Logik, Komplexität und Berechenbarkeit

(Entwurf)

WS 08/09

Sven Eric Panitz

FH Wiesbaden

Version 3. September 2009

Dieses Skript entsteht mit heißer Nadel parallel zur laufenden Vorlesung *Logik, Komplexität und Berechenbarkeit* des Master Studiengangs Informatik an der FH-Wiesbaden. Es wird entsprechend oft neue Versionen geben und entsprechend viele Fehler geben, über die ich bitte stillschweigend hinwegzusehen, oder mich dezent drauf aufmerksam zu machen.

Der Quelltext dieses Skripts ist eine XML-Datei, die durch eine XQuery in eine LaTeX-Datei transformiert und für die schließlich eine pdf-Datei und eine postscript-Datei erzeugt wird. Beispielprogramme werden direkt aus dem Skriptquelltext extrahiert und sind auf der Webseite herunterzuladen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Viele Studenten haben sich angewöhnt, sich auf Klausuren und Prüfungen so vorzubereiten, dass sie mit wenig Verständnis für den eigentlichen Stoff, die Prüfung bestehen können. In mathematischen Vorlesungen liegt die Hoffnung dabei in den sogenannten Rechenaufgaben. In OR-Verfahren kann dieses glücken, wenn man einige Optimierungsverfahren anwenden kann, ohne wirklich zu verstehen, was in diesen Verfahren genau errechnet wird und im wiefern dort eine Optimierungsaufgabe gelöst wird. Leider gibt es immer wieder Prüfer, die diesem Treiben ein Ende bereiten. Das fängt an mit gefürchteten Textaufgaben bereits in der Mittelstufe der Schule und kulminiert in Aufgaben, in denen selbstständig Beweise zu führen sind.

Den Studenten dieser Vorlesung ist ein solches rein auf die Anwendung mechanischer Rechenaufgaben basierendes Lösen natürlich fremd. Deshalb mag es für sie befremdlich sein, wenn am Anfang dieses Skripts ein Verfahren steht, mit dem Aufgaben gelöst werden können.

Was ist zu tun, wenn auf einem Übungsblatt nur chinesische Zeichen stehen?

Betrachten Sie die Zeichen einer Aufgabe, die in einer Zeile stehen. Vor manchen Zeichen steht das Symbol ¬.
Suchen Sie zwei Zeilen, die ein chinesisches Zeichen x gemeinsam haben, so dass in einer Zeile vor diesem Zeichen das Symbol ¬ steht und in der anderen Zeile nicht.
Schreiben Sie dann eine neue Zeile, in der Sie alle Zeichen aus den zwei gewählten Zeilen kopieren, nur das gemeinsame Zeichen x bzw. $\neg x$ ist nicht zu kopieren.
Taucht ein weiteres Zeichen in der resultierende Zeile einmal mit vorangestellten Symbol ¬ und einmal ohne auf, so kann dieses Zeichen inklusiven dem davorstehenden Symbol ¬ gelöscht werden. Tritt ansonsten ein Zeichen doppelt auf, so kann das doppelte Auftreten gelöscht werden.
Fahren Sie so lange rekursiv mit den Zeilen fort, bis Sie eine leere Zeile erzeugt haben. Markieren Sie diese mit dem Symbol □.

Beispiel 1.0.1 *In der Aufgabe sind sieben Zeilen gegeben:*

$$立春雨 \tag{1.1}$$
$$\neg立\neg春 \tag{1.2}$$
$$\neg立\neg雨 \tag{1.3}$$
$$\neg春\neg雨 \tag{1.4}$$
$$春\neg立 \tag{1.5}$$
$$立\neg雨 \tag{1.6}$$
$$\neg春 \tag{1.7}$$

Es lassen sich die folgenden Zeilen mit den Regeln ableiten:

$$\neg立 \quad , \textit{aus 1.2 mit 1.5} \tag{1.8}$$
$$\neg雨 \quad , \textit{aus 1.3 mit 1.6} \tag{1.9}$$
$$春雨 \quad , \textit{aus 1.1 mit 1.8} \tag{1.10}$$
$$春 \quad , \textit{aus 1.9 mit 1.10} \tag{1.11}$$
$$\square \quad , \textit{aus 1.7 mit 1.11} \tag{1.12}$$

Aufgabe 1 Versuchen Sie folgende Probleme zu lösen:

a)

$$\{\neg立, 蛰 \tag{1.13}$$
$$\neg春, 雨 \tag{1.14}$$
$$\neg水, 惊 \tag{1.15}$$
$$\neg雨, \neg蛰 \tag{1.16}$$
$$春, \neg惊 \tag{1.17}$$
$$水 \tag{1.18}$$
$$立 \tag{1.19}$$

Literatur

Es gibt eine Reihe von Büchern, die in die formale Logik auf unterschiedliche Weise einführen. Ein sehr schönes Skript findet sich von Peter Schmitt im Internet[Sch08]. Es ist wesentlich umfangreicher als wir es in dieser Vorlesung behandeln können und kann sicher als Anregung zum Weiterstudium dienen. Dort findet sich auch weitere Literatur angegeben.

Zum Kapitel über Berechenbarkeit mit Hilfe des Lambda-Kalküls sei das kleine Skript von Tobias Nipkow [Nip98] an der TU München empfohlen.

Programme dieses Skripts

In diesem Skript befinden sich mehrere Programme, in denen einige logische Kalküle umgesetzt sind. Die Programme sind in der Programmierspache *Haskell* (`http://www.haskell.org/`) geschrieben.

Die Wahl auf Haskell fiel, um zum einen geistig ein wenig rege zu bleiben, zum anderen, weil sich die Syntax von Haskell stark an den formalen Notation der Logik anlehnt und die Algorithmen verschiedener Kalküle sich elegant in Haskell notieren lassen.

Kapitel 2

Logik

Bengt: Hallo Bert. Was sitzen Sie da so melancholisch wie ein Forsthaus im Walde?

Bert: Ach, wissen Sie, es geht um meinen Freund Ernie.

Bengt: Ich dachte immer, Sie verstehen sich recht gut miteinander.

Bert: Die Sache ist nur, dass Ernie angefangen hat, Gedichte zu schreiben.

Bengt: Nun, das ist doch ein recht leises und angenehmes Hobby.

Bert: Nur Ernie hat mir seine Gedichte zum Lesen gegeben und nach meiner Meinung gefragt.

Bengt: Und.

Bert: Naja, sie sind ziemlich langweilig aber das möchte ich nicht so direkt sagen, schließlich ist er ja mein Freund.

Bengt: Sind die wirklich so schlimm?

Bert: Ja, stellen Sie sich vor, alle seine Gedichte sind über Seifenblasen.

Bengt: Aber das kann doch ganz nett sein.

Bert: Kann, muss aber nicht. Warten sie:

> Oh Seifenblase Seifenblase,
> es spiegelt sich ein kleiner Hase
> Du leuchtest voller Farben bunt
> da spiegelt sich ein kleiner Hund

Bengt: Oh ja, das ist tatsächlich etwas langweilig.

Bert: Aber das muss man ihm ja nicht so direkt sagen.

Bengt: Dann drücken Sie das ihm gegenüber doch etwas verklausuliert aus, etwa so:

- Wirklich interessante Gedichte sind nicht unbeliebt bei Leuten mit guten Geschmack.
- Keine moderne Poesie ist frei von einer gewissen Schwülstigkeit.
- Alle Deine Gedichte sind nun mal über Seifenblasen.

- Schwülstige Poesie ist nicht beliebt bei Leuten mit einen gewissen Geschmack.
- Kein älteres Gedicht beschäftigt sich mit Seifenblasen.

BERT: Das klingt alles sehr allgemein dahergesagt. Ich glaube, wenn ich Ernie es so sage, würde er nicht eingeschnappt sein.

BENGT: Hoffen wir nur das Ernie nicht zuviel Lewis Carrol liest.

Eine der Kernaufgaben eines intelligenten Agenten besteht darin, aus vorhandenen Basiswissen neues Wissen abzuleiten; aus bestehenden Fakten über die Umgebung, weitere Schlüsse zu ziehen oder Pläne zu kreieren. Die Frage, wie der Vorgang des folgerichtigen Schließens genauer präzisiert werden kann, stellten sich schon die alten Griechen. Aristoteles stellte hierzu ein formales System von Schlußregeln auf, die immer zu folgerichtigen Schlüssen führen. Damit begründete Aristoteles die formale Logik.

Je nach den technischen Möglichkeiten, interessertierte die Frage, ob sich die formalen Schlußsysteme der Logik, maschinell zum logischen Schließen ausführen lassen. So verwundert es nicht, dass die KI sehr früh die formale Logik als eines der Hauptwerkzeuge anektierte und entscheidend für seine Zwecke weiterentwickelte.

Charles Lutwidge Dodgson (∗27. Januar 1832 in Daresbury; †14. Januar 1898 in Guildford), besser bekannt unter seinem Künstlernamen Lewis Carroll, war ein britischer Schriftsteller, Mathematiker und Fotograf.
Er ist der Autor von *Alice im Wunderland*, *Alice hinter den Spiegeln* (auch bekannt als *Alice im Spiegelland*) und *Die Jagd nach dem Schnark*. Mit seiner Befähigung für Wortspiel, Logik und Fantasie schaffte er es, weite Leserkreise – von den Naivsten zu den Gebildetsten – zu fesseln. Seine Werke sind bis heute populär geblieben und haben nicht nur die Kinderliteratur, sondern auch Schriftsteller wie James Joyce oder Douglas R. Hofstadter beeinflusst.

Abbildung 2.1: **Wikipediaeintrag** (24. April 2006): Lewis Carroll

Ein Mathematiker, der sich recht unterhaltsam mit logischen Rätsel beschäftigte, war Lewis Carroll, der unzählige von logischen Spielereien und verklausulierten Rätsel aufgestellt hat. Auch die diesem Kapitel vorangestellte in fünf Sätzen verklausulierte Aussage über bestimmte Gedichte stammt von Lewis Carroll.

2.1 Aussagenlogik

Das grundlegendste logischen System, von dem komplexere formale Systeme abgeleitet werden können, ist die Aussagenlogik, die einem Programmierer in jeder Programmiersprache als der Datentyp `boolean` begegnet. Wir werden in diesem Kapitel die Aussagenlogik auf unterschiedliche Weise betrachten. Dabei werden viele Aspekte eines formalen Systems exemplarisch vorgeführt.

Ein formales System besteht in der Regel aus einer Sprache, in der bestimmtes Wissen ausgedrückt werden kann. Diese Sprache ermöglicht es syntaktische Sätze, oder Formeln aufzuschreiben. Zur Unterscheidung zu anderen Sprachen, insbesondere der Sprache, in der wir über die Formelsprache Beweise führen, bezeichnen wir die Formelsprache als *Objektsprache*. Wenn Beweise über Eigenschaften der Objektsprache geführt werden, so werden diese Beweise in einer Metasprache geführt.

Für eine Objektsprache ist eine Semantik zu definieren. Die Semantik wird in der Regel eine mathematische Definition sein, in der jeder Satz der Objektsprache auf bestimmte Elemente mathematischer Mengen abgebildet wird. Die Semantik kann definieren, welche neuen Formeln aus einer Menge von Formeln folgt.

Desweiteren ist für ein formales System ein Kalkül zu definieren. Ein Kalkül besteht aus Rechenregeln. Diese Regeln arbeiten auf den syntaktischen Objekten der Objektsprache und geben an, wie durch symbolisches Rechnen neue Formeln aus einer Menge von Formeln abzuleiten ist.

George Boole (∗2. November 1815 in Lincoln, England; †8. Dezember 1864 in Ballintemple, Irland) war ein englischer Mathematiker (Autodidakt) und Philosoph.
Ursprünglich als Lehrer tätig, wurde er auf Grund seiner wissenschaftlichen Arbeiten 1848 Mathematikprofessor am Queens College in Cork (Irland).
Boole entwickelte 1847 den ersten algebraischen Logikkalkül für Klassen und Aussagen im modernen Sinn. Damit begründete er die moderne mathematische Logik. Booles Kalkül wird gelegentlich noch im Rahmen der traditionellen Begriffslogik interpretiert, obwohl Boole bereits in *The Mathematical Analysis of Logic* ausschließlich von Klassen spricht.
Boole arbeitete in einem kommutativen Ring mit Multiplikation und Addition, die den logischen Verknüpfungen *und* und *entweder...oder* entsprechen; in diesem booleschen Ring gilt zusätzlich zu den bekannten Rechenregeln die Regel aa=a, in Worten (a und a)=a.
In der Boole-Schule wurde die zum booleschen Ring gleichwertige boolesche Algebra entwickelt, die mit *und* und *oder* arbeitet. Sie ist heute in der Aussagenlogik und Mengenlehre bekannter und verbreiteter als der originale, rechnerisch elegantere boolesche Ring.
George Boole ist der Vater von Ethel Lilian Voynich.
Auf George Boole sind die booleschen Variablen zurückzuführen, die zu den Grundlagen des Programmierens in der Informatik zählen.

Abbildung 2.2: **Wikipediaeintrag** (24. April 2006): George Boole

Zwischen einem Kalkül und der Semantik einer Sprache sind zwei fundamentale Eigenschaften zu beweisen. Zum einem soll der Kalkül nur korrekte Ableitungen machen. Jeder Satz, der sich durch die Operationen eines Kalküls aus einer Formelmenge ableiten läßt, soll auch semantisch aus dieser Formelmenge folgern. Ist diese Eigenschaft erfüllt, so ist der Kalkül korrekt. Nun ist es natürlich recht leicht einen korrekten Kalkül für ein beliebiges formales System zu schreiben. Hierzu nehme man einfach den Kalkül, der überhaupt keine Sätze ableiten kann. Ein Kalkül sollte aber möglichst viele Schlußfolgerungen beweisen können, am besten sollte er alles, was semantisch aus einer Formelmenge folgt, aus dieser auch ableiten können. Wenn dieses der Fall ist, so bezeichnet man den Kalkül als vollständig. Abbildung 2.3 gibt einen schematischen Überblick über die Bestandteile eines formales Systems.

In dieser Weise stellt auch jede Programmiersprache ein formales System dar. Der Kalkül einer Programmiersprache ist ihr Ausführungsmodell. Die Semantik einer Programmiersprache ist in den seltesten Fällen eigens definiert. Daher fällt es auch schwer Korrektheitsbeweise über Programme zu führen. Statt von Semantik und Kalkül spricht man bei Programmiersprachen auch von *denotionaler* und *operationaler* Semantik.

2.1.1 Syntax

Die Aussagenlogik ist eine Sprache in der aussagenlogische Formeln ausgedrückt werden können. Informatiker sind es gewohnt Sprachen über einer kontextfreie Grammatik auszudrücken. Wir werden im Folgenden eine eher mathematische Definition der Sprache der aussagenlogischen Formeln benutzen. Wir folgen dabei einem Vorlesungsskript der Universität Karlsruhe[Spe86].

Definition 2.1.1 (Syntax der Aussagenlogik)
Die Menge $\{(,), \neg, \rightarrow\}$ *sei die Menge der Sondersymbole.*

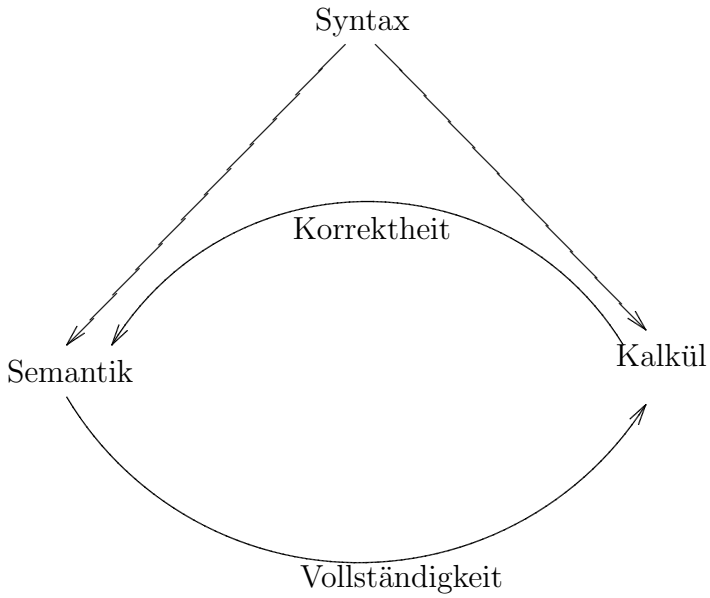

Abbildung 2.3: Syntax, Semantik, Kalkül

Für eine abzählbare Menge $S = \{A, B, C \dots\}$ von aussagenlogischen Variablen sei die Sprache For_S definiert als die kleineste Menge mit:

- $S \subset For_S$
- *wenn* $x \in For_S$ *dann auch* $\neg x \in For_S$
- *wenn* $x \in For_S$ *und* $y \in For_S$ *dann auch* $(x \rightarrow y) \in For_S$

Die Menge S wird als Signatur bezeichnet. Formeln x mit $x \in S$ werden als Atome bezeichnet. Die Menge $S \cup \{\neg x | x \in S\}$ wird als die Menge der Literale bezeichnet.

Wie man sieht, halten wir unsere Sprache extrem klein. Wir verzichten auf die üblichen aussagenlogischen Junktoren für *oder* und *und*, $\vee$ und $\wedge$ und begnügen uns mit dem Symbol $\rightarrow$ für die logischen Implikation und $\neg$ für die Negation. $\vee$ und $\wedge$ werden wir lediglich als Abkürzungen für komplexere Ausdrücke benutzen:

Definition 2.1.2 ($\vee$ und $\wedge$)
Ein Ausdruck der Form $(A \vee B)$, $A, B \in For_S$ stehe für: $(\neg A \rightarrow B)$.
Ein Ausdruck der Form $(A \wedge B)$, $A, B \in For_S$ stehe für: $\neg(\neg A \vee \neg B)$.
Ein Ausdruck der Form $(A \leftrightarrow B)$, $A, B \in For_S$ stehe für: $((A \rightarrow B) \wedge (B \rightarrow A))$.

Unsere aussagenlogischen Formeln sind mit obiger Definition vollständig geklammert. Wie wir es bei arithmetischen Ausdrücken gewohnt sind, bei denen die Punktrechnungsoperatoren stärker binden als Strichrechnungsoperatoren, und somit Klammern vermieden werden können, sei ein Operatorpriorität auf den aussagenlogischen Operatoren in folgender Reihenfolge gegeben: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Aufgabe 2 Die Sprache For_S ist in einen für einen Informatiker, speziell für einen Compilerbauer ungewohnten Form definiert. Es ist eine deklarative Beschreibung als Menge mit bestimmten Eigenschaften. Informatiker beschreiben Sprachen gerne mit einer kontextfreien Grammatik, um damit auch in der Lage zu sein, Parser zum verarbeiten der Sprache zu schreiben.

Entwerfen Sie eine Grammatik, für die Sprache For_S.

Strukturelle Induktion

Die Sprache For_S ist rekursiv definiert. Soll ein Beweis über alle Elemente von For_S geführt werden, oder eine Eigenschaft für alle Elemente von For_S definiert werden, so bietet sich die sogenannte strukturelle Induktion an. Dieses ist eine Form der von den natürlichen Zahlen bekannten vollständigen Induktion, die über den strukturellen Aufbau der Elemente geht. Als Induktionsanfang (bei natürlichen Zahlen die Zahl 0), wird eine Eigenschaft für alle Elemente der Menge S bewiesen bzw. definiert. Beim Induktionsschritt wird dann unter der Annahme, dass die Eigenschaft bereits für bestimmte Formeln bewiesen bzw. definiert wurde, dieses für die aus den Teilformeln zusammengesetzte Formel getan. Dieses entspricht dem Schritt von n auf $n+1$ bei der vollständigen Induktion auf den natürlichen Zahlen.

