arithmetischen Strukturen und nutzt ein SMT-Solver für Arithmetik, also ein SAT-Solver mit Arithmetik-Solver, ebenfalls werden freie Funktionssymbole als Blackbox verwendet. SPASS+T hat keine der vorgestellten Entscheidungsprozeduren implementiert, er nutzt Yices oder CVC3 SMT solver [50] für die arithmetischen Aspekte.

Vampire

Vampire[10] [51, 52] ist seit dem Start der Entwicklung 1994 sukzessive weiterentwickelt worden und ist einer der Theorembeweiser mit den meisten Auszeichnungen in der CASC. Der gerade in *First Order Logic (Logik erster Stufe)* (FOL) sehr performante Beweiser kommt auf 30 Auszeichnungen (Stand Sep. 2019). Für die Behandlung von Arithmetik nutzt Vampire einen SMT solver [50].

[8] SPASS+T (letzter Aufruf: 12.08.19):
`https://people.mpi-inf.mpg.de/~uwe/software/#TSPASS`
[9] SPASS Workbench (letzter Aufruf: 17.09.19):
`https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/`
[10] Vampire (letzter Aufruf: 12.08.19):
`http://voronkov.com/vampire.cgi`

3 Methodik

3.1 Analyse

Die Methodik unterteilt sich in Analyse und Konzeption. Ersteres unterscheidet sich von der üblichen Vorgehensweise in der Art, dass sie sich auf bereits vorhandenen Code bezieht, beziehungsweise an gegebenen Code orientiert und keine Neukonzeption anstrebt. So wird die Betrachtung, um die des zu modernisierenden Codes, erweitert. Letzteres verhält sich adäquat, gegeben der genannten Rahmenbedingungen. Die übliche Vorgehensweise im Bereich der Softwareentwicklung sieht wie folgt aus. Der erste Schritt, bestehend aus Analyse der Anforderungen, erarbeiten einer System-Spezifikation und Projektplanung, ist die Analyse beziehungsweise Planung. Danach kommt der Entwurf oder auch die Konzeptionsphase, welche sich mit dem designen der Software in Form eines Modells oder auch mehrerer Modelle beschäftigt. Diese Modelle werden dann implementiert in einer für den Anwendungsfall passenden Programmiersprache. Danach wird die Software validiert und durch Tests verifiziert, jedoch wird das mit zunehmender Komplexität fast unmöglich. Zum Schluss kommt noch der Betrieb und die Wartung, daran sieht man, dass die Kernphasen sehr Wirtschaftsbezogen sind.[1]
Die Analyse spaltet sich in TOP, LC und die LC-Anbindung an den OC, der Bestandteil des Omega Projektes ist, auf. Zuvor wird jedoch eine allgemeine Betrachtung vorgenommen, um einen klaren Überblick zu schaffen. TOP besteht aus zwei Bestandteilen, wobei nur der zur Vereinfachung und Verifizierung von Presburger-Formeln für uns von Interesse ist. LCO verwendet den OC, um Existenz- und Allquantoren zu eliminieren indem zum Beispiel die Anfrage an OC gesendet wird, ob es für eine Formel $x < 5 \wedge x > 7$ eine erfüllende Belegung gibt. In dem Fall ganz offensichtlich nicht, da eine Variable nicht gleichzeitig kleiner als fünf und größer als sieben sein kann. Somit kann LCO diesen Ast im Beweisbaum fallen lassen, da dieser mit Sicherheit nicht zu einer Lösung führt. Das Projekt ist eine Leistung von William Pugh und dem Omega Project Team - namentlich sind das Evan Rosser, Wayne Kelly, Bill Pugh, Dave Wonnacott, Tatiana Shpeisman und Vadim Maslov.[2] LC ist ein auf dem Konnektionskalkür basierender automatischer Theorem-Beweiser für klassische FOL von Jens Otten.[3] Die Erweiterung von LC für die Nutzung vom OC, der oben genannte Bestandteil, ist von Holger Trölenberg.

[1] Kernphasen der Softwareentwicklung S.4 (letzter Aufruf: 13.08.19):
`https://moodle2.uni-potsdam.de/pluginfile.php/632153/mod_resource/content/2/v01.pdf`

[2] The Omega Project Internetseite (letzter Aufruf: 26.04.19):
`https://www.cs.umd.edu/projects/omega/`
The Omega Project GitHub (letzter Aufruf: 26.04.19):
`https://github.com/davewathaverford/the-omega-project/`

[3] leanCoP Internetseite (letzter Aufruf: 26.04.19):
`www.leancop.de`

Die Anforderungen an das Konvolut sind, dass es in einer schlankeren Fassung auf aktuellen Systemen kompilierbar und ausführbar sein soll. TOP besteht zum Hauptteil aus C/C++ Code und etwas Lex-, Yacc-, Flex- und Bison-Code. Dabei umfasst das Projekt 393 Dateien mit insgesamt 90790 Zeilen Code und 10372 Zeilen Kommentare.[4] LC ist zusammen mit der Erweiterung deutlich kleiner und unterteilt sich in sechs Prolog-Dateien und einer Shellscript-Datei, welche bereinigt[5] nur 160 Zeilen umfasst. Prolog ist eine Programmiersprache zur Logikprogrammierung.[6] Die Prolog-Dateien sind im Prolog-Dialekt SWI-Prolog geschrieben und haben bereinigt 967 Zeilen. Die von LC im CASC-J5 Wettbewerb[7], genutzte Binary von TOP war 3.545.735 Bytes groß.

3.2 Konzeption

Das Konzept unterteilt sich in die Planung der Verkleinerung des TOP und in die Modernisierung von LC mit der Omega-Anbindung. Im ersten Schritt muss ermittelt werden welche Teile des TOP benötigt werden um die OC Binary zu erstellen. Eine Möglichkeit ist bei einer Header-Datei die mit Sicherheit verwendet wird anzufangen und von dort aus sich an den includes ein Abhängigkeitsnetz aufzuspannen und dann alles was nicht von diesem Netz erfasst wird zu entfernen - die Startdatei muss klug gewählt werden. Danach müssen die Makefiles an den kleineren Umfang des Projekts angepasst werden, sodass richtig und vollständig compiliert wird. Sollte die Kompilierung nicht erfolgreich sein, müssen iterativ alle noch benötigten Dateien wieder eingefügt werden. Das ist zwar eine relativ aufwendige Arbeit, aber wie in der Analyse herausgefunden umfasst der wichtige Bereich, also die vier Verzeichnisse basic, code_gen, omega_calc und omega_lib nur 172 Dateien. Da wir nur die Header-Dateien betrachten kann man grob mit der Hälfte rechnen, das wären 86 Dateien die zu untersuchen sind. Das ist eine Anzahl die man manuell oder semi-automatisch analysieren kann. Semi-automatisch soll heißen, dass die Abhängigkeiten von einem Python-Script gesucht und ausgegeben werden. Das Python-Script wurde dazu selbst implementiert und liegt der Arbeit in digitaler Form bei. Somit wird der Großteil der Arbeit an ein Programm übergeben, das erst geschrieben werden muss und Zeit kostet. Jedoch arbeitet das Script fehlerfrei, sofern es richtig implementiert wurde und das schnelle finden aller Abhängigkeiten wiegt die Implementierungszeit auf. Manuell werden dann die Dateien extrahiert und gegebenenfalls Anpassungen an den Makefiles vorgenommen.
Da TOP nun verkleinert und bereit für den Einsatz ist, kann es mit LCO schon verwendet werden, um erste Messergebnisse für die Evaluation zu generieren. Währenddessen wird LCO an eine neue SWI-Prolog Version angepasst. Dazu wird ein Blick in die SWI-Prolog release patchnotes geworfen, um allgemeine Veränderungen und Anpas-

