Natürliche Deduktion in Clojure

Masterarbeit

Studiengang Medieninformatik

vorgelegt von

Tobias Völzel

November 2015

Referent der Arbeit:	Prof. Dr. Burkhardt Renz
Korreferent der Arbeit:	Prof. Dr. Wolfgang Henrich

Selbstständigkeitserklärung

Ich erkläre, dass ich die eingereichte Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die von mir angegebenen Quellen und Hilfsmittel nicht verwendet und die den benutzten Werken wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Die vorgelegte Arbeit wurde zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht.

Friedberg, November 2015

Tobias Völzel

Abstract

Diese Arbeit beleuchtet die Entwicklung eines Programms zur Anwendung der natürlichen Deduktion auf Hypothesen der Aussagen-, Prädikaten- und linearen temporalen Logik. Das Programm überprüft die Parameter für die Inferenzregeln der entsprechenden Logik, wendet diese auf die ausgewählten Formeln eines Beweises an und fügt die Ergebnisse dementsprechend in dessen Struktur ein. Gedacht ist das Programm sowohl für Anfänger zum Einstieg in die Logik, wie auch für fortgeschrittene Nutzer, die es als Unterstützung bei komplizierten Beweisen verwenden können.

Es werden sowohl Grundlagen der natürlichen Deduktion und deren Regeln wie auch die Details der Implementierung genauer erläutert. Zur Umsetzung werden die Programmiersprache Clojure sowie die Bibliothek „core.logic“ verwendet. Im Vergleich zu bestehender Software liegt der Fokus des Programms auf der Erweiterung um neue Regeln und Logiken sowie einer guten Wartbarkeit.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Informatik und Logik verbindet eine lange, gemeinsame Geschichte. Den Fortschritten in der Logik ab dem Ende des 19. Jahrhunderts verdanken wir die Entwicklung der ersten Computer und Programmiersprachen [Val14]. Auch heute noch spielen logische Methoden und Konzepte in der Informatik eine zentrale Rolle [HHI+01]. Die Anwendungsgebiete reichen von Rechnerarchitektur über Softwaretechnik bis hin zu künstlicher Intelligenz.

1.1 Motivation

Ein wichtiger Teil der Logik sind die sog. Kalküle. Diese beschreiben über Regeln und Axiome, wie sich aus gegebenen logischen Aussagen weitere Aussagen ableiten lassen. Solche Kalküle werden genutzt, um etwa die Gültigkeit von logischen Hypothesen zu überprüfen oder um aus gegebenen Aussagen weitere Schlüsse zu ziehen. Das Kalkül, mit welchem wir uns in dieser Arbeit beschäftigen, ist die „Natürliche Deduktion“.

Bei der natürlichen Deduktion werden die sog. Inferenzregeln Schritt für Schritt auf bestehende logische Behauptungen angewandt, um diese entweder zu beweisen oder zu widerlegen. Ihre „Natürlichkeit“ kommt dabei von den einfachen und intuitiv zu verstehenden Regeln. Sie bietet damit auch gerade in der Lehre einen guten Einstiegspunkt in das Themengebiet der Logik. Außerdem stärkt sie den eigenen Fokus für syntaktische Schlussfolgerungen, was besonders in der Informatik von großer Bedeutung ist [Bor05b].

Gerade für Anfänger ist es dabei nicht sofort ersichtlich, welche Regeln an welcher Stelle angewendet werden können und welche Ergebnisse dadurch entstehen. Und auch im Bereich der Logik erfahrene Personen können bei komplizierten Beweisen schnell den Überblick verlieren. Ideal ist daher eine Software, welche prüft, ob eine Regel ausgeführt werden darf, dies im korrekten Fall auch tut und das Ergebnis ordentlich

formatiert ausgibt. So lernt ein Anfänger schnell, was möglich ist und was nicht, während ein fortgeschrittener Nutzer bei komplizierten Beweisen entlastet wird.

1.2 Zielsetzung

Zur Unterstützung von Personen, welche sich mit natürlicher Deduktion beschäftigen, soll ein unterstützendes Programm entwickelt werden. Mit diesem können Interessierte Schritt für Schritt die verschiedenen Regeln auf die zu beweisenden Hypothesen anwenden. Die Schritte werden automatisch ausgewertet und die Ergebnisse in die Beweisstruktur eingepflegt. So lernen Einsteiger die korrekte Anwendung der Regeln und deren Auswirkungen, während fortgeschrittene Nutzer ein nützliches Werkzeug zum Beweisen von logischen Hypothesen erhalten.

Zu Beginn soll das Projekt die natürliche Deduktion für Aussagen-, Prädikaten- und lineare temporale Logik (LTL) unterstützen. Weitere Regeln sollen auch von wenig erfahrenen Nutzern leicht eingebunden werden können. Durch eine offene Architektur soll auch das Erweitern um spezielle Sonderfälle oder neue Logiken einfach möglich sein. Dies soll erlauben, das System nach den eigenen Wünschen zu konfigurieren oder abzuwandeln und es in eigenen Projekten zu verwenden.

Entwickelt wird das Projekt in der Programmiersprache Clojure[1], einem modernen Lisp-Dialekt von Rich Hickey. Clojure ist eine funktionale und dynamische Sprache, welche auf der Java Virtual Machine läuft. Sie verfügt zudem über eine integrierte REPL, welche für die Interaktion mit dem Programm und zur Ausgabe genutzt werden soll. Auf eine grafische Benutzeroberfläche wird aus zeitlichen Gründen zunächst verzichtet. Diese sollte aber bei Bedarf leicht nachzurüsten sein. Für die Anwendung der verschiedenen Regeln der natürlichen Deduktion wird die Clojure-Bibliothek „core.logic“ verwendet. Die Implementierung in Clojure ermöglicht es außerdem, das Projekt zu einem späteren Zeitpunkt mit der von Prof. Dr. Burkhardt Renz entwickelten „Logic Workbench“[2] zusammenzuführen, um diese durch ein zusätzliches Werkzeug für das Arbeiten mit Logik zu erweitern.

1.3 Aufbau der Arbeit

In Kapitel 2 wird erläutert, welche verwandten Arbeiten es bereits gibt, wie ähnliche Programme aussehen, funktionieren und was sie leisten. Es werden Besonderheiten

[1] `www.clojure.org`
[2] `https://github.com/esb-dev/lwb`

und Schwächen der einzelnen Projekte näher beleuchtet. Abschließend wird zusammengefasst, inwiefern sich das hier vorgestellte Projekt von diesen unterscheidet und an welchen Stellen Verbesserungen vorgenommen werden.

Kapitel 3 erläutert die Grundlagen der natürlichen Deduktion. Dazu gehören neben einer Erläuterung der Regeln im Allgemeinen auch die richtige Notation und Gültigkeit von Beweisen. Daraufhin wird der Ablauf eines Beweises inklusive der verschiedenen Anwendungsmöglichkeiten der Regeln und der Einführung von Unterbeweisen erklärt. Schließlich werden die Regeln der implementierten Logiken (Aussagen-, Prädikaten- und lineare temporale Logik) aufgelistet und an einigen Beispiele erläutert.

Im darauffolgenden Kapitel 4 wird die Implementierung genauer betrachtet. Dafür wird zunächst die interne Struktur von Formeln, Regeln und Beweisen näher erläutert. Ebenso wird die Umwandlung von Regeln in Funktionen mittels core.logic, deren Anwendung auf einen Beweis, das Einfügen der Ergebnisse in die Beweisstruktur und das Abspeichern und Wiederverwenden von Beweisen erklärt. Die in Kapitel 3 gezeigten Beispiele der verschiedenen Logiken werden mit dem so entwickelten Programm bewiesen, um zu zeigen, wie das System funktioniert. Anschließend wird auf mögliche Anpassungen und Erweiterungen eingegangen.

Im abschließenden Kapitel 5 wird das Ergebnis noch einmal zusammengefasst und präsentiert. Es wird besprochen, welche Ziele erreicht wurden und wo eventuell Probleme auftraten. Schlussendlich gibt es einen Ausblick auf mögliche Erweiterungen und die Zukunft des Projekts.

Kapitel 2

Verwandte Arbeiten

In diesem Kapitel werden einige verwandte Arbeiten und Programme vorgestellt, die es dem Nutzer ermöglichen, logische Beweise Schritt für Schritt mittels natürlicher Deduktion herzuleiten. Diese Übersicht ist dabei nur ein Ausschnitt der erhältlichen Logiksoftware, welche die natürliche Deduktion in ihrem Funktionsumfang enthält[3]. Dabei sind die Ansätze der hier gezeigten Programme, insbesondere deren Bedienung, doch sehr unterschiedlich.

2.1 Jape

Jape steht für „just another proof editor“ und ist das wohl populärste Programm zum computerunterstützten Lernen von Logik [HKRW10]. Entwickelt wurde es von Richard Bornat und Bernard Sufrin. Die aktuellste Version wurde auf die Programmiersprache OCaml portiert und mit einer grafischen Bedienoberfläche in Java versehen [Bor05a].

Neben der natürlichen Deduktion sind noch weitere Beweissysteme standardmäßig implementiert und einsatzbereit. Außerdem ist es möglich, diese bei Bedarf den eigenen Bedürfnissen anzupassen. So lassen sich etwa Menüeinträge umbenennen oder weitere Funktionen einbauen. Weitere Informationen zum Hinzufügen eigener Logiken finden sich in einer äußerst ausführlichen Anleitung von Richard Bornat [Bor07].

Momentan wird Jape im Kern von seinen Erfindern nicht weiter entwickelt. Der Quellcode ist jedoch öffentlich zugänglich, sodass theoretisch jeder die Entwicklung weiter vorantreiben könnte. Allerdings ist die Software sehr kompliziert, sodass damit laut Bornat eher nicht zu rechnen ist [Bor07].

[3] Eine umfangreiche Liste von Logiksoftware wurde von Hans van Ditmarsch unter `www.ucalgary.ca/aslcle/logic-courseware` zusammengestellt.

Jape ist mit seiner guten Bedienbarkeit und dem großen Funktionsumfang sicherlich eines der besten Programme zur Arbeit mit natürlicher Deduktion. Lediglich die schwierige Wartbarkeit fällt leicht negativ ins Gewicht.

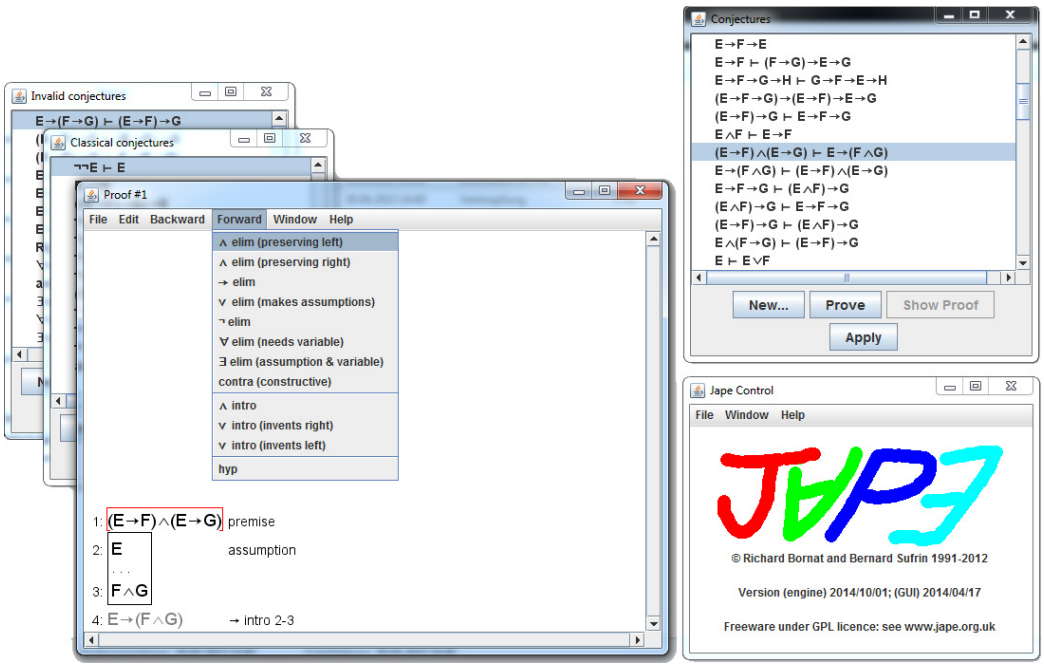


Abbildung 2.1: Jape

2.2 ProofWeb

Das von Cezary Kaliszyk, Freek Wiedijk, Maxim Hendriks und Femke van Raamsdonk entwickelte ProofWeb ist speziell zum Lernen von natürlicher Deduktion an Universitäten und Hochschulen entwickelt worden [HKRW10]. Die Plattform liegt auf einem zentralen Server[4], der über ein Webinterface erreichbar ist und bereits eine große Anzahl an Übungen zur Verfügung stellt.

ProofWeb basiert auf dem Beweisassistenten „coq“[5]. Mit Hilfe der Weboberfläche schreibt der Nutzer ein spezielles Skript, welches dann von dem Assistenten überprüft wird. Um nicht alle Befehle auswendig lernen zu müssen, gibt es eine grafische Oberfläche, aus der sich die verschiedenen Schritte auswählen lassen. Dabei stehen alle Regeln der Aussagen- und Prädikatenlogik zur Verfügung. Zudem gibt es eine umfangreiche Anleitung [KRW$^+$], welche man unbedingt lesen sollte, da nicht alle Schritte in ProofWeb sofort verständlich sind. So werden etwa Zeilen eines Beweises

[4] `http://proofweb.cs.ru.nl/login.php`
[5] `https://coq.inria.fr`

nicht über ihre Nummerierung, sondern über spezielle „Labels“ angesprochen. Diese müssen bei einigen Funktionen vom Nutzer explizit gesetzt werden. Welche Funktionen das sind, ist dabei nicht sofort ersichtlich und muss in der entsprechenden Definition nachgeschlagen werden. Außerdem erkennt ProofWeb nicht automatisch, wenn das Ergebnis einer Regel dem eines schon vorhandenen Labels entspricht. Hier muss der Nutzer ProofWeb manuell mittels der Funktion „exact“ darauf aufmerksam machen (siehe [KRW+] S. 30/31).

Das flexible Webinterface, die zahlreichen Übungsaufgaben sowie die Möglichkeit für Professoren, die Fortschritte ihrer Studenten zu beobachten, gehören zu den Stärken von ProofWeb. Allerdings muss die Bedienung vom Nutzer erst erlernt werden. Die verschiedenen Skript-Befehle sind trotz Vorgaben nicht immer sofort ersichtlich und es gibt einige Besonderheiten, die beachtet werden müssen. Die Möglichkeit, eigene weitere Regeln zu definieren oder Beweise abzuspeichern, um sie an anderer Stelle wiederzuverwenden, fehlt bisher.

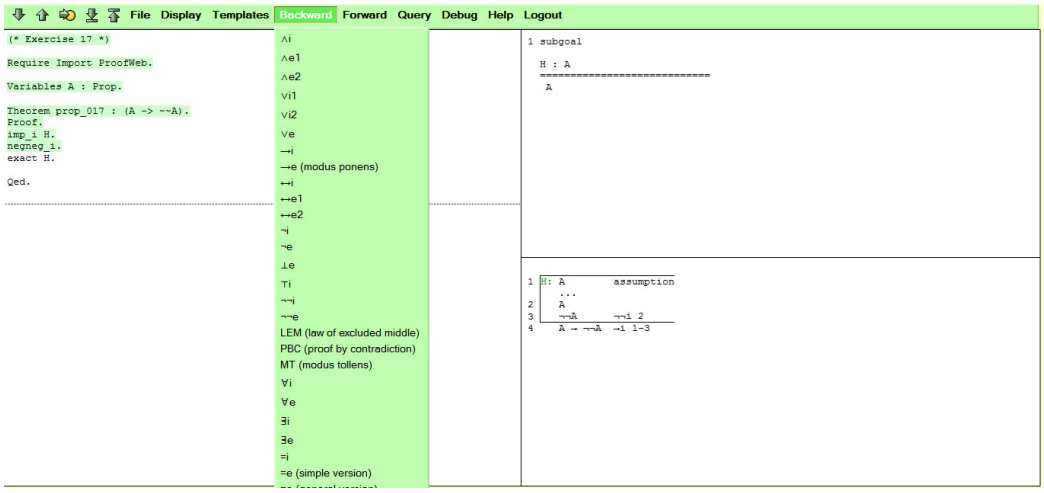


Abbildung 2.2: ProofWeb

2.3 uProve

Das Programm uProve wurde von Niklas Udd entwickelt. Es ermöglicht die natürliche Deduktion der Aussagenlogik und wurde in Java programmiert.

Die Bedienung über die grafische Oberfläche geht äußerst einfach von der Hand. Man wählt eine Zeile aus und bekommt automatisch alle darauf anwendbaren Regeln und deren Ergebnisse angezeigt. Sämtliche Regeln lassen sich nur vorwärts anwenden. Statt Unterbeweisen können jederzeit Annahmen in den Beweis eingefügt werden.

Ein vorzeitiges Lösen eines Beweises durch das Einfügen des Ergebnisses als Annahme wird von uProve zwar verhindert, aber die Übersicht leidet unter den fehlenden Unterbeweisen. Eigene Regeln lassen sich nicht definieren. Auch kann man keine Beweise abspeichern, um sie an anderer Stelle wiederzuverwenden.

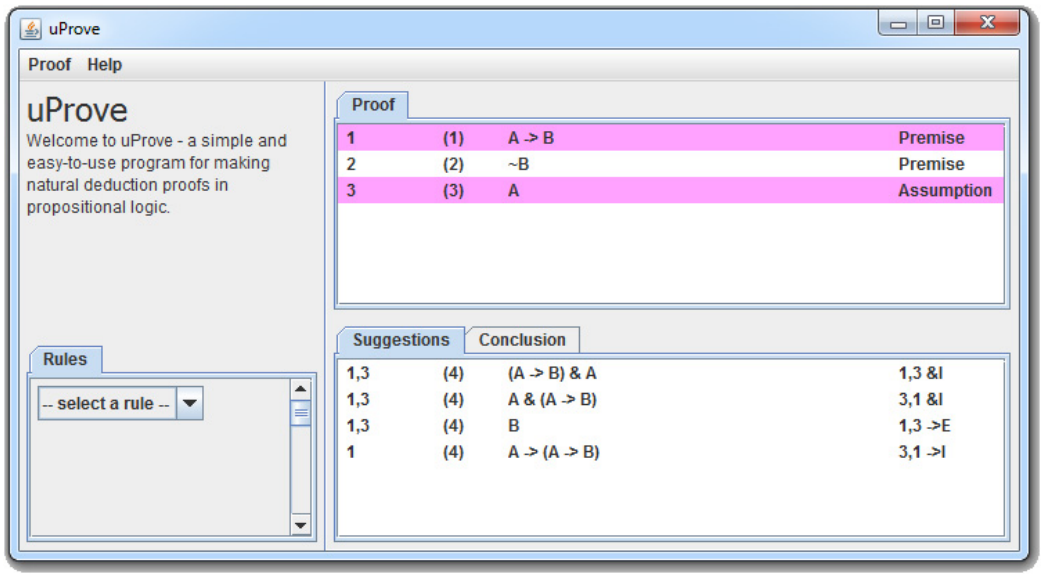


Abbildung 2.3: uProve

2.4 Panda

Die Dozenten Olivier Gasquet, Francois Schwarzentruber und Martin Strecker entwickelten Panda, da sie sich eine Software wünschten, welche ihren Anforderungen und Bedürfnissen in der Lehre entsprach [GSS11]. Programmiert wurde Panda in Java.