Beispiel 2.1.3 *Beweis der relativ trivialen Aussage: Jedes Element F von For_S enthält mindestens ein Symbol aus der Menge S:*

- *Anfang: wenn $F \in S$ dann ist die Behauptung trivial.*
- *Annahme: F und G enthalten ein Symbol aus S. Dann enthalten auch $\neg F$ und $(F \to G)$ ein Symbol aus S.*

Beispiel 2.1.4 *Definition für die Symbolanzahl einer Formel F aus For_S. $no(F)$ für $F \in For_S$ sei wie folgt definiert:*

- *Anfang: wenn $F \in S$ dann sei $no(F) = 1$.*
- *Annahme: wenn $no(F) = n_f$ und $no(G) = n_g$ dann sei $no(\neg F) = n_f$ definiert und $no((F \to G)) = n_f + n_g$ definiert.*

Ein kleiner Datentyp für Aussagenlogische Formeln

Um die eher trockene Materie einer Theorievorlesung mit ein wenig greifbaren zu füllen, soll die Sprache For_S und schließlich auch die Kalküle und auch die Semantik auf dieser implementiert werden. Hierzu benutzen wir die Programmiersprache Haskell mit allen für generische[1] Daten zur Verfügung stehenden Spracherweiterungen des Glasgow Haskell Compilers.

Für diese Kapitel sein ein kleines Modul `PropositionalLogic.hs` entworfen, dass die Sprache der Aussagenlogik implementiert. Hierbei werden ein paar Standardbibliotheken importiert:

[1] Die Generizität, die die Sprache Haskell dabei bietet hat wenig mit dem generischen Typen aus Java zu tun. Javas generische Typen werden in der funktionalen Programmierung als polymorphe Typen bezeichnet.

PropositionalLogic.hs
```
1 {-# OPTIONS -fglasgow-exts #-}
2
3 module PropositionalLogic where
4 import Maybe
5 import List
6 import Data.Generics
7 import ToLatex
```

Datentyp Definition

Als erstes sei ein Datentyp für die Darstellung der Elemente der Sprache For_S definiert. Hierzu sei der Datentyp `Prop` definiert. Rekursive baumartige Strukturen lassen sich in Haskell mit der `data`-Deklaration definieren. Hierzu werden für die verschiedenen Ausprägungen des Typen eigene Konstruktoren definiert. Diese Konstruktoren können dabei auch als binäre Infixkonstruktoren definiert werden.

Die Kernsprache For_S kennt drei Formen von Formeln: die Elemente aus der Menge S, die negierten Formeln, und die Implikationsformeln. Für diese drei Formelarten seien je ein Konstruktor definiert. Der Konstruktor `Atom` für die Elemente aus der Menge S, die wiederum als String dargestellt werden, der Konstruktor `Not` für die Negation einer bestehenden Formel und der binäre Infixoperator `:->` für die Implikation aus zwei Teilformeln:

PropositionalLogic.hs
```
8  data Prop
9   =  Atom String
10    |Not Prop
11    |Prop :-> Prop
```

Zusätzlich soll der Datentyp `Prop` noch in der Lage sein, die abkürzenden Operatoren darzustellen. Für diese seien weitere Infixkonstruktoren definiert. Wie schon beim Konstruktor für die Implikation sollen auch diese im Schriftbild an die Formelnotation erinnern.

PropositionalLogic.hs
```
12      -- erweiterte abkürzende Syntax um Konjunktion,
13      -- Disjunktion und Äquivalenz
14    |Prop :/\ Prop
15    |Prop :\/ Prop
16    |Prop :<-> Prop
```

Die Definition des Datentyps `Prop` in Haskell sei abgeschlossen mit der Bitte an den Compiler bestimmte Eigenschaften für diesen Datentyp standardmäßig zu generieren. Diese betrifft die Gleichheit, eine Umwandlung in Strings, sowie zwei Eigenschaften, die die Nutzung der Generizität erlauben werden:

PropositionalLogic.hs
```
17      deriving (Eq,Show,Typeable,Data)
```

Um zu umfangreiche Klammerungen innerhalb der Formeln zu vermeiden, haben wir für For_S eine Operatorpräzedenz eingeführt. Dieses können wir auch in der Sprache Haskell für die Infixkonstruktoren definieren. Konstruktoren mit einer höheren Infixdeklaration binden dann stärker als welche mit geringerer:

PropositionalLogic.hs
```
18 infixl 8 :/\
19 infixl 6 :\/
20 infixl 4 :->
21 infixl 2 :<->
```

Eine kleine Hilfsfunktion kann testen, ob eine Formel ein Atom ist:

PropositionalLogic.hs
```
22 isAtom (Atom _) = True
23 isAtom _ = False
```

Erste Formelbeispiele

Zeit ein paar erste kleine Formeln in Haskell zu definieren. Hierzu seien Elemente $\{A, B, C, D, E\}$ einer Menge S definiert:

PropositionalLogic.hs
```
24 a = Atom "A"
25 b = Atom "B"
26 c = Atom "C"
27 d = Atom "D"
28 e = Atom "E"
```

Und mit diesen ein paar Beispielformeln definiert:

PropositionalLogic.hs
```
29 ex1 = a
30 ex2 = Not a
31 ex3 = ex1 :\/ ex2
32 ex4 = ex1 :-> ex2
33 ex5 = ex3 :-> b
34 ex6 = Not b :\/ ex5
35 ex7 = a :<-> b
36 ex8 = ex7 :<-> ex7
37 ex9 = ex7 :/\ Not ex7
38 ex10 = a :/\ (b:->c)
39 ex11 = a :/\ (Not a)
40 ex12 = ex8 :<-> ex8
41
42 exs=[ex1,ex2,ex3,ex4,ex5,ex6,ex7,ex8,ex9,ex10,ex11,ex12]
```

Aufgabe 3 Laden Sie das Modul `PropositionalLogic.hs` in einer interaktive Session des Interpreters `ghci` des Glasgow Haskell Compilers. Und lassen Sie sich die Beispielformel einmal durch Aufruf auf der Konsole ausgeben.

Darstellung in LaTeX-Kodierung

Wie beim Lösen der letzten Aufgabe zu sehen war, benutzt Haskell zur textuellen Darstellung des Datentypen `Prop` eine Notation, die exakt den Haskellausdrücken entspricht. Zur Weiterverarbeitung der in Haskell geschriebenen Formeln innerhalb dieses Skripts sollen die Formeln

in LaTeX-Notation umgewandelt werden. Hierzu sei eine Typklasse mit der Funktion zur Umwandlung in einen für LaTeXkodierten String definietr.[2].

PropositionalLogic.hs

```
43
```

Die Funktion `toLatex` wird strukturell über die Struktur der Formel definiert. Hierbei ist schon zu erkennen, wie nah sich in Aufbau die Programmierung in der Sprache Haskell an den Definitionen in der Logik anlehnt.

PropositionalLogic.hs

```
44 instance ToLatex Prop where
45     toLatex (Atom x) = "\\mbox{\\sl "++x++"}"
46     toLatex (Not p) = "\\neg "++toLatex p
47     toLatex (p1 :-> p2)
48      = "("++toLatex p1++" \\rightarrow "++toLatex p2++")"
49     toLatex (p1 :<-> p2)
50       = "("++toLatex p1++" \\leftrightarrow "++toLatex p2++")"
51     toLatex (p1 :/\ p2)
52       = "("++toLatex p1++" \\wedge "++toLatex p2++")"
53     toLatex (p1 :\/ p2)
54       = "("++toLatex p1++" \\vee "++toLatex p2++")"
55
56 instance ToLatex [Prop] where
57   toLatex ps = concat ["$"++toLatex p++"$\n\n"|p<-ps]
58
```

Aufgabe 4 Rufen Sie jetzt einmal in der interaktiven Interpretersession des `ghci` die Funktion `toLatex` für die Beispielformeln auf.

Vereinfachung auf die Kernsprache

In der Sprachdefinition For_S waren die Operatoren für die Konjunktion, Disjunktion und Äquivalenz lediglich Abkürzungen für Formeln, die sich allein mit Negation und Implikation darstellen lassen. Der Datentyp `Prop` kann diese abkürzenden Schreibweisen direkt darstellen. Werden diese drei Operatoren aus den Formeln eliminiert, so können wir uns anschließend lediglich auf die Operatoren der Negation und der Implikation konzentrieren. Deshalb sei eine Haskellfunktion `simplify` definiert, die die drei zusätzlichen Operatoren eleminiert.

PropositionalLogic.hs

```
59 simplify :: Prop -> Prop
60 simplify = everywhere$mkT s
61  where
62    s (p1 :\/ p2)  = Not p1 :-> p2
63    s (p1 :/\ p2)  = Not (s (Not p1 :\/ Not p2))
64    s (p1 :<-> p2) = simplify ((Not p1:\/p2) :/\ (Not p2:\/p1))
65    s x = x
```

Die obige Funktion nutzt die ganze hohe Kunst der Generizität von Haskell. Überall innerhalb eines Formelausdrucks soll die lokale Funktion `s` angewendet werden. Diese ersetzt die drei zu

[2]Die Typklassen aus Haskell entsprechen den Schnittstellen aus Java

eleminierenden Operatoren durch ihre Definition und lässt die übrigen Ausdrücke unberührt. Der Ausdruck `everywhere$mkT` aus der Haskellbibliothek zur Generizität realisiert im Prinzip die strukturelle Induktion. Es soll soll komplett über die Struktur der Formel gegangen werden, um an allen Stellen, an denen es möglich ist, die eigentliche Vereinfachungsfunktion `s` anzuwenden.

Aufgabe 5 Lassen Sie in einer interaktiven Session des `ghci` die Beispielformeln durch Eleminierung der zusätzlichen Operatoren vereinfachen und lassen sich das Ergebnis in LaTeX-Notation ausgeben, um die Vereinfachungen in einen LaTeX-Dokument darzustellen.

Aufgabe 6 **Zusatzaufgabe (schwer):** Die Darstellung der Formeln in LaTeX-Notation ist immer vollständig geklammert. Schreiben Sie die Funktion `toLatex` für den Datentyp `Prop` so neu, dass unter Berücksichtigung der Operatorepräzedenzen möglichst wenig Klammern gesetzt werden.

2.1.2 Semantik

Wir haben im voherigen Abschnitt eine Sprache definiert. Die Ausdrücke der Sprache For_S haben bisher noch keine Bedeutung für uns. Diese definieren wir nun. Hierzu werden Interpretationen für Ausdrücke der Sprache For_S definiert.

Definition 2.1.5 (Interpretation)
Eine Abbildung $I : S \to \{W, F\}$ heißt Interpretation. Eine Interpretation wird auf die folgende Weise zu einer Abbildung $w_I : For_S \to \{W, F\}$ erweitert:

$$
\begin{aligned}
w_I(x) &= I(x) \quad \textit{für } x \in S \\
w_I(\neg x) &= \begin{cases} W & \textit{für } w_I(x) = F \\ F & \textit{sonst} \end{cases} \\
w_I((x \to y)) &= \begin{cases} W & \textit{für } w_I(y) = W \textit{ oder } w_I(x) = F \\ F & \textit{sonst} \end{cases}
\end{aligned}
$$

Eine Interpretation ist also eine Belegung der aussagenlogischen Variablen einer Formel mit den Werten W und F. Für diese Belegung bekommt dann die gesamte Formel einen Wahrheitswert berechnet. Hierzu war es nur notwendig für die beiden Operatoren $\neg$ und $\to$ zu definieren, wie für sie den Wahrheitswert abhängig von der Teilformel lautet.

Es schließen sich ein paar einfache Begriffe an:

Definition 2.1.6 *Sei $A \in For_S$.*

- *A ist eine* Tautologie (allgemeingültig) *gdw. für alle Interpretationen I gilt: $w_I(A) = W$.*
- *A ist* ein Widerspruch (widersprüchlich, unerfüllbar) *gdw. für alle Interpretationen I gilt: $w_I(A) = F$.*
- *A ist* erfüllbar (konsistent) *gdw. es eine Interpretationen I gibt mit: $w_I(A) = W$.*
- *ein* Modell *für eine Formel A ist eine Interpretation I mit $w_I(A) = W$.*

In der Aussagenlogik sind wir in der glücklichen Lage, dass wir in einer Formel nur endlich viele aussagenlogische Variablen haben. Für n aussagenlogische Variablen gibt es 2^n verschiedene Interpretationen. Somit können wir in Form von Wahrheitstafel alle Interpretaionen einer Formel hinschreiben und damit prüfen, ob es eine Tautologie ist. Allerdings ist das mit einem exponentiellen Aufwand verbunden.

Aufgabe 7 Zeigen Sie: $w_I((A \vee B)), w_I((A \wedge B)), w_I((A \rightarrow B)), w_I((A \leftrightarrow B))$ berechnen sich aus $w_I(A)$ und $w_I(B)$ wie folgt:

$w_I(A)$	$w_I(B)$	$w_I((A \vee B))$	$w_I((A \wedge B))$	$w_I((A \leftrightarrow B))$
W	W	W	W	W
W	F	W	F	F
F	W	W	F	F
F	F	F	F	W

Beispiel 2.1.7 *Bauernregeln:*

- *"Abendrot Schlechtwetterbot" kann man übersetzen in*
  Abendrot $\rightarrow$ *Schlechtes_Wetter. Diese Aussage ist weder Tautologie noch Widerspruch, aber erfüllbar.*
- *"Wenn der Hahn kräht auf dem Mist, ändert sich das Wetter oder es bleibt wie es ist." kann man übersetzen in*
  Hahn_kraeht_auf_Mist $\rightarrow$ (*Wetteraenderung* $\vee \neg$ *Wetteraenderung*).
  Man sieht, dass das eine Tautologie ist.

Beispiel 2.1.8 *Beispiele für ein paar allgemeingültige Formelschemata:*

- $A \wedge A \leftrightarrow A$ *(Idemtpotenz)*
- $A \vee A \leftrightarrow A$ *(Idemtpotenz)*
- $(A \wedge B) \wedge C \leftrightarrow A \wedge (B \wedge C)$ *(Assoziativität)*
- $(A \vee B) \vee C \leftrightarrow A \vee (B \vee C)$ *(Assoziativität)*
- $A \wedge B \leftrightarrow B \wedge A$ *(Kommutativität)*
- $A \vee B \leftrightarrow B \vee A$ *(Kommutativität)*
- $A \wedge (B \vee C) \leftrightarrow (A \wedge B) \vee (A \wedge C)$ *(Distributivität)*
- $A \vee (B \wedge C) \leftrightarrow (A \vee B) \wedge (A \vee C)$ *(Distributivität)*
- $A \wedge (A \vee B) \leftrightarrow A$ *(Absorption)*
- $A \vee (A \wedge B) \leftrightarrow A$ *(Absorption)*
- $\neg\neg A \leftrightarrow A$ *(Doppelnegation)*
- $\neg(A \wedge B) \leftrightarrow \neg A \vee \neg B$ *(de Morgan)*
- $\neg(A \vee B) \leftrightarrow \neg A \wedge \neg B$ *(de Morgan)*

Die obigen allgemeingültigen Formeln beinhalten alle das Äquivalenzsymbol $\leftrightarrow$ als obersten Operator. Wir werden diese Äquivalenzen benutzen um Formeln umzuformen.

Nachdem wir nun Formeln interpretiert haben, indem sie unter einer Interpretation auf einen der Werte W und F abgebildet werden, benutzen wir diese Interpretationen, um zu definieren, wann aus einer Formelmenge eine weitere Formel semantisch folgt.

Definition 2.1.9 (semantische Folgerbarkeit)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A folgerbar im Zeichen $M \models_S A$ genau dann wenn:

für jede Interpretation I, für die für jedes $B \in M$ gilt $w_I(B) = W$, gilt auch $w_I(A) = W$.

Wenn nicht gilt, dass A aus einer Formelmenge M entsprechend dieser Definition folgert, dann schreiben wir auch: $M \not\models_S A$.

Im Zusammenhang mit einem rationalen Agenten, identifiziere man die Menge M aus der Definition der Folgerbarkeit mit dem Wissen des Agenten über die Welt. Das ist zum einem Faktenwissen zum anderen Wissen über allgemeine Zusammenhänge in diesem Faktenwissen. Der rationale Agent wird dann versuchen neues aus seinem bestehenden Wissen folgerbares Wissen abzuleiten. Das Wissen, das aus seinem Grundwissen folgt, ist in der obigen Definition die Formel A.

Für den Fall, dass zwei Formeln jeweils auseinander folgern, führen wir die Definition der *äquivalenz* ein.

Definition 2.1.10 (logisch äquivalent)
Für $A, B \in For_S$ sei A logisch äquivalent zu B genau dann wenn: $\{A\} \models_S B$ und $\{B\} \models_S A$.

Für Formeln A mit $\emptyset \models A$ gilt, dass A allgemeingültig ist.

Es gilt das kleine praktische Lemma:

Lemma 2.1.11 *A ist logisch äquivalent zu B genau dann wenn $\{\} \models A \leftrightarrow B$.*

Oder anders ausgedrückt $A \leftrightarrow B$ ist eine Tautologie, genau dann wenn A logisch äquivalent zu B ist.

Implementierung der Semantik

Sofern wir von endlichen Signaturen ausgehen, lassen sich alle unterschiedlichen Interpretationen der Sprache For_Saufzählen. Gibt es n aussagenlogische Variablen in der Signatur S, so gibt es 2^n verschiedene Interpretationen. Im für eine Formel alle Interpretationen aufzustellen, müssen die aussagenlogischen Variablen der Formel gesammelt werden. Daher schreiben wir zunächst eine kleine Funktion, die alle aussagenlogischen Variablen einer Formel auflistet:

PropositionalLogic.hs

```
66 getAtoms :: Prop -> [String]
67 getAtoms= nub . everything (++) ([] `mkQ` atom)
68   where
69     atom (Atom x) = [x]
70     atom _ = []
```

Als nächstes sei der Datentyp definiert, der die zwei Wahrheitswerte ausdrückt, auf die eine Interpretation eine Formel auswertet.

PropositionalLogic.hs
```
71 data Wahrheit = W|F deriving (Eq, Enum,Show)
```

Eine Interpretation kann als eine Abbildung von dargestellt werden, die aussagenlogische Variablen auf einen Wahrheitswert abbildet. Diese Abbildung lässt sich im endlichen Fall als Liste von Paaren darstellen:

PropositionalLogic.hs
```
72 type Interpretation = [(String,Wahrheit)]
```

Die Definition darüber, wie eine Interpretation eine Formel auswertet kennt drei Fälle. Die definition folgt dabei strukturell dem Aufbau der Formeln.

Für aussagenlogische Variablen wird die Abbildung der Variablen in der Abbildung nachgeschlagen:

PropositionalLogic.hs
```
73 interpret it (Atom x) = fromJust$lookup x it
```

Formeln, die mit dem Negationssymbol beginnen, nehmen jeweils den anderen Wahrheitswert, der negierten Teilformel.

PropositionalLogic.hs
```
74 interpret it (Not p)
75  |interpret it p == W = F
76  |otherwise = W
```

Formeln mit dem Implikationssymbol werten entsprechend der Definition nach den Werten der beiden Teilformeln aus:

PropositionalLogic.hs
```
77 interpret it (p1 :-> p2)
78  |interpret it p1 == F = W
79  |otherwise = interpret it p2
```

Alle übrigen Formeln sind mit der Definition der Interpretation nicht direkt abgedeckt. Es handelt sich um Formeln, die die zusätzlichen Symbole enthalten und sind bevor sie zu interpretieren sind, zu vereinfachen.

PropositionalLogic.hs
```
80 interpret it p = interpret it (simplify p)
```

Die folgende kleine Funktion berechnet für eine aussagenlogische Formel `p` zunächst alle möglichen Interpretation. (im Ausdruck `inter$getAtoms p`). Dann gibt sie die Liste der $w_I(\mathtt{p})$ für alle diese Interpretationen als bool'schen Wert zurück.

PropositionalLogic.hs
```
81 allInterpretations p = [interpret int p==W|int <- inter$getAtoms p]
82   where
83    inter []     = [[]]
84    inter (a:as) = concat [[(a,W):ai,(a,F):ai]|ai <- inter as]
```

Damit können nun direkt die semantischen Begriffe definiert werden.