[4] Die Daten wurden mit Hilfe von dem Programm *cloc* (Version: 1.70) ermittelt.
[5] Bereinigt heißt hier ohne Leerzeilen.
[6] Prolog (letzter Aufruf: 13.08.19):
`https://de.wikibooks.org/wiki/Prolog`
[7] Jens Otten - leanCoP-Ω 0.1 (letzter Aufruf: 05.05.19):
`http://www.tptp.org/Seminars/CASC/CASC-J5/`

sungen der Programmiersprache in Erfahrung zu bringen. Die erste Anlaufstelle sollte sein, zu schauen ob es Warnings gibt die zu Errors geändert wurden und das Programm blockieren würden. Darüber hinaus sind Umbenennung von Funktionen sehr unwahrscheinlich, aber die Änderung ihres Zweckes wiederum nicht. Deshalb sollte dahingehend ebenfalls untersucht werden. Sind alle Änderungen aus der Dokumentation umgesetzt wird das Programm getestet, ob es weitere Warnings oder Errors wirft. Diese sind gleicherweise zu entfernen. Läuft LCO mit der aktuellen Prolog Version fehlerfrei kann es mit den selben Problemen gebenchmarkt werden, um einen repräsentativen Vergleich zu haben.

4 Durchführung der Anpassung

In diesem Kapitel werden die aus der Analyse heraus konzeptionierten Anpassungen Schritt für Schritt dargestellt. An einigen Punkten werden auch mehrere Möglichkeiten aufgezeigt und erklärt warum sich für die Gewählte entschieden wurde, um den Prozess klar und nachvollziehbar zu behandeln.
Für die Extraktion des benötigten OC aus dem gesamten Projekt wird als erstes der naive Ansatz gewählt, um Zeit zu sparen. Hierbei wurde von der Headerdatei *omega.h* die Untersuchung gestartet, da nach einer kurzen Prüfung die Nutzung durch OC detektiert wurde. Es wurden sich die in der aktuellen Datei vorhandenen *includes*, welche verwendet werden, um andere Dateien einzubinden, angeschaut. Diese eingebundenen Dateien wurden sukzessive nach dem gleichen Verfahren untersucht. In Abbildung 4.1 kann man nun das Netz, das von *omega.h* aufgespannt wird, erkennen. Die Knotenpunkte sind von der Methode erfasste Headerdateien und die Pfeilrichtung bestimmt in welcher Richtung inkludiert wurde. Diese Erkenntnisse sind ein guten Anfang, aber decken die vom OC benötigten Dateien nicht vollständig ab.

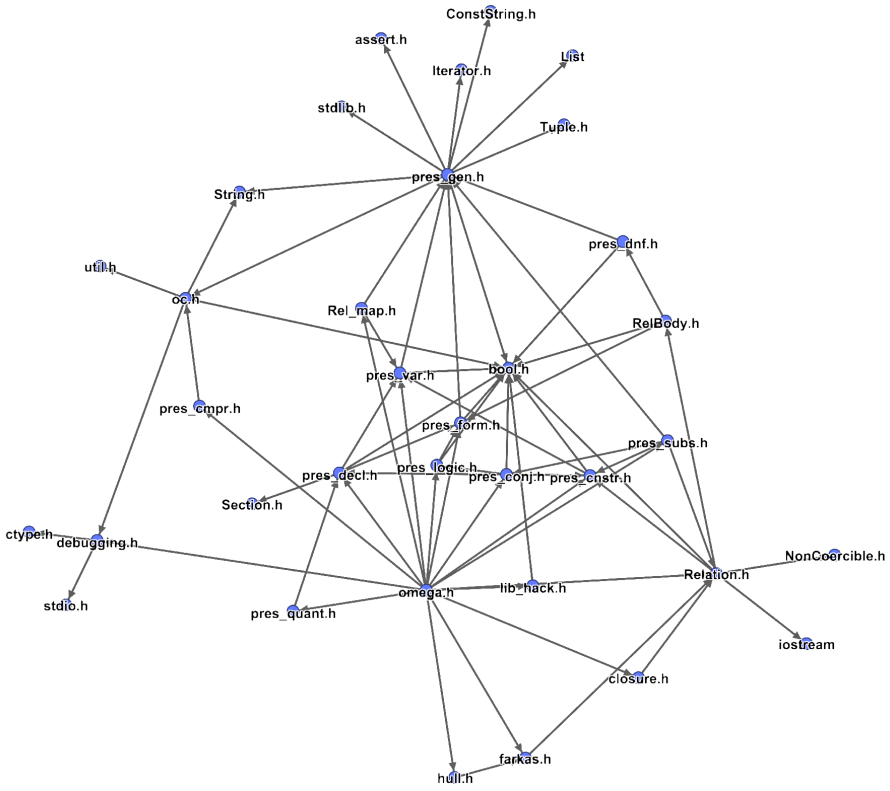

Abbildung 4.1: Abhängigkeitsgraphen von der Datei *omega.h* aus

Da der naive Ansatz ein unvollständiges Bild über die Abhängigkeiten gab, wurde die repetitive Arbeit des Durchsuchens in ein kleines Script ausgelagert. Das Script wurde in der Sprache Python geschrieben, da es eine sehr einfache Sprache ist mit der man schnell kleine Programme schreiben kann. Die Grundidee in Pseudocode:

1. Lege eine Liste an mit Startdatei(en) SD
2. Solange SD nicht leer ist tue:
    2.1 Nehme das erste Element, lösche es aus SD
    2.2 Füge alle includes vom aktuellen Element an SD an
    2.3 Merke das aktuelle Element (inklusive includes)
3. Gebe alle gemerkten Elemente (inklusive includes) aus

Die formale Implementierung befindet sich in digitaler Form anbei der Arbeit und wird hier nicht weiter erläutert, da es sich nicht um Themenspezifischen Code handelt. Die Ausgabe des Scripts sieht folgendermaßen aus:

FILE: ⟨Dateiname⟩
PATH: ⟨Pfad⟩
DEP: ⟨Includes⟩

Beispiel:

FILE: parser.l
PATH: ./omega_calc/src/parser.l
DEP: stdio.h, string.h, AST.h, Dynamic_Array.h, Exit.h, mmap-codegen.h, ...

Beim Compilieren der extrahierten Dateien ist aufgefallen, dass die Sourcedateien (*.c), also Dateien die die eigentlichen Berechnungen enthalten, versuchen andere Dateien einzubinden. Die Datei closure.c versucht SimpleList.h einzubinden, oc.c versucht oc_i.h einzubinden, code_gen.c versucht stmt_builder.h einzubinden, die Datei mmap-codegen.c versucht mmap-sub.h und mmap-checks.h einzubinden. Diese fehlenden Dateien wurde zum Extrakt hinzugefügt und die nötigen Anpassungen an den Makefiles vorgenommen. Diese Anpassungen reduzieren sich darauf, die entfernten Dateien ebenfalls aus dem Makefile zu entfernen, damit keine Dateien versucht werden zu compilieren, die es nicht gibt. Der fertige Abhängigkeitsgraph ist in Abbildung 4.2 zu sehen. Hierbei sind die Dateien in ihre Verzeichnisse farblich unterteilt sind und die Pfeile die Richtung der includes anzeigen.