Panda kommt mit einer grafischen Oberfläche und kann nahezu ausschließlich mit der Maus bedient werden. Wählt man eine Formel aus, werden einem schon passende Rückwärtsschritte oder Hypothesen angeboten, die man durch anklicken hinzufügen bzw. anwenden kann. Alternativ lassen sich alle Regeln aus einer separaten Box auswählen. Für Vorwärtsschritte fügt man die entsprechenden Bedingungen nebeneinander und zieht mit der Maus einen Querstrich unter diesen. Das Ergebnis wird daraufhin automatisch eingefügt. Die Darstellung entspricht einem Baum nach Gentzen.

Das ungewöhnliche Bedienkonzept muss erst verinnerlicht werden, geht aber dann gut von der Hand. Durch die automatischen Vorschläge und die sehr knappe Beschriftung der Regeln passiert es aber leicht, dass man sich durch einen Beweis „durchklickt“, ohne den Hintergrund wirklich verstanden zu haben. Weitere Logiken oder

Regeln lassen sich nicht hinzufügen und auch ein Abspeichern und Wiederverwenden von Beweisen fehlt.

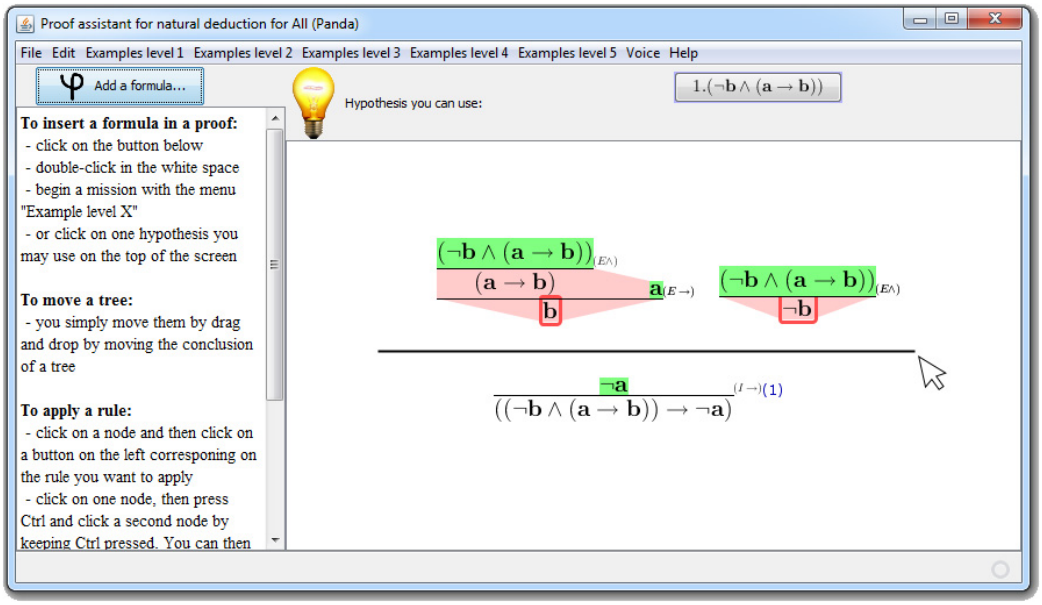


Abbildung 2.4: Panda

2.5 Natural Deduction

Das von Daniel Kirsten entwickelte „Natural Deduction“ [Kir14] kommt diesem Projekt mit Abstand am nächsten. Es wurde ebenfalls an der THM in Clojure entwickelt und bietet natürliche Deduktion mit Aussagen- und Prädikatenlogik. Es ist zum Zeitpunkt dieser Arbeit das einzige Projekt, welches in Clojure implementiert wurde.

Es verfügt über keine grafische Nutzeroberfläche, sondern wird in der Clojure REPL ausgeführt. Regeln lassen sich vorwärts wie rückwärts anwenden, Unterbeweise können eingefügt werden. Außerdem ist es möglich, erfolgreich gelöste Beweise als Theoreme abzuspeichern und in anderen Beweisen als Regel wiederzuverwenden.

2.6 Abgrenzung

Alle hier gezeigten Programme können Hypothesen der Aussagen- und Prädikatenlogik (uProve nur Aussagenlogik) mittels natürlicher Deduktion beweisen. Ansätze und Bedienung mögen sich zwar unterscheiden, die Kernfunktionen der Regelanwendung sind aber nahezu immer dieselben. Dabei sind die entsprechenden Inferenzregeln

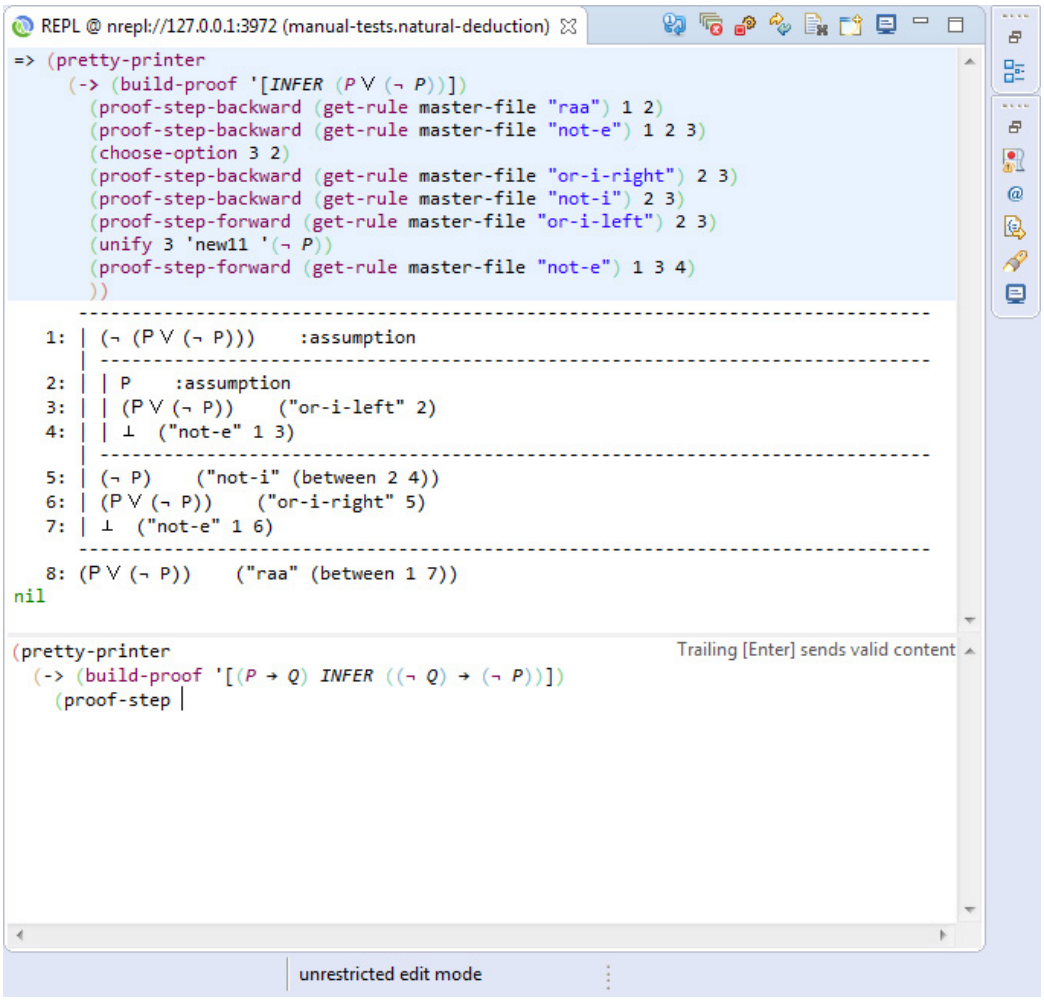


Abbildung 2.5: Natural Deduction

meist fest im Programm verankert und der Quellcode oft nicht zugänglich, wodurch eine Erweiterung sehr erschwert wird.

Das in dieser Arbeit vorgestellte Projekt soll die lineare temporale Logik (LTL) als eine weitere Logik ergänzen und die Möglichkeiten der Regelanwendung um einige dafür notwendige Optionen erweitern. Dazu gehören das Anwenden von Regeln auf Teile von Formeln, triviale Theoreme (Regeln, die mit den Wahrheitswerten *true* und *false* interagieren) sowie die Möglichkeit, bewiesene Hypothesen als Theoreme abzuspeichern und wiederzuverwenden. Letzteres ist gerade bei langen Beweisketten eine enorme Erleichterung.

Darüber hinaus liegt der Fokus auf möglichen Erweiterungen und der Wartbarkeit des Projekts. Das beinhaltet das Hinzufügen und Abändern von Regeln, sowie die Programmierung neuer Logiken und deren Sonderfälle. Dafür wird der Quellcode entsprechend kommentiert und öffentlich zugänglich gemacht.

Anders als bei den meisten der hier vorgestellten Programme wird zunächst auf eine grafische Bedienoberfläche verzichtet. Als Schnittstelle dient die Clojure REPL. Ein späteres Hinzufügen einer GUI wird durch den Quellcode aber nicht ausgeschlossen.

Ein sehr ähnliches Projekt wurde bereits mit „Natural Deduction“ von Daniel Kirsten vorgestellt. Es wurde ebenfalls in Clojure entwickelt und hat ähnliche Ansprüche. Grund für die hier vorgestellte „Neuentwicklung“ sind einige Änderungen und Verbesserungen, welche die Anwendung noch erleichtern und erweitern sollen. So sollen etwa die Formeln ohne Sonderzeichen auskommen und von Infixnotation auf Präfixnotation umgestellt werden (siehe Kapitel 4.1), um sich besser in den Stil der Programmiersprache Clojure einzufügen. Außerdem wird die Definition von eigenen Regeln weiter vereinfacht. Dazu gesellen sich weitere neue Funktionen sowie einige Verbesserungen.

Auch wenn die Grundfunktionen dieselben sind, wurde zur Entwicklung kein Programmcode übernommen. Das hier vorgestellte Projekt kann somit als nahezu komplette Neuentwicklung angesehen werden.

Kapitel 3

Grundlagen

In diesem Kapitel sollen die Grundlagen der natürlichen Deduktion näher erläutert werden. Ziel dabei ist es, ein grundlegendes Verständnis für die Arbeit mit ihr aufzubauen und ihren Ablauf zu verstehen. Dabei kratzt dieses Kapitel verständlicherweise nur an der Oberfläche des Themas. Es ersetzt daher keineswegs Vorlesungen oder Literatur, welche sich mit diesem Thema üblicherweise noch viel tiefer und ausführlicher auseinandersetzen.

Für einen tieferen Einblick empfehle ich das Buch „Natural Deduction Proof and Disproof in Formal Logic“ von Richard Bornat [Bor05b].

3.1 Definition

Die natürliche Deduktion ist ein Kalkültyp der Logik. Über sog. Inferenzregeln werden dabei aus gegebenen Aussagen andere Aussagen abgeleitet, i.d.R. mit dem Ziel, logische Hypothesen zu beweisen. Jene Regeln lehnen sich dabei eng an die Denkweise des Menschen und dessen „natürliche“ Art zu argumentieren an. Das macht die natürliche Deduktion besonders einfach zu verstehen und bietet damit auch Anfängern einen leichten Einstieg in die Logik. Entwickelt wurde die natürliche Deduktion von Stanisław Jaśkowski [Jaś34] und Gerhard Gentzen [Gen35] nahezu zeitgleich und unabhängig voneinander.

Die natürliche Deduktion gehört dabei zu den Kalkültypen, die ohne semantische Bedeutung auskommen. Ihre Regeln und Schlussfolgerungen operieren unabhängig vom Wahrheitsgehalt der betrachteten Aussagen.

Ein Kalkül der natürlichen Deduktion ist nicht streng definiert. Stattdessen besitzt es eine Reihe von Eigenschaften, welche je nach Kalkül unterschiedlich stark zutreffen können [Pel01]. In der Regel besteht ein Kalkül der natürlichen Deduktion jedoch immer aus wenigen bis gar keinen Axiomen (Grundsätzen) und vielen Inferenzregeln.

Letztere sollen vor allem einfach und intuitiv zu verstehen und zu rechtfertigen sein (siehe Kapitel 3.3).

3.2 Elemente

Im Folgenden sollen nun die Struktur und Funktionsweise der Regeln und Beweise erläutert werden.

3.2.1 Regeln

Die Inferenzregeln sind der Kern der natürlichen Deduktion. Welche Regeln gelten, hängt von der entsprechenden Logik ab. Für die folgenden Beispiele bewegen wir uns innerhalb der Aussagenlogik. Genauere Informationen zu weiteren Logiken und den dazugehörigen Regeln folgen in Kapitel 3.4.

Jede Regel hat bestimmte Voraussetzungen und daraus resultierende Schlussfolgerungen. Die Schreibweise erklärt sich wie folgt: Alle Voraussetzungen stehen über dem Querstrich, alle daraus resultierenden Schlussfolgerungen darunter. Die verschiedenen Großbuchstaben stehen für beliebige Aussagen:

$$\frac{A \qquad B}{(A \wedge B)} \qquad \frac{(A \wedge B)}{A} \qquad \frac{(A \wedge B)}{B}$$

(a) Einführung (b) Elimination 1 (c) Elimination 2

Abbildung 3.1: Einführungs- und Eliminationsregeln der Konjunktion ($\wedge$, AND)

In Abbildung 3.1 werden die Regeln für die Einführung sowie die Elimination der Konjunktion gezeigt. Eine Einführungsregel schlussfolgert aus bestimmten Voraussetzungen eine Aussage mit dem entsprechenden Operator als Hauptoperator. Dagegen nimmt eine Eliminationsregel eine Aussage mit dem entsprechenden Operator als Hauptoperator und bildet daraus eine Aussage ohne diesen. Üblicherweise gibt es in einem Kalkül der natürlichen Deduktion für jeden logischen Operator zumindest eine Einführungs- und eine Eliminationsregel. Es gibt aber auch diverse Kalküle, bei denen dies nicht der Fall ist [Pel01].

Manche dieser Regeln benötigen Unterbeweise als Voraussetzung(en). Unterbeweise werden in einem Kasten dargestellt, wobei die obere Aussage eine Annahme und die untere Aussage die zu beweisende darstellt. Ein Beispiel für eine solche Regel ist die Einführung der Implikation:

$$\frac{\boxed{\begin{array}{c} A \\ \dots \\ B \end{array}}}{(A \to B)}$$

(a) Einführung

$$\frac{(A \to B)\ A}{B}$$

(b) Elimination

Abbildung 3.2: Einführungs- und Eliminationsregel der Implikation ($\to$, IMPL)

Der Unterbeweis in Abbildung 3.2a erklärt sich wie folgt: Wenn man logisch beweisen kann, dass sich aus der Annahme A die Aussage B ableiten lässt, dann kann man sagen „aus A folgt B“ $(A \to B)$.

3.2.2 Beweise

Mit Hilfe solcher Regeln versucht man nun, Ableitungen oder logische Aussagen zu beweisen. Dies geschieht entweder, indem man aus ein oder mehreren Vorbedingungen $A_1, \dots, A_n$ die Schlussfolgerung C ableitet (siehe Abb. 3.3a) oder indem man die Schlussfolgerung C ohne Vorbedingungen beweisen kann (siehe Abb. 3.3b).

$$A_1, \dots, A_n \vdash C$$

(a) Beweis einer Ableitung

$$\vdash C$$

(b) Beweis einer Aussage

Abbildung 3.3: Schreibweise von Beweisen

Um mit Beweisen arbeiten zu können, bringen wir sie in die Zeilenform nach Huth und Ryan [HR04], wie sie auch von Bornat benutzt wird [Bor05b]. Zeilen sind auf der linken Seite nummeriert, gefolgt von der logischen Formel und ihrem Ursprung. Jede bewiesene Zeile ist entweder eine Vorbedingung (premise), eine Annahme (assumption) oder eine Ableitung aus vorherigen Zeilen. In letzterem Fall gibt der Ursprung die Regel sowie die Zeilennummern an, aus denen die Zeile gebildet wurde. Bei noch nicht bewiesenen Zeilen ist dementsprechend noch kein Ursprung angegeben. Die Reihenfolge der Zeilen ist: Vorbedingungen, Ableitungen, Schlussfolgerung.

Ich bin ein Mensch, Mein Name ist Hans
$\vdash$ Ich bin ein Mensch und mein Name ist Hans

1.	*Ich bin ein Mensch*	Vorbedingung
2.	*Mein Name ist Hans*	Vorbedingung
3.	*Ich bin ein Mensch und mein Name ist Hans*	$\wedge$ Einführung (1, 2)

Abbildung 3.4: Ein beispielhafter Beweis. Die ersten beiden Zeilen sind als Vorbedingungen gekennzeichnet. Die Schlussfolgerung steht ganz unten und wurde durch die $\wedge$ Einführung mit den Zeilen eins und zwei bewiesen.

Manche Beweise haben neben einfachen Zeilen zusätzlich noch Unterbeweise. Diese werden immer dann benötigt, wenn man eine Annahme machen muss. Dargestellt werden Unterbeweise durch Kästen, die um sie gezogen sind (siehe Abb. 3.5). Jeder Beweis kann beliebig viele Zeilen und Unterbeweise beinhalten. Unterbeweise wiederum können ebenfalls beliebig viele Zeilen und weitere Unterbeweise beinhalten. So ist es möglich, Beweise beliebig tief zu verschachteln.

(Es ist Herbst $\rightarrow$ Es regnet), (Es regnet $\rightarrow$ Ich werde nass)
$\vdash$ (Es ist Herbst $\rightarrow$ Ich werde nass)

1.	(*Es ist Herbst* $\rightarrow$ *Es regnet*)	Vorbedingung
2.	(*Es regnet* $\rightarrow$ *Ich werde nass*)	Vorbedingung
3.	*Es ist Herbst*	Annahme
4.	*Es regnet*	$\rightarrow$ Elimination (1, 3)
5.	*Ich werde nass*	$\rightarrow$ Elimination (2, 4)
6.	(*Es ist Herbst* $\rightarrow$ *Ich werde nass*)	$\rightarrow$ Einführung (3-5)

Abbildung 3.5: Ein Beispiel für einen Beweis mit einem Unterbeweis. Zeile sechs wird durch die $\rightarrow$ Einführung bewiesen und verweist als Argument auf den kompletten Unterbeweis/Kasten von Zeile drei bis fünf.