Für eine Tautologie müssen alle Interpretationen wahr ergeben. Das ist die Verundung aller Interpretationsergebnisse:

PropositionalLogic.hs
```
85 tautology  = and . allInterpretations
```

Für eine erfüllbare Formel muss mindestens eine Interpretation wahr ergeben. Das ist die Veroderung aller Interpretationsergebnisse:

PropositionalLogic.hs
```
86 erfuellbar  = or . allInterpretations
```

Die Unerfüllbarkeit ist das Negat der Erfüllbarkeit:

PropositionalLogic.hs
```
87 unerfuellbar = not . erfuellbar
```

Schreiben wir ruhig auch die Formeln hin, die wir bereits als Beispiele von Tautologien kennengelernt haben:

PropositionalLogic.hs
```
88 t01 = a :/\ a :<-> a
89 t02 = a :\/ a :<-> a
90 t03 = (a :/\ b) :/\ c :<-> a :/\ (b :/\ c)
91 t04 = (a :\/ b) :\/ c :<-> a :\/ (b :\/ c)
92 t05 = a :/\ b :<-> b:/\a
93 t06 = a :\/ b :<-> b:\/a
94 t07 = a :/\ (b:\/c) :<-> (a:/\b):\/(a:/\c)
95 t08 = a :\/ (b:/\c) :<-> (a:\/b):/\(a:\/c)
96 t09 = a :/\ (a:\/b) :<-> a
97 t10 = a :\/ (a:/\b) :<-> a
98 t11 = Not(Not a) :<-> a
99 t12 = Not (a:/\b) :<-> Not a:\/ Not b
100 t13 = Not (a:\/b) :<-> Not a:/\ Not b
```

Diese Formeln seien auch noch einmal in einer Liste gesammelt:

PropositionalLogic.hs
```
101 ts=[t01,t02,t03,t05,t06,t07,t08,t09,t10,t11,t12,t13]
```

Probieren Sie in einer interaktiven Session des `ghci` aus, ob die Formeln `ts` tatsächlich Tautologien sind.

2.1.3 Kalkül des natürlichen Schließens

> Mathematik, so lautet die gängige Meinung, beruht im Gegensatz zu den Naturwissenschaften nicht auf Erfahrung, sondern auf reiner Logik. Aus einer überschaubaren Menge von Grundannahmen, so genannten Axiomen, finden die Mathematiker durch die Anwendung logischer Schlussregeln zu immer neuen Erkenntnissen, dringen immer tiefer ins Reich der mathematischen Wahrheit vor. Die menschliche Subjektivität (der Geisteswissenschaft) und die schmutzige Realität (der Naturwissenschaft) bleiben außen vor. Es gibt nichts Wahreres als die Mathematik, und vermittelt wird uns diese Wahrheit durch den Beweis. So ein Beweis ist zwar von Menschen gemacht, und es erfordert Inspiration und Kreativität, ihn zu finden, aber wenn er einmal dasteht, ist er unumstößlich. Allenfalls kann er durch einen einfacheren oder eleganteren ersetzt werden.
>
> *Christoph Drösser, Die Zeit 27.4.2006*

Wir werden zwei Beispiele für Kalküle der Aussagenlogik vorstellen. Zunächst einen Kalkül, der versucht ähnlich wie im Idealfall ein Mathematiker, den formalen Beweis einer Formel mit einer Kette kleiner aufeinanderfolgender Schritte zu führen; und ein zweiter Kalkül, der möglichst gut als einfacher Algorithmus umgesetzt und automatisiert werden kann.

Obiges Zitat aus einem Wochenzeitungsartikel beschreibt, wie im Idealfall ein mathematischer Beweis aussieht. Der Kalkül, der in diesem Abschnitt vorgestellt wird, spiegelt genau diese Idee, Beweise zu führen, wider. Daher definieren wir zunächst Axiome, die wir im Kalkül des natürlichen Schließens benutzen wollen:

Definition 2.1.12 (Axiome)
Die Menge $\mathcal{A}x$ der Axiome über eine Signatur S sei die Vereinigung der folgenden drei Mengen:

- $Ax_1 = \{(A \to (B \to A)) | A, B \in For_S\}$
- $Ax_2 = \{(A \to (B \to C)) \to ((A \to B) \to (A \to C))) | A, B, C \in For_S\}$
- $Ax_3 = \{((\neg A \to \neg B) \to (B \to A)) | A, B \in For_S\}$

Als kleine Übung läßt sich folgene Behauptung beweisen:

Lemma 2.1.13 *Alle Axiome sind Tautologien.*

Aufgabe 8 Beweisen Sie Lemma 2.1.13.

Man vergegenwärtige sich noch einmal, dass es nicht drei Axiome gibt, sondern unendlich viele Axiome, die alle nach einem der drei obigen Schemata gebildet sind.

Im Kalkül des logischen Schließens wird eine Beweiskette aufgebaut: In dieser Kette dürfen Axiome auftauchen, Formeln des Basiswissens, aus dem neues Wissen abgeleitet werden soll, und über eine bestimmte Schlußregel erhaltene neue Schlüsse.

Definition 2.1.14 (syntaktische Ableitbarkeit)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A S-ableitbar im Zeichen $M \vdash_S A$ genau dann wenn: es existiert eine Folge $(A_1, \ldots, A_n)$ mit $A_i \in For_S$, so dass für jedes A_i gilt:

- $A_i \in M$

- *oder* $A_i \in \mathcal{A}x$
- *oder es existieren* $j, k < i$ *mit* $A_j = (A_k \to A_i)$

Die Folge $(A_1, \ldots, A_n)$ *wird als S-Ableitung von* A_n *bezeichnet.*

Wenn es für eine Formel A kein S-Ableitung für eine Formelmenge M gibt, schreiben wir auch: $M \nvdash_S A$.

Der letzte Punkt in dieser Definition ist die eigentliche Schlußregel. Sie ist eine der Schlußregeln, die bereits Aristoteles aufgestellt hat, und wir seit der Antike als *modus ponens* bezeichnet. Sie hat die schematische Form:

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

Diese Schreibweise von logischen Schlußregeln ist wie folgt zu lesen: wenn die Formeln oberhalb des Querstriches angenommen werden können, so darf die Formel unterhalb des Querstriches geschlossen werden.

Für die erste unserer beiden Bauernregeln kommt der *modus ponens* zur Anwendung, wenn Abendrot beobachtet wird. Dann läßt sich aus den Faktum der Beobachtung und der Regel schließen, dass mit schlechten Wetter zu rechnen ist.

$$\frac{\text{Abendrot} \quad \text{Abendrot} \to \text{Schlechtes\_Wetter}}{\text{Schlechtes\_Wetter}}$$

Jetzt wollen wir zunächst einmal auch einen Beweis im Kalkül des natürlichen Schließens führen:

Beispiel 2.1.15 *Versuchen wir einmal eine Ableitung der sehr einfachen Formel* $(A \to A)$ *zu finden:*

$(A \to ((A \to A) \to A))$	(Ax_1)
$((A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A)))$	(Ax_2)
$((A \to (A \to A)) \to (A \to A))$	(MP)
$((A \to (A \to A))$	(Ax_1)
$(A \to A)$	(MP)

Wie man sieht, ist es eine mühsehlige Aufgabe, einen Beweis in diesem Kalkül zu finden. Es ist ein kreativer Anteil notwendig, in dem geschickte Instanziierungen der Axiomenschemata gefunden werden müssen. Algorithmisch ließe sich ein solcher Beweis tatsächlich nur über eine Suche realisieren, in der alle möglichen Ableitungsketten aufgezählt werden, um endlich eine Beweiskette zu finden, die zur zu beweisenden Formel führt.

Da wir mit Syntax, Semantik und Kalkül alle drei Komponenten eines formalen Systems definiert haben, ist der fundamentale Zusammenhang zwischen diesen noch zu zeigen

Satz 2.1.16 *Der Kalkül des natürlichen Schließens ist korrekt und vollständig, d.h.:*

- *aus* $M \vdash_S A$ *folgt* $M \models A$ *(Korrektheit)*
- *aus* $M \models A$ *folgt* $M \vdash_S A$ *(Vollständigkeit)*

Beweis 2.1.17 *Zum Beweis sind zwei Richtungen zu zeigen. Zum einen in der Korrektheit, dass wann immer eine Interpretation I alle Formeln aus der Menge M zu W auswertet, dass dann die Interpretation I auch die Formel A zu W auswertet. Diese Richtung ist meist leichter zu zeigen. Zum anderen ist für die Vollständigkeit zu zeigen, dass wann immer eine Interpretaion alle Formeln aus $M \cup \{A\}$ zu W auswertet, gibt es eine S-Ableitung für A aus der Menge M. Die Vollständigkeit ist in der Regel wesentlich schwerer zu beweisen, als die Korrektheit. Einen korrekten Kalkül zu entwickeln ist nicht schwer. Ein Kalkül, der keine Ableitung zulässt, ist bereits korrekt. Ein vollständiger Kalkül ist hingegen nicht so naheliegend zu entwickeln. In ihm müssen sich alle Wahrheiten syntaktisch errechnen lassen.*

Beginnen wir also mit dem Beweis. Zunächst die Korrektheit:

Korrektheit*:*
Es gelte $M \vdash_S A$. Somit existiert eine Ableitung für A. Sei $(A_1, \ldots, A_n)$ eine S-Ableitung von A aus M. Es ist zu zeigen: $M \models A$. Sei I eine Interpretation mit $w_I(m) = W$ für alle $m \in M$. Es ist zu zeigen: $w_I(A) = W$, d.h. $w_I(A_n) = W$.

Es lässt sich zeigen, dass für alle $i \in \{1, \ldots, n\}$ gilt: $w_I(A_i) = W$.

Hierzu kann eine vollständige Induktion über i benutzt werden.

Annahme: für $j < i$ ist bereits gezeigt: $w_I(A_j) = W$. A_i wurde aufgrund einer der drei folgenden Fälle in die Beweiskette aufgenommen:

- *$A_i \in M$: Dann gilt nach der Voraussetzung über I: $w_I(A_i) = W$.*
- *$A_i \in Ax_1 \cup Ax_2 \cup Ax_3$: Nach Lemma 2.1.13 sind die Axiome des Kalküls Tautologien. Und damit gilt: $w_I(A_i) = W$.*
- *Es gibt $j < i$ und $l < i$ mit: $A_j = (A_k \to A_i)$. Nach der Induktionsvoraussetzung gilt: $w_I(A_j) = w_I(A_k) = W$. Nach der Definition 2.1.5 der Interpretation gilt mit $w_I(A_k \to A_i) = W$ und $w_I(A_k) = W$ auch gelten $w_I(A_i) = W$, was zu beweisen war.*

Vollständigkeit:
Der Beweis der Vollständigkeit ist um einiges kniffiger. Wir werden ihn in mehreren Schritten mit ein paar Hilfsbehautungen führen. Der Beweis wird in vier Schritten erfolgen:

1. *Nachweis diverser Ableitungsaussagen mit Hilfe des Deduktionstheorems.*
2. *Definition der Konsistenz und Vollständigkeit einer Formelmenge.*
3. *Erweiterungslemma.*
4. *Vollständigkeitssatz.*

Lemma 2.1.18 *Sei S eine Signatur, $M \subseteq For_S, A \in For_S, B \in For_S, C \in For_S$.*
Wenn $M \vdash_S A$ und $M \vdash_S (A \to B)$, dann auch $M \vdash_S B$.

Beweis 2.1.19 *Triviale Übung. Einfache Anwendung des Modus Ponens.*

Satz 2.1.20 (Deduktionstheorem)
$M \vdash_S (A \to B)$ *gdw.* $M \cup \{A\} \vdash_S B$.

Das Deduktionstheorem stellt eine Verbindung zwischen dem Ableitungsbegriff des Kalküls und dem objektsprachlichen Pfeilsymbol $\rightarrow$ her. Es soll uns helfen, auf einfachere Weise zu zeigen, dass eine Formel aus einer bestimmten Formelmenge ableitbar ist. In Beispiel 2.1.15 war schon zu sehen, wie kompliziert es ist eine Ableitungskette für die Tautologie $(A \rightarrow A)$ herzuleiten. Trivial hingegen ist die Aussage $\{A\} \vdash_S A$. Mit dem Deduktionstheorem lässt sich daraus direkt schließen: $\emptyset \vdash_S (A \rightarrow A)$.

Beweis 2.1.21 *Es gibt zwei Richtungen zu zeigen:*

- *Es gelte: $M \vdash_S (A \rightarrow B)$.*
  Dann gilt auch:

$M \cup \{A\} \vdash_S (A \rightarrow B)$	(erst recht)
$M \cup \{A\} \vdash_S A$	(trivial)
$M \cup \{A\} \vdash_S B$	(nach Lemma 2.1.18)

- *Es gelte: $M \cup \{A\} \vdash_S B$.*
  Sei $(A_1, \ldots, A_n)$ eine S-Ableitung von B aus $M \cup \{A\}$.
  Es ist zu zeigen: $M \vdash_S (A \rightarrow B)$ also: $M \vdash_S (A \rightarrow A_n)$.
  Es lässt sich sogar für alle $i = 1, \ldots, n$ zeigen, dass $M \vdash_S (A \rightarrow A_i)$.
  Der Beweis ist per Induktion über i zu führen. Sei i mit $1 \leq i \leq n$ gegeben.
  Induktionsannahme: für $j < i$ sei schon gezeigt: $M \vdash_S (A \rightarrow A_j)$.

  Es gibt drei Fälle, über die die Formel A_i in die Ableitungskette aufgenommen wurde:

  - *A_i ist ein Axiom oder Element aus M: $A_i \in M \cup Ax_1 \cup Ax_2 \cup Ax_3$. Dann gilt:*

$M \vdash_S A_i$	(trivial da Axiom)
$M \vdash_S (A_i \rightarrow (A \rightarrow A_i))$	(Ax_1)
$M \vdash_S (A \rightarrow A_i)$	(mit Lemma 2.1.18)

  - *$A_i = A$. Im Beispiel 2.1.15 wurde bereits gezeigt: $\emptyset \vdash_S (A \rightarrow A)$. Damit gilt auch $M \vdash_S (A \rightarrow A)$*
  - *A_i wurde durch Anwednung des Modus Ponens in die Beweiskette aufgenommen. Es gibt also $k < i$ und $j < i$ mit $A_j = (A_k \rightarrow A_i)$.*

    Nach der Induktionsvoraussetzung dürfen wir annehmen:
    $M \vdash_S (A \rightarrow A_k)$ und $M \vdash_S (A \rightarrow (A_k \rightarrow A_i))$.

    Es lässt sich weiterhin folgende Formel aus der Axiomenmenge Ax_2 ableiten:
    $M \vdash_S ((A \rightarrow (A_k \rightarrow A_i)) \rightarrow ((A \rightarrow A_k) \rightarrow (A \rightarrow A_i)))$

    Mit Lemma 2.1.18 folgt:
    $M \vdash_S ((A \rightarrow A_k) \rightarrow (A \rightarrow A_i))$
    Mit nochmaliger Anwendung von Lemma 2.1.18 folgt:
    $M \vdash_S (A \rightarrow A_i)$
    Was zu zeigen war.

Satz 2.1.22 *Folgende Formelschemata sind aus $\emptyset$ ableitbar:*

$$((A \to B) \to ((B \to C) \to (A \to C))) \quad \textit{(Transitivität)} \tag{2.1}$$
$$(A \to ((A \to B) \to B)) \quad \textit{(Modus Ponens)} \tag{2.2}$$
$$(\neg\neg A \to A) \quad \textit{(Tertium Non Datur)} \tag{2.3}$$
$$(A \to \neg\neg A) \quad \textit{(Tertium Non Datur)} \tag{2.4}$$
$$((A \to B) \to (\neg B \to \neg A)) \quad \textit{(Kontraposition)} \tag{2.5}$$
$$(A \to (\neg B \to \neg(A \to B))) \quad \textit{(Bedeutung von } \to\textit{)} \tag{2.6}$$
$$(\neg A \to (A \to B)) \quad \textit{(Bedeutung von } \to\textit{)} \tag{2.7}$$
$$(A \to (B \to (A \to B))) \quad \textit{(Bedeutung von } \to\textit{)} \tag{2.8}$$
$$(\neg(A \to B) \to A) \quad \textit{(Bedeutung von } \to\textit{)} \tag{2.9}$$
$$(\neg(A \to B) \to \neg B) \quad \textit{(Bedeutung von } \to\textit{)} \tag{2.10}$$
$$((A \to \neg A) \to \neg A) \quad \textit{(Selbstwiderlegung)} \tag{2.11}$$
$$((\neg A \to A) \to A) \quad \textit{(Selbstwiderlegung)} \tag{2.12}$$
$$((A \to B) \to ((\neg A \to B) \to B)) \quad \textit{(Fallunterscheidung)} \tag{2.13}$$
$$(\neg(A \to A) \to B) \quad \textit{(ex falso quod libet)} \tag{2.14}$$

Beweis 2.1.23 *Die Ableitbarkeit lässt sich wesentlich einfacher jeweils zeigen, wenn möglichst oft das Deduktionstheorem zur Anwendung kommt.*

- *Ableitung von 2.1:*

$$\begin{array}{rcll}
\{(A \to B), (B \to C), A\} & \vdash_S & A & \text{(trivial)} \\
\{(A \to B), (B \to C), A\} & \vdash_S & (A \to B) & \text{(trivial)} \\
\{(A \to B), (B \to C), A\} & \vdash_S & B & \text{(2.1.18)} \\
\{(A \to B), (B \to C), A\} & \vdash_S & (B \to C) & \text{(trivial)} \\
\{(A \to B), (B \to C), A\} & \vdash_S & C & \text{(2.1.18)} \\
\{(A \to B), (B \to C)\} & \vdash_S & (A \to C) & \text{(2.1.20)} \\
\{(A \to B)\} & \vdash_S & ((B \to C) \to (A \to C)) & \text{(2.1.20)} \\
\{\} & \vdash_S & ((A \to B) \to ((B \to C) \to (A \to C))) & \text{(2.1.20)}
\end{array}$$

- *Ableitung von 2.2:*

$$\begin{array}{rcll}
\{A, (A \to B)\} & \vdash_S & A & \text{(trivial)} \\
\{A, (A \to B)\} & \vdash_S & (A \to B) & \text{(trivial)} \\
\{A, (A \to B)\} & \vdash_S & B & \text{(2.1.18)} \\
\{A\} & \vdash_S & ((A \to B) \to B) & \text{(2.1.20)} \\
\{\} & \vdash_S & (A \to ((A \to B) \to B)) & \text{(2.1.20)}
\end{array}$$

- *Ableitung von 2.3:*

$$\begin{array}{rcll}
\{\} & \vdash_S & (\neg\neg A \to (\neg\neg\neg\neg A \to \neg\neg A)) & (\mathrm{Ax}_1) \\
\{\neg\neg A\} & \vdash_S & (\neg\neg\neg\neg A \to \neg\neg A)) & \text{(2.1.20)} \\
\{\neg\neg A\} & \vdash_S & ((\neg\neg\neg\neg A \to \neg\neg A) \to (\neg A \to \neg\neg\neg A)) & (\mathrm{Ax}_3) \\
\{\neg\neg A\} & \vdash_S & (\neg A \to \neg\neg\neg A) & \text{(2.1.18)} \\
\{\neg\neg A\} & \vdash_S & ((\neg A \to \neg\neg\neg A) \to (\neg\neg A \to A)) & (\mathrm{Ax}_3) \\
\{\neg\neg A\} & \vdash_S & (\neg\neg A \to A) & \text{(2.1.18)} \\
\{\neg\neg A\} & \vdash_S & A & \text{(2.1.20)} \\
\{\} & \vdash_S & (\neg\neg A \to A) & \text{(2.1.20)}
\end{array}$$

- *Ableitung von 2.4:*

$$\begin{array}{rcll} \{\} & \vdash_S & (\neg\neg\neg A \to \neg A) & (2.3) \\ \{\} & \vdash_S & ((\neg\neg\neg A \to \neg A) \to (A \to \neg\neg A)) & (\mathrm{Ax}_3) \\ \{\} & \vdash_S & (A \to \neg\neg A) & (2.1.18) \end{array}$$