Tabelle 4.1: Codeumfang (TOP) der relevanten Bereiche vor und nach der Extraktion

	new	files	comment	code	old	files	comment	code
./basic		28	156	2027		44	158	2110
./code_gen		25	1031	5763		27	1047	5930
./omega_calc		8	723	8472		21	772	9338
./omega_lib		50	1428	12073		80	1766	19965
sum		111	3338	28335		172	3743	37343
project		113	3354	29516		393	10372	90790

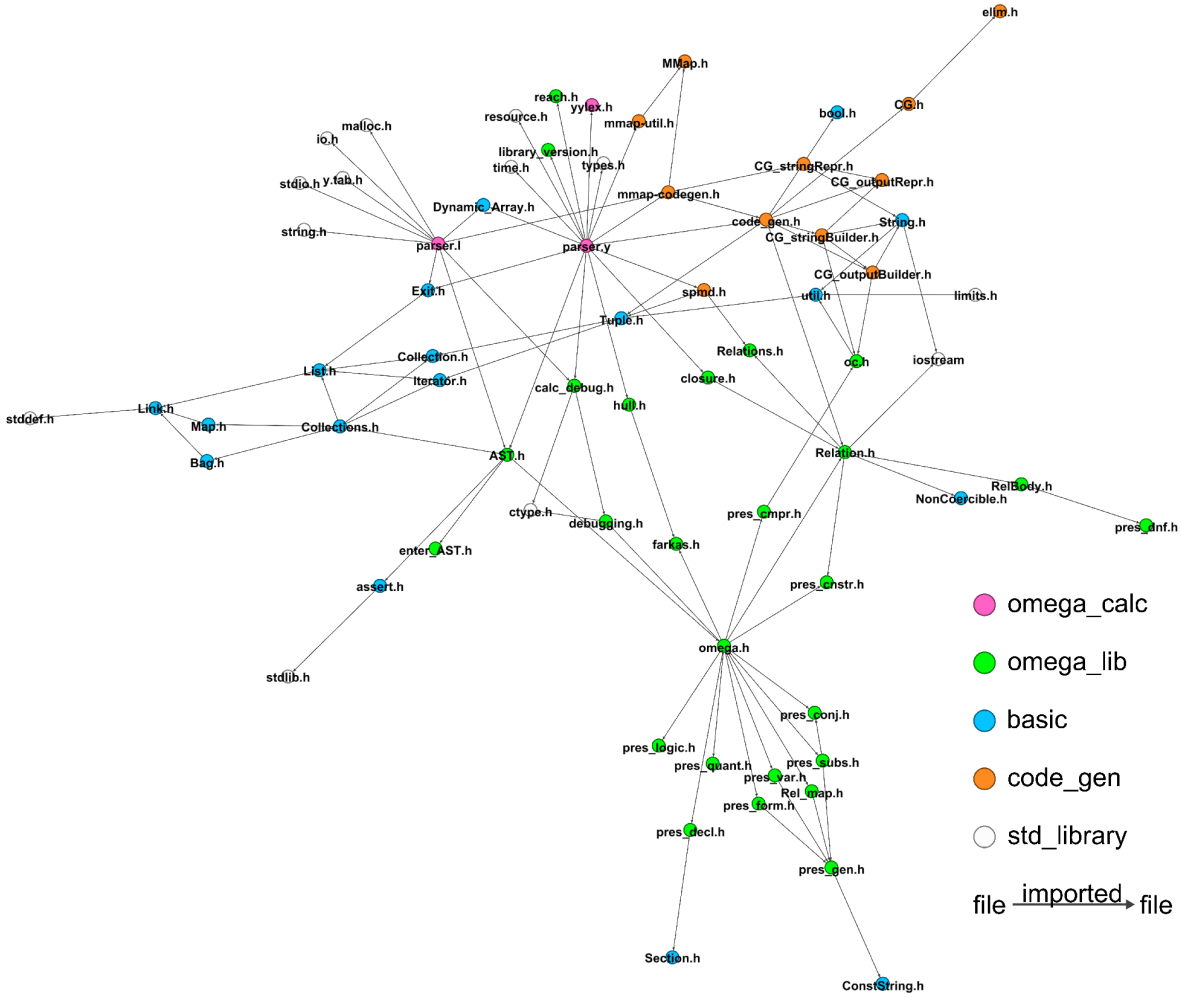

Abbildung 4.2: Abhängigkeitsgraphen von den Dateien *parser.l, parser.y* aus

In der Tabelle Tabelle 4.1 sind die Daten vor *old* und nach *new* der Extraktion zu-

sammengefasst. In der Kopfzeile befinden sich die Beschreibungen, um welche Daten es sich handelt. In der ersten Spalte befinden sich die wichtigen Verzeichnisse vom TOP. Die Zeile mit der Beschreibung *sum* fasst die Daten dieser vier Bereiche zusammen. In der letzten Zeile sind die Daten des kompletten Projektes aufgelistet. Der Anteil, den der OC vom gesamten TOP verwendet, ist ein Drittel des gesamten Projektes. Das Projekt konnte von 393 Dateien auf 113 Dateien reduziert werden, dabei verkleinerte sich die Anzahl der Zeilen Code von 90790 auf 29516. Compiliert wurde das TOP damals mit dem G++ 2.7.2 Compiler von 1995 (Veröffentlichungsdatum: 30.11.1995). Hier wurde der GCC 4.8.5 aus der selben GNU Compiler Sammlung verwendet (Veröffentlichungsdatum: 23.06.2015). Der aktuellste Compiler-Treiber dieser Sammlung ist zur Zeit Version GCC 9.2 (Veröffentlichungsdatum: 12.08.2019).[1] [2]

Dann wurden die Anpassungen an LCO vorgenommen die notwendig waren damit der Beweiser mit der neueren SWI-Prolog Version 7.7.19 läuft. Da die Releasenotes im Vorfeld studiert wurden ist eine Modifikation aufgefallen (* MODIFIED [May 11 2011]). In dieser wurde eine Warnung in einen Fehler geändert. Die Warnung trat bei den senkrechten Balken | auf, wenn sie nicht mit einfachen Anführungsstrichen umschlossen waren. Abschnitte mit | wurden zu $'|'$ angepasst, da Entwicklertools damit nicht gut zurecht kamen. Ebenfalls wurde die Präfix-Notation notwendig. Das heißt unären Funktionen der Form $!Fo$ werden zu $'!'(Fo)$ ergänzt. Bei binären Funktionen werden die Variablen durch ein Komma getrennt, das heißt Ausdrücke der Form $(Fo\&Go)$ wurden zu $'\&'(Fo, Go)$. Damit waren alle Fehler behoben, jedoch wurden noch einige Warnungen beseitigt. Unter anderem ein paar Singleton Variablen ausgewiesen und eine alte Funktion ausgetauscht, die nur noch aus Gründen der Kompatibilität gehalten wurde. Die Funktion *concat_atom* wurde durch die neue Funktion *atomic_list_concat* ausgetauscht, welche den selben Zweck erfüllt. Um sicherzustellen, dass die Änderung der Funktion keine Auswirkungen hat, wurden alle Probleme mit *concat_atom* und *atomic_list_concat* gelöst und die Zeiten verglichen. Dabei ergab sich, dass die Funktion keinen signifikanten Einfluss auf die Lösungsdauer hat.

[1] GNU Compiler releases (letzter Aufruf: 16.08.19):
`https://www.gnu.org/software/gcc/releases.html`

[2] Zusätzliche Compiler Informationen (letzter Aufruf: 16.08.19):
`https://gcc.gnu.org/onlinedocs/gcc/Invoking-G_002b_002b.html`
`https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html`

5 Evaluation

5.1 Vorbetrachtung

Es war nicht zu erwarten, dass es durch Extraktion des OC zu einem speed-up der Problemlösung kommt. Das ist zu begründen mit der Tatsache das die OC-Binary sich in ihrer Größe und Funktion im Wesentlichen nicht verändert hat. Mit dem Umstieg von der Prolog Version 5.10 auf die Version 7.7.x ist im best-case schätzungsweise ein speed-up von fünf Prozent zu erzielen. Da mit der Prolog Version 7.6.x eine bessere multi-threaded Performance[1] unterstützt wird. Jedoch profitiert das Programm nicht sehr davon, da in der automatisierten Beweisführung Parallelisierung sehr schwer genutzt werden kann. Die Problemsammlung umfasst 75 Probleme mit Arithmetik. Bei der CASC-J5 hat LCO mit 64 gelösten Problemen die meisten Probleme gelöst, also 9 Probleme nicht gelöst. Der Anspruch ist, dass diese 64 Probleme auch mit dem modernisierten LCO gelöst werden.