Gültigkeit von Beweisen

Um die Gültigkeit eines Beweises zu wahren, müssen die folgenden zwei Bedingungen zu jeder Zeit gegeben sein:

- Jede bewiesene Zeile ist entweder eine Vorbedingung, eine Annahme oder ist durch die Anwendung einer Regel entstanden, welche auf **vorherige** Zeilen und/oder Unterbeweise angewandt wurde (siehe Abb. 3.6 als Gegenbeispiel).
- Bezieht sich eine Regel auf eine vorherige Zeile innerhalb eines Unterbeweises, so muss dieser Unterbeweis auch die entstandene Zeile umschließen (siehe Abb. 3.7 als Gegenbeispiel).

Oder verkürzt nach Bornat [Bor05b]:

> - no looking downwards
> - no peeking inside a box

1.	$(Ich\ stehe\ im\ Regen \rightarrow Ich\ werde\ nass)$	Vorbedingung
2.	$(Ich\ werde\ nass \rightarrow Ich\ stehe\ im\ Regen)$	Vorbedingung
3.	$Ich\ werde\ nass$	$\rightarrow$ Elimination (1, 4)
4.	$Ich\ stehe\ im\ Regen$	$\rightarrow$ Elimination (2, 3)
5.	$Ich\ stehe\ im\ Regen\ \wedge Ich\ werde\ nass$	$\wedge$ Einführung (3, 4)

Abbildung 3.6: Ungültiger Beweis, da die $\rightarrow$ Elimination in Zeile drei auf die noch folgende Zeile vier verweist. Diese wiederum verweist auf Zeile drei, ein Tauschen der beiden Zeilen würde das Problem also nicht lösen: Eine zyklische Abhängigkeit liegt vor.

1.	A	Vorbedingung
2.	B	Annahme
3.	$A \wedge B$	$\wedge$ Einführung (1, 2)
4.	$B \to A \wedge B$	$\to$ Einführung (2-3)
5.	C	Annahme
6.	$B \wedge C$	$\wedge$ Einführung (2, 5)
7.	$C \to B \wedge C$	$\to$ Einführung (5-6)
8.	$(B \to (A \wedge B)) \wedge (C \to (B \wedge C))$	$\wedge$ Einführung (4, 7)

Abbildung 3.7: Ungültiger Beweis, da die $\wedge$ Einführung in Zeile sechs auf Zeile zwei innerhalb eines Unterbeweises (2-3) verweist, welcher Zeile sechs nicht umschließt.

3.3 Ablauf

Es wird mit einer Hypothese, also einem Beweis, in dem noch nicht alle Zeilen einen Ursprung besitzen bzw. bewiesen sind, begonnen. Um zu zeigen, dass eine Zeile noch bewiesen werden muss, fügt man eine Zeile mit „. . . “ vor der betroffenen Zeile ein.

$$A \to B, C \wedge D \vdash A \to (B \wedge C)$$

1.	$A \to B$	Vorbedingung
2.	$C \wedge D$	Vorbedingung
3.	. . .	
4.	$A \to (B \wedge C)$	

Abbildung 3.8: Eine Hypothese in Zeilenform

Regeln lassen sich auf zwei unterschiedliche Arten anwenden: vorwärts und rückwärts. Meistens müssen beide Ansätze kombiniert werden, um ein Ergebnis zu erreichen.

Soll eine Regel vorwärts angewendet werden, so werden alle entsprechenden Vorbedingungen als bewiesene Zeilen im vorliegenden Beweis benötigt. Dann kann das Ergebnis der Regel in den Beweis eingefügt werden. Natürlich muss auf die Gültigkeit des Beweises geachtet werden. Die neue Zeile wird demnach unterhalb der Vorbedingungen eingefügt und darf nur auf Zeilen in ihrem Gültigkeitsbereich verweisen.

1.	$A \to B$	Vorbedingung
2.	$C \wedge D$	Vorbedingung
3.	C	$\wedge$ Elimination (2)
4.	...	
5.	$A \to (B \wedge C)$	

Abbildung 3.9: Die $\wedge$ Elimination wurde vorwärts auf Zeile zwei angewandt. Das Ergebnis wurde unterhalb dieser in den Beweis eingeführt.

Zum rückwärts Anwenden einer Regel muss die Schlussfolgerung als unbewiesene Zeile im Beweis vorliegen. In diesem Fall werden alle Vorbedingungen der Regel oberhalb der ausgehenden Zeile eingefügt. Die neu eingefügten Zeilen sind zunächst unbewiesen, es sei denn, dass einige von ihnen schon als bewiesene Zeilen im Gültigkeitsbereich existieren. Die Zeile, welche zur Regelanwendung als Schlussfolgerung gewählt wurde, verweist nun in ihrer Ursprungsangabe auf die neu eingefügten Zeilen. Das rückwärts Anwenden ist die einzige Möglichkeit, Unterbeweise mit Annahmen in den Beweis einzufügen.

1.	$A \to B$	Vorbedingung
2.	$C \wedge D$	Vorbedingung
3.	C	$\wedge$ Elimination (2)
4.	A	Annahme
5.	...	
6.	$B \wedge C$	
7.	$A \to (B \wedge C)$	$\to$ Einführung (4-6)

Abbildung 3.10: Die $\to$ Einführung wurde rückwärts auf die letzte Zeile des Beweises angewandt. Dadurch wurde ein Unterbeweis mit einer Annahme eingefügt, der jetzt bewiesen werden muss.

Mit diesen beiden Möglichkeiten der Anwendung und allen zur Verfügung stehenden Regeln werden nun so lange weitere Schritte vollzogen, bis alle Zeilen des Beweises bewiesen sind. Sollte dies gelingen, wird aus der Hypothese ein Theorem. Dieses kann anschließend in anderen Beweisen wiederverwendet werden, indem es wie eine Regel eingesetzt wird.

1.	$A \to B$	Vorbedingung
2.	$C \wedge D$	Vorbedingung
3.	C	$\wedge$ Elimination (2)
4.	A	Annahme
5.	B	$\to$ Elimination (1, 4)
6.	$B \wedge C$	$\wedge$ Einführung (3, 5)
7.	$A \to (B \wedge C)$	$\to$ Einführung (4-6)

(a)

$$\frac{A \to B \quad C \wedge D}{A \to (B \wedge C)}$$

(b)

Abbildung 3.11: Die Hypothese wurde durch die Vorwärts-Anwendung der $\to$ Elimination und der $\wedge$ Einführung vollständig bewiesen (a). Nun kann das vorliegende Theorem in anderen Beweisen als Regel verwendet werden (b).

3.4 Regeln

In diesem Abschnitt werden nun alle für das Projekt relevanten Logiken und ihre zugehörigen Regeln kurz vorgestellt. Darunter fallen die Aussagenlogik, die Prädikatenlogik und die lineare temporale Logik (LTL).

3.4.1 Aussagenlogik

Die Aussagenlogik beschäftigt sich mit dem Verknüpfen und Modifizieren von verschiedenen Aussagen, um kompliziertere Aussagen zu formen. In der klassischen Aussagenlogik[6] wird jeder Aussage entweder der Wert „wahr“ oder „falsch“ zugeordnet. Der Wahrheitswert einer zusammengesetzten Aussage kann dabei immer aus den Wahrheitswerten ihrer Teilaussagen hergeleitet werden [Kle].

Regeln

Die für die natürliche Deduktion der Aussagenlogik wichtigen Konnektoren sind:

$$\wedge\ Konjunktion,\ \vee\ Disjunktion,\ \to\ Implikation,\ \neg\ Negation$$

Außerdem die beiden elementaren Wahrheitswerte:

$$\top\ Wahrheit,\ \bot\ Widerspruch$$

[6]Die klassische Aussagenlogik besagt, dass jede Aussage entweder „wahr“ oder „falsch“ sein muss. Die intuitionistische Logik dagegen sagt, dass es für eine Aussage entweder einen Beweis oder einen Widerspruch geben muss. Aussagen, die weder bewiesen noch widerlegt werden können, sind demnach nicht gültig (siehe [Bor05b] S. 35). Für dieses Projekt beschäftigt uns nur die klassische Aussagenlogik.

Eine Konjunktion beschreibt Situationen, in denen mehrere Aussagen wahr sind. Die Formel $A \wedge B$ wird „A und B“ ausgesprochen. Sind zwei wahre Aussagen gegeben, so können diese mit einer Konjunktion verbunden werden (siehe Abb. 3.12a). Aus einer gegebenen Konjunktion kann gefolgert werden, dass jede ihrer Teilaussagen wahr ist. Daher können aus dieser Konjunktion die beiden Teilaussagen „herausgelöst“ werden (siehe Abb. 3.12b und 3.12c).

$$\frac{A \qquad B}{(A \wedge B)} \qquad \frac{(A \wedge B)}{A} \qquad \frac{(A \wedge B)}{B}$$

(a) Einführung (b) Elimination 1 (c) Elimination 2

Abbildung 3.12: Regeln der Konjunktion ($\wedge$, AND)

Eine Disjunktion beschreibt eine Situation, in der mindestens eine Aussage wahr ist. Die Formel $A \vee B$ wird „A oder B“ ausgesprochen und bedeutet: A ist wahr oder B ist wahr oder beide Aussagen sind wahr. Wann immer eine wahre Aussage vorliegt, kann mittels einer Disjunktion eine weitere beliebige Aussage hinzugefügt werden (siehe Abb. 3.13a und 3.13b). So wird beispielsweise aus der wahren Aussage A die Aussage „A oder B“. Mit dem Wissen, dass Aussage A wahr ist, kann jede beliebige Aussage angehängt werden, ohne dass der Wahrheitswert der Gesamtaussage verändert wird.

Eine Disjunktion aufzulösen ist etwas komplizierter, da man nicht weiß, welche der Teilaussagen wahr sind: A, B oder beide. Daher nimmt man eine beliebige Schlussfolgerung C und zeigt, dass sich sowohl aus der Aussage A als auch aus der Aussage B diese Schlussfolgerung herleiten ließe. Kann dies bewiesen werden, ist es egal, welche der Teilaussagen wahr ist, in jedem Fall kann von der Disjunktion „A oder B“ auf die Schlussfolgerung C geschlossen werden (siehe Abb. 3.13c).

$$\frac{A}{(A \vee B)} \qquad \frac{B}{(A \vee B)} \qquad \frac{(A \vee B) \quad \boxed{\begin{array}{c} A \\ \dots \\ C \end{array}} \quad \boxed{\begin{array}{c} B \\ \dots \\ C \end{array}}}{C}$$

(a) Einführung 1 (b) Einführung 2 (c) Elimination

Abbildung 3.13: Regeln der Disjunktion ($\vee$, OR)

Die Implikation beschreibt eine Verbindung zwischen verschiedenen Aussagen. Die Formel $A \to B$ wird „A impliziert B“ oder „aus A folgt B“ ausgesprochen. Kann logisch bewiesen werden, dass sich aus der Aussage A die Aussage B herleiten lässt, dann kann man diese mit einer Implikation verbinden (siehe Abb. 3.14a). Sind eine Implikation der Form „aus A folgt B“ und die Aussage A gegeben, so kann daraus gefolgert werden, dass auch B wahr sein muss (siehe Abb. 3.14b).

$$\frac{\boxed{\begin{array}{c} A \\ \dots \\ B \end{array}}}{(A \to B)} \qquad\qquad \frac{(A \to B) \quad A}{B}$$

(a) Einführung (b) Elimination

Abbildung 3.14: Regeln der Implikation ($\to$, IMPL)

Die Negation verneint eine Aussage. Die Formel $\neg A$ wird „nicht A“ ausgesprochen. Außerdem wird das Symbol $\bot$ für „Widerspruch“ benötigt. Ein Widerspruch steht für eine Situation, die nicht eintreten kann.

Wenn sich aus der Annahme A ein Widerspruch ableiten lässt, kann die Aussage A nicht wahr sein, daher wird geschlussfolgert, dass „nicht A“ richtig sein muss (siehe Abb. 3.15a). Es kann nie sein, dass die Aussagen A und „nicht A“ gleichzeitig wahr sind. Aus eben diesen Vorbedingungen folgert man daher einen Widerspruch (siehe Abb. 3.15b).

$$\frac{\boxed{\begin{array}{c} A \\ \dots \\ \bot \end{array}}}{\neg A} \qquad\qquad \frac{A \quad \neg A}{\bot}$$

(a) Einführung (b) Elimination

Abbildung 3.15: Regeln der Negation ($\neg$, NOT)

Die Regeln der Kontradiktion beschreiben, wie mit einem Widerspruch $\bot$ in Beweisen umgegangen wird. Ein Widerspruch steht für eine unmögliche Situation. Die Regel „ex falso quodlibet“ bedeutet „aus Falschem folgt Beliebiges“ bzw. „aus einem Widerspruch folgt Beliebiges“. Ein Widerspruch in einem Beweis steht für eine

Situation, welche unter keinen Umständen jemals eintreten kann. Daher ist es auch unerheblich, wie man den Beweis an dieser Stelle weiter fortführt (siehe Abb. 3.16a).

Die Regel "reductio ad absurdum" ist auch als „Widerspruchsbeweis" bekannt. Um eine Aussage A zu beweisen, versucht man nicht, diese herzuleiten, sondern ihr Gegenteil zu widerlegen. Kann man beweisen, dass aus dem Gegenteil $\neg A$ ein Widerspruch folgt, so muss folglich die Aussage A der Wahrheit entsprechen (siehe Abb. 3.16b). Um etwa die Aussage „Nicht alle Menschen sind Griechen" zu beweisen, genügt es zu zeigen, dass die Aussage „Alle Menschen sind Griechen" zu einem Widerspruch führt. Dies lässt sich z.B. dadurch erreichen, indem man einen Menschen findet, der kein Grieche ist (was nicht weiter schwer sein sollte). Diese Regel wird von der intuitionistischen Logik abgelehnt, da sie nicht konstruktiv ist. In der klassischen Aussagenlogik dagegen werden viele Beweise mit dieser Regel geführt.

$$\frac{\bot}{A} \qquad\qquad \frac{\boxed{\begin{array}{c}\neg A \\ \vdots \\ \bot\end{array}}}{A}$$

(a) EFQ (ex falso quodlibet) (b) RAA (reductio ad absurdum)

Abbildung 3.16: Regeln der Kontradiktion (EFQ, RAA)

3.4.2 Prädikatenlogik

Die Prädikatenlogik ist eine Erweiterung der Aussagenlogik. Sie erlaubt es, genauere Aussagen über Individuen, die Existenz von Individuen oder eine Menge von Individuen zu definieren. Zusätzlich zu den Aussagen und Konnektoren führt sie daher noch die sog. Prädikate und Quantoren ein.

Prädikate geben dabei genauere Eigenschaften von Individuen an. So ist der Satz „x ist ein Mensch" ein solches Prädikat. Je nachdem was für „x" eingesetzt wird, entspricht der Satz der Wahrheit (z.B. „Sokrates ist ein Mensch") oder nicht (z.B. „Mein Hund ist ein Mensch"). So lassen sich die Eigenschaften von Individuen genauer bestimmen, als dies in der Aussagenlogik möglich wäre.

Quantoren geben an, auf welche Individuen des betrachteten Universums ein oder mehrere Prädikate zutreffen. Der Existenzquantor ($\exists$) zeigt, dass es mindestens ein Individuum gibt, auf welches die angegebenen Prädikate zutreffen (z.B. $\exists$(x)(„ist ein Mensch" x) $\Rightarrow$ „es gibt mindestens ein Individuum, das ein Mensch ist"). Der Allquantor ($\forall$) dagegen zeigt, dass für alle Individuen eines betrachteten Universums die

zugehörigen Prädikate zutreffen (z.B. ∀(x)(„besteht aus Atomen“ x) ⇒ „alle Individuen bestehen aus Atomen“).

Regeln

Die Regeln der natürlichen Deduktion für die Prädikatenlogik werden zu den Regeln der Aussagenlogik hinzugefügt. Die wichtigen Quantoren für die natürliche Deduktion der Prädikatenlogik sind:

$$\forall\ \textit{Allquantor},\ \exists\ \textit{Existenzquantor}$$

Diese Quantoren werden auf Prädikate angewandt (in den Regeln mit P gekennzeichnet). Außerdem enthalten die Regeln zusätzlich den Begriff „actual“, welcher ein existierendes Individuum angibt (z.B. *actual* x ⇒ x ist ein Individuum), sowie die Schreibweise $\phi[x_0/x]$, welche für eine Substitution steht ($\phi[i/x]$ ⇒ jedes Vorkommen von x in der Formel ϕ wird durch i ersetzt).

Der Allquantor macht eine Aussage über alle Individuen des betrachteten Universums. Die Formel $\forall x P(x)$ wird „für alle x gilt, $P\ x$“ ausgesprochen. Das bedeutet, dass jedes einzelne Individuum des Universums das Prädikat P erfüllt, z.B. $\forall x Katze(x)$ hieße, dass alle Individuen des Universums Katzen sind.

Kann bewiesen werden, dass die in der Formel ϕ notierten Eigenschaften für ein beliebiges Individuum eines Universums *actual* i[7] gelten, so kann dies für alle Individuen des Universums angenommen werden (siehe Abb. 3.17a). Aus einem gegebenen Allquantor und einem Individuum des betrachteten Universums folgt, dass jenes Individuum die im Allquantor notierten Eigenschaften besitzen muss (siehe Abb. 3.17b). Aus der Aussage $\forall x Katze(x)$ (alle Individuen des Universums sind Katzen) und dem Individuum des Universums „Freddy“ folgt „Katze(Freddy)“ (Freddy ist eine Katze).

$$\frac{\boxed{\begin{array}{c}\textit{actual } i\\ \dots\\ \phi[i/x]\end{array}}}{\forall x\phi} \qquad\qquad \frac{\textit{actual } i \quad \forall x\phi}{\phi[i/x]}$$

(a) Einführung (b) Elimination

Abbildung 3.17: Regeln des Allquantors (∀, FORALL)

[7]Das Individuum ist absolut beliebig gewählt, kann also durch jedes andere Individuum des Universums ersetzt werden. Die Beweisführung ist daher auch für unendlich große Universen möglich.

Der Existenzquantor versichert, dass es mindestens ein Individuum gibt, welches die notierten Eigenschaften erfüllt. Die Formel $\exists x P(x)$ wird „es existiert ein x für das gilt, $P\ x$“ ausgesprochen.