- *Ableitung von 2.5:*

$$\begin{array}{rcll} \{(A \to B)\} & \vdash_S & ((\neg\neg A \to \neg\neg B) \to (\neg B \to \neg A)) & (\mathrm{Ax}_3) \\ \{(A \to B)\} & \vdash_S & (\neg\neg A \to A) & (2.3) \\ \{(A \to B)\} & \vdash_S & (A \to B))) & (\text{trivial}) \\ \{(A \to B)\} & \vdash_S & ((\neg A \to A) \to ((A \to B) \to (\neg\neg A \to B))) & (2.1) \\ \{(A \to B)\} & \vdash_S & ((A \to B) \to (\neg\neg A \to B)) & (2.1.18) \\ \{(A \to B)\} & \vdash_S & (\neg\neg A \to B) & (2.1.18) \\ \{(A \to B)\} & \vdash_S & (B \to \neg\neg B) & (2.4) \\ \{(A \to B)\} & \vdash_S & ((\neg\neg A \to B) \to ((B \to \neg\neg B) \to (\neg\neg A \to \neg\neg B))) & (2.1) \\ \{(A \to B)\} & \vdash_S & ((B \to \neg\neg B) \to (\neg\neg A \to \neg\neg B)) & (2.1.18) \\ \{(A \to B)\} & \vdash_S & (\neg\neg A \to \neg\neg B) & (2.1.18) \\ \{(A \to B)\} & \vdash_S & (\neg B \to \neg A) & (2.1.18) \\ \{\} & \vdash_S & ((A \to B) \to (\neg B \to \neg A)) & (2.1.20) \end{array}$$

- *Ableitung von 2.6:*

$$\begin{array}{rcll} \{\} & \vdash_S & (A \to ((A \to B) \to B)) & (2.2) \\ \{A\} & \vdash_S & ((A \to B) \to B) & (2.1.20) \\ \{A\} & \vdash_S & (((A \to B) \to B) \to (\neg B \to \neg(A \to B))) & (2.5) \\ \{A\} & \vdash_S & (\neg B \to \neg(A \to B)) & (2.1.18) \\ \{\} & \vdash_S & (A \to (\neg B \to \neg(A \to B))) & (2.1.20) \end{array}$$

- *Ableitung von 2.7:*

$$\begin{array}{rcll} \{\} & \vdash_S & (\neg A \to (\neg B \to \neg A)) & (\mathrm{Ax}_1) \\ \{\neg A\} & \vdash_S & (\neg B \to \neg A) & (2.1.20) \\ \{\neg A\} & \vdash_S & ((\neg B \to \neg A) \to (A \to B)) & (\mathrm{Ax}_3) \\ \{\neg A\} & \vdash_S & (A \to B) & (2.1.18) \\ \{\} & \vdash_S & (\neg A \to (A \to B)) & (2.1.20) \end{array}$$

- *Ableitung von 2.8:*

$$\begin{array}{rcll} \{A\} & \vdash_S & (B \to (A \to B)) & (\mathrm{Ax}_1) \\ \{\} & \vdash_S & (A \to (B \to (A \to B))) & (2.1.20) \end{array}$$

Aufgabe 9 *Zeigen Sie, dass die Formeln der Form 2.9 bis 2.14 abgeleitet werden können.*

Definition 2.1.24 *Sei S eine Signatur und $M \subset For_S$.*

- *M heißt S-konsistent, falls es kein $A \in For_S$ gibt mit $M \vdash_S A$ und $M \vdash_S \neg A$.*
- *M heißt S-vollständig, falls für jedes $A \in For_S$ gilt: $M \vdash_S A$ oder $M \vdash_S \neg A$.*

Lemma 2.1.25 *Sei S eine Signatur, $M \subset For_S$ und $A \in For_S$. Dann gilt:*

1. *Die folgenden drei Aussagen sind äquivalent:*

(a) *M ist S-konsistent.*

(b) *Es gibt kein $B \in For_S$ mit $M \vdash_S \neg(B \to B)$.*

(c) *Es gibt ein $C \in For_S$ mit $M \nvdash_S C$.*

2. *Wenn es eine Interpretation I gibt mit $w_I(M) \subseteq \{W\}$, dann ist M S-konsistent.*

3. *$M \nvdash_S A$ gdw. $M \cup \{\neg A\}$ ist S-konsistent.*

Beweis 2.1.26 *1. Die Äquivalenz der drei Fälle wird in einem Ringschluss gezeigt.*

- *Zunächst die der Beweis für aus a) folgt b):*
  Sei M S-konsistent. Beweis per Widerspruch. Annahme es gibt ein $B \in For_S$ mit $M \vdash_S \neg(B \to B)$. Mit 2.14 folgt: $M \vdash_S B$ und $M \vdash_S \neg B$. Dieser ist jedoch ein Widerspruch zur S-Konsistenz von M.
- *Nun der Beweis für aus b) folgt c):*
  Es gibt kein $B \in For_S$ mit $M \vdash_S \neg(B \to B)$. Damit gilt für jedes $B_0 \in For_S$: $M \nvdash_S \neg(B_0 \to B_0)$. Damit ist ein $C \in For_S$ mit $M \nvdash_S C$ gefunden.
- *Und schließlich der Beweis für aus c) folgt a):*
  Sei $C \in For_S$ mit $M \nvdash_S C$. Beweis per Widerspruch. Annahme: M ist nicht S-konsistent. Dann gibt es ein $A \in For_S$ mit $M \vdash_S A$ und $M \vdash_S \neg A$. Mit 2.7 kann aber gezeigt werden, dass jede Formel nun aus M ableitbar ist. Widerspruch zur Voraussetzung!

2. *Sei $w_I(M) \subseteq \{W\}$. Beweis per Widerspruch. Annahme: M ist nicht S-konsistent. Dann gibt es ein $A \in For_S$ mit $M \vdash_S A$ und $M \vdash_S \neg A$. Aus der Korrektheit des Kalküls folgt: $M \models_S A$ und $M \models_S \neg A$. Also folgt: $w_I(A) = W$ und $w_I(\neg A) = W$. Widerspruch!*

3. - *Zunächst die Richtung von rechts nach links per Kontraposition:*
     Es gelte $M \vdash_S A$. Dann folgt: $M \cup \{\neg A\} \vdash_S A$ und $M \cup \{\neg A\} \vdash_S \neg A$. Also ist $M \cup \{\neg A\}$ nicht S-konsitent.
   - *Nun der Beweis der umgekehrten Folgerung per Kontraposition:*
     Sei $M \cup \{\neg A\}$ nicht S-konsistent. Damit gilt nach 2.1.25 1.: $M \cup \{\neg A\} \vdash_S A$. Und mit 2.1.20 schließlich: $M \vdash_S (\neg A \to A)$. Mit 2.12 und Modus Ponens ergibt sich: $M \vdash_S A$.

Um die Vollständigkeit des Kalküls zu beweisen brauchen wir noch ein weiteres Hilfslemma, das Erweiterungslemma. Das Erweiterungslemma sagt aus, dass jede S-konsistente Formelmenge zu einer S-konsistenten und S-vollständigen Formelmenge erweitert werden kann.

Lemma 2.1.27 **Erweiterungslemma**
Sei S eine Signatur und $M \subseteq For_S$.
Wenn M S-konsistent ist, dann gibt es eine Formelmenge $M' \subseteq For_S$ mit:

- *$M \subseteq M'$*
- *M' ist S-konsistent.*
- *M' ist S-vollständig.*

Beweis 2.1.28 *Sei M S-konsistent.*
For_Sist eine abzählbar unendliche Menge. Wir wählen eine Aufzählung aller Formeln in For_S: $A_0, A_1, A_2, \ldots, A_i, A_{i+1}, \ldots$.
Nun wird eine Folge von Teilmengen von For_S:
$M_0 \subseteq M_1 \subseteq M_2 \subseteq M_3 \subseteq M_4 \ldots$
wie folgt konstruiert:

- *$M_0 = M$. (und damit S-konsistent)*
- $$M_{i+1} = \begin{cases} M_i, & \text{wenn } M_i \vdash_S A_i \\ M_i \cup \{\neg A_i\}, & \text{wenn } M_i \nvdash_S A_i \end{cases}$$

Mit 2.1.25 gilt: Da M_i S-konsistent ist, ist auch M_{i+1} S-konsistent.

Nun definiere $M' \bigcup_{i \in \mathbb{N}} M_i$.

M' ist die gesuchte Menge:

- *$M \subseteq M'$, denn es gilt $M = M_0 \subseteq M_i$ für alle $i > 0$.*
- *M' ist S-konsistent.*

  Annahme: M' sei nicht S-konsistent. Dann gilt: $M' \vdash_S \neg(A \to A)$ für eine beliebige Formel A. Also gibt es ein $i \in \mathbb{N}$ mit $M_i \vdash \neg(A \to A)$. Damit wäre M_i nicht S-konsistent. Dieses wäre ein Widerspruch.
- *M' ist S-vollständig.*

  Sei $A \in For_S$ gegeben, etwa $A = A_i$. Es gibt zwei Fälle in Bezug auf A_i und M_i.
  - *$M_i \vdash_S A_i$. Dann gilt erst recht: $M' \vdash_S A_i$.*
  - *$M_i \nvdash_A A_i$. Dann ist $M_{i+1} = M_i \cup \{\neg A_i\}$ also $M' \vdash_S \neg A_i$. Somit gilt für jede Formel, dass sie oder ihr Negat aus M' ableitbar ist.*

Das nächste kleine Hilfslemma fasst ein paar Eigenschaften zusammen, die für S-konsistente, S-vollständige Formelmengen gelten.

Lemma 2.1.29 *Sei S eine Signatur, $M \subseteq For_S$, $B, C \in For_S$.*
Sei M S-konsistent und S-vollständig. es gilt:

1. *$M \vdash_S \neg B$ gdw.: $M \nvdash_S B$.*
2. *$M \vdash_S \neg(B \to C)$ gdw.: $M \vdash_S B$ und $M \vdash_S \neg C$.*

Beweis 2.1.30 *1. Folgt direkt aus der Definition der S-Konsistenz (links nach rechts) und der S-Vollständigkeit (rechts nach links).*

2. - *links nach rechts: Hier lassen sich 2.1.22 2.9 und 2.1.22 2.10 anwenden.*
   - *rechts nach links: Sei $M \vdash_S B$ und $M \vdash_S \neg C$.*
     Beweis per Widerspruch. Annahme: $M \nvdash_S \neg(A \to B)$. Aus 1) folgt: $M \vdash_S \neg\neg(B \to C)$. Also mit 2.1.22 2.2: $M \vdash_S (B \to C)$ und schließlich: $M \vdash_S C$. Widerspruch zur Annahme.

Wir können nun langsam zum eigentlichen Beweis des Vollständigkeitssatzes gehen. Es ist zu zeigen, dass aus $M \models_S A$ *folgt* $M \vdash_S A$.

Der Beweis wird per Kontraposition geführt. Es wird also gezeigt, dass aus $M \nvdash_S A$ *folgt* $M \not\models_S A$.

Beide Aussagen werden zunächst zu zwei äquivalenten Aussagen umgeformt.

- $M \nvdash_S A$ *ist nach 2.1.25 3) äquivalent zu* $M \cup \{\neg A\}$ *ist S-konsistent.*
- $M \not\models_S A$ *heisst nach der Definition der Folgerbarkeit: Es gibt eine Interpretation* I *mit* $w_I(M) \subseteq \{W\}$, *aber* $w_I(A) = F$ *und damit* $w_I(M \cup \{\neg A\}) \subseteq \{W\}$.

Damit lässt sich die Vollständigkeit jetzt über die allgemeinere Aussage beweisen, ob eine S-konsistente Formelmenge eine Interpretation hat, die die alle Formel zu W *auswertet. Eine solche Interpretation wird auch Modell genannt.*

Lemma 2.1.31 **Modelllemma**
Sei S *eine Signatur. Dann gilt:*
Zu jeder S-konsistenten Formelmenge M *gibt es eine S-Interpretation* I *mit* $w_I(M) \subseteq (\{W\})$.

I *heisst dann ein Modell von* M.

Beweis 2.1.32 *Zu* M *gibt es nach dem Erweiterungslemma ein* $M' \in For_S$ *mit* $M \subseteq M'$. *und* M' *ist S-konsistent und S-vollständig.*

Nun lässt sich über M' *eine Interpretation* $I : S \to \{W, F\}$ *wie folgt definieren:*

$$I(P) = \begin{cases} W, & \text{wenn } M' \vdash_S P \\ F, & \text{sonst} \end{cases}$$

Nun werden wir mit struktureller Induktion zeigen, dass für alle $A \in For_S$ *gilt:*

$w_I(A) = W$ *gdw.:* $M' \vdash_S A$.

- *Induktionsanfang:* A *ist ein Atom, d.h.* $A \in S$. *Dann folgt die Behauptung direkt aus der Definition, nach der* I *gebaut wurde.*
- *1. Induktionsschritt:* $A = \neg B$. *Dann folgt:*

	$w_I(A) = W$
gdw.	$w_I(B) = F$
gdw. (nach Induktionsvorraussetzung)	$M' \nvdash_S B$
gdw. (nach 2.1.29 1)	$M' \vdash_S \neg B$
gdw.	$M' \vdash_S A$

- *2. Induktionsschritt:* $A = (B \to C)$. *Dann folgt:*

	$w_I(A) = F$
gdw.	$w_I(B) = W$ und $w_I(C) = F$
gdw. (nach Induktionsvorraussetzung)	$M' \vdash_S B$ und $M' \nvdash_S C$
gdw.gdw. (nach 2.1.29 1)	$M' \vdash_S B$ und $M' \vdash_S \neg C$
gdw. (nach 2.1.29 2)	$M' \vdash_S \neg(A \to B)$

gdw. (nach 2.1.29 1) $M' \nvdash_S A$

Insgesamt haben wir die Vollständigkeit nun auf folgende Weise gezeigt:

Statt:
aus $M \models_S A$ *folgt* $M \vdash_S A$ *zu zeigen*
wurde:
aus $M \nvdash_S A$ *folgt* $M \not\models_S A$ *gezeigt.*
Diese Aussage wurde umformoliert zu:
aus $M \cup \{A\}$ *ist S-konsistent folgt:* $M \cup \{A\}$ *hat ein Modell. Diese Aussage ist nach dem Modelllemma wahr, da jede S-konsistente Formelmenge ein Modell hat. Die Vollständigkeit des Kalküls des natürlichen Schließens ist damit bewiesen.*

Implementierung des Kalküls des natürlichen Schließens

Der Kalkül des natürlichen Schließens bietet sich nicht besonders für eine Implementierung an, die zielgerichtet eine Ableitungskette für die zu beweisende Formel erzeugt. Hierzu ist die geschickte Instanziierung der Axiome notwendig. Was aber möglich ist, ist eine Aufzählung aller ableitbaren Formeln. Hierzu vergewissere man sich, dass die Sprache For_S abzählbar ist. Es folgt, dass alle möglichen Instanziierungen der drei Axiome abzählbar sind. In die Aufzählung der Axiome lassen sich dann schließlich die zusätzlich durch Modus Ponens ableitbaren Formeln einfügen. Mit diesem Hauruckverfahren lassen sich alle ableitbaren Formeln generieren. Für einen Beweis ist dann zu schauen, ob die als allgemeingültig zu beweisene Formel in dieser Ableitung auftaucht.

Beginnen wir die Implementierung mit der Aufzählung der Formelmenge For_S. Hierzu seien zunächst drei simple Hilfsfunktionen definiert: `atom` erzeugt aus einer Liste von Strings, die Atome der Sprache. `negated` erstellt das Negat aller Formeln einer Liste. `implies` erzeugt alle Implikationen mit einer Vorbedingung aus dem ersten Parameter und einer Nachbedingung aus der zweiten Menge. Dabei werden allerdings keine Implikationen zwischen derselben Formel erzeugt.

PropositionalLogic.hs

```
102 atom = map Atom
103 negated = map Not
104 implies xs ys = [a:->b|a<-xs,b<-ys,not (a==b)]
```

Nun lasses sich schrittweise alle Formeln als For_S aufzählen. Dieses wird durch die Funktion `newgen` bewerkstelligt. Diese hat zwei Parameter: der erste Parameter sind alle bereits berechneten Formeln, die zweite Liste sind die im letzten Iterationsschritt neu berechneten Formeln.

Für eine neue Generation werden neu erzeugt:

- alle Negate der zuletzt erzeugten Formel.
- alle Implikationen zwischen den zuletzt erzeugten Formeln.
- alle Implikationen einer Formel mit sich selbst aus den zuletzte erzeugten Formeln.
- alle Implikationen aus den bisherigen Formeln mit den zuletzt erzeugten Formeln.
- die umgekehrten Implikationen der letzten Implikationen.

PropositionalLogic.hs

```
105 newgen sofar lastnew
106   = lastnew
107       ++newgen
108        (sofar++lastnew)
109        (    negated lastnew
110          ++ implies lastnew lastnew
111          ++ [a:->a|a<-lastnew]
112          ++ implies sofar lastnew
113          ++ implies lastnew sofar )
```

Alle Formeln lassen sich mit dieser Generierungsfunktion erzeugen, indem sie mit den Atomen der Sprache begonnen wird.

PropositionalLogic.hs

```
114 allformulae vars = newgen [] (atom vars)
```

Nachdem jetzt alle Formeln aufgezählt sind, können diese in die Axiomenschemata eingesetzt werden. Das erste und das dritte Axiom wird jeweils aus zwei Teilformeln zusammengesetzt, das zweite Axiom aus drei Formeln. Aus der unendlichen Folge von allen Formeln in For_S sind also alle Paare zweier Formeln zu Erzeugen, die dann zu benutzen sind, um Instanzen des ersten und dritten Axioms zu bilden.

Die folgende Funktion ist in der Lage alle Paare aus den Elemente zweier potentiell unendlicher Liste zu bilden.

PropositionalLogic.hs

```
115 diagonal (x:xs) (y:ys) = (x,y):merge3 xs2 ys2 dia2
116  where
117   dia2= diagonal xs ys
118   xs2 = [(x,y)|y<-ys]
119   ys2 = [(x,y)|x<-xs]
```

Um alle Instanzen des zweiten Axioms zu bilden, brauchen wir gar alle Tripel aus Formeln. Diese lassen sich durch zweifache Anwendung der Funktion `diagonal` erzielen:

PropositionalLogic.hs

```
120 diagonal3 xs ys zs = diagonal xs$diagonal ys zs
```

Die Hilfsfunktionen `merge` und `merge3` sind in der Lage unendliche Listen zu einer zu vereinigen:

PropositionalLogic.hs

```
121 merge (x:xs) (y:ys)= x:y:merge xs ys
122 merge xs ys = xs++ys
123
124 merge3 xs ys zs = merge xs$merge ys zs
```

Jetzt können wir mit Hilfe dieser Diagonalisierung alle Axiome aufzählen. Die drei Axiomenschemata sind wie folgt definiert:

PropositionalLogic.hs

```
125 ax1 a b = a:-> (b:->a)
126 ax2 a b c = (a:->(b:->c)):->((a:->b):->(a:->c))
127 ax3 a b = (Not a:-> Not b):->(b:->a)
```

Für jedes dieser Schemata können nun für eine potentiell unendliche Formelmenge alle Instanzen erzeut werden:

PropositionalLogic.hs

```
axioms1 fors = map (\(a,b)->ax1 a b)$diagonal fors fors
axioms3 fors = map (\(a,b)->ax3 a b)$diagonal fors fors
axioms2 fors = map (\(a,(b,c))->ax2 a b c)$diagonal3 fors fors fors
```

Alle Axiome für eine Signatur sind nun aufzählbar:

PropositionalLogic.hs

```
allAxioms sigs = merge3 (axioms1 fs) (axioms2 fs) (axioms3 fs)
 where
   fs= allformulae sigs
```

Schließlich gilt es noch die per Modus Ponens ableitbaren Formeln in diese unendliche Liste einzufügen. Hierzu folgt die Definition, des Modus Ponens:

PropositionalLogic.hs

```
mp a1 (a2:->b)
 |a1==a2=[b]
mp (a2:->b) a1
 |a1==a2=[b]
mp _ _ =[]
```

Folgende Hilfsfunktion berechnet alle per Modus Ponens mit einer Formel aus und den Formeln einer einer Liste ableitbaren Formeln:

PropositionalLogic.hs

```
mps x = concat . (map (mp x))
```

Schließlich können wir alle ableitbaren Formeln, sprich alle Tautologien aufzählen:

PropositionalLogic.hs

```
allTauts vars = newgen [] axs
  where
   axs = allAxioms vars
   newgen old (n:ext) = n:(newgen ((n:old)) (ponens++ext))
     where
       ponens = [x|x<-mps n old,not (elem x old )]
```

Ein Beweis für eine Formel ist gefunden, wenn sie in der Liste aller Tautologien enthalten ist. Es sei hier für $S = \{A, B, C\}$ definiert.