5.2 Ausführung

Zuerst wurde untersucht welche Auswirkung die OC-Binary aus dem extrahiertem TOP hat. Dazu hat LCO die Probleme einmal mit der ursprünglichen OC-Binary und einmal mit der Binary aus dem extrahierten Projekt gelöst. Die verwendete SWI-Prolog Version ist beide Male die Version 5.10. Das Ergebnis dieses Experimentes bestätigte die Hypothese, dass die OC-Binary aus dem extrahiertem TOP keine signifikante Zeitveränderung hervorbringt. Genauer hat sie nicht einmal eine nicht signifikante Zeitveränderung bewirkt, die Zeiten waren identisch. Das ist ein gutes Indiz dafür ist, dass die Extraktion erfolgreich war. Ebenfalls ist zu erwähnen, dass die selbe Anzahl an Problemen, ferner die selben Probleme gelöst wurden wie bei der CASC-J5. Anschließend wurden die Laufzeiten der unterschiedlichen SWI-Prolog Versionen untersucht.
Es wurde die OC-Binary aus dem extrahiertem TOP für beide Versionen verwendet. Alle Probleme wurden drei mal versucht zu lösen. Dann wurde der Durchschnittswert dieser drei Versuche gebildet und als Lösungsdauer des Problems verwendet. Das Experiment wurde auf einem CentOS Linux 7 System durchgeführt mit einer Intel Core i7-2630QM CPU 2.00GHz x 8 und 16GB RAM. Die LCO Variante mit SWI-Prolog 5.10 löste von den gegebenen 75 TFAs 64, wie bereits bekannt war. Die durchschnittliche Lösungszeit aller gelösten Probleme betrug dabei 18,15 Sekunden und somit die Gesamtzeit 19,36 Minuten. Die LCO Variante mit SWI-Prolog 7.7.19 löste von den gegebenen 75 TFAs ebenfalls 64, somit bietet diese den gleichen Funktionsumfang in diesem Experiment. Für das Lösen eines Problems hat diese Variante durchschnittlich 17,44 Sekunden benötigt, wieder ohne Mitbetrachtung der nicht gelösten Proble-

[1] SWI-Prolog Releasenotes (letzter Aufruf: 14.08.2019):
`https://www.swi-prolog.org/versions.txt`

me. Daraus ergibt sich eine Gesamtlaufzeit von 18,6 Minuten für die LCO Variante mit SWI-Prolog 7.7.19. Bei einer Lösungsrate von 88%, in diesem Experiment, war die Variante mit der aktuellen SWI-Prolog Version 3,9% schneller, somit wurde der prognostizierte best-case speed-up nur knapp verfehlt. Das ist ein guter average-case speed-up, jedoch sieht man in Abbildung 5.1 und Abbildung 5.2 gut, dass es sehr vom Anwendungsfall abhängt.
Die Datentabelle ist in zwei Abbildungen unterteilt, dabei ist die Tabelle nach der CASC-Zeit sortiert. Die dritte Spalte gibt die Maximale Formeltiefe an und wird hier als Maß für die Schwierigkeit eines Problems verwendet. In der darauffolgenden Spalte gibt das OC an ob der OC für die Lösung des Problems verwendet wird. Die letzten beiden Spalten beinhalten die Zeiten der LCO-Version mit SWI-Prolog 5.10 und 7.7.19. Bei Problemen mit der Formeltiefe bis drei, also sehr kleine Probleme kommt es zu einer Verdopplung der Laufzeit (siehe auch Abbildung 6.3). Doch fällt das durch die geringe Laufzeit nicht ins Gewicht. Bei den vier Problemen 252, 236, 260 und 243 mit der Formeltiefe zwischen 400 und 500 kam es zu einer Halbierung der Zeit durch die neuere SWI-Prolog Version (siehe auch Abbildung 6.4).

				5.10	7.7.19
Problem TFA	CASC CPU-Time	Maximal formula depth	PP / OC	user	user2
170	0,10 s	1	OC	0,06 s	0,13 s
008	0,10 s	1	PP	0,06 s	0,11 s
162	0,10 s	1	OC	0,07 s	0,12 s
058	0,10 s	1	OC	0,07 s	0,14 s
155	0,10 s	1	OC	0,07 s	0,13 s
101	0,10 s	1	OC	0,08 s	0,13 s
015	0,10 s	1	PP	0,06 s	0,12 s
022	0,10 s	1	PP	0,06 s	0,13 s
186	0,10 s	1	OC	0,07 s	0,13 s
001	0,10 s	1	PP	0,05 s	0,12 s
066	0,10 s	1	OC	0,07 s	0,14 s
122	0,10 s	1	OC	0,06 s	0,12 s
110	0,10 s	1	OC	0,06 s	0,12 s
013	0,10 s	2	OC	0,05 s	0,11 s
029	0,10 s	2	PP	0,06 s	0,13 s
126	0,10 s	2	PP	0,06 s	0,10 s
166	0,10 s	2	PP	0,07 s	0,13 s
038	0,10 s	2	OC	0,06 s	0,11 s
036	0,10 s	2	PP	0,04 s	0,12 s
026	0,10 s	2	OC	0,07 s	0,13 s
040	0,10 s	2	OC	0,06 s	0,11 s
178	0,10 s	2	PP	0,07 s	0,13 s
018	0,10 s	2	OC	0,06 s	0,12 s
070	0,10 s	2	PP	0,08 s	0,14 s
043	0,10 s	2	PP	0,05 s	0,10 s
174	0,10 s	2	PP	0,06 s	0,12 s
062	0,10 s	2	OC	0,06 s	0,12 s
117	0,10 s	2	OC	0,06 s	0,12 s
050	0,10 s	2	PP	0,06 s	0,12 s
004	0,10 s	2	OC	0,07 s	0,13 s
182	0,10 s	2	PP	0,06 s	0,12 s
114	0,10 s	2	PP	0,06 s	0,12 s

Abbildung 5.1: Benchmark-Daten Durchschnitte mit Formeltiefe Teil 1

Eine weitere Auffälligkeit ist, dass hauptsächlich zur Lösung der einfachen TFAs der OC verwendet wurde (siehe Abbildung 5.1). Ab einer maximalen Formeltiefe von 100 wird der OC kein einziges mal verwendet (siehe Abbildung 5.2). Die Gründe dafür wurden nicht untersucht, da sie nicht Teil der Zielstellung waren.