Existiert ein Individuum *actual i*, welches die Eigenschaften der Formel ϕ erfüllt, dann kann daraus ein Existenzquantor formuliert werden (siehe Abb. 3.18a). Das Auflösen eines Existenzquantors ist etwas komplizierter, da das genaue Individuum bzw. die Individuen, welche die entsprechenden Eigenschaften erfüllen, nicht bekannt sind. Man nimmt daher eine beliebige Schlussfolgerung C und zeigt, dass unter der Annahme eines beliebigen Individuums, welches die entsprechenden Eigenschaften besitzt, diese Schlussfolgerung herleitet werden kann (siehe Abb. 3.18b). Es ist nicht wichtig, welches genaue Individuum gewählt wurde. Solange es die angegebenen Eigenschaften besitzt, kann C immer herleitet werden. Man geht dabei immer von nicht leeren Universen aus, sodass immer mindestens ein Individuum vorhanden ist.

$$\frac{\textit{actual } i \quad \phi[i/x]}{\exists x \phi}$$

(a) Einführung

$$\frac{\exists x \phi \qquad \boxed{\begin{array}{c}\textit{actual } i \\ \phi[i/x] \\ \dots \\ C\end{array}}}{C}$$

(b) Elimination

Abbildung 3.18: Regeln des Existenzquantors (∃, EXISTS)

Mit dem Gleichheitsoperator kann gezeigt werden, dass ein Term einem anderen entspricht. Ein Term ist immer gleich zu sich selbst. Es braucht keinerlei Vorbedingungen, um dies zu beweisen (siehe Abb. 3.19a). Existiert eine Gleichheit zweier Terme „*t1* = *t2*“ und eine Formel, die einen der beiden Terme beinhalten kann $\phi[t_1/x]$, so kann jedes Vorkommen von *t1* durch *t2* ersetzt werden (siehe Abb.3.19b).

Dabei ist äußerst wichtig, dass der eingesetzte Term keine Variablen enthält, die durch diese Substitution in den Gültigkeitsbereich eines Quantors fallen. Beispielsweise kann die Substitution $\phi[f(x)/y]$ (jedes Vorkommen von y wird durch den Term $f(x)$ ersetzt) nicht auf die Formel $\exists x \exists y P(y)$ angewendet werden. Die im Term $f(x)$ enthaltene Variable x würde in den Gültigkeitsbereich des ersten Existenzquantors $\exists x$ fallen und damit ihre eigentliche Bedeutung verlieren. Da die Regeln der Quantoren mit Individuen arbeiten, fällt diese Prüfung dort nicht weiter ins Gewicht. Bei den

Regeln der Gleichheit wird aber mit beliebigen Termen agiert, daher muss bei jeder Substitution entsprechend geprüft werden.

$$\frac{}{t = t} \qquad \frac{t_1 = t_2 \quad \phi[t_1/x]}{\phi[t_2/x]}$$

(a) Einführung (b) Elimination

Abbildung 3.19: Regeln der Gleichheit (=, EQUAL)

3.4.3 Temporale Logik

Mit der Erweiterung der Aussagenlogik um eine zeitliche Komponente erhält man die temporalen Logiken. So lassen sich logische Aussagen in Relation zu bestimmten Zeitpunkten stellen. Neben der hier vorgestellten „Linear Temporal Logic“ (LTL), welche von einer linearen Zeitfolge ausgeht, gibt es auch noch die „Computation Tree Logic“ (CTL), die eine sich verzweigende Zeitfolge beschreibt.

Regeln

Zunächst soll nur die natürliche Deduktion der LTL nach [BGS07] eingeführt werden. Dabei wurde die von Bolotov, Grigoriev und Shangin verwendete Schreibweise der Regeln an einigen Stellen der hier verwendeten angepasst.

Die LTL erweitert die klassischen Konnektoren der Aussagenlogik ($\wedge$, $\vee$, $\rightarrow$, $\neg$) um vier weitere, die den Zeitpunkt einer Aussage weiter spezifizieren[8]:

$\Box$ *immer in der Zukunft,* $\bigcirc$*zum nächst möglichen Zeitpunkt,*
$\Diamond$ *irgendwann in der Zukunft,* $\mathcal{U}$ *bis*

Zusätzlich zu den Konnektoren und den eigentlichen Aussagen, müssen nun auch die verschiedenen Zeitpunkte berücksichtigt werden. Diese werden mit Kleinbuchstaben (*i, j, k*) angegeben und mit den Aussagen wie folgt verknüpft:

$$i : A \Rightarrow \text{zum Zeitpunkt } i \text{ gilt die Aussage } A$$

Zunächst folgt eine Darstellung der modifizierten Regeln für die klassischen Konnektoren der Aussagenlogik, angepasst auf die LTL. In den meisten Fällen entsprechen die Regel denen der Aussagenlogik erweitert um die Angabe der Zeitpunkte:

[8] Die Angabe von temporalen Konnektoren bezieht sich lediglich auf den Zeitpunkt einer Aussage. Der Wahrheitswert jener bleibt davon gänzlich unangetastet.

$$\frac{i : A \quad i : B}{i : A \wedge B} \qquad \frac{i : A \wedge B}{i : A} \qquad \frac{i : A \wedge B}{i : B}$$

(a) Einführung (b) Elimination 1 (c) Elimination 2

Abbildung 3.20: Regeln für die Konjunktion ($\wedge$, AND)

Die Elimination einer Disjunktion wurde in der LTL stark vereinfacht. Existiert eine Disjunktion $A \vee B$ und eine Verneinung eines dieser beiden Elemente z.B. $\neg B$ so kann daraus geschlossen werden, dass das entsprechend andere Element wahr sein muss (siehe Abb. 3.21c).

$$\frac{i : A}{i : A \vee B} \qquad \frac{i : B}{i : A \vee B} \qquad \frac{i : A \vee B \quad i : \neg A}{i : B}$$

(a) Einführung 1 (b) Einführung 2 (c) Elimination

Abbildung 3.21: Regeln für die Disjunktion ($\vee$, OR)

$$\frac{\boxed{\begin{array}{c} i : A \\ \dots \\ i : B \end{array}}}{i : A \to B} \qquad \frac{i : A \to B \quad i : A}{i : B}$$

(a) Einführung (b) Elimination

Abbildung 3.22: Regeln für die Implikation ($\to$, IMPL)

Die Einführung der Negation entspricht der Regel der Aussagenlogik, allerdings verwenden Bolotov, Grigoriev und Shangin anstelle des Widerspruchs $\perp$ eine entsprechende Aussage der Form „$B \wedge \neg B$“[9] (siehe Abb. 3.23a). Auch die Elimination der Negation verwendet kein Widerspruchssymbol, stattdessen eliminiert die Regel die doppelte Negation einer Aussage (siehe Abb. 3.23b).

[9]Es ist nicht wichtig, zu welchem Zeitpunkt diese Aussage vorliegt. Der Zeitpunkt (hier: j) kann vor, gleich oder nach dem ausgehenden Zeitpunkt (hier: i) liegen.

$$\frac{\boxed{\begin{array}{c} i : A \\ \dots \\ j : B \wedge \neg B \end{array}}}{i : \neg A}$$

(a) Einführung

$$\frac{i : \neg\neg A}{i : A}$$

(b) Elimination

Abbildung 3.23: Regeln für die Negation (¬, NOT)

Die Regeln für die Zeitpunkte geben an, wie verschiedene Zeitpunkte zueinander in Relation gesetzt werden können. Mit den Operatoren $<$, $\leq$ und $\simeq$ lässt sich angeben, dass ein Zeitpunkt i kleiner ($i < j$), kleiner oder gleich ($i \leq j$) oder ungefähr gleich ($i \simeq j$) einem zweiten Zeitpunkt j ist. Die Formel „*Next(i, i')*“ besagt, dass der Zeitpunkt *i'* unmittelbar auf den Zeitpunkt i folgt. Es existieren keine weiteren Zeitpunkte zwischen i und *i'*. Zeitpunkte in der LTL sind transitiv, seriell, reflexiv und linear abfolgend.

$$\frac{}{i \leq i}$$

(a) Reflexivität

$$\frac{}{Next(i, i')}$$

(b) ○Serialität

$$\frac{i < j}{i \leq j}$$

(c) $</\leq$

$$\frac{Next(i, i')}{i \leq i'}$$

(d) $\bigcirc/\leq$

$$\frac{i \leq j,\ j \leq k}{i \leq k}$$

(e) Transitivität

$$\frac{i \leq j,\ i \leq k}{(j \leq k) \vee (j \simeq k) \vee (k \leq j)}$$

(f) Linearität

Abbildung 3.24: Regeln für die Relationen der Zeitpunkte

Der Operator $\Box$ steht für eine Aussage, die zum gegebenen Zeitpunkt wahr ist und in Zukunft immer wahr sein wird. Lässt sich zeigen, dass zu einem beliebigen Zeitpunkt j, welcher größer oder gleich einem Zeitpunkt i ist ($i \leq j$), die Aussage A wahr ist ($j : A$), dann ist zum Zeitpunkt i die Aussage A in Zukunft immer wahr (siehe Abb. 3.25a).

Ist die Aussage A zum Zeitpunkt i in Zukunft immer wahr ($i : \Box A$) und haben wir einen Zeitpunkt j, welcher gleich oder größer i ist ($i \leq j$), dann muss zu diesem Zeitpunkt j die Aussage A wahr sein (siehe Abb. 3.25b).

$$\frac{\boxed{\begin{array}{c} i \leq j \\ \dots \\ j : A \end{array}}}{i : \Box A}$$

(a) Einführung

$$\frac{i : \Box A \quad i \leq j}{j : A}$$

(b) Elimination

Abbildung 3.25: Regeln für „immer in der Zukunft“ ($\Box$, ALWAYS)

Der Operator $\Diamond$ steht für eine Aussage, die vom aktuellen Zeitpunkt aus gesehen irgendwann in der Zukunft wahr ist. Wenn zum Zeitpunkt j die Aussage A wahr ist ($j : A$) und dieser Zeitpunkt gleich oder größer dem Zeitpunkt i ist ($i \leq j$), dann kann vom Zeitpunkt i aus gesagt werden, dass die Aussage A irgendwann in der Zukunft wahr ist (siehe Abb. 3.26a).

Die Regel funktioniert auch umgekehrt. Weiß man zum Zeitpunkt i, dass die Aussage A irgendwann in der Zukunft wahr ist, so lässt sich daraus schließen, dass es einen Zeitpunkt j in der Zukunft ($i \leq j$) gibt, zu dem die Aussage A wahr ist (siehe Abb. 3.26b).

$$\frac{j : A \quad i \leq j}{i : \Diamond A}$$

(a) Einführung

$$\frac{i : \Diamond A}{i \leq j \quad j : A}$$

(b) Elimination

Abbildung 3.26: Regeln für „irgendwann in der Zukunft“ ($\Diamond$, SOMETIME)

Der Operator $\bigcirc$ steht für den nächst möglichen Zeitpunkt. Ist die Aussage A zum Zeitpunkt *i'* wahr ($i' : A$) und weiß man, dass der Zeitpunkt i unmittelbar vor diesem Zeitpunkt *i'* liegt (*Next(i, i')*), dann ist vom Zeitpunkt i aus gesehen die Aussagen A zum nächst möglichen Zeitpunkt wahr (siehe Abb. 3.27a).

Weiß man, dass zum Zeitpunkt i die Aussage A zum nächst möglichen Zeitpunkt wahr ist ($i : \bigcirc A$) dann kann eben dieser Zeitpunkt mit der entsprechenden Aussage daraus abgeleitet werden (siehe Abb. 3.27b).

$$\frac{i' : A \quad Next(i, i')}{i : \bigcirc A} \qquad \frac{i : \bigcirc A}{i' : A}$$

(a) Einführung (b) Elimination

Abbildung 3.27: Regeln für „zum nächst möglichen Zeitpunkt“ ($\bigcirc$, ASAP)

Der Operator $\mathcal{U}$ steht für „bis“. Die Formel $A\mathcal{U}B$ bedeutet, dass die Aussage B zum aktuellen oder einem beliebigen nachfolgenden Zeitpunkt wahr ist. Die Aussage A gilt dabei mindestens so lange bis eben dieser Zeitpunkt erreicht ist. Sie muss zu dem entsprechenden Zeitpunkt nicht mehr wahr sein.

Gilt zu einem Zeitpunkt i die Aussage B, so kann zum selben Zeitpunkt immer eine Aussage A mit dem „bis“ Operator hinzufügt werden (siehe Abb. 3.28a). Die Aussage A kann zum Zeitpunkt i schon wieder ungültig sein. Sie ist nur mindestens gültig bis die Aussage B wahr ist, was zum Zeitpunkt i ja der Fall ist.

$$\frac{i : B}{i : A\mathcal{U}B} \qquad \frac{i : \Box(B \to C) \quad i : \Box((A \wedge \bigcirc C) \to C)}{i : (A\mathcal{U}B) \to C}$$

(a) Einführung (b) Elimination

Abbildung 3.28: Regeln für „bis“ ($\mathcal{U}$, UNTIL)

Die drei Regeln in Abb. 3.29 stellen Sonderformen für die Negation der entsprechenden Operatoren dar. Sie wurden von Bolotov, Grigoriev und Shangin in ihrer Arbeit [BGS07] als zusätzliche Regeln eingeführt. Sie sollen ihre erste Definition von der natürlichen Deduktion der linearen temporalen Logik [BBGS06] stark vereinfachen.

$$\frac{i : \neg\Diamond A}{i : \Box\neg A} \qquad \frac{i : \neg(A \vee B)}{i : \neg A \wedge \neg B}$$

(a) Negation von „irgendwann in der Zukunft“ ($\neg\Diamond$) (b) Negation der Disjunktion ($\neg\vee$)

$$\frac{i : \neg(A\mathcal{U}B)}{i : (\Box\neg B) \vee (B\mathcal{U}(\neg A \wedge \neg B))}$$

(c) Negation von „bis“ ($\neg\mathcal{U}$)

Abbildung 3.29: Sonderregeln für die Negation spezieller Operatoren

3.5 Beispiele

Im letzten Abschnitt des Grundlagenkapitels sollen nun einige Hypothesen der vorgestellten Logiken beispielhaft bewiesen werden. Zu jeder Logik wurden dabei sowohl einfache wie auch kompliziertere Beispiele ausgesucht.

Aussagenlogik

Der „Modus Tollens“ ist ein Theorem der Logik, welches in vielen anderen Beweisen verwendet wird. Die Vorbedingungen sind „aus A folgt B“ und „nicht B“. Man weiß also, dass aus der Aussage A die Aussage B folgen muss. Da aber die Aussage „nicht B“ wahr ist, kann dieser Fall nicht auftreten und damit die Aussage A nicht wahr sein. Daraus wird „nicht A“ geschlussfolgert (siehe Abb. 3.30).

$$A \to B, \neg B \vdash \neg A$$

1.	$A \to B$	Vorbedingung
2.	$\neg B$	Vorbedingung
3.	A	Annahme
4.	B	$\to$ Elimination (3, 1)
5.	$\bot$	$\neg$ Elimination (4, 2)
6.	$\neg A$	$\neg$ Einführung (3-5)

Abbildung 3.30: Modus Tollens

Der „Satz vom ausgeschlossenen Dritten“ (Tertium Non Datur) ist ein Grundprinzip der Logik. Er besagt, dass immer eine Aussage selbst oder ihr logisches Gegenteil wahr sein muss. Etwas Drittes, das weder die Aussage selber noch ihr Gegenteil ist, kann es nicht geben. Hergeleitet wird dieses Theorem durch einen Widerspruchsbeweis (siehe Abb. 3.31).

Die intuitionistischen Logik lehnt dieses Theorem ab. Für sie gilt: Eine Aussage A ist entweder bewiesen oder widerlegt. Allerdings gibt es Aussagen, die sich weder beweisen noch widerlegen lassen (z.B. Goldbachsche Vermutung).

$$\vdash A \vee \neg A$$

1.	$\neg(A \vee \neg A)$	Annahme
2.	A	Annahme
3.	$A \vee \neg A$	$\vee$ Einführung 1 (2)
4.	$\bot$	$\neg$ Elimination (3, 1)
5.	$\neg A$	$\neg$ Einführung (2-4)
6.	$A \vee \neg A$	$\vee$ Einführung 2 (5)
7.	$\bot$	$\neg$ Elimination (1, 6)
8.	$A \vee \neg A$	RAA (1-7)

Abbildung 3.31: Tertium Non Datur

Die „De Morganschen Gesetze“ gehören zu den grundlegenden Regeln für logische Aussagen. Mit ihrer Hilfe lässt sich eine Konjunktion ($\wedge$) in eine Disjunktion ($\vee$) umwandeln und umgekehrt. Dies wird zum Beispiel häufig bei digitalen Schaltkreisen genutzt, um logische Schaltelemente gegeneinander auszutauschen oder entsprechend einzusparen.

Die Aussage „Es ist nicht sonnig und windig“ ($\neg(A \wedge B)$) kann man umformen in die Aussage „Es ist nicht sonnig oder es ist nicht windig“ ($\neg A \vee \neg B$). Die Konjunktion ist deshalb negiert, da eines oder beide ihrer Elemente falsch sind. Dies lässt sich durch eine entsprechende Disjunktion ausdrücken (siehe Abb. 3.32).