PropositionalLogic.hs

```
allts = allTauts ["A","B","C"]
beweis f = (takeWhile (\x->not (x==f)) (allts))++[f]
```

Damit haben wir theoretisch einen Beweiser für alle Tautologien gefunden. Wie sieht es aus? Kann dieser Beweiser die einfachen Formeln als 2.1.22 ableiten?

PropositionalLogic.hs

```
f00 = a:->a
f01 = (a:->b):->((b:->c):->(a:->c))
f02 = a:->((a:->b):->b)
f03 = Not(Not a) :-> a
```

```
152 f04 = a:-> Not(Not a)
153 f05 = (a:->b):->(Not b:-> Not a)
154 f06 = a:-> (Not b :-> Not (a:->b))
155 f07 = Not a :-> (a:->b)
156 f08 = a :-> (b:-> (a:->b))
157 f09 = Not (a:-> b) :-> a
158 f10 = Not (a:-> b) :-> Not b
159 f11 = (a:->Not a):-> Not a
160 f12 = (Not a:->a):-> a
161 f13 = (a:->b):->((Not a:-> b):->b)
162 f14 = Not (a:->a):->b
```

Aufgabe 10 Probieren Sie, welche der Formeln `f01` bis `f14` mit dem Beweiser des Kalküls des natürlichen Schließen ableitbar sind. Benutzen Sie hierbei die Funktion `natweis` aus dem Anhang, um ein pdf-Dokument für den gefundenen Beweis zu erhalten.

2.1.4 Der Resolutionskalkül

Der Kalkül des natürlichen Schließens im letzten Abschnitt, war für algorithmische Zwecke und als Grundlage eines automatischen Beweissystems auf aussagenlogischen Formeln denkbar ungeeignet. Daher werden wir jetzt einen Kalkül vorstellen, der sich gut automatisieren läßt. Hierzu werden wir die Formeln zunächst in eine bestimmte Normalform umformen.

Klauselnormalform

Definition 2.1.33 (konjunktive Normalform (CNF), Klauselnormalform (KNF))
Eine Formel, die eine Konjunktion von Disjunktionen von Literalen ist, ist in konjunktive Normalform oder auch Klauselnormalform.

D.h. die Formel ist von der Form

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge \ldots \wedge (L_{m,1} \vee \ldots \vee L_{m,n_m})$$

wobei $L_{i,j}$ Literale sind.

Die Klauselnormalform wird oft als Menge von Mengen notiert und auch behandelt. Dies ist gerechtfertigt, da sowohl $\wedge$ als auch $\vee$ assoziativ, kommutativ und idempotent sind, so dass Vertauschungen und ein Weglassen der Klammern erlaubt ist. Wichtig ist die Idempotenz, die z.B. erlaubt, eine Klausel $\{A, B, A, C\}$ als unmittelbar äquivalent zur Klausel $\{A, B, C\}$ zu betrachten. Eine Klausel mit einem Literal bezeichnet man auch als *1-Klausel*. Eine Klausel (in Mengenschreibweise) ohne Literale wird als *leere Klausel* bezeichnet. Diese ist äquivalent zu einem Widerspruch. Sie wird auch mit dem Symbol $\square$ notiert.

Lemma 2.1.34 *Eine Klausel C ist eine Tautologie genau dann wenn es eine Variable A gibt, so dass sowohl $A \in C$ als auch $\neg A \in C$.*

Beweis: Übungsaufgabe.

Lemma 2.1.35 *Eine Klauselmenge M ist unerfüllbar, wenn es eine Variable A gibt, so dass sowohl $\{A\} \in M$ als auch $\{\neg A\} \in M$.*

Beweis: Übungsaufgabe.

Der Resolutionskalkül ist nur auf Klauselmengen definiert. Es wird also die ursprüngliche Struktur der Formeln vollständig zerstört. Für jede Formel existiert eine äquivalente konjunktive Normalform. Der folgende Algorithmus formt jede aussagenlogische Formel in eine semantisch äquivalente Formel in Klauselnormalform um.

Umformen in Klauselnormalform

- ersetze sukzessive Teilformeln der Gestalt
  - $\neg\neg A$ durch A,
  - $\neg(A \vee B)$ durch $(\neg A \wedge \neg B)$,
  - $\neg(A \wedge B)$ durch $\neg A \vee \neg B$

  bis keine der Umformungen mehr möglich ist.
- dann ersetze Teilformeln der Gestalt
  - $(A \vee (B \wedge C))$ durch $(A \vee B) \wedge (A \vee C)$
  - und $((B \wedge C) \vee A)$ durch $(B \vee A) \wedge (C \vee A)$

  bis keine der Umformungen mehr möglich ist.

Mit diesen Umformungen läßt sich jede Formel in eine semantisch Klauselnormalform transformieren. Die Disjunktionen der Konjuktiven-Normalform werden als Klauseln bezeichnet.

Lemma 2.1.36 *Der obige Algorithmus terminiert mit einer Formel in KNF.*

Beweis 2.1.37 *Das Lemma ist eine Aussage über die totale Korrektheit eines Algorithmus. Ein Programm wird als total korrekt bezeichnet wenn:*

- *es für jede Eingabe terminiert*
- *und wenn es partiell korrekt ist. Als partiell korrekt wird bezeichnet, dass das Programm nur korrekte Ergebnisse liefert.*

Es sind also zwei Eigenschaften zu beweisen:

- **partielle Korrektheit:** *TO DO*
- **Terminierung:** *Der obige Algorithmus wendet sukzessiv Transformationsregeln auf eine Formel an. Um zu zeigen, dass irgendwann der Algorithmus terminiert, muss so irgendwann eine Formel erzeugt werden, auf die keine der regeln mehr anwendbar ist. Dieses lässt sich am einfachsten zeigen, wenn man ein Terminierungsmaß findet, in dem die Formeln durch die Transformationsregeln minimiert werden.*

  Der Algorithmus hat zwei Ersetzungsschritte, für die getrennt die Terminierung gezeigt werden muss. Im ersten Ersetzungsschritt werden nacheinander die Negationssymbole näher an die Atome der Formel geschoben.

Für diesen Teil der Umformung lässt sich ein Terminierungsmaß als Funktion $\tau_1 : For_S \to \mathbb{N}$ definieren. Ein solches τ_1 wiederum lässt sich am besten per struktureller Induktion definieren:

Es sei $\tau_1 : For_S \to \mathbb{N}$ definiert durch:

- *Anfang: wenn $F \in S$ dann sei $\tau_1(F) = 0$.*
- *Wenn $\tau_1(G) = n_g$ dann sei $\tau_1(\neg G) = \tau_1(G) + \sigma(G)$.*
- *Wenn $\tau_1(F) = n_f$ und $\tau_1(G) = n_g$ dann sei $\tau_1((F \vee G)) = n_f + n_g$ und $\tau_1((F \wedge G)) = n_f + n_g$.*

Die obige Definition nimmt Bezug auf die Hilfsfunktion σ, die auch strukturell für die Formelmenge definiert sei:

Es sei $\sigma : For_S \to \mathbb{N}$ definiert durch:

- *Anfang: wenn $F \in S$ dann sei $\sigma(F) = 0$.*
- *Wenn $\sigma(F) = n_f$ und $\sigma(G) = n_g$ dann sei $\sigma((F \vee G)) = 1 + n_f + n_g$, $\sigma((F \wedge G)) = 1 + n_f + n_g$ und $\sigma(\neg F) = 1 + n_f$.*

Die Ersetzungregeln im ersten Teil des Algorithmus minimiert die Formeln im Terminierungsmaß τ_1. Es gibt drei Regeln:

- $\tau_1(\neg\neg A) = \tau_1(\neg A) + \sigma(\neg A) = \tau_1(A) + \sigma(A) + \sigma(\neg A) = \tau_1(A) + 2 * \sigma(A) + 1 > \tau_1(A)$.
- $\tau_1(\neg(A \vee B)) = \tau_1(A \vee B) + \sigma(A \vee B) = \tau_1(A \vee B) + \sigma(A) + \sigma(B) + 1 = \tau_1(A) + \tau_1(B) + \sigma(A) + \sigma(B) + 1 >= \tau_1(A) + \sigma(A) + \tau_1(B) + \sigma(B) = \tau_1(\neg A) + \tau_1(\neg B) = \tau_1((\neg A \wedge \neg B))$
- $\tau_1(\neg(A \wedge B))$ *ist analog zum zweiten Fall.*

Im zweiten Teil des Algorithmus werden durch Ersetzungen die $\vee$-Symbole über die $\wedge$-Symbole geschoben. Hierzu sei ein weiteres Terminierungsmaß definirt: TODO

Resolution

Der Resolutionskalkül bedient sich eines beweistechnischen Tricks, der auch in der Mathematik gerne angewendet wird. Er realisiert einen Widerspruchsbeweis. Hierbei wird die zu beweisende Aussage zunächst negiert. Wenn die Annahme des Gegenteils schließlich zu einem Widerspruch führt, so war diese negierte Annahme falsch und das Gegenteil ist korrekt.

Der Grundschritt einer Ableitung im Resolutionskalkül ist das Bilden einer neuen Klausel aus zwei bestehenden Klauseln. Dieses ist der Rolutionsschritt. Die neue Klausel heißt *Resolvente* und wird der Klauselmenge hinzugefügt.

Definition 2.1.38 (Resolutionsschritt, Resolvente)
Seien C_1 und C_2 zwei Klauseln, so dass es eine aussagenlogische Variabel A gibt, mit $A \in C_1$ und $\neg A \in C_2$. Dann kann in einem Resolutionsschritt aus C_1 und C_2 die Resolvente C_3 gebildet werden mit:
$C_3 = \{x | x \in C_1, x \neq A\} \cup \{x | x \in C_2, x \neq \neg A\}$

Für einen Resolutionsbeweis wird die zu beweisende Formel negiert und in Klauselnormalform umgeformt. Zusätzlich werden alle Formeln des angenommenen Basiswissens in Klauselnormalform ungeformt. Auf der enstehenden Klauselmenge werden so lange Resolventen gebildet, bis die leere Klausel entstanden ist.

Definition 2.1.39 (Resolution)
Sei $M \subset For_S$ und $A \in For_S$. Aus M sei A per Resolution ableitbar im Zeichen $M \vdash_R A$ genau dann wenn:
Aus der Klauselmenge, die durch Umformung in die Klauselnormalform für die Formeln der Menge M und für die Formel $\neg A$ entsteht, nach endlich vielen Resolutionsschritten die leere Klausel resolviert werden kann.

Beispiel 2.1.40 *In diesem Beispiel soll per Resolution zum Beweis die Allgemeingültigkeit der Formel: $((\neg A \to B) \to C) \to ((\neg B \to A) \to C)$ gezeigt werden. Der enstsprechende Beweis ist in Abbildung 2.4 zu finden. Der entsprechende Beweis wurde automatisch mit der nachfolgenden Implementierung erzeugt.*

Als Tautologie zu beweisende Formel:
$(((\neg A \to B) \to C) \to ((\neg B \to A) \to C))$
Negat der Formel:
$\neg(((\neg A \to B) \to C) \to ((\neg B \to A) \to C))$
Elemination der Pfeiloperatoren:
$\neg(\neg(\neg(\neg\neg A \vee B) \vee C) \vee (\neg(\neg\neg B \vee A) \vee C))$
Negation direkt vor die Atome:
$(((\neg A \wedge \neg B) \vee C) \wedge ((B \vee A) \wedge \neg C))$
Klauselform:
$(((C \vee \neg A) \wedge (C \vee \neg B)) \wedge ((B \vee A) \wedge \neg C))$
Klauselmenge:
$\{\{C, \neg A\}, \{C, \neg B\}, \{B, A\}, \{\neg C\}\}$
Beweis:
$\{B, A\}$
$\{C, \neg A\}$
$\{\neg C\}$
$\{\neg A\}$
$\{B\}$
$\{C, \neg B\}$
$\{\neg C\}$
$\{\neg B\}$
$\square$

Abbildung 2.4: Resolutionsbeweis

Der aufmerksame Student hat dabei sicher schon bemerkt, dass der Resolutionskalkül bereits im ersten Kapitel dieses Skripts beschrieben war. Die Anleitung, wie mit chinesischen Buchstaben auf einem Übungsblatt zu verfahren sei, stellte bereits den Resolutionskalkül dar.

Es ist nun an der Zeit auch einen Beweis im Resolutionskalkül zu führen. Hierzu erinnern wir uns an die anfänglich verklausuliert gemachten Aussagen über Ernies Gedichte.

Beispiel 2.1.41 *In den fünf Aussagen über Gedichte tauchen sechs verschiedene Arten von Gedichten auf. Für jede dieser Gedichtarten sei eine aussagenlogische Variabel vorgesehen, so dass die folgende Signatur erhalten wird:*
$S = \{Modern, Beliebt, Interessant, Schwülstig, Ernies, Seifenblasen\}$

Mit dieser Signatur, lassen sich recht schnell die fünf Aussagen in aussagenlogische Formeln ausdrücken:

- $Interessant \to \neg\neg\ Beliebt$
- $Modern \to Schwülstig$

- *Ernies* $\rightarrow$ *Seifenblasen*
- *Schwülstig* $\rightarrow \neg$ *Beliebt*
- $\neg$ *Modern* $\rightarrow \neg$ *Seifenblasen*

Diese fünf Aussagen sind die Formeln der Menge M. Wir wollen jetzt beweisen, dass Ernies Gedichte nicht interessant sind, also den Satz:
Ernies $\rightarrow \neg$ *Interessant*

Hierzu negieren wir den zu beweisenden Satz und erhalten:
$\neg$*(Ernies* $\rightarrow \neg$ *Interessant)*

Nun sind zunächst die sechs Aussagen in Klauselnormalform umzuformen. Wir erhalten die folgenden Formeln:

- $\neg$ *Interessant* $\vee$ *Beliebt*
- $\neg$ *Modern* $\vee$ *Schwülstig*
- $\neg$ *Ernies* $\vee$ *Seifenblasen*
- $\neg$ *Schwülstig* $\vee \neg$ *Beliebt*
- *Modern* $\vee \neg$ *Seifenblasen*
- *Ernies wedge Interessant*

Wir erhalten die sieben folgenden Klauseln

$$\{\neg Interessant, Beliebt\} \tag{2.15}$$
$$\{\neg Modern, Schwülstig\} \tag{2.16}$$
$$\{\neg Ernies, Seifenblasen\} \tag{2.17}$$
$$\{\neg Schwülstig, \neg Beliebt\} \tag{2.18}$$
$$\{Modern, \neg Seifenblasen\} \tag{2.19}$$
$$\{Ernies\} \tag{2.20}$$
$$\{Interessant\} \tag{2.21}$$

Es lassen sich die folgende neue Klausen resolvieren:

$$\{Beliebt\}, \textit{ aus 2.15 mit 2.21} \tag{2.22}$$
$$\{\neg Schwülstig\}, \textit{ aus 2.18 mit 2.22} \tag{2.23}$$
$$\{\neg Modern\}, \textit{ aus 2.16 mit 2.23} \tag{2.24}$$
$$\{\neg Seifenblasen\}, \textit{ aus 2.19 mit 2.24} \tag{2.25}$$
$$\{\neg Ernies\}, \textit{ aus 2.17 mit 2.25} \tag{2.26}$$
$$\{\}, \textit{ aus 2.17 mit 2.20} \tag{2.27}$$

Damit konnte die leere Klausel durch Resolution erzeugt werden. Die Annahme, dass nicht folgt, Ernies Gedichte seien interessant führen zum Widerspruch und muss fallen gelassen werden. Wir haben bewiesen, dass aus den fünf Aussgane folgt, dass Ernies Gedichte sind uninteressant.

Aufgabe 11 Beweise mittels des Resolutionskalküls, dass die drei Axiome des Kalküls des natürlichen Schließens Tautologien sind.

Wir werden in diesem Skript keine Korrektheitsaussagen und Vollständigkeitsaussagen über den Resolutionskalkül beweisen. Interessierte Studenten seinen an das Skript von Herrn Schmidt-Schauß verwiesen[SS06].

Dem aufmerksamen Leser wird aufgefallen sein, dass der hier beschriebene Resolutionsalgorithmus große Freiheitsgrade aufweist. Es dürfen beliebige Klauseln gewählt werden, um eine Resolvente zu bilden. Damit sind wir wieder mitten in einem Suchproblem gelandet. Gesucht wird die leere Klausel. Die Knotenmarkierungen des Suchbaums sind Klauselmengen. Ein Suchschritt ist, eine Resolvente zu bilden.

Implementierung der Resolution

Auch der Resolutionskalkül soll implementiert werden. Die Datenstuktur, auf der die Resolution arbeitet sind Mengen von Klauseln. Eine Klausel ist eine Menge von Literalen. Um den Kalkül vereinfacht zu implementieren, sie eine Klausel aus einem Paar von zwei Listen dargestellt. Die erste Liste repräsentiert die nicht-negierten Literale, die zweite die negierten:

PropositionalLogic.hs

```
163 type Clause a = ([a],[a])
164 type Clauses a = [Clause a]
```

Um Resolutionsbeweise auszudrucken, sei für Klauseln auch eine Instanz der Klasse `ToLatex` implementiert:

PropositionalLogic.hs

```
165 instance ToLatex (Clause String) where
166  toLatex ([],[])  = "\\Box"
167  toLatex (pos,neg)= "\\{"++ concatWith ", " (pos++map ((++)"\\neg ") neg)
168                          ++ "\\}"
169
170 instance ToLatex (Clauses String) where
171   toLatex cs = "\\{"++ concatWith ", " (map toLatex  cs)++"\\}"
172
173 concatWith sep [] = ""
174 concatWith sep [a] = a
175 concatWith sep (x:xs) = x++sep++concatWith sep xs
```

Der Operator `+++` sei benutzt, um zwei Klauseln zu vereinigen:

PropositionalLogic.hs

```
176 (+++) (a1,a2) (b1,b2) = (nub (a1++b1),nub (a2++b2))
```

Erstellen der konjunktiven Normalform Bevor es an die Bildung von Resolventen geht, sind die Formeln in die konjunktive Normalform umzuwandeln. Hierzu sind vier Schritte notwendig:

- Elimination der Pfeiloperatoren.
- Die Negationen sind vor die Atome zu schieben.

- Disjunktionen sind unter Konjunktionen zu schieben.
- Die enstandene konjunktive Normalform als Klausel zu schreiben.

Die Pfeiloperatoren sind bei jedem Auftreten innerhalb der Formel durch Konjunktion und Disjunktionen zu ersetzen:

PropositionalLogic.hs

```
177 noArrows :: Prop -> Prop
178 noArrows = everywhere (mkT noA) where
179   noA (a :-> b) = Not a :\/ b
180   noA (a :<-> b) = (Not a :\/ b) :/\ (Not b :\/ a)
181   noA x = x
```

Anschließend lassen sich doppelte Negationen überall eliminieren und über Konjunktionen und Disjunktionen schieben:

PropositionalLogic.hs

```
182 notInside :: Prop -> Prop
183 notInside = (everywhere (mkT noI)) where
184    noI (Not (Not p))    = p
185    noI (Not (p1:\/p2))  = (noI (Not p1):/\ noI(Not p2))
186    noI (Not (p1:/\p2))  = (noI (Not p1):\/ noI(Not p2))
187    noI x = x
```

Dann sind Disjunktionen überall innerhalb der Formel unterhalb der Konjunktionen zu schieben.