086	0,10 s	3	OC	0,07 s	0,12 s
079	0,10 s	3	OC	0,06 s	0,12 s
046	0,10 s	3	OC	0,06 s	0,11 s
216	0,10 s	4	OC	0,07 s	0,13 s
138	0,10 s	12	PP	0,06 s	0,13 s
192	0,20 s	4	OC	0,09 s	0,15 s
198	0,30 s	4	OC	0,19 s	0,39 s
190	0,30 s	5	OC	0,11 s	0,17 s
195	0,40 s	3	OC	0,14 s	0,19 s
210	1,10 s	8	OC	0,34 s	0,39 s
220	5,80 s	103	PP	10,60 s	10,64 s
228	5,90 s	120	PP	10,93 s	10,72 s
245	25,10 s	452	PP	8,88 s	13,91 s
255	30,50 s	482	PP	13,01 s	15,26 s
293	33,10 s	554	PP	75,61 s	77,12 s
252	36,50 s	469	PP	35,45 s	19,56 s
236	38,80 s	435	PP	37,34 s	20,65 s
260	39,20 s	492	PP	43,11 s	19,35 s
243	41,00 s	443	PP	36,57 s	20,65 s
265	46,90 s	504	PP	49,78 s	55,22 s
085	55,50 s	12	OC	62,87 s	60,73 s
082	58,00 s	5	OC	24,67 s	24,96 s
316	64,10 s	588	PP	61,43 s	62,11 s
286	66,10 s	543	PP	60,60 s	64,11 s
310	82,90 s	582	PP	72,10 s	71,02 s
272	85,20 s	516	PP	72,09 s	73,60 s
276	88,90 s	523	PP	74,98 s	74,55 s
294	91,50 s	557	PP	76,51 s	77,31 s
298	94,70 s	566	PP	78,04 s	79,12 s
290	101,10 s	550	PP	79,63 s	80,57 s
302	107,00 s	573	PP	75,35 s	86,00 s
305	129,40 s	582	PP	98,70 s	93,31 s
SUMME	**22,2 Min.**		**35 OC / 75**	**19,4 Min.**	**18,6 Min.**

Abbildung 5.2: Benchmark-Daten Durchschnitte mit Formeltiefe Teil 2

6 Ausblick

Der OC ist aus dem gesamten TOP extrahiert und funktioniert mit LCO. Es gibt jedoch noch weitere Anpassungen, die vorgenommen werden können. Als Erstes ist es möglich alle Funktionen die noch im OC enthalten sind, aber nicht verwendet werden zu entfernen, um das Projekt weiter in seiner Größe zu reduzieren. Obwohl es das Projekt handhabbarer und einfacher verständlich machen würde, wäre das viel Arbeit für einen sehr kleinen Nutzen und ist deshalb nicht sinnvoll. Da das TOP einige Funktionalitäten, wie spezielle Listen selbst implementiert hat, könnte man ebenfalls das Projekt umbauen, um die C++ Standardlibrary zu nutzen. Jedoch sind diese Anpassungen umfangreicher Natur und bringen vermutlich ebenfalls keinen Performance-Vorteil. Sinnvoller ist es da schon den OC mit *foreign function interface* (FFI) anstatt mit OS-Systemcall von LCO aufrufen zu lassen. Das könnte die Kommunikation zwischen LCO und OC stark beschleunigen, da Kommunikation über FFI einen geringeren Overhead hat als Kommunikation über die System-Konsole. Den automatischen Theorembeweiser könnte man mit Hilfe von Docker[1] automatisch in einem lauffähigen Zustand isolieren. Dort wird der Beweiser in einer virtuellen Umgebung als Container von Docker verpackt und kann als Plugin verwendet werden. Jedoch ist nichts über den verursachten Overhead bei der Performance bekannt und müsste zunächst untersucht werden. Außerdem ist die Docker-Architektur eher als eine Server-Client Struktur aufgebaut. Sinnvoller ist es eine eigene Erweiterung (engl. auch extension) zu bauen die geläufige Schnittstellen von etablierten Beweissystemen nutzt, um diese um die Arithmetikfunktionalität erweitern zu können.