Auch die Aussage „Es ist nicht sonnig oder windig“ ($\neg(A \vee B)$) lässt sich umformen. Man weiß durch die Aussage, dass es weder sonnig noch windig ist, daher kann die Aussage auch wie folgt umformuliert werden: „Es ist nicht sonnig und es ist nicht windig“ ($\neg A \wedge \neg B$) (siehe Abb. 3.33).

$$\neg(A \wedge B) \vdash \neg A \vee \neg B$$

1.	$\neg(A \wedge B)$	Vorbedingung
2.	$\neg(\neg A \vee \neg B)$	Annahme
3.	$\neg A$	Annahme
4.	$\neg A \vee \neg B$	$\vee$ Einführung 1 (3)
5.	$\bot$	$\neg$ Elimination (2, 4)
6.	A	RAA (3-5)
7.	$\neg B$	Annahme
8.	$\neg A \vee \neg B$	$\vee$ Einführung 2 (7)
9.	$\bot$	$\neg$ Elimination (2, 8)
10.	B	RAA (7-9)
11.	$A \wedge B$	$\wedge$ Einführung (6, 10)
12.	$\bot$	$\neg$ Elimination (1, 11)
13.	$\neg A \vee \neg B$	RAA (2-12)

Abbildung 3.32: De Morgansches Gesetz 1

$$\neg(A \vee B) \vdash \neg A \wedge \neg B$$

1.	$\neg(A \vee B)$	Vorbedingung
2.	A	Annahme
3.	$A \vee B$	$\vee$ Einführung 1 (2)
4.	$\bot$	$\neg$ Elimination (1, 3)
5.	$\neg A$	$\neg$ Einführung (2-4)
6.	B	Annahme
7.	$A \vee B$	$\vee$ Einführung 2 (6)
8.	$\bot$	$\neg$ Einführung (1, 7)
9.	$\neg B$	$\neg$ Einführung (6-8)
10.	$\neg A \wedge \neg B$	$\wedge$ Einführung (5, 9)

Abbildung 3.33: De Morgansches Gesetz 2

Die Kontraposition bezeichnet die Umkehrung einer Implikation. Aus einer Implikation wie z.B. „Wenn es regnet, ist der Wald nass“ lässt sich ableiten „Wenn

der Wald nicht nass ist, regnet es nicht“. Bei der Beweisführung wird das Theorem „Modus Tollens“ (siehe Abb. 3.30) herangezogen (siehe Abb. 3.34).

$$A \to B \vdash \neg B \to \neg A$$

1.	$A \to B$	Vorbedingung
2.	$\neg B$	Annahme
3.	$\neg A$	Modus Tollens (1, 2)
4.	$\neg B \to \neg A$	$\to$ Einführung (2-3)

Abbildung 3.34: Kontraposition

Prädikatenlogik

Wenn kein Individuum existiert, welches ein bestimmtes Prädikat P erfüllt ($\neg\exists x P(x)$), dann gilt für alle Individuen des betrachteten Universums, dass das betreffende Prädikat nicht zutrifft ($\forall x \neg P(x)$). Aus der Aussage „es existiert kein Lebewesen, das im luftleeren Raum überleben kann“ lässt sich leicht schließen „alle Lebewesen können nicht im luftleeren Raum überleben“. Dieses Theorem lässt sich recht kurz beweisen (siehe Abb. 3.35).

$$\neg\exists x P(x) \vdash \forall x \neg P(x)$$

1.	$\neg\exists x P(x)$	Vorbedingung
2.	$actual\ x_0$	Annahme
3.	$P(x_0)$	Annahme
4.	$\exists x P(x)$	$\exists$ Einführung (2, 3)
5.	$\bot$	$\neg$ Elimination (1, 4)
6.	$\neg P(x_0)$	$\neg$ Einführung (3-5)
7.	$\forall x \neg P(x)$	$\forall$ Einführung (2-6)

Abbildung 3.35: Beispiel 1

In Abb. 3.36 sieht man, dass ein Quantor in seinem Inneren nicht nur Prädikate, sondern auch logische Operatoren halten kann. Eine Formel wie etwa „es gibt einen Ball, der rot oder blau ist“ kann leicht umgeformt werden. Die Disjunktion im inneren

des Quantors wird nach außen verlagert: „es gibt einen Ball, der blau ist, oder es gibt einen Ball der rot ist“ (möglicherweise ist der Ball auch blau-rot).

$$\exists x(P(x) \vee Q(x)) \vdash \exists x P(x) \vee \exists x Q(x)$$

1.	$\exists x(P(x) \vee Q(x))$	Vorbedingung
2.	*actual* x_0	Annahme
3.	$P(x_0) \vee Q(x_0)$	Annahme
4.	$P(x_0)$	Annahme
5.	$\exists x P(x)$	$\exists$ Einführung (2, 4)
6.	$\exists x P(x) \vee \exists x Q(x)$	$\vee$ Einführung 1 (5)
7.	$Q(x_0)$	Annahme
8.	$\exists x Q(x)$	$\exists$ Einführung (2, 7)
9.	$\exists x P(x) \vee \exists x Q(x)$	$\vee$ Einführung 2 (8)
10.	$\exists x P(x) \vee \exists x Q(x)$	$\vee$ Elimination (3, 4-6, 7-9)
11.	$\exists x P(x) \vee \exists x Q(x)$	$\exists$ Elimination (1, 2-10)

Abbildung 3.36: Beispiel 2

Temporale Logik

Aus einer Aussage der Form „nicht irgendwann in Zukunft A“ folgt logischerweise „in Zukunft immer nicht A“ (siehe Abb. 3.37). Ein Beispiel: „Es gibt nicht irgendwann eine Reunion der Beatles“ wird zu “Es wird in Zukunft keine Reunion der Beatles geben“.

$$\neg\Diamond A \vdash \Box\neg A$$

1.	$i : \neg\Diamond A$	Vorbedingung
2.	$i \leq j$	Annahme
3.	$j : A$	Annahme
4.	$i : \Diamond A$	$\Diamond$ Einführung (2, 3)
5.	$i : \Diamond A \wedge \neg\Diamond A$	$\wedge$ Einführung (1, 4)
6.	$j : \neg A$	$\neg$ Einführung (3-4)
7.	$i : \Box\neg A$	$\Box$ Einführung (2-5)

Abbildung 3.37: Beispiel 1

Das Theorem in Abb. 3.38 beweist, dass „zum nächst möglichen Zeitpunkt gilt, dass irgendwann die Aussage A wahr ist“ gleichzusetzen ist mit „irgendwann gilt die Aussage A“. Da der $\Diamond$ Operator den aktuellen Zeitpunkt miteinschließt, macht es keinen Unterschied, ob man mit diesem oder dem nächst möglichen Zeitpunkt beginnt.

$$\bigcirc\Diamond A \vdash \Diamond A$$

1.	$i : \bigcirc\Diamond A$	Vorbedingung
2.	$Next(i, j)$	$\bigcirc$ Serialität
3.	$j : \Diamond A$	$\bigcirc$ Elimination (1)
4.	$i \leq j$	Next/$\leq$ (2)
5.	$j \leq k$	$\Diamond$ Elimination (3)
6.	$k : A$	$\Diamond$ Elimination (3)
7.	$i \leq k$	Transitivität (4, 5)
8.	$i : \Diamond A$	$\Diamond$ Einführung (6, 7)

Abbildung 3.38: Beispiel 2

Kapitel 4

Implementierung

Im folgenden Kapitel wird die Implementierung des Projekts genauer beleuchtet. Dazu gehören die verwendeten Datenstrukturen, Funktionen zur Beweisführung und die Schnittstelle für die Nutzer (REPL). Abschließend folgt noch ein Teil über mögliche Erweiterungen. Zum Verständnis dieses Kapitels werden Grundlagen der Programmiersprache Clojure (z.B. Listen, Hashmaps, Vektoren) und ihrer Notation vorausgesetzt.

Der Quellcode des Projekts kann unter folgendem Link eingesehen werden: `https://github.com/TheBlob42/deduction-playground`.

4.1 Formeln

Im Gegensatz zu der in Kapitel 3 verwendeten Infixnotation[10] für logische Formeln, entspricht die hier verwendete Notation der von Clojure verwendeten Prefixnotation[11] (siehe Tabelle 4.1). Grund dafür ist die Annäherung an die zugrunde liegende Programmiersprache sowie die einfachere Verarbeitung im Clojure Code selbst. Außerdem wird auf den Einsatz von Sonderzeichen (z.B. $\vdash$, $\wedge$, $\neg$) verzichtet, was die Schreibweise vereinfacht.

4.2 Beweise

Beweise werden oftmals nach der in Kapitel 3.2.2 gezeigten Schreibweise handschriftlich festgehalten. Für das hier vorgestellte Projekt muss diese Schreibweise in eine angepasste Datenstruktur übersetzt werden. Auf dieser werden dann sämtliche Operationen der natürlichen Deduktion ausgeführt.

[10] Der Operator wird zwischen die Operanden gesetzt (z.B. 1 + 2).
[11] Der Operator wird vor die Operanden gesetzt (z.B. + 1 2).

Infixnotation	*Projektnotation*	*Bedeutung*
a $\vdash$ b	(infer a b)	Schlussfolgerung
a $\wedge$ b	(and a b)	Und
a $\vee$ b	(or a b)	Oder
a $\rightarrow$ b	(impl a b)	Implikation
$\neg$a	(not a)	Negation
$\top$	truth	Wahrheit
$\bot$	contradiction	Widerspruch
$\forall$xP(x)	(forall [x] (P x))	Allquantor
$\exists$xP(x)	(exists [x] (P x))	Existenzquantor
ϕ[t/x]	(substitution phi x t)	Substitution
i : a	(at i a)	Zeitpunktangabe
$\Box$a	(always a)	„Immer in der Zukunft“
$\bigcirc$a	(asap a)	„Zum nächst möglichen Zeitpunkt“
$\Diamond$	(sometime a)	„Irgendwann in der Zukunft“
a$\mathcal{U}$b	(until a b)	„bis“

Tabelle 4.1: Infixnotation für logische Formeln und die für das Projekt verwendete Clojure ähnliche Prefixnotation in der direkten Gegenüberstellung.

Die interne Beweisstruktur entspricht der von Daniel Kirsten [Kir14]. Dabei wird ein Beweis als Vektor angegeben. Die Reihenfolge der Elemente in diesem Vektor entspricht der Reihenfolge der Zeilen des Beweises. Unterbeweise sind ebenfalls Vektoren, die weitere Zeilen und Unterbeweise beinhalten können (siehe Abb. 4.1). Zeilen werden durch Hashmaps mit den folgenden Schlüsseln repräsentiert:

- **:id** Eine eindeutige ID zur Identifikation der Zeile (entspricht *:hash* in [Kir14]).
- **:body** Die Formel der entsprechenden Zeile (*:todo* für leere Zeilen).
- **:rule** Die Ursprungsangabe der Zeile: Vorbedingung (*:premise*), Annahme (*:assumption*), abgeleitet nach einer Regel (z.B. „impl-e“ (1 3))[12] oder unbewiesen (*nil*).

4.3 Regeln

Die Inferenzregeln sind der Kern der natürlichen Deduktion. Im Folgenden werden die interne Struktur der Regeln (Kapitel 4.3.1), die Umwandlung dieser Definitionen in Funktionen (Kapitel 4.3.2) sowie die Anwendung der Regelfunktionen auf gegebene Argumente erklärt (Kapitel 4.3.3).

[12]Die angegebenen Zahlen verweisen auf die IDs der entsprechenden Zeilen.

1.	$a \to b$	Vorbedingung
2.	$\neg a \to b$	Vorbedingung
3.	$\neg b$	Annahme
4.	...	
5.	$\bot$	
6.	b	RAA (3-5)

(a) Zeilenform

```
[{:id 1 :body (impl a b) :rule :premise}
 {:id 2 :body (impl (not a) b)
  :rule :premise}
 [{:id 5 :body (not b) :rule :assumption}
  {:id 6 :body :todo :rule nil}
  {:id 7 :body contradiction :rule nil}]
 {:id 4 :body b :rule "\"raa\" ([5-7])"}]
```

(b) Interne Struktur

Abbildung 4.1: Derselbe Beweis in klassischer Zeilenform (a) und in der internen Beweisstruktur (b).

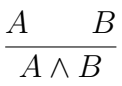
4.3.1 Struktur

Die Struktur der Regeln orientiert sich an der von Kirsten [Kir14]. Regeln werden dabei als Hashmaps dargestellt mit den folgenden Schlüsseln:

- **:name** Ein String mit dem Namen der Regel. Über diesen werden die verschiedenen Regeln zur Anwendung aufgerufen.
- **:given** Ein Vektor mit den Vorbedingungen der Regel.
- **:conclusion** Ein Vektor mit den Schlussfolgerungen der Regel.
- **:prereq** Ein Vektor mit zusätzlichen Vorbedingungen der Regel. Wird nur für die Elimination der Gleichheit verwendet (siehe Kapitel 4.3.1.1).
- **:forwards** Gibt an, ob die Regel vorwärts angewendet werden darf (*true* oder *false*)[13].
- **:backwards** Gibt an, ob die Regel rückwärts angewendet werden darf (*true* oder *false*)[13].

Im direkten Vergleich zwischen der hier vorgestellten Definition und der ursprünglich von Kirsten entwickelten fällt auf, dass der Eintrag *:forms* wegfällt (siehe Codebeispiel 1). Unter diesem Eintrag werden logische Formeln gruppiert und mit einem Alias versehen, welcher dann in den Vorbedingungen und Schlussfolgerungen der Regel verwendet wird. Dies ist entscheidend für die spätere Umwandlung der Definition zu einer anwendbaren Funktion. Allerdings macht der Eintrag das Schreiben von

[13]Ein Verzicht auf den entsprechenden Schlüssel entspricht *false*.



$$\frac{A \quad B}{A \wedge B}$$

(a) Klassische Schreibweise

```
{:name "and-i"
 :given [a b]
 :conclusion [(and a b)]
 :forwards true
 :backwards true}
```

(b) Interne Struktur

Abbildung 4.2: Die Definition der $\wedge$ Einführung in klassischer Schreibweise (a) und in der internen Struktur als Hashmap (b)

eigenen Regeln wesentlich aufwendiger. Deshalb wurde das Umwandlungsverfahren verändert, sodass die Definition nun wesentlich simpler ausfällt.

Auch der Umstieg auf eine Schreibweise mit entsprechenden Worten für die logischen Operatoren wirkt sich vereinfachend aus. Durch den Wegfall der Sonderzeichen lassen sich neue Definitionen schneller verfassen und eventuelle Schwierigkeiten mit verschiedenen Zeichensätzen verschwinden.

```
{:name "and-e-1"
 :premise [$and]
 :consequence $a
 :forms [[$and ($a ∧ $b)]]
 :forward true}
```

(a) Regeldefinition nach Kirsten

```
{:name "and-e1"
 :given [(and a b)]
 :conclusion [a]
 :forwards true}
```

(b) Regeldefinition des Projekts

Codebeispiel 1: Vergleich der Regeldefinitionen des Projekts und von Kirsten

Regeln werden in eigenen Dateien gespeichert und über die Funktion **import-rules** des Namenraums „io“ eingelesen. Dabei können auch mehrere Regeldefinitionen hintereinander in einer Datei stehen. Wer sofort loslegen möchte, findet die Regeln für Aussagen- und Prädikatenlogik („rules-prop-pred.clj“) sowie für die lineare temporale Logik („rules-temporal.clj“) schon vordefiniert.

4.3.1.1 Sonderfall: Elimination der Gleichheit

Die Elimination der Gleichheit (siehe Abb. 3.19 S. 26) stellt einen Sonderfall der Regeln dar. Sie verfügt über zusätzliche Vorbedingungen (*:prereq*, siehe Codebeispiel 2a), welche überprüfen, ob eine Substitution mit den entsprechenden Werten überhaupt erlaubt ist. Nur falls alle so definierten Vorbedingungen den Wert *true* zurückliefern, wird die Regel ausgeführt. Die dort aufgeführte Funktion **substitutionable?** liegt im Namensraum „prereqs“.

Außerdem enthält die Regel eine spezielle Schreibweise für Nutzereingaben. Normalerweise werden beim späteren Ausführen der Regeln die Zeilennummern des Beweises angegeben, welche als Argumente dienen (siehe Kapitel 4.4). Das Präfix **__:** kennzeichnet dagegen eine Variable, die vom Nutzer direkt eingegeben werden muss (siehe Abb. 2b). Dadurch kann der Nutzer explizit entscheiden, welche der Variablen substituiert werden soll.

```
{:name "equal-e"
 :given [(= a b) phi _:x]
 :conclusion [(substitution phi _:x b)]
 :prereq [(substitution? phi _:x a)]
 :forwards true}
```

(a) Regeldefinition der = Elimination

```
(step-f proof
        "equal-e"
        1 2 'x)
```

(b) Aufruf der = Elimination

Codebeispiel 2: Das Schlüsselwort *:prereq* kennzeichnet zusätzliche Vorbedingungen der Regel. Das Präfix **__:** des Parameters *__:x* kennzeichnet, dass dieser vom Nutzer im späteren Aufruf explizit mit angegeben werden muss (siehe (b) der Parameter *'x*)

Diese Sonderformen lassen sich theoretisch auch für andere Regeln nutzen, um entsprechende Vorbedingungen und Nutzereingaben zu ermöglichen. Die beiden Features wurden jedoch zunächst speziell für die Elimination der Gleichheit geschaffen. Dementsprechend ist ihre allgemeine Einsetzbarkeit noch nicht ausreichend getestet worden. Prinzipiell sollten diese Optionen aber auch bei anderen Regeln funktionieren.

4.3.2 Umwandlung

Um die Regeldefinitionen auf logische Formeln anzuwenden, müssen jene zunächst für die Nutzung mit „core.logic“[14] umgewandelt werden. Diese Bibliothek bietet relationale und logische Programmierung für Clojure. Mit diesen Techniken werden aus den Definitionen entsprechende Funktionen erstellt, die anschließend, ebenfalls mit Hilfe von core.logic, auf gegebene Formeln angewendet werden können.

4.3.2.1 Einführung in core.logic

Um die Funktionsweise der Regelfunktionen besser verstehen zu können, soll ein grundlegendes Wissen der Bibliothek core.logic und der logischen Programmierung vermittelt werden. Dabei werden nur die für dieses Projekt nötigen Funktionalitäten

[14] `https://github.com/clojure/core.logic`

beleuchtet. Für einen umfangreicheren Einstieg in core.logic siehe `https://github.com/clojure/core.logic/wiki/A-Core.logic-Primer`.

Bei der logischen Programmierung mit core.logic schreibt man sog. logische Ausdrücke. Diese bestehen aus logischen Variablen und Restriktionen. Von einer „Logic Engine“ werden diese Ausdrücke angenommen und ausgewertet. Dabei werden alle möglichen Belegungen der logischen Variablen zurückgegeben, die sämtliche Restriktionen erfüllen. Das ermöglicht auch mehrere Ergebnisse oder überhaupt keine, je nach Wahl von Variablen und Restriktionen.

Man werfe nun einen Blick auf ein einfaches Beispiel:

```
1 => (run* [q]
2       (== q 1))
3 (1)
```

Codebeispiel 3: Eine einfache Logikfunktion ausgeführt mit core.logic

Die Funktion **run*** übergibt alle logischen Variablen und Restriktionen an die Engine und liefert deren Ergebnisse zurück. In diesem Fall gibt es nur die logische Variabel *q* und lediglich eine Restriktion *(== q 1)*. Die Funktion == (unify) zwingt die beiden Werte sich anzugleichen und ist immer dann erfüllt, wenn dies gelingt. Im Codebeispiel 3 muss *q* daher den Wert *1* annehmen. Da es sonst keine weiteren Restriktionen gibt, ist dies die Lösung des Programms. Weitere Beispiele zu == und möglichen Ergebnissen finden sich in Codebeispiel 4.

Die letzte wichtige Funktion, die benötigt wird ist **fresh**. Diese führt neue logische Variablen ein, welche für logische Restriktionen genutzt werden können. Allerdings werden ihre Belegungen nicht mit im Ergebnis ausgegeben (siehe Codebeispiel 5).