PropositionalLogic.hs

```
188 clauselForm = everywhere$mkT cf where
189      cf (a:\/(b:/\c))    = clauselForm ((a:\/b) :/\ (a:\/c))
190      cf ((b:/\c):\/a)    = cf (a:\/(b:/\c))
191      cf x                = x
```

Und schließlich sind die entstandenen Formeln in der konjunktive Normalform als Klauseln auszudrücken

PropositionalLogic.hs

```
192 clauses = cf  where
193      cf (Atom x)       = [([x],[])]
194      cf (Not (Atom x)) = [([],[x])]
195      cf (a:/\b)          = clauses a++clauses b
196      cf (a:\/b)          = [ca+++cb|ca<-clauses a,cb<-clauses b]
```

Die Verkettung dieser vier Funktionen ergibt den Gesamtprozess zum Erstellen der Klauselform.

PropositionalLogic.hs

```
197 inCls=clauselForm.notInside.noArrows
198 mkClauses = clauses.inCls
```

Resolventenbildung Nachdem nun die Formeln in Klauselform umgewandelt werden können, gilt es aus zwei Klauseln per Resolutionsschritt eine Resolvente zu bilden. Die Kernfunktion nimmt hierzu zwei Klauseln und erzeugt eine Liste von Resolventen aus den zwei Argumentklauseln. Ist die Ergebnisliste leer, dann konnte keine Resolvente gebildet werden.

PropositionalLogic.hs
```
199 resolve c1@(p1,n1) c2@(p2,n2)
```

In einem ersten Fall wird verhindert, dass eine Klausel mit sich selbst resolviert wird:

PropositionalLogic.hs
```
200  |c1==c2 =[]
```

Ansonsten sind für alle positiven Literale der ersten Klausel (`l<-p1`), die als negatives Literal in der zweiten Klausel auftreten (`elem l n2`) Resolventen zu bilden. Dieses geschieht durch die Vereinigung der jeweiligen positiven und negativen Literallisten aus beiden Klauseln, ohne das entsprechende Literal über das resolviert wurde.

Duplikate werden aus den Ergebnisliteralmenge durch die Standardlistenfunktion `nub` entfernt.

PropositionalLogic.hs
```
201  |otherwise = [(nub((p1\\[l])++p2),nub(n1++(n2\\[l])))|l<-p1,elem l n2]
```

Die Funktion `resolve` erzeugt somit nicht alle Resolventen der beiden Argumentklauseln, sondern nur die Resolventen, die aus einem positiven Literal der ersten Klausel und einem negativen der zweiten gebildet wurde. Alle Resolventen können erzeugt werden wenn die Funktion zweimal aufgerufen wird: `resolve c1 c2 ++ resolve c2 c1`

Es sollen für zwei Klauselmengen sämtliche Resolventen gebildet werden. Allgemein sollen alle Paare, die sich aus einer Listen bilden lassen, durch die Resolutionsfunktion verknüpft werden. Hierbei kann folgende kleine Hilfsfunktion benutzt werden:

PropositionalLogic.hs
```
202 breadth f xs =  [f x y|x<-xs,y<-xs]
```

Mit deren Hilfe lässt sixh dann aus einer Klauselmenge eine neue Klauselmenge definieren, die alle möglichen Resolventen enthält:

PropositionalLogic.hs
```
203 allresolvents xs = xs++(concat$ breadth resolve xs)
```

Klauseln, die sowohl ein Literal negiert als auch unnegiert enthalten sind Tautologien und können nicht zum Widerspruchsbeweis beitragen. Die folgende kleine Funktion testet, ob eine Klausel eine Tautologie ist:

PropositionalLogic.hs
```
204 tautologyClause (pos,neg) = or [at1==at2|at1<-pos,at2<-neg]
```

Die Beweisfunktion schließlich, entscheidet, ob die leere Klausel aus einer Klauselmenge hergeleitet werden kann. Hierzu wird eine Formel in Klauselform gebracht und alle Tautologieklauseln aus der entsprechenden Klauselmenge gefiltert. Dann werden sukzessive immer wieder alle Resolventen gebildet. Mit der Standardlistenfunktion `elem` wird geprüft ob die leere Klausel in der Liste aller Resolventen enthalten ist. Das Ergebnis ist ein bool'scher Wert, der angibt, ob der Beweis geglückt ist.

PropositionalLogic.hs
```
205 proof p = elem ([],[])(concat(iterate allresolvents
206                       (filter (not.tautologyClause)$mkClauses p)))
```

Beweise Ausdrucken Die acht Zeilen Haskell im vorherigen Abschnitt sind ein kompletter Reolutionsbeweiser für die Aussagenlogik. Der Beweiser hat leider nur ein Ja/Nein-Ergebnis. Er zeigt nicht an, wie im Erfolgsfall die leere Klausel hergeleitet werden konnte. Deshalb folgt hier eine zweite Implementierung, die als Ergebnis im Erfolgsfall den Beweis angibt.

Der Beweis besteht aus einer Kette von Resolutionen, die schließlich zur leeren Klausel führen. Der eine Resolvente immer aus zwei Klauseln gebildet wird, lässt sich der Beweis als Binärbaum darstellen. An der Wurzel steht die leere Klausel. Die Kinder eines Knotens sind die beiden Klauseln, die per Resolution den Elternknoten erzeugt haben. An den Blättern des Beweisbaumes stehen Klauseln aus der ursprünglichen Klauselmende.

Die entsprechende Baumstruktur lässt sich für einen Binärbaum in Haskell wie folgt definieren:

PropositionalLogic.hs

```
207 data T a = TN|TC (T a) a (T a) deriving (Eq, Show,Data,Typeable)
```

Die Reimplementierung der obigen Resolutionsfunktion agiert nun nicht nur auf den Klauseln, sondern auch auf den Beweisbäumen, die zu den Klauseln geführt haben:

PropositionalLogic.hs

```
208 resolveT (c1@(p1,n1),t1) (c2@(p2,n2),t2)
209  = [(x,TC t1 x t2)|l<-p1
210                   ,elem l n2
211                   ,x<-[(nub((p1\\[l])++p2),nub(n1++(n2\\[l])))]]
```

Die Beweisfunktionen sind für die Beweisbäume fast wörtlich die aus dem vorherigen Abschnitt:

PropositionalLogic.hs

```
212 allresolventsT xs = xs++(concat$ breadth resolveT xs)
213
214 proofs p = filter (\(x,_) -> x==([],[]))
215      (concat(iterate allresolventsT (map (\x->(x,TC TN x TN)) p)))
```

Die folgende Funktion zweigt schließlich den Beweis in einer LaTeX-Darstellung:

PropositionalLogic.hs

```
216 showProof TN = ""
217 showProof (TC TN x TN) = "$"++toLatex x++"$"
218 showProof (TC l x r)
219   =   showProof l ++"\\\\\n"
220     ++showProof r ++"\\\\\n"
221     ++"$"++toLatex x++"$"
222
223 moin e = writeFile "test"$showProof$snd$head$proofs$mkClauses$Not e
```

Aufgabe 12 Formalisieren Sie die folgenden Aussagen in der Aussagenlogik und beweisen Sie mit dem Resolutionskalkül, dass Kleinkinder nicht mit Krokodilen umgehen können.

1. Kleinkinder sind nicht logisch.
2. Niemand wird nicht geachtet, der mit Krokodilen umgehen kann.
3. Unlogische Personen werden nicht geachtet.

Aufgabe 13 Formalisieren Sie die folgenden Aussagen in der Aussagenlogik und beweisen Sie mit dem Resolutionskalkül, dass alle Pfandleiher ehrlich sind.

1. Leuten, die Versprechen nicht einhalten, kann man nicht trauen.
2. Weintrinker sind sehr kommunikative Menschen.
3. Jemand, der seine Versprecher einhält, ist ehrlich.
4. Kein Weinabstinenzler ist Pfandleiher.
5. Kommunikative Menschen kann man prinzipiell trauen.

2.1.5 Der Tableaukalkül

Wir haben bereits zwei Kalküle für die Aussagenlogik kennengelernt. [3] Während der Kalkül des logischen Schließens denkbar ungeeignet war um auf systematische Weise den Beweis für eine bestimmte Formel zu finden, so hat der Resolutionskalkül die Formeln in eine Mengenschreibweise gebracht, die die Formelstuktur vollkommen zerstört hat. Der eigentliche Beweis besteht aus einer Abfolge von Mengen und lässt wenig Rückschlüsse über die ursprünglich gefundene Formel zu. Zum Abschluss des Kapitels über die Aussagenlogik sei ein weiterer Kalkül vorgestellt, der Tableaukalkül. Er wurde erstmals von Raymond M. Smullyan vorgestell[Smu68].

Dieser Kalkül ist ein Kalkül, der sich binärer Bäume bedient. Die Baumknoten sind mit Formeln markiert. Die Pfade entlang des Baumes entsprechen der Suche nach Interpretationen, die die Formel an der Wurzel erfüllen. Wenn für jeden Pfad zu einem Blatt bewiesen werden kann, dass auf diesem Pfad keine Interpration gefunden werden kann, dann ist die Formel an der Wurzel unerfüllbar. Somit kann der Tableaukalkül auch als Widerlegungskalkül benutzt werden. Ein Tableu für das Negat der als allgemeingültig zu beweisenden Formel wird aufgestelt. Wenn dieses Tableau zeigen kann, dass die negierte Formel unerfüllbar war, dann ist die unnegierte Formel eine Tautologie.

Nach dieser Vorrede sollen nun die Begriffe formal definiert werden:

Definition 2.1.42 (Tableaux)
Ein Tableau ist ein Binärbaum, dessen Knoten mit Formeln aus For_S markiert sind.

Ein Pfad von der Wurzel zu einem Blatt des Tableau heisst geschlossen, wenn es auf diesem Pfad ein Atom A gibt, so dass auch $\neg A$ auf diesem Pfad liegt.

Ein Tableau heisst geschlossen, wenn alle Pfade zu einem Blatt geschlossen sind.

Blätter, an denen ein Tableau geschlossen ist, werden mit dem zusätzlichen Kind $\square$ gekennzeichnet.

Natürlich soll ein Tableau nicht aus beliebigen Formelmarkierungen bestehen. Die Kinder eines Knotens sollen systematisch die Formel der Wurzel dekonstruieren. Hierzu gibt es Erweiterungsregeln.

Definition 2.1.43 (Erweiterungsregeln)
Ein Tableau kann nach zwei Arten von Regelschemata erweitert werden:

[3] Betrachten wir das Aufstellen von Wahrheitstafeln als Kalkül, dann waren es sogar schon drei Kalküle.

- **Und-Regeln (auch als α-Regeln bezeichnet)**:

$$\begin{array}{cccc} (A \wedge B) & \neg(A \vee B) & \neg(A \to B) & \neg\neg A \\ | & | & | & | \\ A & \neg A & A & A \\ | & | & | & \\ B & \neg B & \neg B & \end{array}$$

- **Oder-Regeln (auch als β-Regeln bezeichnet)**:

$$\begin{array}{ccc} (A \vee B) & \neg(A \wedge B) & (A \to B) \\ \diagup\ \diagdown & \diagup\ \diagdown & \diagup\ \diagdown \\ A \quad B & \neg A \quad \neg B & \neg A \quad B \end{array}$$

Ein Tableau kann erweitert werden, indem für einen Knoten, der die Form einer der Wurzeln der obigen Regeln hat, an alle Blätter, die von diesem Knoten erreichbar sind, die entsprechenden Kinder aus der benutzten Regel angehängt werden.

Wir betrachten nun nur noch Tableaux, die durch sukzessive Anwendung der Regeln von einem Wurzelknoten aus aufgebaut wurden.

Ein Tableau heisst komplett[4]*, wenn auf allen Pfaden, von der Wurzel zu einem Blatt, die nicht geschlossen sind, bereits alle möglichen Erweiterungen angewendet wurden.*

Beispiel 2.1.44 *In diesem Beispiel soll ein komplettes Tableau zum Beweis der Allgemeingültigkeit der Formel:* $((\neg A \to B) \to C) \to ((\neg B \to A) \to C)$ *aufgestellt werden. Hierzu wird die Formel negiert und an die Wurzel eines Tableaus gestellt. Systematisch wird das Tableau expandiert. Anschließend wird geschaut, ob das Tableau an allen Blättern geschlossen werden kann. Das entstandene Tableau ist in Abbildung 2.5 zu finden.*

Der entsprechende Beweis wurde automatisch mit der nachfolgenden Implementierung generiert.

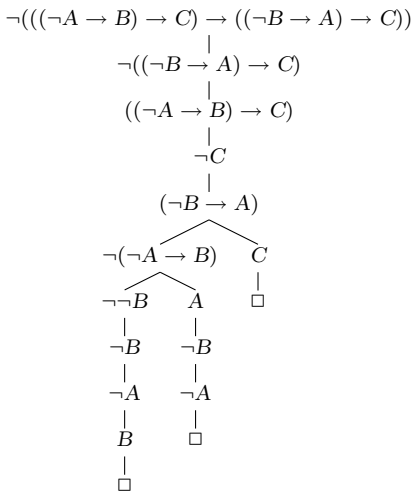

Abbildung 2.5: Komplettes geschlossenes Tableau

[4]Eigentlich ist hier der Begriff *vollständig* auf Deutsch auch gebräuchlich. Dieser kollidiert aber abermals mit dem Begriff der Vollständigkeit eines Kalküls bezüglich einer Semantik, deshalb sei hier der Bergriff *komplett* gewählt.

Implementierung des Tableaukalküls

Überraschender Weise ist, obwohl der Tableaukalkül die meiste Strukturinformation und die eleganteste graphische Darstellung hat, dieser am einfachsten kürzesten von allen betrachteten Kalkülen implementierbar.

Ein Tableau ist eine Baumstruktur. Ein Baumknoten habe:

- eine Formel als Knotenmarkierung.
- eine Liste der Kinderknoten.
- eine Liste der Vorfahren. In dieser sei das erste Element die Markierung des Elternknoten, das zweite Element die Markierung des Großelternknotens, das dritte Element die Markierung des Urgroßelternknotens usw. Es entspricht also dem umgekehrten Pfad vom Knoten zur Wurzel.

Der Datentyp für Tableaux, hat demnach einen Konstruktor für Knoten mit drei Argumenten:

PropositionalLogic.hs

```
224 data Tableau = Node Prop [Tableau] [Prop]
225      deriving (Eq,Show,Typeable,Data)
```

Eine wichtige Information, die zum Beweis eine Rolle spielt, ist die Frage, ob ein Tableau geschlossen ist. Wenn der entsprechende Tableauknoten ein Blatt ist, dann ist das Tableau geschlossen, wenn auf dem Pfad von der Wurzel zu diesem Blatt ein Widerspruch zu finden ist. Dieser ist dadurch charakterisiert, dass ein Atom und dessen Negat auf dem Pfad zu finden ist. Der Pfad von der Wurzel zu diesem Blatt ist in den Vorfahren zu finden.

PropositionalLogic.hs

```
226 closed (Node f [] ancestors) = contradiction (f:ancestors)
```

Handelt es sich bei dem Tableauknoten nicht um ein Blatt, so ist das entsprechende Tableau geschlossen, wenn alle seine Kinder geschlossen sind:

PropositionalLogic.hs

```
227 closed (Node f children _) = and(map closed children)
```

Es bleibt die Definition eines Widerspruchs auf einem Pfad zu implementieren. Hierzu muss eines der Atome auf dem Pfad auch in negierter Form auf dem Pfad enthalten sein. Dieses drückt die folgende Funktionsdefinition aus:

PropositionalLogic.hs

```
228 contradiction xs = or[elem (Not x) xs|x<-xs,isAtom x]
```

Schließlich sind Tableaux zu erweitern. Bei der Erweiterung eines Tableaus werden an bestimmten Knoten zusätzliche Kinder angehängt. Die folgende Funktion fügt der Kinderliste weitere Kinder hinzu. Dabei ist darauf zu achten, dass für die neuen Kinder, die Vorfahren korrekt gesetzt werden:

PropositionalLogic.hs

```
229 addChildren xs (Node f children ancestors)
230  = Node f ([(Node x [] (f:ancestors))|x<-xs]++children) ancestors
```

Zum Erweitern der Tableaux werden neue Kinder an alle erreichbaren Blätter eines Knotens angehängt, und nicht direkt an den Knoten selbst, wie es die vorherige Funktion bewerkstelligt. Daher folgt die Definition einer Funktion, die zusätzliche Knoten an alle erreichbaren Blätter als Kinder anhängt.

Ist der fragliche Knoten selbst ein Blatt, dann sind die Knoten direkt diesem als Kinder zuzufügen, allerdings nur, wenn es sich nicht bereits um einen geschlossenen Pfad handelt:

PropositionalLogic.hs

```
231 addToLeaves xs t@(Node f [] ancestors)
232  |closed t  = t
233  |otherwise =  addChildren xs t
```

Handelt es sich hingegen nicht um ein Blatt, so sind die Knoten an alle Blätter, die von einem der Kinderknoten erreichbar sind anzuhängen:

PropositionalLogic.hs

```
234 addToLeaves xs (Node f children ancestors)
235   = Node f (map (addToLeaves xs) children) ancestors
```

Mit diesem Möglichkeiten der Erweiterung an den Blättern, lassen sich die Regeln zur Tableauerweiterung direkt ausdrücken:

Und-Regeln Bei der Anwendung der vier Und-Regeln sind die Pfade jeweils um zwei Knoten zu verlängern (bzw. um einen Knoten bei der Regel zur doppelten Negation). Dieses mündet in zweifachen Aufruf der Funktion `addToLeaves` mit jeweils einer einelementigen Liste:

PropositionalLogic.hs

```
236 expand t@(Node (Not(Not a)) children ancestors)
237  = addToLeaves [a] t
238 expand t@(Node (a:/\b) children ancestors)
239  = addToLeaves [b]$ addToLeaves [a] t
240 expand t@(Node (Not (a:\/b)) children ancestors)
241  = addToLeaves [Not b]$ addToLeaves [Not a] t
242 expand t@(Node (Not (a:->b)) children ancestors)
243  = addToLeaves [a]$ addToLeaves [Not b] t
```

Oder-Regeln Bei den drei Oder-Regeln sind an alle Blätter zwei neue Kinder anzuhängen. Dieses mündet in einem einmaligen Aufruf der Funktion `addToLeaves`, dieses mal aber mit der zweielementigen Liste der beiden neuen Kinder.

PropositionalLogic.hs

```
244 expand t@(Node (a:\/b) children ancestors)
245  = addToLeaves [a,b] t
246 expand t@(Node (Not (a:/\b)) children ancestors)
247  = addToLeaves [Not a,Not b] t
248 expand t@(Node (a:->b) children ancestors)
249  = addToLeaves [Not a,b] t
```

Handelt es sich bei dem zu expandierenden Knoten um ein Literal, so ist keine Regel anwendbar und der Knoten bleibt unverändert:

PropositionalLogic.hs

```
250 expand t = t
```

Um einen Tablaux komplett zu erweitern, ist systematisch auf alle Baumknoten die Erweiterung anzuwenden. Dieses bedeutet, dass nach der Expandierung des Wurzelknotens alle Kindknoten zu expandieren sind:

PropositionalLogic.hs

```
251 expandAll = expandChildren.expand
```

Zur Expansion der Kindknoten eines Knotens, ist die Funktion `expandAll` rekursiv auf alle Kinder anzuwenden:

PropositionalLogic.hs

```
252 expandChildren (Node f children ancestors)
253  = Node f (map expandAll children) ancestors
```

Ein komplettes Tableau für eine Formel ist ein komplett expandierter Tableau für die Formel an der Wurzel des Tableaus:

PropositionalLogic.hs

```
254 tableau f = expandAll (Node f [] [])
```

Ein Tableaubeweis ist ein Widerspruchsbeweis, so dass nach der Negation der Formel für diese ein komplettes Tableau aufzubauen ist:

PropositionalLogic.hs

```
255 tableauProof = tableau.Not
```

Eine Formel ist durch ein Tableau als allgemeingültig bewiesen, wenn das Tableau des Negats der Fromel geschlossen werden kann:

PropositionalLogic.hs

```
256 isTableauProven = closed.tableauProof
```

LaTeX-Darstellung von Tableaus Mit dem Paket `qtree` lassen sich in LaTeX auf einfache Weise Bäume darstellen. Die folgende Instanz der Klasse `ToLatex` benutzt dieses Paket um LaTeX-Code zur Darstellung eines Tableaus zu erzeugen.