[1] Docker (letzter Aufruf: 20.08.2019):
`https://www.docker.com/`

Anhang

Tabellen

α	α_1	α_2
$\neg\neg F$	F	F
$F \wedge G$	F	G
$\neg(F \vee G)$	$\neg F$	$\neg G$

Tabelle 6.1: α-Formeln

β	β_1	β_2
$F \vee G$	F	F
$\neg(F \wedge G)$	$\neg F$	$\neg G$

Tabelle 6.2: β-Formeln

γ	$\gamma(t)$
$\forall x F$	F_x^t
$\neg\exists x F$	$\neg F_x^t$

Tabelle 6.3: γ-Formeln

δ	$\delta(a)$
$\exists x F$	F_x^a
$\neg\forall x F$	$\neg F_x^a$

Tabelle 6.4: δ-Formeln

													SWI-Prolog 5.10	SWI-Prolog 7.7.19
Problem TFA	CASC CPU-Time	Number of formulae	Number of atoms	Maximal formula depth	Number of connectives	Number of type conns	Number of predicates	Number of functors	Number of variables	Maximal term depth	Arithmetic symbols	PP / OC	user	user2
170	0,10 s	1	1	1	0	0	1	4	0	2	5	OC	0,06 s	0,13 s
008	0,10 s	1	1	1	0	0	1	2	0	1	3	PP	0,06 s	0,11 s
162	0,10 s	1	1	1	0	0	1	4	0	2	5	OC	0,07 s	0,12 s
058	0,10 s	1	1	1	0	0	1	4	0	2	4	OC	0,07 s	0,14 s
155	0,10 s	1	1	1	0	0	1	4	0	3	5	OC	0,07 s	0,13 s
101	0,10 s	1	1	1	0	0	1	4	0	2	4	OC	0,08 s	0,13 s
015	0,10 s	1	1	1	0	0	1	2	0	1	3	PP	0,06 s	0,12 s
022	0,10 s	1	1	1	0	0	1	2	0	1	3	PP	0,06 s	0,13 s
186	0,10 s	1	1	1	0	0	1	9	0	3	11	OC	0,07 s	0,13 s
001	0,10 s	1	1	1	0	0	1	2	0	1	3	PP	0,05 s	0,12 s
066	0,10 s	1	1	1	0	0	1	4	0	2	4	OC	0,07 s	0,14 s
122	0,10 s	1	1	1	0	0	1	4	0	2	4	OC	0,06 s	0,12 s
110	0,10 s	1	1	1	0	0	1	4	0	2	4	OC	0,06 s	0,12 s
013	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,05 s	0,11 s
029	0,10 s	1	1	2	1	0	1	2	0	1	3	PP	0,06 s	0,13 s
126	0,10 s	1	1	2	0	0	2	3	1	2	4	PP	0,06 s	0,10 s
166	0,10 s	1	1	2	0	0	2	3	1	2	5	PP	0,07 s	0,13 s
038	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,06 s	0,11 s
036	0,10 s	1	1	2	1	0	1	2	0	1	3	PP	0,04 s	0,12 s
026	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,07 s	0,13 s
040	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,06 s	0,11 s
178	0,10 s	1	1	2	0	0	2	3	1	2	5	PP	0,07 s	0,13 s
018	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,06 s	0,12 s
070	0,10 s	1	1	2	0	0	2	3	1	2	4	PP	0,08 s	0,14 s
043	0,10 s	1	1	2	1	0	1	2	0	1	3	PP	0,05 s	0,10 s
174	0,10 s	1	1	2	0	0	2	3	1	2	5	PP	0,06 s	0,12 s
062	0,10 s	1	1	2	1	0	1	4	0	2	4	OC	0,06 s	0,12 s
117	0,10 s	1	1	2	1	0	1	4	0	2	4	OC	0,06 s	0,12 s
050	0,10 s	1	1	2	1	0	1	2	0	1	3	PP	0,06 s	0,12 s
004	0,10 s	1	1	2	0	0	2	1	1	1	3	OC	0,07 s	0,13 s
182	0,10 s	1	1	2	0	0	2	2	1	2	5	PP	0,06 s	0,12 s
114	0,10 s	1	1	2	0	0	2	2	1	2	4	PP	0,06 s	0,12 s
086	0,10 s	2	4	3	1	1	4	4	0	2	5	OC	0,07 s	0,12 s
079	0,10 s	1	1	3	0	0	2	1	2	2	3	OC	0,06 s	0,12 s
046	0,10 s	1	1	3	0	0	2	0	2	1	3	OC	0,06 s	0,11 s
216	0,10 s	1	2	4	1	0	2	3	2	2	8	OC	0,07 s	0,13 s
138	0,10 s	1	5	12	4	0	2	1	7	2	11	PP	0,06 s	0,13 s
192	0,20 s	1	2	4	1	0	2	4	2	2	6	OC	0,09 s	0,15 s
198	0,30 s	1	2	4	1	0	3	2	2	2	6	OC	0,19 s	0,39 s
190	0,30 s	1	3	5	2	0	4	0	2	1	4	OC	0,11 s	0,17 s
195	0,40 s	1	2	3	1	0	3	4	1	2	6	OC	0,14 s	0,19 s
210	1,10 s	9	23	8	4	3	10	8	7	6	9	OC	0,34 s	0,39 s
220	5,80 s	8	985	103	1090	2	12	18	236	2	518	PP	10,60 s	10,64 s
228	5,90 s	8	1251	120	1404	2	12	18	300	2	642	PP	10,93 s	10,72 s
245	25,10 s	9	7232	452	8460	2	13	20	1620	2	3688	PP	8,88 s	13,91 s
255	30,50 s	9	7902	482	9304	2	13	20	1765	2	3975	PP	13,01 s	15,26 s
293	33,10 s	9	9591	554	11437	2	13	20	2135	2	4655	PP	75,61 s	77,12 s
252	36,50 s	9	7646	469	8982	2	13	20	1710	2	3864	PP	35,45 s	19,56 s
236	38,80 s	8	7019	435	8210	2	12	19	1579	2	3564	PP	37,34 s	20,65 s
260	39,20 s	9	8153	492	9621	2	13	20	1820	2	4078	PP	43,11 s	19,35 s
243	41,00 s	8	7161	443	8373	2	12	19	1607	2	3651	PP	36,57 s	20,65 s
265	46,90 s	9	8404	504	9938	2	13	20	1875	2	4179	PP	49,78 s	55,22 s
085	55,50 s	1	5	12	4	0	2	1	7	2	11	OC	62,87 s	60,73 s
082	58,00 s	1	3	5	2	0	2	3	2	2	8	OC	24,67 s	24,96 s
316	64,10 s	9	10348	588	12392	2	13	20	2300	2	4961	PP	61,43 s	62,11 s
286	66,10 s	9	9316	543	11090	2	13	20	2075	2	4547	PP	60,60 s	64,11 s
310	82,90 s	9	10208	582	12216	2	13	20	2270	2	4903	PP	72,10 s	71,02 s
272	85,20 s	9	8701	516	10313	2	13	20	1940	2	4299	PP	72,09 s	73,60 s
276	88,90 s	9	8857	523	10511	2	13	20	1975	2	4358	PP	74,98 s	74,55 s
294	91,50 s	9	9642	557	11500	2	13	20	2145	2	4681	PP	76,51 s	77,31 s
298	94,70 s	9	9839	566	11751	2	13	20	2190	2	4750	PP	78,04 s	79,12 s
290	101,10 s	9	9479	550	11295	2	13	20	2110	2	4613	PP	79,63 s	80,57 s
302	107,00 s	9	9997	573	11951	2	13	20	2225	2	4810	PP	75,35 s	86,00 s
305	129,40 s	9	10186	582	12188	2	13	20	2265	2	4894	PP	98,70 s	93,31 s
SUMME	22,2 Min.											35 OC / 75	19,4 Min.	18,6 Min.

Tabelle 6.5: Benchmark-Daten Durchschnitte mit Formelparameter

Tabelle 6.6: Alle Benchmark-Daten mit Formelparameter

Abbildungen

Abbildung 6.1: Valide Schlussregeln in der Refinement Logic I

Axiom	$A \vdash B$	(1)
Verdünnung	$\dfrac{A,\Gamma \vdash \Phi}{\Gamma \vdash \Phi}$	(2)
	$\dfrac{\Gamma \vdash \Phi,A}{\Gamma \vdash \Phi}$	(3)
Kontraktion	$\dfrac{A,\Gamma \vdash \Phi}{A,A,\Gamma \vdash \Phi}$	(4)
	$\dfrac{\Gamma \vdash \Phi,A}{\Gamma \vdash \Phi,A,A}$	(5)
Vertauschung	$\dfrac{\Gamma,B,A,\Delta \vdash \Phi}{\Gamma,A,B,\Delta \vdash \Phi}$	(6)
	$\dfrac{\Gamma \vdash \Phi,B,A,\Psi}{\Gamma \vdash \Phi,A,B,\Psi}$	(7)
Schnittregel	$\dfrac{\Gamma,\Delta \vdash \Phi,\Psi}{\Gamma \vdash \Phi,A \qquad A,\Delta \vdash \Psi}$	(8)
$\neg - A$	$\dfrac{\neg A,\Gamma \vdash \Phi}{\Gamma \vdash \Phi,A}$	(9)
$\neg - S$	$\dfrac{\Gamma \vdash \Phi,\neg A}{A,\Gamma \vdash \Phi}$	(10)
$\wedge - A$	$\dfrac{A \wedge B,\Gamma \vdash \Phi}{A,\Gamma \vdash \Phi}$	(11)
	$\dfrac{A \wedge B,\Gamma \vdash \Phi}{B,\Gamma \vdash \Phi}$	(12)
$\wedge - S$	$\dfrac{\Gamma \vdash \Phi,A \wedge B}{\Gamma \vdash \Phi,A \qquad \Gamma \vdash \Phi,B}$	(13)
$\vee - A$	$\dfrac{A \vee B,\Gamma \vdash \Phi}{A,\Gamma \vdash \Phi \qquad B,\Gamma \vdash \Phi}$	(14)
$\vee - S$	$\dfrac{\Gamma \vdash \Phi,A \vee B}{\Gamma \vdash \Phi,A}$	(15)
	$\dfrac{\Gamma \vdash \Phi,A \vee B}{\Gamma \vdash \Phi,B}$	(16)
$\Rightarrow - A$	$\dfrac{A \Rightarrow B,\Gamma,\Delta \vdash \Phi\Psi}{\Gamma\Phi,A \qquad B,\Delta \vdash \Psi}$	(17)
$\Rightarrow - S$	$\dfrac{\Gamma \vdash \Phi,A \Rightarrow B}{A,\Gamma \vdash \Phi,B}$	(18)

Abbildung 6.2: Valide Schlussregeln in der Refinement Logic II

$$\forall - A \qquad \frac{\forall x F, \Gamma \vdash \Phi}{F', \Gamma \vdash \Phi} \tag{19}$$

wobei $F' = F\{x \setminus t\}$ und t ist ein Term, der keine gebundene Variable von F enthält.

$$\forall - S \qquad \frac{\Gamma \vdash \Phi, \forall x F}{\Gamma \vdash \Phi, F'} \tag{20}$$

wobei $F' = F\{x \setminus a\}$ und a ist eine Variable, die nicht in Γ, Φ und F erscheint.

$$\exists - A \qquad \frac{\exists x F, \Gamma \vdash \Phi}{F', \Gamma \vdash \Phi} \tag{21}$$

wobei $F' = F\{x \setminus a\}$ und a ist eine Variable, die nicht in Γ, Φ und F erscheint.

$$\exists - S \qquad \frac{\Gamma \vdash \Phi, \exists x F}{\Gamma \vdash \Phi, F'} \tag{22}$$

wobei $F' = F\{x \setminus t\}$ und t ist ein Term, der keine gebundene Variable von F enthält.