Mit den Funktionen **run***, **fresh** und == stehen alle Werkzeuge zur Verfügung, welche für die Konstruktion der Regelfunktionen benötigt werden. Doch core.logic bietet noch andere Funktionen für verschiedenste weitere Anwendungsfälle. Für mehr Informationen lohnt sich ein Blick in das offizielle Wiki unter `https://github.com/clojure/core.logic/wiki`.

```
=> (run* [q]
     (== q 1)
     (== q 2))
()
```

(a) Alle Restriktionen müssen erfüllt sein. Da die Variable *q* nicht gleichzeitig *1* und *2* sein kann, gibt es in diesem Beispiel keine Lösung.

```
=> (run* [q]
     (== {:a q :b "Tom"}
         {:a "Hallo" :b "Tom"}))
("Hallo")
```

(b) Die == Funktion kann auch komplexere Werte angleichen. Wenn *q* den Wert *„Hallo“* annimmt, sind die beiden Hashmaps gleich und die Restriktion erfüllt. Daher ist *„Hallo“* die Lösung.

```
=> (run* [q]
     (== 1 1))
(_0)
```

(c) Eine Lösung der Form *_0* heißt, dass die Variable *q* jeden beliebigen Wert annehmen kann. Die Restriktion ist immer erfüllt. Die Nummerierung hält „verschieden beliebige“ Variablen auseinander.

Codebeispiel 4: Weitere Beispiele für Logikfunktionen mit core.logic

```
=> (run* [q]
     (fresh [p]
       (== 1 p)
       (== q {:a p})))
({:a 1})
```

Codebeispiel 5: Die Variable *p* verhält sich wie eine lokale Variable, die innerhalb der Funktion verwendet werden kann. Zum Ergebnis gehört aber nur die Variable, die auf der obersten Ebene definiert wurde (hier: *q*).

4.3.2.2 Regelfunktionen

Mit den Funktionen, die im vorherigen Abschnitt erläutert wurden, lassen sich nun die gewünschten Regelfunktionen erstellen. Man werfe beispielsweise einen Blick auf die ∧ Elimination 1:

$$\frac{A \wedge B}{A}$$

```
1 => (defn and-elim
2       [and1 q]
3       (fresh [a b]
4         (== and1 `(~'and ~a ~b))
5         (== q a)))
6 => (run* [q] (and-elim '(and x y) q))
7 (x)
```

Codebeispiel 6: Die Regelfunktion der ∧ Elimination 1

Die erste Restriktion in Zeile vier überprüft, ob die Form des übergebenen Arguments einer Konjunktion entspricht, d.h. eine Liste, die mit dem Schlüsselwort „and“ beginnt und genau zwei weitere Argumente besitzt. Gleichzeitig werden dabei den beiden fresh-Variablen a und b die Werte dieser Argumente zugeordnet. Sollte das Argument nicht dieser Form entsprechen, ist die Restriktion nicht erfüllt und die Funktion liefert kein Ergebnis. Sonst wird die logische Variable q in Zeile fünf mit dem Wert der fresh-Variablen a gleichgesetzt. Dies ist schlussendlich auch das Endergebnis der Funktion.

Erklärung der Schreibweise `(~'and ~a ~b)

Zum Unifizieren des Arguments mit einer bestimmten Form werden die Reader Makros ' (Quote), ` (Syntax-Quote) und ~ (Unquote) von Clojure genutzt. Ohne diese Makros würde Clojure versuchen, die Liste *(and a b)* als Funktion zu interpretieren (siehe Codebeispiel 7a). Dieses Verhalten lässt sich in Clojure durch das Quote-Makro unterbinden (siehe Codebeispiel 7b). Damit aber die Variablen a und b als solche erkannt und eingesetzt werden, brauchen wir das Unquote-Makro, welches jedoch nur mit einem Syntax-Quote funktioniert. Dieser hat jedoch die Besonderheit, Symbole in den momentanen Kontext zu setzen und den aktuellen Namensraum anzuhängen (siehe Codebeispiel 7c). Für die hiesigen Zwecke ist diese Besonderheit jedoch hin-

derlich, daher wird das Konstrukt ∼’ genutzt. Mittels des Unquote-Makros ist es möglich innerhalb des Syntax-Quotes ein normales Quote-Makro zu benutzten, sodass kein Namensraum angehängt wird (siehe Codebeispiel 7d).

```
1 => (and a b)
2 Unable to resolve symbol:
3 a in this context [...]
4 => (impl a b)
5 Unable to resolve symbol:
6 impl in this context [...]
```

(a) Clojure versucht die gegebene Liste als Funktion auszuführen. Dabei behandelt der Compiler die Listenelemente wie Symbole und versucht auf die dahinter liegenden Funktionen und Werte zuzugreifen (*and* ist eine Standardfunktion von Clojure, deshalb wird dort kein Fehler geworfen).

```
1 => '(and a b)
2 (and a b)
```

(b) Das Quote-Makro ’ verhindert das Ausführen als Funktion und gibt einfach den „gequoteten“ Wert zurück.

```
1 => (def a 1)
2 => (def b 2)
3 [...]
4 => `(and ~a ~b)
5 (clojure.core/and 1 2)
6 => `(impl ~a ~b)
7 (current-namespace/impl 1 2)
```

(c) Im Gegensatz zum normalen Quote ’ ergänzt der Syntax-Quote ‘ alle Symbole, welche er findet, um den zugehörigen Namensraum (im Zweifelsfall den aktuellen). Außerdem ermöglicht er es, das Unqote-Makro ∼ zu benutzen.

```
1 => (def a 1)
2 => (def b 2)
3 [...]
4 => `(~'and ~a ~b)
5 (and 1 2)
```

(d) Durch das Konstrukt ∼’ wird das Anhängen des Namenraum verhindert. Trotzdem hat man durch den Syntax-Quote ‘ weiterhin Zugriff auf das Unquote-Makro ∼.

Codebeispiel 7: Erklärung von Quote ’, Syntax-Quote ‘ und Unquote ∼ Makros von Clojure

4.3.2.3 Automatische Umwandlung

Mit dem Wissen, wie sich mit Hilfe von core.logic Regelfunktionen schreiben lassen, kann dieser Prozess nun automatisiert werden (hier am Beispiel der ∧ Elimination 1, siehe Codebeispiel 8).

Zunächst werden die Funktionsargumente erzeugt. Dazu konvertiert man die Vor-

```
{:name "and-e1"
 :given [(and a b)]
 :conclusion [a]
 :forwards true}
```

Codebeispiel 8: Die Regeldefinition der ∧ Elimination 1

bedingungen der Regel nach folgendem Muster: Ist eine Vorbedingung ein Symbol (z.B. *a*, *b*) wird es einfach übernommen, ist es eine Liste (z.B. *(and a b)*), wird ein Alias erzeugt. Dieser besteht aus dem Operator der Liste plus einer fortlaufenden Ziffer[15]. Daraufhin wird für jede Schlussfolgerung ein Argument, bestehend aus einem *q* und einer fortlaufenden Ziffer (z.B. *q1*, *q2* usw.)[15], erzeugt. Diese verschiedenen Argumente werden nun hintereinander in einen Vektor gepackt (siehe Codebeispiel 9).

```
:given [(and a b)]
:conclusion [a]
```

(a)

```
[and1 q1]
```

(b)

Codebeispiel 9: Die Vorbedingungen und Schlussfolgerungen der ∧ Elimination 1 (a) sowie die daraus resultierenden Funktionsargumente (b)

Nun werden die Argumente für die **fresh** Funktion von core.logic erstellt. Dazu werden Vorbedingungen wie Schlussfolgerungen der Regel nach Symbolen durchsucht (auch innerhalb von Listen). Von dieser Menge werden all jene Symbole aussortiert, die in den Vorbedingungen als alleinstehende Argumente vorkommen (nur in den Vorbedingungen)(siehe Codebeispiel 10).

```
:given [(and a b)]
:conclusion [a]
```

(a)

```
[a b]
```

(b)

Codebeispiel 10: Die Vorbedingungen und Schlussfolgerungen der ∧ Elimination 1 (a) sowie die daraus resultierenden **fresh** Argumente (b)

[15]Die genaue Erzeugung ist unwichtig. Entscheidend ist, dass jedes so gebildete Argument eindeutig ist.

Es folgt die Erstellung der ersten Unifikationen. Dafür werden die Funktionsargumente der Vorbedingungen (ohne die angehängten Logikargumente) sowie die Vorbedingungen der Regel durchlaufen. Für jedes Funktionsargument, welches kein Symbol, sondern eine Liste ist, wird eine Unifikation wie in Codebeispiel 11 erzeugt. Dasselbe passiert für die Logikargumente und die Schlussfolgerungen. Dabei wird jedes Logikargument mit der entsprechenden Schlussfolgerung unifiziert.

```
1 :given [(and a b)]
2 [and1]
```

(a)

```
1 (== and1 `(~'and ~a ~b))
```

(b)

```
1 :conclusion [a]
2 [q1]
```

(c)

```
1 (== q1 a)
```

(d)

Codebeispiel 11: Die Unifikationen für die Vorbedinungen (a + b) sowie die Schlussfolgerungen (c + d)

Abschließend wird aus allen Elementen eine anonyme Funktion zusammengesetzt (siehe Codebeispiel 12). Die so erstellte Funktion kann nun in einem entsprechenden Aufruf genutzt werden.

```
1 (fn
2   [and1 q1] ;; die kombinierten Funktions- und Logikargumente
3   (fresh [a b] ;; die fresh Argumente
4     (== and1 `(~'and ~a ~b))
5     (== q1 a))) ;; die erstellten Unifikationen
```

Codebeispiel 12: Die fertige Regelfunktion der ∧ Elimination 1

4.3.3 Anwendung

Die für das Ausführen einer Regel zuständige Funktion heißt **apply-rule**. Ihr wird der Name der Regel, die Ausführungsrichtung (vorwärts oder rückwärts) sowie die Argumente übergeben. Außerdem ist es möglich, optionale Argumente mit anzugeben.

Über den Namen der Regel wird die entsprechende Regeldefinition herausgesucht. Ist die Anwendungsrichtung „vorwärts“ (*forward? = true)* wird die Regelfunktion aus dieser Definition erstellt. Bei „rückwärts“ (*forward? = false)* dagegen werden vor dem Erstellen der Funktion die Vorbedingungen und Schlussfolgerungen der Regel vertauscht. Durch diesen simplen Schritt wird die Regel rückwärts angewendet.

Da die Reihenfolge der übergebenen Argumente entscheidet, ob eine Regelfunktion ein Ergebnis hat bzw. wie dieses Ergebnis aussieht (siehe Abb. 13), werden alle möglichen Permutationen der Argumente ausprobiert. Dies geschieht in einer for-Schleife, bei der am Ende leere Ergebnismengen aussortiert werden. Die Logikargumente sind davon ausgeschlossen, sie bilden immer die letzten Argumente einer Regelfunktion und werden in diesem Schritt nicht mit permutiert.

```
1 => (run* [q] (impl-elim '(impl a b) 'a q))
2 (b)
3 => (run* [q] (impl-elim 'a '(impl a b) q))
4 ()
```

(a) → Elimination

```
1 => (run* [q] (and-intro 'a 'b q))
2 (and a b)
3 => (run* [q] (and-intro 'b 'a q))
4 (and b a)
```

(b) ∧ Einführung

Codebeispiel 13: Bei der → Elimination entscheidet die Reihenfolge, ob ein Ergebnis zustande kommt oder nicht (a). Bei der ∧ Einführung gibt es zwei verschiedene Ergebnisse je nach Reihenfolge (b).

Außerdem bietet die Funktion **apply-rule** die Möglichkeit, optionale Argumente anzugeben, um das Ergebnis genauer zu spezifizieren. Optionale Argumente funktionieren nur bei Regeln, die mehr als eine Schlussfolgerung haben (z.B. ∨ Elimination „rückwärts“). In diesem Fall kann man bis zu „Anzahl der Schlussfolgerungen – 1“ optionale Argumente angeben. Diese werden anstelle der logischen Variablen eingefügt und formen so das Ergebnis (siehe Codebeispiel 14). Auch hier werden wieder alle möglichen Permutationen von Logikargumenten und optionalen Argumenten getestet. Dies geschieht in einer zweiten for-Schleife. So werden alle möglichen Permutationen

der Vorbedingungen mit allen möglichen Permutationen von optionalen Argumenten und Logikargumenten verglichen und deren Ergebnisse ausgewertet. Nützlich ist dies, wenn bestimmte Ergebnisse im Beweis schon vorhanden sind, so können diese bei der Ausführung der Regel berücksichtigt werden.

```
1 => (apply-rule "or-e" false '[X])
2 ([(or _0 _1) (infer _0 X) (infer _1 X)])
```

(a) Ohne optionale Argumente sind einige der Felder des Ergebnisses nicht spezifiziert. Stattdessen werden die Platzhalter *_0* und *_1* von core.logic eingefügt.

```
1 => (apply-rule "or-e" false '[X] '[(or p q)])
2 ([(infer p X) (infer q X)] [(infer q X) (infer p X)])
```

(b) Durch den optionalen Parameter *(or p q)* sind die Platzhalter aus dem Ergebnis verschwunden. Außerdem erscheint das angegebene Argument nicht mit in der Ergebnismenge. Durch die verschiedenen Permutationen erhält man zwei mögliche Ergebnisse, die sich jedoch einzig und allein in der Reihenfolge ihrer Teilergebnisse unterscheiden.

Codebeispiel 14: Anwendung der $\vee$ Elimination ohne (a) und mit (b) optionalen Argumenten

Die Funktion **apply-rule** sollte außer für Testzwecke niemals direkt vom Nutzer aufgerufen werden. Sie verwendet keinerlei Überprüfung, ob die übergebenen Argumente korrekt sind und ist daher bei falscher Benutzung äußerst fehleranfällig. Sie wird von den verschiedenen Beweisschritten in Kapitel 4.4 verwendet, wobei dort auch eine entsprechende Überprüfung der Argumente stattfindet.

4.4 Deduktion

Die eigentlichen Funktionen zur Deduktion, also das Überprüfen der Argumente, das Anwenden der Regeln und das Einfügen der Ergebnisse an den richtigen Stellen, befinden sich im Namensraum „deduction“ des Projekts. Diese Funktionen sind die Schnittstelle für Nutzer oder andere Programme zur Führung von logischen Beweisen. Folgende Funktionen sind wichtig und werden in den folgenden Abschnitten näher beleuchtet:

- **(proof formula)/(proof premises formula)** Konstruiert aus den gegebenen Vorbedingungen und der Schlussfolgerung einen neuen Beweis
- **(step-f proof rule & lines)** Wendet eine Regel vorwärts im gegebenen Beweis auf die angegebenen Zeilen an
- **(step-f-inside proof rule line)** Sonderfall von **step-f**, wendet eine Regel vorwärts innerhalb einer Zeile an
- **(step-b proof rule & lines)** Wendet eine Regel rückwärts im gegebenen Beweis auf die entsprechenden Zeilen an
- **(choose-option proof line option)** Ermöglicht das Auswählen einer Option, sollte eine Regel verschiedene Ergebnisse ausgeben
- **(rename-var proof old new)** Benennt eine Variable um
- **(trivial proof line)** Wendet sämtliche triviale Theoreme innerhalb einer Zeile an

Zudem müssen die Funktionen **check-args** und **check-duplicates** noch erwähnt werden. Beide werden von den hier aufgelisteten Funktionen verwendet und führen wichtige Überprüfungen durch.

Die Funktion **check-args** prüft Argumente für die Anwendung mit Regeln. Dabei wird etwa sichergestellt, dass eine Regel existiert, die Anzahl der Argumente korrekt ist, alle Zeilen im gleichen Gültigkeitsbereich liegen usw. Sollte eine der Bedingungen nicht erfüllt sein, wird eine entsprechende Fehlermeldung ausgegeben.

Fügt eine Funktion neue Zeilen in den Beweis ein oder verändert bestehende, so wird anschließend **check-duplicates** aufgerufen. Diese Funktion sucht innerhalb der Beweisstruktur nach doppelten Einträgen, welche sowohl bewiesen als auch unbewiesen vorliegen und löscht die „überflüssigen“ unbewiesenen Zeilen. Anschließend werden die IDs angepasst, indem die gelöschten IDs durch die ihrer entsprechenden Duplikate ersetzt werden. Dabei wird immer darauf geachtet, dass keine Zeilen aus einem anderen Gültigkeitsbereich gelöscht und durch Löschungen keine Unterbeweise „zerstückelt“ werden (siehe Codebeispiel 15).

```
=> (check-duplicates
     '[{:id 1 :body (at x a) :rule :premise}
       [{:id 4 :body (<= x x) :rule :assumption}
        {:id 5 :body :todo :rule nil}
        {:id 6 :body (at x a) :rule nil}]
       {:id 3 :body (at x (always a)) :rule "\"always-i\" ([4-6])"}])
[{:id 1, :body (at x a), :rule :premise}
 [{:id 4, :body (<= x x), :rule :assumption}
  {:id 6, :body (at x a), :rule "\"already proved\" (1)"}]
 {:id 3, :body (at x (always a)), :rule "\"always-i\" ([4-6])"}]
```

Codebeispiel 15: Die in Zeile fünf notierte Zeile des Beweises ist unbewiesen und von der Formel gleich der in Zeile zwei zu sehenden. Da das Löschen dieser Zeile allerdings den Unterbeweis um seine Schlussfolgerung berauben würde, wird mit den Worten „already proven“ auf diesen Umstand hingewiesen.

4.4.1 Vorwärtsschritt

Der einfachste Fall einer Regelanwendung ist der Vorwärtsschritt. Man gibt die Regel an, welche man ausführen möchte, sowie die Zeilen, die als Vorbedingungen dienen sollen. Sollten alle Argumente korrekt sein, wird das Ergebnis bzw. werden die Ergebnisse hinter der letzten angegebenen Zeile oder vor einer eventuell im Gültigkeitsbereich vorhanden leeren Zeile (...) eingefügt (siehe Codebeispiel 16). Jede so eingefügte Zeile hat in ihrer Ursprungsangabe die Regel und die IDs der Zeilen, die als Vorbedingungen dienten.

```
=> (step-f
     '[{:id 1 :body (and a b) :rule :premise}
       {:id 2 :body :todo :rule nil}
       {:id 3 :body (and b a) :rule nil}]
     "and-e1" 1)
[{:id 1 :body (and a b) :rule :premise}
 {:id 4 :body a :rule "\"and-e1\" (1)"}
 {:id 2 :body :todo :rule nil}
 {:id 3 :body (and b a) :rule nil}]
```

Codebeispiel 16: Die Anwendung der $\wedge$ Elimination 1 vorwärts mittels **step-f**

Ein Vorwärtsschritt ist die einzige Möglichkeit, Regeln auszuführen, die keine Vorbedingungen besitzen. Dies wird zum Beispiel genutzt, um Relationen von Zeitpunk-

ten oder allgemeingültige Theoreme in den Beweis einzufügen.