Alle aussagenlogische Tableaux in diesem Skript sind mit Hilfe dieser Funktion erzeugt worden.

PropositionalLogic.hs

```
257 instance ToLatex Tableau where
258     toLatex t
259       = "\\Tree "++aux t++""
260       where
261         aux (Node f [] ancestors)
262          |contradiction (f:ancestors)
263            = "[ .\\mbox{$"++toLatex f++"$}\n \\mbox{$\\Box$}\n] "
264          |otherwise = " \\mbox{$"++toLatex f++"$} "
265         aux (Node f cs ancestors)
266           = "[ .\\mbox{$"++toLatex f++"$}\n "++concat(map aux cs)++" ]"
267
```

2.1.6 Bert Bresgen: Axiome der Liebe

Bevor wir uns im nächsten Abschnitt der Prädikatenlogik widmen, soll der Autor Bert Bresgen noch einmal zu Wort kommen, und ein paar Gedanken zur Logik aus Sicht eines Nicht-Informatikers verlieren:

Bert: Manchmal am Ende eines langen Tages beginnen Mathematiker zu träumen. Sie sind es leid, immer irgendjemand irgendetwas beweisen zu müssen. Sie träumen von Sätzen und Formeln, die unmittelbar wahr sind, ohne dass man sie beweisen muss. Solche Sätze nennt man Axiome.

Bereits Aristoteles träumte davon und er fand drei dieser Sätze. Der erste Satz lautet:

- $A = A$, d.h. etwas, das mit sich selbst nicht identisch ist, kann nicht gedacht werden.

Was bedeutet das in Hinsicht auf die Liebe?

Ich liebe A. Aber wer ist A? A ist ... gleich A. Deshalb liebe ich A. Allerdings macht es sich A ein bisschen leicht damit, einfach A zu sein. Ich liebe A, weil A A ist, aber natürlich liebe ich nicht alles an A. Vielleicht würde ich sogar B noch mehr lieben als A, wenn ... B nur so wäre wie A.

Ein Lied der Lassie Singers stellt dieses Problem dar. *(Lied wird eingespielt)*

Die beiden anderen Axiome des Aristoteles, die nicht bewiesen werden müssen, lauten:

- A kann nicht zugleich nicht A sein, und
- Alles, was ist, ist entweder A oder nicht A. Dies ist der Satz vom ausgeschlossenen Dritten.

Der Satz vom ausgeschlossenen Dritten bedeutet im Zustand akuter Verliebtheit z.B.:
Das ist Andrea. Ich weiss, dass das Andrea ist. Alles was ist, ist entweder Andrea oder es ist Nicht–ANDREA. Nicht-Andrea ist zum Beispiel Matthias. Oder Birgit. Matthias oder Birgit sind unwichtig, denn sie sind nicht: Andrea. Andrea ist die Welt für mich. Komischerweise ist trotzdem alles, was ist, Nicht–Andrea, mit Ausnahme Andreas. Meine Arbeit, der Schmutz unter meinen Fingernägeln oder die sixtinische Kapelle sind nicht Andrea. Nicht–Andrea ist die neue Strassenbahn der Linie 17, die bis zum Rebstockbad fährt. Es ist sinnlos, mit Birgit in der Linie 17 zum Rebstockbad zu fahren, wo sie mir ihren neuen Badeanzug zeigt, denn Birgit ist nicht Andrea. Ich sage zu Birgit "Was für ein schöner Tag, Birgit. Was für ein schöner, neuer Badeanzug, Birgit. Er erinnert mich an Andreas Badeanzug. Ich weiss nicht, ob ich das schon gesagt habe, Birgit, aber: Andrea schwimmt die 50 Meter Bahn ohne Badeanzug in 7 Stunden. Und während ich ertrinke, schmiert sie diese fabelhaften Clubsandwiches. Möchtest Du Thunfisch oder Huhn, Birgit?"

Draußen dämmert frühes Licht.

Die drei nicht beweisnotwendigen Sätze des Aristoteles gelten bis zum heutigen Tag.

2.2 Prädikatenlogik

2.2.1 Einführende Beispiele

Der vorangegangene Text von Bert Bresgen läßt schon eine gewisse Unzufriedenheit mit der Atomarität von Aussagen durchschimmern. Die kleinste Einheit in der Aussagenlogik sind aus-

sagenlogische Variablen, die entweder als wahre oder als falsche Aussagen angenommen werden.

Bei dem Versuch mathematische Aussagen in Formeln zu schreiben, um über diese automatisch Beweise zu führen, werden wir häufig mit Problemfeldern und Welten konfrontiert sein, in denen es mehrere verschiedene Individuen gibt, die bestimmten Menge zugehörig sind und in bestimmten Beziehungen zueinander stehen. Derartige Zusammenhänge und Aussagen über verschiedene Individuen lassen sich nicht mit der Aussagenlogik beschreiben. Hierzu bedarf es einer mächtigeren Sprache, die mit der Prädikatenlogik zur Verfügung steht.

Bevor wir in gleicher Weise, wie im Kapitel über die Aussagenlogik, zunächst eine Syntax, anschließend Semantik und Tableaux- und Resolutionskalkül für die Prädikatenlogik definieren, wollen wir uns der Sprache mit ein paar informellen Beispielen annähern, die ein Gefühl für die Ausdrucksstärke geben können.

Betrachten wir hierzu zunächst ein kleines Szenario einer Bauklötzchenwelt In Abbildung 2.6 findet sich eine kleine Skizze von Bauklötzen.

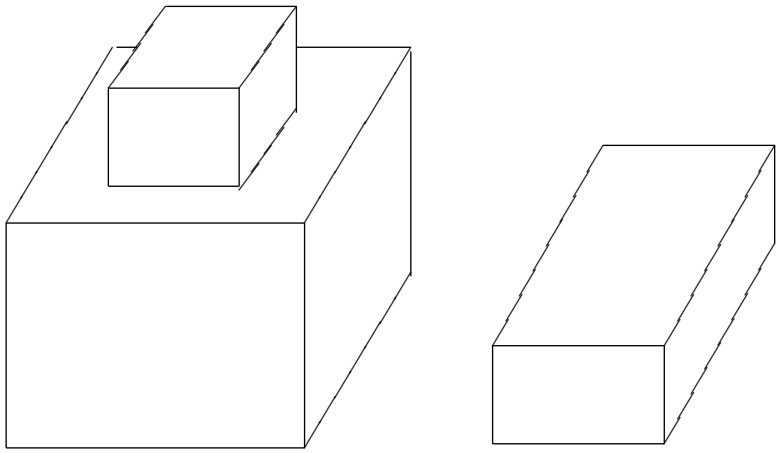

Abbildung 2.6: In Prädikatenlogik zu modellierendes Bauklötzchen Szenario

Es sind hier drei Bauklötze zu erkennen, zwei von ihnen stehen auf dem Boden, ein dritter steht auf einem weiterem Bauklotz. Wenn wir die Bauklötze mit den Namen A, B und C bezeichnen, so lassen sich folgende Fakten über die Anordnung der Bauklötze als Prädikate aufschreiben:

$$Klotz(A)$$

$$Klotz(B)$$

$$Klotz(C)$$

$$StehtAuf(B, A)$$

$$AufBoden(A)$$

$$AufBoden(C)$$

Drei unterschiedliche Prädikate sind in diesen Fakten benutzt. Zwei einstellige Prädikate, die somit eine Mengenzugehörigkeit ausdrücken und ein zweistelliges Prädikat, das eine Relation ausdrückt.

Stellen wir uns vor, wir wollen einen Roboter modellieren, der mit einem Greifarm ausgestattet ist. Er kann Bauklötze greifen, sofern kein weiterer Bauklotz auf ihnen steht. Dieser Zusammenhang läßt sich durch eine Regel ausdrücken.

$$\forall\, x\, (\neg \exists\, y\, StehtAuf(y, x)) \rightarrow\ GreifBar(x)$$

Die Formel ist natürlichsprachlich zu lesen als:
Für alle Individuen x, für die nicht ein Indivuduum y existiert, das auf x steht, folgt, dass das x ergreifbar ist.

Damit ist ausgedrückt, dass alle Individuen (in unserem Fall Bauklötze) ergriffen werden können, wenn kein anderes Individuum existiert, das auf dem fraglichen Individuum steht.

Man hätte das durchaus auch anders ausdrücken können, nämlich durch:

$$\forall\, x\, (\forall\, y\, \neg StehtAuf(y, x)) \rightarrow\ GreifBar(x)$$

Nämlich, die greifbaren Objekte sind die, für die gilt, dass für alle anderen Objekte gezeigt werden kann, dass sie nicht auf dem fraglichen stehen.

Neu gegenüber der Aussagenlogik sind nicht nur die Prädikate, sondern auch die Konstanten, die bestimmte Individuen bezeichnen sollen (hier A, B und C) sowie Variablen, die durch die Quantoren $\forall$ und $\exists$ eingeführt werden können.

Was ist jetzt gegenüber der Aussagenlogik zu beachten, wenn versucht wird eine Schlußregel in der Prädikatenlogik anzuwenden. Hierzu betrachte man eine alt bekannte Aussage über die allgemeine Sterblichkeit und ein spezielles Faktum über ein spezielles Individuum (auch wenn das von Manchen angezweifelt wird):

$$\forall x\ Mensch(x) \rightarrow Sterblich(x)$$
$$Mensch(Elvis)$$

Hier wird eine allgemeine Implikationregel für alle Individuen, die der Menge Mensch zugehörig sind, aufgestellt. Zusätzlich wird von einem bestimmten Element die Zugehörigkeit zu der Menge Mensch als Fakt gegeben. Um jetzt aus diesen beiden Formeln das Fakt abzuleiten, dass selbst Elvis zur Menge der sterblichen Dinge gehört, ist die allgemeine Regel, die für alle Individuen x gilt, auf ein bestimmtes Individuum anzuwenden. Mann könnte auch sagen, dass

$$\forall\, x\, Mensch(x) \rightarrow Sterblich(x)$$

ein Schema ist, in das man für die Variabel x beliebige Individuen einsetzen kann, z.B.:

$$Mensch(Elvis) \rightarrow Sterblich(Elvis)$$

oder auch für andere Individuen:

$$Mensch(Jesus) \rightarrow Sterblich(Jesus)$$

Hat man eine solche Einsetzung für eine Variabel gefunden, lässt sich mittels des Modus Ponens wie in der Aussagenlogik das neue Fakt herleiten. Der Schluß lässt sich analog zur Aussagenlogik schematisch aufschreiben:

$$\frac{\text{Mensch(Elvis)} \quad \text{Mensch(Elvis)} \rightarrow \text{Sterblich(Elvis)}}{\text{Sterblich(Elvis)}}$$

Wie man sieht, wird es notwendig werden, um spezielle Schlüsse zu ziehen, eventuell Variablen durch konkrete Ausdrücke zu ersetzen. Eine solche Ersetzung heißt Substitution. Wie wir sehen werden, spielen Substitutionen eine entscheidene Rolle in Kalkülen der Prädikatenlogik.

Bisher haben wir Variablen, Konstanten und Prädikate als Bestandteile der Prädikatenlogik gesehen. Es gibt noch ein zusätzliches Konzept, die Funktionen. Eine Funktion soll für ein oder mehere Elemente, nämlich für ihre Argumente, ein neues Element liefern. Wir können uns z.B. für das obige Beispiel zusätzlich eine Funktion `vater` vorstellen, die für jedes Element ein Element zurückgibt, das den Vater dieses Elements darstellen soll. Damit ließe sich folgender Sachverhalt formalisieren:

$$\forall\, x\, Mensch(x) \rightarrow Mensch(vater(x))$$

Für jeden Menschen gilt, dass sein Vater auch ein Mensch ist.

So ließe sich also aus dem Fakt `Mensch(Jesus)` mit Modus Ponens ableiten: `Mensch(vater(Jesus))` und in einem weitern Schritt: `Sterblich(vater(Jesus))`.

Zum Abschluß der informellen Einführung der Prädikatenlogik seien noch zwei Aussagen aus bekannten Schlagern formalisiert:

Z. B der alten Dean Martin Schlager *Everybody Loves Somebody Sometime* könnte man wie folgt formalisieren:

$$\forall\, x\, \exists\, y\, \exists\, t\, LovesAtTime(x, y, t)$$

Damit hätten wir auch ein erstes Beispiel eines Prädikats, das mehr als zwei Argumente hat, nämlich eines dreistelligen Prädikats.

Oder es lässt sich die bekannte Tatsache formalisieren, dass fast alles ein Ende hat, nur eben die Wurst nicht (wobei wir außer acht lassen, dass die zwei Enden hat):

$$\forall\, x\, (HatEnde(x) \vee Wurst(x))$$

2.2.2 Syntax

Nachdem schon eine Reihe von Beispielen für prädikatenlogische Formeln gezeigt wurden, wird es notwendig, die Syntax endlich auch formal zu beschreiben. Ebenso wie in der Aussagenlogik werden wir die Syntax nicht über eine kontextfreie Grammatik definieren, sondern eine strukturelle induktive Definition vorziehen.

Die Definition geht in zwei Stufen. Zunächst werden Terme definiert. Terme sind die Ausdrücke, über die die Prädikate Aussagen machen werden. Dann werden die Formeln definiert.

Wir definieren die Prädikatenlogik erster Stufe, kurz PL1.[5]

Zunächst seien ein paar Symbole als ausgezeichnete Sonderzeichen der Sprache PL1 aufgezählt.

[5] Prädikatenlogik höherer Ordnung begegnet man selten. Wir werden uns nicht damit beschäftigen und wann immer wir von der Prädikatenlogik sprechen ist sie erster Stufe gemeint. In höheren Ordnungen kann man auch über Prädikatensymbole quantifizieren.

Definition 2.2.1 (Sonderzeichen)
Die Menge der Sonderzeichen der Prädikatenlogik sei: $\{\neg, \rightarrow, (,), , , \forall\}$. *Das Symbol* $\forall$ *wird als Allquantor bezeichnet.*

Auch hier haben wir die Sprache so klein wie möglich gehalten und in der formalen Definition auf die weiteren Sonderzeichen $\exists, \vee, \wedge$ und $\leftrightarrow$ verzichtet. Diese Symbole werden wir wieder als abkürzende Schreibweisen einführen.

Die Syntax einer Prädikatenlogik setzt sich aus vier Komponenten zusammen. Drei Mengen von Bezeichnern, nämlich Bezeichnern für Variablen, für Funktionen und für Prädikate. Und einer Funktionen, die den Funktions- und Prädikatensymbolen eine Stelligkeit zuordnet.

Definition 2.2.2 (Signatur)
Eine Signatur S *sei ein 4-Tupel* $(I_S, F_S, P_S, \alpha_S)$ *mit:*

- F_S, P_S, I_S *sind abzählbare Mengen.*
- *Die drei Mengen sind gegenseitig disjunkt und enthalten kein Sonderzeichen, also:* $F_S \cap P_S = \emptyset, P_S \cap I_S = \emptyset$, $F_S \cap I_S = \emptyset$, $\{\neg, \rightarrow, (,), , , \forall\} \cap (F_S \cup P_S \cup I_S) = \emptyset$. .
- *Die Funktion* α_S *bildet jedes Funktions- und Prädikatensymbol auf eine natürliche Zahl ab.:*
  $\alpha_S : P_S \cup F_S \rightarrow \mathbb{N}$

I_S wird als die Menge der Individuenvariablen bezeichnet.

P_S heisst die Menge der Prädikatensymbole. Ein $p \in P_S$ mit $\alpha_S(P_S) = n$ wird als n-stelliges Prädikat bezeichnet.

F_S heisst die Menge der Funktionssymbole. Ein $f \in F_S$ mit $\alpha_S(F_S) = n$ wird als n-stellige Funktion bezeichnet. Ist $\alpha_S(f) = 0$ so wird f als Konstante bezeichnet.

Als Konvention werden wir Prädikatensymbole mit einem Großbuchstaben beginnen lassen, und Funktionssymbole mit einem Kleinbuchstaben. Für die Individuenvariablen werden die Buchstaben $\{x, y, z, u, v, \dots\}$ eventuell mit Index verwendet.

Die Definition der Syntax der Prädikatenlogik wird in drei Stufen vorgenommen. Zunächst werden Terme definiert. Dann atomare Formeln, die aus genau einem Prädikat bestehen und schließlich Formeln, die aus atomaren Formeln und logischen Operatoren zusammengesetzt werden. Die Definitionen der Terme und der Formeln erfolgt wieder strukturell induktiv.

Beginnen wir mit den Termen:

Definition 2.2.3 (Terme)
Sei S *eine Signatur. Die Menge* T_S *der Terme sei die kleinste Menge von Wörtern über* $\{(, , ,)\} \cup F_S \cup I_S$ *mit:*

- $I_S \subseteq T_S$.
- *Wenn* $f \in F_S$ *mit* $\alpha_S(f) = n$, $t_1, \dots, t_n \in T_S$ *so ist auch:* $f(t_1, \dots, t_n) \in T_S$.

Damit sind Terme ähnlich aufgebaut, wie wir es aus Programmiersprachen kennen. Variablen und Konstanten sind Terme und Funktionsaufrufe. Die Argumente der Funktionen können dabei wieder beliebige Terme sein.

Die Terme dienen nun als Ausdrücke, über die Prädikatenaussagen getroffen werden können. Hierzu werden die atomaren Formeln definiert.

Definition 2.2.4 (atomare Formeln)
Sei S eine Signatur. Die Menge A_S der atomaren Formeln sei definiert als:

$$A_S = \{P(t_1, \ldots, t_n) | P \in P_S, \alpha_S(P) = n, t_1, \ldots, t_n \in T_S\}$$

In Analogie zur Aussagenlogik lässt sich sagen, dass die atomaren Formeln dort den aussagenlogischen Variablen entsprechen. Was in der Aussagenlogik noch die atomaren Einheiten waren, hat in der Prädikatenlogik eine zusätzliche Struktur bekommen.

Schließlich in der dritten Definition werden die Formeln von PL1 definiert. Auch dieses ist wieder eine strukturell induktive Definition:

Definition 2.2.5 (Formeln)
Sei S eine Signatur. Die Menge For_S der Terme sei die kleinste Menge von Wörtern über $\{(, , ,), \neg, \rightarrow, \forall\} \cup F_S \cup I_S \cup P_S$ mit:

- $A_S \subseteq For_S$.
- *Wenn $A, B \in For_S$ und $x \in I_S$, so sind auch: $\neg A, (A \rightarrow B)$ und $\forall x A$ in For_S.*

Analog zu den Begriffen in der Aussagenlogik, sei eine Formel, die nur aus einem Prädikat besteht als ein Atom bezeichnet, und eine Formel, die ein Atom oder ein negiertes Atom ist, als Literal.

Terme, in denen keine Variabeln vorkommen, werden als Grundterme bezeichnet. Atome, die nur Grundterme enthalten, werden als Grundatome bezeichnet.

Wie man sieht, halten wir unsere Sprache extrem klein. Wir verzichten auf die üblichen aussagenlogischen Junktoren für *oder* und *und*, $\vee$ und $\wedge$ und begnügen uns mit dem Symbol $\rightarrow$ für die logischen Implikation und $\neg$ für die Negation. $\leftrightarrow$, $\vee$ und $\wedge$ werden wir lediglich als Abkürzungen für komplexere Ausdrücke benutzen. Ebenso stellt der Existenzquantor eine abkürzende Schreibweise für eine negierte allquantifizierte Formel dar:

Definition 2.2.6 ($\exists$, $\leftrightarrow$, $\vee$ und $\wedge$)
Ein Ausdruck der Form $(A \vee B)$, $A, B \in For_S$ stehe für: $(\neg A \rightarrow B)$.
Ein Ausdruck der Form $(A \wedge B)$, $A, B \in For_S$ stehe für: $\neg(\neg A \vee \neg B)$.
Ein Ausdruck der Form $(A \leftrightarrow B)$, $A, B \in For_S$ stehe für: $((A \rightarrow B) \wedge (B \rightarrow A))$.
Eine Formel der Form $\exists x A$ stehe für: $\neg\forall(\neg A)$

Implementierung der Syntax

Damit ist die Syntax von PL1 vollständig definiert. Wie schon in der Aussagemlogik soll auch eine Haskellimplementierung und ein zusätzliches Gefühl für die Sprache geben und Gelegenheit geben mit ihr herumzuexperimentieren.