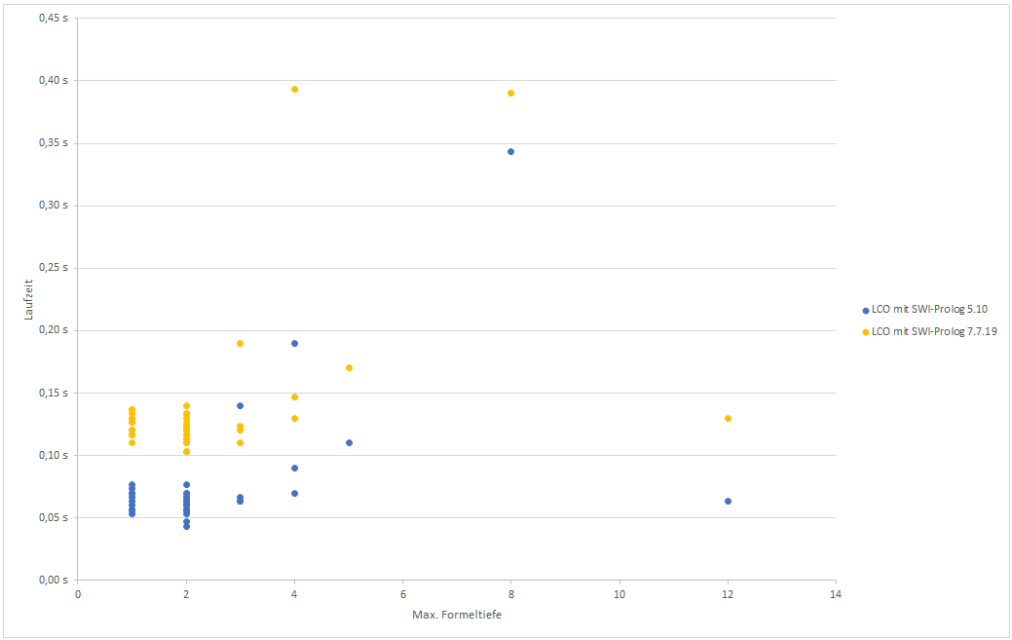

Abbildung 6.3: Einfache TFA mit Laufzeit bis 0,5s

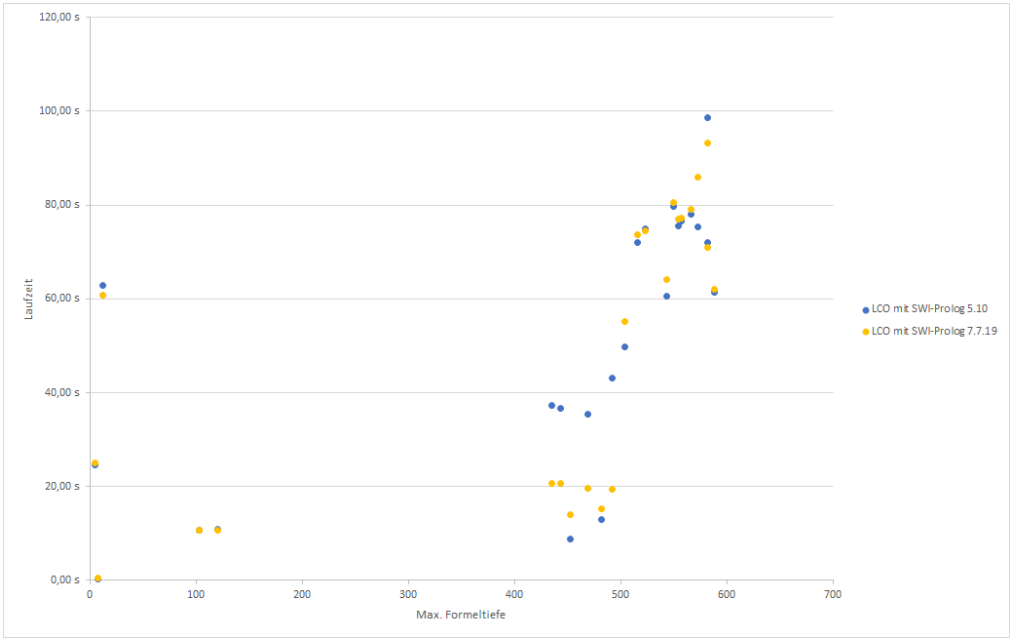

Abbildung 6.4: Komplexe TFA mit Laufzeit ab 0,5s

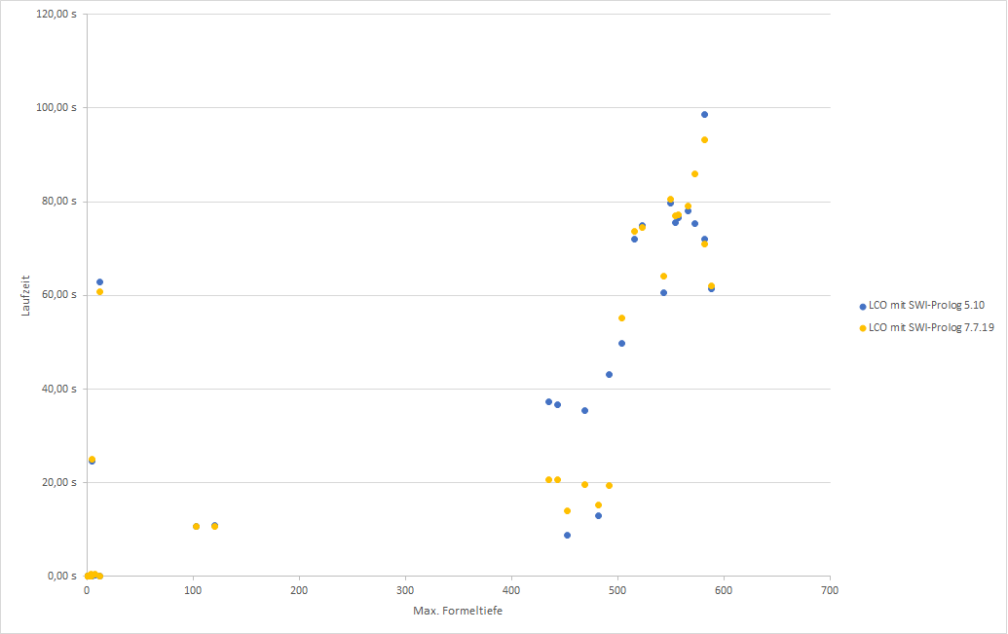

Abbildung 6.5: TFA mit Laufzeit beider LCO-Versionen

Abbildungsverzeichnis

Tabellenverzeichnis

Fußnotenverzeichnis

Literaturverzeichnis

[1] H.M. Edwards. An appreciation of Kronecker. In *The Mathematical Intelligencer*, volume 9, pages 28–35. Springer-Verlag, 1987. URL https://doi.org/10.1007/BF03023570. Zugriff am 27.08.2019.

[2] C. Kreitz and N. Brede. Theoretische Informatik I: Automaten und formale Sprachen. URL https://www.cs.uni-potsdam.de/ti/lehre/06-Theorie-I/slides.html. Einheit 1, S.5; Zugriff am 25.08.2019.

[3] K. Appel and W. Haken. Every planar map is four colorable. In *Contemp. Math.*, volume 98, 1989.

[4] G. Sutcliffe. The TPTP problem library and associated infrastructure. from CNF to TH0, TPTP v6.4.0. In *Journal of Automated Reasoning*, volume 59, pages 483–502, 2017.

[5] W. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. 1991.

[6] Sutcliffe G., Schulz S., Claessen K., and Baumgartner P. The TPTP Typed First-Order Form with Arithmetic. In N. Bjørner and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 7180 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2012.