4.4.1.1 Innerer Vorwärtsschritt

Manchmal möchte oder muss man eine Regel nur auf Teile einer gegebenen Formel anwenden. Mit der Funktion **step-f-inside** ist genau dies machbar. Diese benötigt die Regel sowie die Zeile, die behandelt werden soll. Die Funktion sucht nun innerhalb dieser Zeile nach Teilen, welche sich mit der angegebenen Regel auflösen lassen. Das Ergebnis wird als neue Zeile direkt hinter der ursprünglichen in die Beweisstruktur eingefügt (siehe Codebeispiel 17).

Das Ganze funktioniert nur mit Regeln, die vorwärts anwendbar sind und genau eine Vorbedingung sowie genau eine Schlussfolgerung besitzen. Sollte es innerhalb der Zeile mehrere Möglichkeiten geben, die Regel anzuwenden oder entstehen durch das Anwenden weitere Möglichkeiten, so wird die Regel auch auf diese Teile angewendet (siehe Codebeispiel 18).

```
1 => (step-f-inside
2      '[{:id 1 :body (or a (and b c)) :rule :premise}
3        {:id 2 :body :todo :rule nil}
4        {:id 3 :body c :rule nil}]
5      "and-e2" 1)
6 [{:id 1 :body (or a (and b c)) :rule :premise}
7  {:id 4 :body (or a c) :rule "\"and-e2\" (1)"}
8  {:id 2 :body :todo :rule nil}
9  {:id 3 :body c :rule nil}]
```

Codebeispiel 17: Die ∧ Elimination 2 wird innerhalb der ersten Zeile des Beweises durch **step-f-inside** angewandt. Das Ergebnis der Anwendung wird direkt hinter der entsprechenden Zeile eingefügt.

4.4.2 Rückwärtsschritt

Ein Rückwärtsschritt funktioniert äquivalent zum Vorwärtsschritt. Neben der anzuwendenden Regel und den entsprechenden Zeilen kommt es oft vor, dass bei einem Rückwärtsschritt optionale Zeilen angegeben werden (dies ist bei Vorwärtsschritten auch möglich, wird aber kaum verwendet). Optionale und obligatorische Zeilennummern müssen dabei nicht getrennt angegeben werden. Die Funktion sortiert die Zeilen automatisch und überträgt diese korrekt an die Regelfunktion. Die Ergebnisse eines Rückwärtsschritt werden vor der ausgehenden Zeile eingefügt.

```
=> (step-f-inside
     '[{:id 1 :body (and a (and b c)) :rule :premise}
       {:id 2 :body :todo :rule nil}
       {:id 3 :body c :rule nil}]
     "and-e2" 1)
[{:id 1 :body (and a (and b c)) :rule :premise}
 {:id 3 :body c :rule "\"and-e2\" (1)"}]
```

Codebeispiel 18: Sollte sich die Möglichkeit ergeben wird die Regel innerhalb der Zeile mehrfach bzw. auch auf Teilergebnisse mittels **step-f-inside** angewandt.

```
=> (step-b
     '[{:id 1 :body a :rule :premise}
       {:id 2 :body :todo :rule nil}
       {:id 3 :body contradiction :rule nil}]
     "not-e" 3 1)
[{:id 1 :body a :rule :premise}
 {:id 2 :body :todo :rule nil}
 {:id 4 :body (not a) :rule nil}
 {:id 3 :body contradiction :rule "\"not-e\" (1 4)"}]
```

Codebeispiel 19: Die Rückwärtsanwendung mit **step-b** der ¬ Elimination auf Zeile drei des Beweises. Durch die optionale Angabe der Zeile eins des Beweises wurde das Ergebnis spezifiziert. Ein Aufruf mit vertauschten Zeilennummern (1 3) führt zum selben Ergebnis.

4.4.3 Auswahl von Optionen

Manchmal kann die Anwendung einer Regel zu mehreren möglichen Ergebnissen führen. Dies geschieht durch die Anwendung der verschiedenen Permutationen auf die Regel (siehe Kapitel 4.3.3). In solchen Fällen wird eine Zeile eingefügt, welche die zur Auswahl stehenden Resultate anzeigt. Der Nutzer kann dann mittels der Funktion **choose-option** selbst entscheiden, welches der möglichen Ergebnisse das richtige bzw. das gesuchte ist (siehe Codebeispiel 20).

Die entsprechende Zeile hält unter dem *:body* Schlüssel keine Formel, sondern eine Hashmap. Diese enthält unter nummerierten Schlüsseln die verschiedenen möglichen Ergebnisse. Durch das Auswählen einer entsprechenden Option, wird der entsprechende Eintrag aus der Hashmap anstelle der Zeile eingefügt.

```
1  => (choose-option
2       '[{:id 1 :body a :rule :premise}
3         {:id 2 :body b :rule :premise}
4         {:id 5 :body {1 (and a b) 2 (and b a)}
5          :rule "\"and-i\" (1 2)"}
6         {:id 3 :body :todo :rule nil}
7         {:id 4 :body (and b a) :rule nil}]
8       3 2)
9  [{:id 1 :body a :rule :premise}
10  {:id 2 :body b :rule :premise}
11  {:id 4 :body (and b a) :rule "\"and-i\" (1 2)"}]
```

Codebeispiel 20: Auswahl eines Ergebnisses der ∧ Einführung mittels **choose-option**. Die *3* in Zeile acht gibt die Zeilennummer des Beweises an, die *2* dahinter die zu wählende Option.

4.4.4 Variablen

Bei der Anwendung bestimmter Regeln kann es vorkommen, dass bestimmte Teile einer Formel jeden beliebigen Wert annehmen können. Diese Positionen werden von core.logic standardmäßig mit Platzhaltern der Form *_0*, *_1*, usw. besetzt (siehe Codebeispiel 4, S. 43). Die Nummer zeigt an, welche Stellen den gleichen (beliebigen) Wert annehmen müssen. Da core.logic diese Nummerierung bei jeder neuen Regelanwendung immer wieder bei null beginnt, werden diese Platzhalter durch eigene fortlaufend nummerierte Variablen ersetzt (V1, V2, V3 usw.). So werden eventuelle Überschneidungen, durch gleiche Platzhalter von verschiedenen Ergebnissen, verhindert.

4.4.4.1 Umbenennung von Variablen

Die automatisch generierten und nummerierten Variablen sind nur Platzhalter und führen meist nicht zum gewünschten Ergebnis. Um sie für die eigene Beweisführung brauchbar zu machen, kann man sie daher umbenennen. Dafür existiert die Funktion **rename-var**. Diese durchsucht den kompletten Beweis nach Auftritten der gesuchten Variable und ersetzt diese mit der neuen Bezeichnung (siehe Codebeispiel 21). Man kann Variablen nicht nur durch Symbole (z.B. *a, b, P, Q* usw.) ersetzten, sondern auch durch ganze Formeln (z.B. *(not a), (or a b)* usw.). Generell können alle Symbole eines Beweises umbenannt werden. Der Nutzer ist hier selbst dafür verantwortlich, die Funktion nicht zu missbrauchen.

```
=> (rename-var
     '[{:id 1 :body a :rule :premise}
       {:id 4 :body (or a V1) :rule "\"or-i1\" (1)"}
       {:id 2 :body :todo :rule nil}
       {:id 3 :body (or a b) :rule nil}]
     'V1 'b)
[{:id 1 :body a :rule :premise}
 {:id 3 :body (or a b) :rule "\"or-i\" (1)"}]
```

Codebeispiel 21: Dieser Beweis wird durch die Umbenennung der Variablen *V1* mittels **rename-var** gelöst.

4.4.5 Triviale Theoreme

Triviale Theoreme bezeichnen Regeln, welche mit den Wahrheitswerten *true* und *false* interagieren. In Abbildung 4.3 werden einige Beispiele aufgezeigt.

$$\frac{A \wedge false}{false} \qquad \frac{A \vee false}{A} \qquad \frac{A \wedge true}{A} \qquad \frac{A \to A}{true}$$

Abbildung 4.3: Einige Beispiele für triviale Theoreme

Diese trivialen Regeln sind für jeden Logiker sofort ersichtlich, für das hier entwickelte Programm jedoch nicht. Um Beweise führen zu können, welche mit solchen Formeln arbeiten, braucht es daher einer Möglichkeit, diese Regeln anzuwenden.

Sämtliche trivialen Theoreme werden standardmäßig in einer separaten Datei namens „trivial-theorems.clj“ gespeichert. Sie werden mit einigen Einschränkungen genau wie alle anderen Regeln aufgeschrieben. Sie müssen genau eine Vorbedingung und eine Schlussfolgerung haben, außerdem können die Schlüssel *:forwards* und *:backwards* weggelassen werden, triviale Theoreme werden nur vorwärts ausgeführt. Das Auslagern in eine externe Datei ermöglicht es dem Nutzer, weitere triviale Theoreme hinzuzufügen, etwa für weitere Logiken oder zum Experimentieren. Geladen wird diese externe Datei mit der Funktion **import-trivials** des Namensraums „io“.

Angewendet werden die trivialen Theoreme mit der Funktion **trivial**. Diese funktioniert sehr ähnlich wie ein innerer Vorwärtsschritt (siehe Kapitel 4.4.1.1). Auf die angegebene Zeile werden von innen heraus nach und nach alle geladenen trivialen Theoreme angewandt. Die neu entstandene Zeile wird wie bei einem Vorwärtsschritt eingefügt. Sollte keine Regel passen, wird eine Nachricht ausgegeben, die den Nutzer darauf hinweist.

```
=> (trivial
     '[{:id 1 :body (and a (or b false)) :rule :premise}
       {:id 2 :body :todo :rule nil}
       {:id 3 :body b :rule nil}])
[{:id 1 :body (and a (or b false)) :rule :premise}
 {:id 4 :body (and a b) :rule "\"trivial\" ()"}
 {:id 2 :body :todo :rule nil}
 {:id 3 :body b :rule nil}]
```

Codebeispiel 22: Ein Beispiel für die Anwendung der Funktion **trivial**. Ohne die Nutzung von trivialen Theoremen wäre dieser Beweis für das Programm nicht lösbar.

4.5 Theoreme

Wird eine Hypothese bewiesen, d.h. sie enthält keine Zeilen ohne Ursprungsangabe mehr, so wird sie zu einem Theorem. Dieses kann nun wie eine Regel in anderen Beweisen verwendet werden. Dabei sind die Vorbedingungen und Schlussfolgerungen dieser Regel gleich denen des Theorems. Durch diese Wiederverwendbarkeit spart man eine Menge Aufwand und vereinfacht komplizierte und lange Beweise.

Um Theoreme zu nutzen, müssen bewiesene Hypothesen als solche exportiert werden. Alternativ lassen sich auch abgespeicherte Theoreme jederzeit importieren. Wurde ein entsprechendes Theorem exportiert oder importiert, kann es wie eine Regel verwendet werden.

4.5.1 Import/Export

Die für den Export zuständige Funktion ist **export-theorem** aus dem Namensraum „io“. Sie überprüft zunächst, ob der übergebene Beweis komplett bewiesen wurde und exportiert diesen im Erfolgsfall in die angegebene Datei (siehe Codebeispiel 23). Das Theorem wird der entsprechenden Datei angehängt, es können daher mehrere Theoreme in einer einzigen Datei gespeichert werden. Über den angegebenen Namen wird das Theorem zu einem späteren Zeitpunkt aufgerufen. Der vergebene Name des Theorems sollte sich nicht mit dem anderer Theoreme oder Regeln überschneiden.

Mit dem Export ist das Theorem sofort für weitere Beweise in derselben Session nutzbar. Bei einem Neustart des Programms müssen Theoreme jedoch erst geladen werden, um sie einsetzen zu können. Mit der Funktion **import-theorems** aus dem Namensraum „io“ werden alle in der Datei gespeicherten Theoreme importiert. Sollte es mehrere Dateien mit Theoremen geben, welche man benutzten möchte, muss

die Funktion dementsprechend mehrfach aufgerufen werden (mit den entsprechenden Dateinamen).

```
1  => (io/export-theorem
2       '[{:id 1 :body (impl a b) :rule :premise}
3         {:id 2 :body (not b) :rule :premise}
4         [{:id 4 :body a :rule :assumption}
5          {:id 7 :body b :rule „impl-e" (1 4)}
6          {:id 6 :body contradiction :rule "\"not-e\" (2 7)"}]
7         {:id 3 :body (not a) :rule "\"not-i\" ([4-6])"}]
8       "theorems.clj"
9       "modus-tollens"}
10 [...]
```

Codebeispiel 23: Export des Theorems „Modus Tollens “ für die spätere Verwendung

4.5.2 Anwendung

Theoreme werden genau wie Regeln angewendet mit der Einschränkung, dass sie nur vorwärts angewendet werden können. Es gibt keine zusätzlichen Funktionen für Theoreme, die normalen Funktionen zur Regelanwendung **step-f** und **step-f-inside** können einfach mit dem Namen des Theorems aufgerufen werden (siehe Codebeispiel 24).

```
1  => (step-f
2       '[{:id 1 :body (impl a b) :rule :premise}
3         [{:id 3 :body (not b) :rule :assumption}
4          {:id 4 :body :todo :rule :nil}
5          {:id 5:body (not a) :rule nil}]
6         {:id 6 :body (impl (not b) (not a)) :rule "\"impl-i\" (3-5)"}]
7       "modus-tollens"
8       1 2)
9  [{:id 1 :body (impl a b) :rule :premise}
10  [{:id 3 :body (not b) :rule :assumption}
11   {:id 5:body (not a) :rule "\"modus-tollens\" (1 3)"}]
12  {:id 6 :body (impl (not b) (not a)) :rule "\"impl-i\" (3-5)"}]
```

Codebeispiel 24: Lösung eines Beweises durch die Anwendung des in Codebeispiel 23 exportierten Theorems „Modus Tollens“

4.6 REPL

Die Bezeichnung REPL steht für „read eval print loop“. Sie beschreibt eine Umgebung, in der man mit der entsprechenden Programmiersprache sofort interagieren und experimentieren kann. Eingaben des Nutzers werden eingelesen, ausgewertet und die Ergebnisse werden ausgegeben. Code muss nicht erst in Dateien abgespeichert oder lange kompiliert werden, die Ausgabe erfolgt zügig und im selben Fenster. Eine REPL bietet damit den idealen Startpunkt für Einsteiger, welche eine neue Programmiersprache lernen oder Nutzer, die nur schnell bestimmte Funktionen testen wollen.

Um die Anwendung über diese Schnittstelle zu vereinfachen, wurde ein extra Namensraum „repl“ implementiert. Hier wird die Schreibarbeit reduziert, indem der aktuelle Beweis automatisch zwischengespeichert wird. So muss dieser nicht in jeder Funktion erneut komplett aufgeschrieben werden. Außerdem werden die Ergebnisse nicht mehr als Clojure Datenstruktur ausgegeben, sondern passend formatiert[16]. So erhält man eine einfach zu bedienende und zu lesende Schnittstelle.

Ein neuer Beweis startet immer mit dem Aufruf der Funktion **proof**. Diese erstellt die interne Beweisstruktur und speichert diese auch in den Zwischenspeicher. Außerdem wird der Beweis direkt ordentlich formatiert angezeigt. Alle Funktionen zur Manipulation des Beweises sind einsetzbar[17], mit dem Unterschied, dass die Beweisstruktur nicht mit angegeben werden muss (siehe Codebeispiel 25). Diese wird jedes Mal automatisch aus dem Zwischenspeicher geladen. Außerdem steht dem Nutzer die Funktion **undo** zur Verfügung, mit der er bis zum Startzustand des zwischengespeicherten Beweises zurückgehen kann.

4.6.1 Beispiele

Mit dem Wissen um die Implementierung und der hier vorgestellten Schnittstelle sollen nun die in Kapitel 3.5 vorgestellten Beispiele logischer Beweise neu bewiesen werden. Die verwendeten Regelnamen entsprechen dabei den Standardbenennungen aus dem Projekt. Aus Platzgründen wird auf die Anzeige der Zwischenergebnisse verzichtet.

[16] Die Funktionen zur Formatierung der Ausgaben befindet sich im Namensraum „printer“. Diese können auch unabhängig von der REPL verwendet werden.

[17] step-f, step-f-inside, step-b, trivial, choose-option, rename-var, export-theorem

```
=> (proof '[a b] '(and a b))
   ------------------------------
1: a          premise
2: b          premise
3: ...
4: (and a b)
   ------------------------------
=> (step-f "and-i" 1 2)
   ------------------------------
1: a          premise
2: b          premise
3: {1 (and a b) 2 (and b a)} "and-i" (1 2)
4: ...
5: (and a b)
   ------------------------------
=> (choose-option 3 1)
   ------------------------------
1: a          premise
2: b          premise
3: (and a b) "and-i" (1 2)
   ------------------------------
```

Codebeispiel 25: Beispielhafte Beweisführung im „repl“ Namensraum

Aussagenlogik

```
=> (proof '[(impl a b) (not b)] '(not a))
=> (step-b "not-i" 4)
=> (step-f "impl-e" 1 3)
=> (step-f "not-e" 2 4)
   ------------------------------
1: (impl a b)              premise
2: (not b)                 premise
    ----------------------------
3:  | a                    assumption
4:  | b                    "impl-e" (1 3)
5:  | contradiction         "not-e" (2 4)
    ----------------------------
6: (not a)                 "not-i" ([3 5])
   ------------------------------
```

Codebeispiel 26: Modus Tollens

```
1  => (proof '(or a (not a)))
2  => (step-b "raa" 2)
3  => (step-b "not-e" 1 3)
4  => (choose-option 3 2)
5  => (step-b "or-i2" 3)
6  => (step-b "not-i" 3)
7  => (step-f "or-i1" 2)
8  => (rename-var 'V1 '(not a))
9  => (step-f "not-e" 1 3)
10    ---------------------------------------
11     ---------------------------------------
12 1:  | (not (or a (not a)))       assumption
13     | -----------------------------------
14 2:  | | a                        assumption
15 3:  | | (or a (not a))           "or-i1" (2)
16 4:  | | contradiction            "not-e" (1 3)
17     | -----------------------------------
18 5:  | (not a)                    "not-i" ([2 4])
19 6:  | (or a (not a))             "or-i2" (5)
20 7:  | contradiction              "not-e" (1 6)
21     ---------------------------------------
22 8: (or a (not a))                "raa" ([1 7])
23      -------------------------------------
```

Codebeispiel 27: Tertium Non Datur

```
1  => (proof '(not (and a b)) '(or (not a) (not b)))
2  => (step-b "raa" 3)
3  => (step-b "not-e" 4 1)
4  => (choose-option 4 2)
5  => (step-b "and-i" 4)
6  => (step-b "raa" 4)
7  => (step-b "raa" 8)
8  => (step-f "or-i2" 3)
9  => (rename-var 'V1 '(not a))
10 => (step-f "not-e" 4 2)
11 => (step-f "or-i1" 6)
12 => (rename-var 'V2 '(not b))
13 => (step-f "not-e" 2 7)
14     ------------------------------------------------
15  1: (not (and a b))                    premise
16      ----------------------------------------------
17  2:  | (not (or (not a) (not b)))      assumption
18      | ----------------------------------------------
19  3:  | | (not b)                       assumption
20  4:  | | (or (not a) (not b))          "or-i2" (3)
21  5:  | | contradiction                 "not-e" (2 4)
22      | ----------------------------------------------
23      | ----------------------------------------------
24  6:  | | (not a)                       assumption
25  7:  | | (or (not a) (not b))          "or-i1" (6)
26  8:  | | contradiction                 "not-e" (2 7)
27      | ----------------------------------------------
28  9:  | b                               "raa" ([3 5])
29 10:  | a                               "raa" ([6 8])
30 11:  | (and a b)                       "and-i" (10 9)
31 12:  | contradiction                   "not-e" (1 11)
32      ----------------------------------------------
33 13: (or (not a) (not b))               "raa" ([2 12])
34    ------------------------------------------------
```

Codebeispiel 28: De Morgansches Gesetz 1

```
1  => (proof '(not (or a b)) '(and (not a) (not b)))
2  => (step-b "and-i" 3)
3  => (step-b "not-i" 3)
4  => (step-b "not-i" 7)
5  => (step-f "or-i2" 2)
6  => (rename-var 'V1 'a)
7  => (step-f "not-e" 1 3)
8  => (step-f "or-i1" 5)
9  => (rename-var 'V2 'b)
10 => (step-f "not-e" 1 6)
11     ------------------------------------------
12  1: (not (or a b))                premise
13      ----------------------------------------
14  2:  | b                          assumption
15  3:  | (or a b)                  "or-i2" (2)
16  4:  | contradiction             "not-e" (1 3)
17      ----------------------------------------
18      ----------------------------------------
19  5:  | a                          assumption
20  6:  | (or a b)                  "or-i1" (5)
21  7:  | contradiction             "not-e" (1 6)
22      ----------------------------------------
23  8: (not b)                       "not-i" ([2 4])
24  9: (not a)                       "not-i" ([5 7])
25 10: (and (not a) (not b))         "and-i" (8 9)
26     ------------------------------------------
```

Codebeispiel 29: De Morgansches Gesetz 2

```
=> (proof '(impl a b) '(impl (not b) (not a)))
=> (step-b "impl-i" 3)
=> (step-f "modus-tollens" 1 2)
   ------------------------------------------------
1: (impl a b)                          premise
    -----------------------------------------------
2:  | (not b)                          assumption
3:  | (not a)                          "modus-tollens" (1 2)
    -----------------------------------------------
4: (impl (not b) (not a))              "impl-i" ([2 3])
   ------------------------------------------------
```

Codebeispiel 30: Kontraposition - Das Theorem „Modus Tollens“ (siehe Codebeispiel 26) muss zunächst mit der Funktion **export-theorem** exportiert werden.

Prädikatenlogik

```
=> (proof '(not (exists [x] (P x)))
          '(forall [x] (not (P x))))
=> (step-b "forall-i" 3)
=> (step-b "not-i" 4)
=> (step-b "not-e" 5 1)
=> (choose-option 5 2)
=> (step-b "exists-i" 5 2)
   ------------------------------------------------
1: (not (exists [x] (P x)))          premise
    -----------------------------------------------
2:  | (actual V1)                    assumption
    | -----------------------------------------------
3:  | | (P V1)                    assumption
4:  | | (exists [x] (P x))           "exists-i" (2 3)
5:  | | contradiction                "not-e" (1 4)
    | -----------------------------------------------
6:  | (not (P V1))                   "not-i" ([3 5])
    -----------------------------------------------
7: (forall [x] (not (P x)))          "forall-i" ([2 6])
   ------------------------------------------------
```

Codebeispiel 31: Prädikatenlogik - Beispiel 1

```
=> (proof '(exists [x] (or (P x) (Q x)))
          '(or (exists [x] (P x)) (exists [x] (Q x))))
=> (step-b "exists-e" 3 1)
=> (step-b "or-e" 5 3)
=> (choose-option 5 1)
=> (step-b "or-i2" 6)
=> (step-b "exists-i" 6 2)
=> (step-b "or-i1" 9)
=> (step-b "exists-i" 9 2)
    ------------------------------------------------
 1: (exists [x] (or (P x) (Q x)))           premise
     ----------------------------------------------
 2:  | (actual V1)                          assumption
 3:  | (or (P V1) (Q V1))                   assumption
     | ---------------------------------------------
 4:  | | (Q V1)                             assumption
 5:  | | (exists [x] (Q x))                 "exists-i" (2 4)
 6:  | | (or (exists [x] (P x)) (exists [x] (Q x)))"or-i2" (5)
     | ---------------------------------------------
     | ---------------------------------------------
 7:  | | (P V1)                             assumption
 8:  | | (exists [x] (P x))                 "exists-i" (2 7)
 9:  | | (or (exists [x] (P x)) (exists [x] (Q x)))"or-i1" (8)
     | ---------------------------------------------
10:  | (or (exists [x] (P x)) (exists [x] (Q x)))"or-e"(3 [4 6] [7 9])
     ----------------------------------------------
11: (or (exists [x] (P x)) (exists [x] (Q x)))"exists-e" (1 [2 10])
    ------------------------------------------------
```

Codebeispiel 32: Prädikatenlogik - Beispiel 2

Temporale Logik

```
1  => (proof '(at i (not (sometime a)))
2            '(at i (always (not a))))
3  => (step-b "always-i" 3)
4  => (rename-var 'V1 'j)
5  => (step-b "not-i" 4)
6  => (rename-var 'V2 'i)
7  => (rename-var 'V3 '(sometime a))
8  => (step-b "and-i" 5)
9  => (step-f "sometime-i" 2 3)
10    ------------------------------------------------
11 1: (at i (not (sometime a)))                premise
12     ----------------------------------------------
13 2:  | (<= i j)                              assumption
14     | ------------------------------------------
15 3:  | | (at j a)                            assumption
16 4:  | | (at i (sometime a))                 "sometime-i" (2 3)
17 5:  | | (at i (and (sometime a) (not (sometime a))))"and-i" (1 4)
18     | ------------------------------------------
19 6:  | (at j (not a))                        "not-i" ([3 5])
20     ----------------------------------------------
21 7: (at i (always (not a)))                  "always-i" ([2 6])
22    ------------------------------------------------
```

Codebeispiel 33: Lineare temporale Logik - Beispiel 1

```
1  => (proof '(at i (asap (sometime a)))
2            '(at i (sometime a)))
3  => (step-f "asap-seriality")
4  => (rename-var 'V1 'i)
5  => (rename-var 'V2 'j)
6  => (step-f "asap-e" 1)
7  => (rename-var 'V3 'j)
8  => (step-f "asap/<=" 2)
9  => (step-f "sometime-e" 3)
10 => (rename-var 'V4 'k)
11 => (step-f "transitivity" 4 5)
12 => (step-f "sometime-i" 6 7)
13    ------------------------------------------------
14 1: (at i (asap (sometime a)))          premise
15 2: (next i j)                          "asap-seriality"
16 3: (at j (sometime a))                 "asap-e" (1)
17 4: (<= i j)                            "asap/<=" (2)
18 5: (<= j k)                            "sometime-e" (3)
19 6: (at k a)                            "sometime-e" (3)
20 7: (<= i k)                            "transitivity" (4 5)
21 8: (at i (sometime a))                 "sometime-i" (6 7)
22    ------------------------------------------------
```

Codebeispiel 34: Lineare temporale Logik - Beispiel 2

4.7 Erweiterung

Eine der Ideen des Projekts war es, die Implementierung möglichst offen zu halten, um auch anderen Entwicklern die Möglichkeit zu geben, das Programm nach ihrem Bedarf zu erweitern. So wurde beispielsweise das Hinzufügen von neuen Regeln oder Theoremen bewusst sehr einfach gehalten. Dennoch gibt es Sonderfälle oder bestimmte Logiken, für welche weitere Anpassungen und Erweiterungen implementiert werden müssen.

Um es anderen Entwicklern zu erleichtern, weitere Logiken zu implementieren, wurden wichtige Stellen mit dem Schlüsselwort „NEW LOGIC“ und einem erklärenden Kommentar versehen. Diese so markierten Stellen im Code behandeln Sonderfälle und sind daher der erste Einstiegspunkt für Entwickler, sollte sich die gewünschte Logik nicht nur durch neue Regeln ausdrücken lassen. So muss nicht erst der komplette

Code durchforstet werden, um eigene Sonderfälle zu implementieren[18]. Im Folgenden sollen die wichtigsten dieser Stellen kurz erläutert werden.

4.7.1 Schlüsselwörter

Schlüsselwörter sind jene, die innerhalb der Regeldefinitionen nicht wie eine Variable behandelt werden. Sie können dementsprechend bei der späteren Anwendung einer Regel nicht durch beliebige Elemente ersetzt werden, sondern müssen ähnlich wie die logischen Operatoren, den vorgegebenen Wert besitzen (siehe Codebeispiel 35). Voreingestellt sind die Wörter *true, false, truth* und *contradiction*. Das entsprechende Clojure Set befindet sich im Namensraum „rules“ und kann beliebig erweitert werden.

4.7.2 Funktionen für Sonderformen

Manche Voraussetzungen oder Schlussfolgerungen von Regeln lassen sich nur schwer in selbigen darstellen. Für diese Sonderformen müssen daher Funktionen geschrieben werden, um sie zwischen der Beweisstruktur und der Regelanwendung zu übersetzen. Beispiele sind etwa Unterbeweise (infer) und Substitutionen (substitution).

Wurden entsprechenden Funktionen implementiert, so müssen diese noch bei der Auswertung von Regelergebnissen berücksichtigt werden. Dazu wird die Funktion **eval-body** im Namensraum „deduction“ entsprechend erweitert.

[18]Natürlich lässt sich nicht mit 100-prozentiger Sicherheit sagen, dass nicht doch Anpassungen an anderer Stelle gemacht werden müssen. Es wird jedoch davon ausgegangen, dass das Grundgerüst nicht verändert werden muss.

```
{:name „and-e1"
 :given [(and a b)]
 :conclusion [a]
 :forwards true}

=> (apply-rule „and-e1" true '[(and Hans Peter])
(Hans)
=> (apply-rule „and-e1" true '[(and 1 2)])
(1)
```

(a) Regeln ohne voreingestellte Schlüsselworte

```
{:name "trivial-3"
 :given      [(and a true)]
 :conclusion [a]
 :forwards   true}

=> (apply-rule „trivial-3" true '[(and Hans Peter)])
()
=> (apply-rule „trivial-3" true '[(and Hans true)])
(Hans)
```

(b) Regeln mit dem voreingestellten Schlüsselwort *true*

Codebeispiel 35: Vergleich einer Regel ohne voreingestellte Schlüsselworte ($\wedge$ Elimination 1) (a) und einer Regel mit einem solchen (triviales Theorem) (b). Man sieht, dass die als Schlüsselwort gekennzeichneten Bezeichner auch nur als solche akzeptiert werden. Dagegen lassen sich für die Variablen a und b beliebige Werte einsetzen.

```
1  (defn substitution
2    [formula old new]
3    (cond
4      (not (list? formula))
5      (throw (Exception.
6        (str "The argument \"" formula "\" is not a list and therefore
7         can't be substituted." "Maybe you have to provide optional
8         arguments for the step you trying to accomplish.")))
9
10     (contains? (set (flatten formula)) new)
11     (throw (Exception.
12       (str "Substitution failed. The identifier \"" new "\" is
13        already used inside the formula \"" formula "\"")))
14
15     :else (clojure.walk/postwalk-replace {old new} formula)))
```

Codebeispiel 36: Die Funktion **substitution** ersetzt jedes Vorkommen von *old* in der Formel *formula* mit dem Argument *new*. Bei Fehlern innerhalb der Argumente wird eine entsprechende Fehlermeldung ausgegeben

Kapitel 5

Fazit

Mit den hier beschriebenen Methoden sind die Kernziele des Projekts erreicht worden. Das Programm ist lauffähig und in der Lage, Hypothesen der Aussagen-, Prädikaten- und linearen temporalen Logik korrekt zu beweisen. Ein Nutzer kann mit wenig Aufwand seine eigenen Regeln definieren und bei Bedarf eigene Sonderfälle für neue Logiken hinzufügen. Der Quellcode ist öffentlich zugänglich[19] und großzügig kommentiert.

Für die Schnittstelle wurde die Clojure REPL ausgewählt. Durch einen extra geschriebenen Namensraum bleibt die Arbeit auch ohne grafische Bedienoberfläche immer übersichtlich und lesbar. Die so angepasste Schnittstelle kann jederzeit und einfach durch eine entsprechende grafische Lösung ersetzt werden. Mögliche Bedienkonzepte zeigen sich bei den verwandten Arbeiten (siehe Kapitel 2).

5.1 Ausblick

Ein Punkt, der noch ausgebaut werden sollte, sind die „zusätzlichen Bedingungen“ für Regeln. Momentan sind diese nur für den Sonderfall der = Elimination getestet und umgesetzt worden (siehe Kapitel 4.3.1.1). Aber gerade für die natürliche Deduktion der linearen temporalen Logik gibt es noch einige Zusatzbedingungen zu Regeln, die im momentanen Stand nicht abgebildet werden können.

So darf beispielsweise der Zeitpunkt aus dem Ergebnis einer $\bigcirc$ Elimination nicht als Vorbedingung für eine $\Box$ Einführung genutzt werden (siehe [BGS07] S. 6). Oftmals sind diese Einschränkungen kein Problem und können durch etwas Achtsamkeit und logisches Denken des Nutzers ausgeglichen werden. Um solche Zusatzbedingungen aber systemseitig abbilden zu können (auch im Hinblick auf zukünftige Logiken) muss das für = Elimination genutzte System entsprechend ausgebaut werden.

[19] `https://github.com/TheBlob42/deduction-playground`

Darüber hinaus wäre sicher die Vervollständigung der natürlichen Deduktion für die temporale Logik mit der „Computation Tree Logic“ (CTL) eine naheliegende Erweiterung. Eine entsprechende Arbeit von Bolotov, Grigoriev und Vasilyi zu diesem Thema liegt vor [BGS06]. Dazu müsste evaluiert werden, ob die formulierten Regeln bereits umgesetzt werden können oder ob es noch Sonderfälle gibt, welche das Programm nicht abdecken kann.

Abbildungsverzeichnis

Codeverzeichnis

Referenzen

[BBGS06] Bolotov, Alexander ; Basukoski, Artie ; Grigoriev, Oleg ; Shangin, Vasilyi: Natural Deduction Calculus for Linear-Time Temporal Logic. In: Fisher, Michael (Hrsg.) ; Hoek, Wiebe van d. (Hrsg.) ; Konev, Boris (Hrsg.) ; Lisitsa, Alexei (Hrsg.): *Logics in Artificial Intelligence* Bd. 4160. Springer Berlin Heidelberg, 2006, S. 56–68

[BGS06] Bolotov, Alexander ; Grigoriev, Oleg ; Shangin, Vasilyi: Natural deduction calculus for computation tree logic. In: *In IEEE John Vincent Atanasoff Symposium on Modern Computing*, 2006, S. 175–183

[BGS07] Bolotov, Alexander ; Grigoriev, Oleg ; Shangin, Vasilyi: A Simpler Formulation of Natural Deduction Calculus for Linear-Time Temporal Logic. In: *Proceedings of the 3rd Indian International Conference on Artificial Intelligence*, 2007, S. 1253–1266

[Bor05a] Bornat, Richard: *Natural Deduction Proof and Disproof in Jape*. `https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/jape.org/MANUALS/natural_deduction_manual.pdf`, 2005. – Abrufdatum: 05.11.2015

[Bor05b] Bornat, Richard: *Proof and Disproof in Formal Logic*. Oxford University Press, 2005

[Bor07] Bornat, Richard: *Roll your own Jape logic, Encoding logics for the Jape proof calculator*. `https://www.cs.ox.ac.uk/people/bernard.sufrin/personal/jape.org/MANUALS/roll_your_own.pdf`, 2007. – Abrufdatum: 05.11.2015

[Gen35] Gentzen, Gerhard: Untersuchungen über das logische Schließen. In: *Mathematische Zeitschrift* 39 (1935), S. 176–210, 405–431

[GSS11] Gasquet, Olivier ; Schwarzentruber, François ; Strecker, Martin: Panda: A Proof Assistant in Natural Deduction for All. A Gentzen Style Proof Assistant for Undergraduate Students. In: Blackburn, Patrick (Hrsg.) ; Ditmarsch, Hans van (Hrsg.) ; Manzano, María (Hrsg.) ; Soler-Toscano, Fernando (Hrsg.): *Tools for Teaching Logic* Bd. 6680. Springer Berlin Heidelberg, 2011, S. 85–92

[HHI+01] Halpern, Joseph Y. ; Harper, Robert ; Immerman, Neil ; Kolaitis, Phokion G. ; Vardi, Moshe Y.: On the Unusual Effectiveness of Logic in Computer Science. In: *The Bulletin of Symbolic Logic* 7 (2001), Nr. 2

[HKRW10] Hendriks, Maxim ; Kaliszyk, Cezary ; Raamsdonk, Femke van ; Wiedijk, Freek: Teaching logic using a state-of-the-art proof assistant. In: *Acta Didactica Napocensia* 3 (2010), Nr. 2, S. 35–48

[HR04] Huth, Michael ; Ryan, Mark: *Logic in Computer Science*. Cambridge University Press, 2004

[Jaś34] Jaśkowski, Stanisław: On the Rules of Suppositions in Formal Logic. In: *Studia logica* 1 (1934), S. 5–32

[Kir14] Kirsten, Daniel: *Natürliche Deduktion in Clojure*, Technische Hochschule Mittelhessen, Masterarbeit, 2014

[Kle] Klement, Kevin C.: *Propositional Logic*. `http://www.iep.utm.edu/prop-log/`, . – Internet Encyclopedia of Philosophy, Abrufdatum: 10.09.2015

[KRW+] Kaliszyk, Cezary ; Raamsdonk, Femke V. ; Wiedijk, Freek ; Hendriks, Maxim ; Vrijer, Roel D.: *Deduction using the ProofWeb system*

[Pel01] Pelletier, Francis J.: A History of Natural Deduction and Elementary Logic Textbooks. In: Woods, John (Hrsg.) ; Brown, Bryson (Hrsg.): *Logical consequence. Rival approaches. Proceedings of the 1999 Conference of the Society of Exact Philosophy*, Hermes Science, 2001, S. 105–138

[Val14] Vallerú, Jordi: Logic and Computers. A Sketch of Their Symbiotic Relationships. In: *Kairos. Revista de Filosofia & Ciência* 9 (2014), S. 45–72