[7] A. Colmerauer and P. Roussel. History of programming languages—ii. chapter The Birth of Prolog, pages 331–367. ACM, New York, NY, USA, 1996.

[8] W. Bibel. *Deduktion - Automatisierung der Logik*. R. Oldenbourg Verlag, 1992.

[9] W. Pugh and the entire Omega Project Team. The Omega Project. URL https://www.cs.umd.edu/projects/omega/handout.html. Zugriff am 09.04.2019.

[10] CASC-J5. Results - TFA Problems, Jul 2010. URL http://tptp.cs.miami.edu/CASC/J5/WWWFiles/Results.html#TFAProblems. Zugriff am 09.04.2019.

[11] G. Peano. *Arithmetices principia, nova methodo exposita*. Fratres Bocca, 1889. URL https://archive.org/details/arithmeticespri00peangoog/page/n8. Zugriff am 27.08.2019.

[12] D. Hilbert and P. Bernays. Grundlagen der Mathematik 1. In *Die Grundlehren der mathematischen Wissenschaften*, Band 40. Springer-Verlag, Berlin Heidelberg New York, 1968. Zweite Auflage.

[13] U. Hebisch and H.J. Weinert. *Halbringe: algebraische Theorie und Anwendungen in der Informatik*. Teubner, Stuttgart, 1993.

[14] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen,in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congres de Mathematiciens des Pays Slaves*, pages 92–101, 1929.

[15] R.L. Constable and M.J. O'Donnell. A programming logic: with an introduction to the PL/CV verifier. In *McGraw-Hill International Editions*. Winthrop Publishers, 1978.

[16] T. Chan. An algorithm for checking PL/CV - arithmetic inferences. 1977.

[17] D.C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, 1972.

[18] L. Hodes. Solving Problems by Formula Manipulation in Logic and Linear Inequalities. In *Artificial Intelligence*, volume 3, pages 165–174. North-Holland Publishing Company, 1972.

[19] H. Trölenberg. Arithmetik im automatischen Theorembeweisen. Master's thesis, Universität Potsdam, Institut für Informatik, 2009. Aufgabenstellung und Betreuung: Prof. Dr. Christoph Kreitz.

[20] C. C. Chang. G. Kreisel and J. L. Krivine. Elements of mathematical logic. (Model theory). North-Holland Publishing Company, Amsterdam1967, xi 222 pp. In *Journal of Symbolic Logic*, volume 34, pages 112–112. Cambridge University Press, Amsterdam, 1969.

[21] J.B.J. Fourier. Solution d'une question particulière du calcul des inégalités. In *Nouveau Bulletin des Sciences par la Société Philomathique de Paris*, pages 99–100, 1826.

[22] L. Dines. Systems of linear inequalities. In *Annals of Mathematics*, volume 20, pages 191–199, 1919.

[23] T.S. Motzkin. *Beiträge zur Theorie der linearen Ungleichungen*. PhD thesis, Universität Zürich, 1936.

[24] G. Dantzig. Linear programming and extensions. Princeton Univ. Press, 1963.

[25] A. Bockmayr and V. Weispfenning. Solving Numerical Constraints. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 12, pages 771–772. Elsevier Science B.V. Co-pub.: The MIT Press, 2001.

[26] H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. In *Journal of Combinatorial Theory*, volume 21, pages 118–123. Academic Press, Inc., 1976.

[27] W.W. Bledsoe. A Now Method for Proving Certain Presburger Formulas. In *IJCAI'75 Proceedings of the 4th International Joint Conference on Artificial Intelligence*, volume 1, pages 15–21, 1975.

[28] R.E. Shostak. On the SUP-INF method for proving presburger formulas. In *Journal of the ACM (JACM)*, volume 24, pages 529–543, 1977.

[29] W. Bibel. *Automated Theorem Proving*. Friedr. Vieweg & Sohn Verlagsgesellschaft mbH, 1982.

[30] A. Blake. *Canonical Expression in Boolean Algebra*. PhD thesis, University of Chicago, Illinois, 1937.

[31] J.A. Robinson. A machine-oriented logic based on the resolution principle. In *Journal of the ACM (JACM)*, volume 12, pages 23–41, 1965.

[32] C. Kreitz. Automatisierte Logik und Programmierung, . URL `https://www.cs.uni-potsdam.de/ti/lehre/downloads/ALUPI/slides-03.pdf`. Kapitel 3; Zugriff am 01.09.2019.

[33] C. Kreitz. Inferenzmethoden, . URL `https://www.cs.uni-potsdam.de/ti/lehre/06-Inferenzmethoden/slides/slides-02.pdf`. Kapitel 2; Zugriff am 01.09.2019.

[34] J. Otten. leanCoP 2.0 and ileanCoP 1.2: High Performance Lean Theorem Proving in Classical and Intuitionistic Logic (System Descriptions). In A. Armando, P. Baumgartner, and Dowek G., editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2008. IJCAR 2008.

[35] L. Volkmann. *Fundamente der Graphentheorie*. Springer-Verlag, 1996.

[36] L.C. Paulson. Isabelle A Generic Theorem Prover. In *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[37] M. Norrish. Complete Integer Decision Procedures as Derived Rules in HOL. In D. Basin and B. Wolff, editors, *TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 71–86, Berlin Heidelberg, 2003. Springer-Verlag.

[38] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliâtre, and et al. *The Coq Proof Assistant Reference Manual : Version 6.1.*, 1997. URL `https://hal.inria.fr/inria-00069968/document`. [Research Report] RT-0203, INRIA. 1997,pp.214. inria-00069968. Zugriff am 27.09.2019.

[39] P. Crégut. *Omega: a solver for quantifier-free problems in Presburger Arithmetic*. URL `https://coq.inria.fr/refman/addendum/omega.html`. Coq 8.9.1 Documentation. Zugriff am 28.09.2019.

[40] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl proof development system*. Prentice Hall, 1986.

[41] S. Allen, R. Constable, R. Eaton, C. Kreitz, and L. Lorigo. The Nuprl open logical environment. In D. McAllester, editor, *17th Conference on Automated Deduction*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer Verlag, 2000.

[42] P. Martin-Löf. Intuitionistic Type Theory. In *Studies in Proof Theory Lecture Notes*, volume 1. Bibliopolis, Napoli, 1984.

[43] C. Kreitz. *The Nuprl Proof Development System, Version 5*. Department of Computer Science, Cornell-University, Ithaca, NY 14853-7501, U.S.A., .

[44] J. Otten and W. Bibel. leancop: lean connection-based theorem proving. In *Journal of Symbolic Computation*, volume 36, pages 139–161. Elsevier Science B.V., 2003.

[45] W. Pugh, W. Kelly, V. Maslov, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Calculator and Library*, Nov 1996. version 1.1.0.

[46] W. Pugh, W. Kelly, V. Maslov, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Library*, Nov 1996. URL `http://www.cs.umd.edu/projects/omega`. Version 1.1.0, Zugriff am 11.09.2019.

[47] J. Otten. Restricting backtracking in connection calculi. In *AI Communications*, 2018.

[48] R. Nieuwenhuis and A. Rubio. Paramodulation-Based Theorem Proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume 1, chapter 7. Elsevier Science B.V. Co-pub.: The MIT Press, 2001.

[49] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS Version 3.5. In R.A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer, Berlin, Heidelberg, 2009.

[50] G. Sutcliffe, S. Schulz, K. Claessen, and P. Baumgartner. The TPTP typed first-order form with arithmetic.

[51] A. Voronkov and L. Kovács. First-Order Theorem Proving and VAMPIRE. 2013.

[52] A. Voronkov and L. Kovács. Integrating Linear Arithmetic into Superposition Calculus. 2007.