Wir schreiben ein Haskellmodul PL1 mit den entsprechenden Imports:

PL1.hs

```
1 {-# OPTIONS -fglasgow-exts #-}
2 module PL1 (module PL1, module ToLatex)where
3
4 import Maybe
5 import List
```

```
6  import Data.Generics
7  import ToLatex
```

Entsprechend der Syntax von PL1 ist als erstes ein Datentyp für die Terme zu definieren. Ein Term ist entweder ein Variablensymbol oder ein Funktionsaufruf, der eine Liste von Argumenttermen enthält-

PL1.hs

```
8  data Term =
9     Var String
10   |Fun String [Term]
11         deriving (Eq,Show,Typeable,Data)
```

Auch für diesen Datentyp seinen möglichst viele Haskellklassen, wie `Eq` oder `Show` bereits automatisch implementiert.

Für die späteren Beispielen seien ein paar Variablen definiert:

PL1.hs

```
12 vx = Var "x"
13 vy = Var "y"
14 vz = Var "z"
15 vs = Var "s"
16 vt = Var "t"
17 vu = Var "u"
18 vv = Var "v"
19
20 vxi i = Var ("x_"++show i)
21
22 vx1 = vxi 1
23 vx2 = vxi 2
24 vx3 = vxi 3
25 vx4 = vxi 4
```

Die Definition von Formeln der PL1 ging in zwei Stufen. Erst haben wir atomare Formeln definiert und daraus komplexere Formeln zusammengesetzt. In Haskell werden wir dieses in einem Datentyp machen. Eine Formel ist entweder eine atomare Formel, eine allquantifizierte Formel, eine negierte Formel oder eine Implikation.

PL1.hs

```
26 data Formula =
27    Pred String [Term]
28   |Forall String Formula
29   |Not Formula
30   |Formula :-> Formula
```

Existenzquantifizierte Formeln, Und- und Oder-Verknüpfungen und die Äquivalenz tauchen in der Definition von PL1 nicht aus, sondern wurden nur als abkürzende Schreibweisen betrachtet. Im Haskelldatentyp sehen wir aber auch diese Fälle vor.

PL1.hs

```
31   |Exists String Formula
32   |Formula :/\ Formula
33   |Formula :\/ Formula
34   |Formula :<-> Formula
35          deriving (Eq,Show,Typeable,Data)
```

Wie schon in der Implementierung der Aussagenlogik haben wir von der Möglichkeit Haskells Infixkonstruktoren zu definieren Gebrauch gemacht. Die Operatoren haben wieder eine unterschiedlich starke Bindung:

PL1.hs

```
infixl 8 :/\
infixl 6 :\/
infixl 4 :->
infixl 2 :<->
```

Ein paar der Beispielformeln aus dem einführenden Abschnitt seien exemplarisch als Haskellausdrücke geschrieben:

PL1.hs

```
sterblich = Pred "Sterblich"
mensch = Pred "Mensch"
vater = Fun "vater"
elvis = Fun "elvis" []

f1 = (Forall "x" (mensch[vx] :-> sterblich[vx]) :/\ mensch[elvis])
       :-> sterblich[elvis]
f2 = Forall "x" (mensch[vx] :-> mensch[vater[vx]])
f3 = mensch[vater[elvis]]

liebZumZeitpunkt = Pred "LovesAtT"
f4 = Forall "x"$Exists "y"$Exists "t"$liebZumZeitpunkt[vx,vy,vt]
```

Um die Formeln in eine gut lesbare Druckform zu formatieren, werden die beiden Datentypen, die zusammen die Syntax Prädikatenlogik ausdrücken, zu Instanzen der Klasse `ToLatex` implementiert. Damit lassen sich dann in vertrauter Form die Formeln in Latexdokumente einbinden:

PL1.hs

```
instance ToLatex Term where
 toLatex (Var x) = x
 toLatex (Fun f [])
  ="\\mathrm{"++f++"}"
 toLatex (Fun f ts)
  ="\\mathit{"++f++"}("++concatWith "," (map toLatex ts)++")"

instance ToLatex Formula where
 toLatex (Pred p ts)
  = "\\mathrm{"++p++"}("++concatWith "," (map toLatex ts)++")"
 toLatex (Forall x f)
  = "\\forall \\, "++x++" \\, "++toLatex f
 toLatex (Exists x f)
  = "\\exists \\, "++x++" \\, "++toLatex f
 toLatex (Not f)
  = "\\neg "++toLatex f
 toLatex (f :-> g)
  = "\\left("++toLatex f++"\\rightarrow "++toLatex g++"\\right)"
 toLatex (f :<-> g)
  = "\\left("++toLatex f++"\\rightleftarrow "++toLatex g++"\\right)"
 toLatex (f :\/ g)
  = "\\left("++toLatex f++"\\vee "++toLatex g++"\\right)"
 toLatex (f :/\ g)
  = "\\left("++toLatex f++"\\wedge "++toLatex g++"\\right)"
```

Variablenbindungen Die Syntax der Prädikatenlogik sieht Variablen vor. Diese Variablensymbole sind ähnlich zu verstehen, wie die formalen Parameter einer Funktion in Programmiersprachen. Dort sind die Variablen durch den Funktionsrumpf als lokal für die Funktion in der Gültigkeit gebunden anzusehen. Hierbei gilt wie in Programmiersprachen, dass innere Bindungen äußere überdecken. Man Betrachte so die Formel $\forall x \exists x P(x)$. Hier werden zwei Variablen x eingeführt. Einmal durch den Allquantor, das zweite mal durch den inneren Existenzquantor. Die zweite Bindung überdeckt die erste. Die beiden Quantoren führen jeweils eine neue Variable x ein. In der atomaren Formel $P(x)$ ist das x durch den Existenzquantor gebunden.

Für unsere Implementierung der Syntax der Prädikatenlogik seinen Funktionen definiert, die die freien Variablen einer Formel berechnen.

Da sich die Syntax der Prädikatenlogik über zwei Datentypen erstreckt, sei zunächst eine Haskellschnittstelle definiert, die dann sowohl von Termen als auch von Formeln implementiert wird. Die Schnittstelle soll alle Variablennamen aus einem Term bzw. einer Formel extrahieren:

PL1.hs

```
76 class GetVars a where
77   getVars:: a -> [String]
```

Für Terme sind alle Variablen zu sammeln. Die Standardfunktion `nub` entfernt doppelte ais der Ergebnisliste.

PL1.hs

```
78 instance GetVars Term where
79   getVars = nub . everything (++) ([] `mkQ` var)
80    where
81     var (Var x) = [x]
82     var _ = []
```

Wir lassen glauch auch noch eine Listen von Termen die Schnittstelle `GetVars` implementieren:

PL1.hs

```
83 instance GetVars [Term] where
84   getVars = nub . concat . map getVars
```

Für Formeln gilt: Variablen tauchen nur in Termen auf. Die Terme sind die Argumente der Prädikatensymbole. Also sind für die Implementierumg der Klasse `GetVars` auf Formeln für jedes Prädikatensymbol aus den Argumenttermen die Variablen zu extrahieren:

PL1.hs

```
85 instance GetVars Formula  where
86   getVars = nub . everything (++) ([] `mkQ` var)
87    where
88     var (Pred _ ts) = getVars ts
89     var _ = []
```

Die Funkion `getVars` extrahiert jeweils alle Variablen, sowohl freie als auch gebundene. Die folgende Funktion soll nur die freien, also durch keinen Quantor gebundenen Variablen aus einer Formel extrahieren. Hierzu sei eine Hilfsfunktion `fV` definiert, die zwei Argumente hat. Die bereits gebundenen Variablen und die Formel, aus der die freien Variablen extrahiert werden sollen. Die beiden Quantoren fügen eine weitere Variablen zu der Mende der gebundenen Variablen hinzu. Die fünf logischen Operatoren binden keine Variablen neu. Bei Atomen werden mit `getVars` zunächst alle Variablen der Argumentterme extrahiert, von diesen aber die bereits gebundenen wieder abgezogen: `(getVars ts \\ vs)`.

PL1.hs

```
90 freeVars f = nub $ fV [] f
91   where
92     fV vs (Pred p ts)  = getVars ts \\ vs
93     fV vs (Forall x f) = fV (x:vs) f
94     fV vs (Exists x f) = fV (x:vs) f
95     fV vs (Not f)      = fV vs f
96     fV vs (f :-> g)    = fV vs f ++ fV vs g
97     fV vs (f :<-> g)   = fV vs f ++ fV vs g
98     fV vs (f :\/ g)    = fV vs f ++ fV vs g
99     fV vs (f :/\ g)    = fV vs f ++ fV vs g
```

2.2.3 Substitution

Die Quantoren, die Variablen binden, sollen Aussagen darüber machen, ob die Variablen durch einen beliebigen Term ersetzt werden dürfen oder nicht. Deshalb spielen Variablenersetzungen in der Prädikatenlogik eine entscheidene Rolle. Solche Variablenersetzungen werden als Substitution bezeichnet.

Definition 2.2.7 (Substitution)
Eine Substition $\sigma : V \to \mathcal{T}$ sei eine Abbildung von Variable auf Terme. Substitutionen mit endlich vielen $v \in V$ so dass $\sigma(v) \neq v$ lassen sich andlich aufzählen. Wir schreiben dann $\sigma = \{x_1 \rightsquigarrow t_1, \ldots, x_n \rightsquigarrow t_n\}$.

Für eine Substitution σ wird als Funktionsanwendung die übliche mathematische Notation benutzt, sprich σ angewendet auf t wird $\sigma(t)$ bezeichnet. Ist die Substitution hingegegen endlich aufgezählt, wird sie dem Ausdruck, auf dem Sie angewendet wird auch hinten dran geschrieben in der Notation: $t\{x_1 \rightsquigarrow t_1, \ldots, x_n \rightsquigarrow t_n\}$. Dieses ist zu lesen als: führe in t die folgende Variablenersetzung durch. Hier werden mitunter statt der geschweiften Mengenklammern auch eckige Klammern verwendet.

Die Notation $\sigma[x \rightsquigarrow t]$ bezeichne die Substition, die entsteht, wenn die Substitution σ abgeändert wird, so dass für die Variable x der Term t ersetzt wird, für alle anderen Variablen die Ersetzung, die σ vorsieht:

$$\sigma[x \rightsquigarrow t](y) = \begin{cases} t, & \text{wenn } y = x \\ \sigma(y), & \text{sonst} \end{cases}$$

Eine Substitution σ wird erweitert auf eine Abbildung $\sigma : T \to \mathcal{T}$ von Termen auf Termen durch folgende strukturelle Definition. Für den Basisfall einer Variablen ist die Substitution bereits definiert, für den Rekursionsfall sei sie definiert durch:

- $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$

Für Formeln wird eine Substitution erweitert durch folgende strukturelle Definition:

- $\sigma(P(t_1, \ldots, t_n)) = P(\sigma(t_1), \ldots, \sigma(t_n))$
- $\sigma(\neg P) = \neg\sigma(P)$
- $\sigma(F \to P) = \sigma(F) \to \sigma(P)$
- $\sigma(\forall x P) = \forall x \sigma'(P)$ *mit* $\sigma' = \sigma[x \rightsquigarrow x]$.

Implementierung von Substitutionen

Auch Substitutionen seien zur Illustration in Haskell implementiert. Eine Substition sei zunächst als eine Abbildung auf einem beliebigen Typ `a` definiert:

PL1.hs
```
100 type Substitution a = a -> a
```

In unseren Fall ist eine Substition zunächst definiert als eine Abbildung von Variablennamen auf Terme. Dieses lässt sich als endliche Liste von Paaren darstellen. Mit der Haskellklasse `Substitute` wird eine Funtion angeboten, die aus einer Liste von Variablennamen-Termpaaren eine Substitution macht:

PL1.hs
```
101 class Substitute a where
102   substitute:: [(String,Term)] -> Substitution a
```

Für Terme gibt es zwei Fälle. Eine Liste von Paaren kann mit der Standardfunktion `lookup` direkt als Map verwendet werden

PL1.hs
```
103 instance Substitute Term where
104   substitute sub v@(Var x) = maybe v id$lookup x sub
105   substitute sub (Fun f ts)= Fun f$map (substitute sub) ts
```

Für Formeln lässt sich die raffinierte Funktion `everywhere` der generischen Haskellerweiterung für eine sehr kurze Implementierung benutzen:

PL1.hs
```
106 instance Substitute Formula where
107   substitute sub = everywhere (mkT s)
108    where
109     s (Pred p ts) = Pred p$map (substitute sub) ts
110     s x = x
```

Der Bequemlichkeit halber seien Substitutionen auch auf Listen und Paaren substituierbarer Typen gehoben:

PL1.hs
```
111 instance Substitute a => Substitute [a] where
112   substitute sub = map (substitute sub)
113
114 instance Substitute a => Substitute (a,a) where
115   substitute sub (x,y)= (substitute sub x,substitute sub y)
```

2.2.4 Semantik

Nun soll in gleicher Weise wie auch schon für die Aussagenlogik eine Semantik definiert werden. Während dieses für die Aussagenlogik recht einfach war, und in Abbildungen der aussagenlogischen Variablen auf die Werte *wahr* oder *falsch* mündete, wird die Sache jetzt ein wenig vertrakter. Zwar soll im Endeffekt eine prädikatenlogische Formel für eine bestimmte Interpretation wieder zu *wahr* oder zu *falsch* ausgewertet werden, doch Terme und Prädikate sollen ja komplexe Beziehungen zwischen Elementen verschiedener Mengen darstellen. Um also Formeln

zu interpretieren, brauchen wir mindestens eine Menge von Elementen, sozusagen die Individuen, für die wir die Formeln aufgestellt haben. Um eine Formel zu interpretieren sind wir in der Wahl der Menge, auf der Funktionen und Prädikate interpretieren weitgehendst frei, so lange diese Menge abzählbar und nicht leer ist. Diese Menge wird als Trägermenge bezeichnet. Wir können ansonsten zur Interpretation einer Formel eine beliebige solche Trägermenge wählen, sei es die Menge aller Studenten der Hochschule RheinMain, die Menge aller Menschen, die wir einmal geliebt haben, die Menge der Bauklötze in einem bestimmten Kinderzimmer, die Menge der natürlichen Zahlen oder die Menge aller Sandkörner in der Sahara.

Auf dieser Trägermenge werden dann jedes Funktionssymbol als eine entsprechende totale Funktion interpretiert, und jedes Prädikatensymbol als eine Relation auf der Trägermenge, wobei einstellige Prädikate Teilmengen der Trägermenge darstellen.

Eine Interpretaion wertet somit einen Term zu einem Element der Trägermenge aus, ein Prädikat zu einem der beiden Werte `W` oder `F` und auch jede Formel zu einen der Werte `W` oder `F`.

Definition 2.2.8 (Interpretation von PL1)
Eine Interpretation prädikatenlogischer Formeln Sei ein 4-Tupel $I = (\mathcal{D}, \mathcal{F}, \mathcal{P}, \text{Iv})$ *aus:*

- *einer nichtleeren abzählbaren Trägermenge* $\mathcal{D}$*.*
- *einer Abbildung für jedes Funktionssymbol* $f \in F_S$ *auf eine totale Funktion* f_i *in* $\mathcal{D}$ *mit entsprechender Stelligkeit.*
- *einer Abbildung für jedes Prädikatensymbol* $P \in P_S$ *auf eine Relation* P_I *in* $\mathcal{D}$ *mit entsprechender Stelligkeit. Nullstellige Prädikate werden auf eines der beiden Symbole* $\{T, F\}$ *abgebildet.*[6]
- *einer Variablenbelegung* Iv: $I \to \mathcal{D}$*, die jede Variabel auf ein Element der Trägermenge abbildet.*

Für eine gegebene Interpretation I *definieren wir eine abgeänderte Interpretation* $I[x \rightsquigarrow a]$ *mit* $I[x \rightsquigarrow a](x) = a$ *und* $I[x \rightsquigarrow a](y) = I(x)$ *wenn* $x \neq y$*.*

Eine Interpretation sieht also auf einer Trägermenge für jedes Funktionssymbol in der Signatur eine totale Funktion vor und für jede Prädikatensymbol eine Relation. Eine Interpretation wertet Terme zu einem Element der Trägermenge aus. Formeln werden nun wie auch schon in der Aussagenlogik durch eine Interpretation auf einen Wert aus der Menge $\{W, F\}$ abgebildet.

Zunächst sei die Auswertung von Termen durch eine Interpretation definiert:

Definition 2.2.9 (Auswertung von PL1-Termen)
Sei $I = (\mathcal{D}, \mathcal{F}, \mathcal{P}, \text{Iv})$ *eine Interpretation. Die Interpretation bilde Terme auf Elemente der Trägermenge ab.*

Die Auswertung der Terme einer für eine Signatur S *sei eine Funktion* $w_I : T_S \to \mathcal{D}$*, die strukturell Induktiv definiert ist durch:*

- $w_I(x) = \text{Iv}(x)$ *für* $x \in I_S$*.*
- $w_I(f(t_1, \ldots, t_n) = \mathcal{F}(f)(w_I(t_1), \ldots, w_I(t_n))$*.*

[6] Nullstellige Prädikate entsprechen den aussagenlogischen Variablen in der Aussagenlogik.

Es wird also die Funktion, als die ein Funktionssymbol in der Semantik abgebildet wird, auf die Auswertung der Argumente angewendet.

Definition 2.2.10 (Auswertung von PL1-Formeln)
Sei $I = (\mathcal{D}, \mathcal{F}, \mathcal{P}, \mathrm{Iv})$ *eine Interpretation.*
Unter der Interpretation I *wird eine Formel auf einen der Werte* W *und* F *abgebildet.*

Die Auswertung der Formeln einer für eine Signatur S *sei eine Funktion* $w_I : For_S \to \{W,F\}$*, die strukturell Induktiv definiert ist durch:*

Basisfälle:

Fall: $H = P(t_1, \ldots t_n)$	*falls* $(w_I(t_1), \ldots, w_I(t_n)) \in \mathcal{P}(P)$,	*dann* $w_I(H) := W$.
	falls $(w_I(t_1), \ldots, w_I(t_n)) \notin \mathcal{P}(P)$, ,	*dann* $w_I(H) := F$.
Fall $H = P$		$I(P) := \mathcal{P}(P)$.

Rekursionsfälle:

Fall: $H = \neg F$	*dann* $w_I(H) = W$ *falls* $w_I(F) = F$
Fall: $H = F \to G$	*dann* $w_I(H) = W$ *falls* $w_I(F) = F$ *oder* $w_I(G) = W$
Fall: $H = \forall x : F$	*dann* $I(H) = W$ *falls für alle* $a \in D_S : I[a \mapsto x](F) = W$

Die Semantik der erweiterten Syntax, insbesondere des Existenzquantors ergibt sich aus der obigen Definition.

Wir können nun direkt die schon aus der Aussagenlogik bekannten Begriffe eines *Modells*, der *Erfüllbarkeit*, *Allgemeingültigkeit*, *Tautologie* und *Widersprüchlichkeit* übernehmen. Insbesondere den Begriff der semantischen Folgerbarkeit ist wieder über die Interpretationen von Formeln definiert.

Definition 2.2.11 (semantische Folgerbarkeit in PL1)
Sei $M \subset For_{(\Sigma,V)}$ *und* $A \in For_{(\Sigma,V)}$ *. Aus* M *sei* A *folgerbar im Zeichen* $M \models_S A$ *genau dann wenn:*

für jede Interpretation I*, für die für jedes* $B \in M$ *ein Modell ist, gilt* I *ist auch ein Modell von* A*.*

2.2.5 Der Tableauxkalkül

Auch für die Prädikatenlogik kann in analoger Weise der Kalkül des natürlichen Schließens definiert wird. Wegen der schwer automatisierbaren Natur diesen Kalküls wollen wir darauf verzichten und uns gleich dem Tableauxkalkül für die Prädikatenlogik zuwenden.

Wie bereits in der Aussagenkogik ist auch in der Prädikatenlogik der Tableauxkalkül ein Widerlegungskalkül. Es wird gezeigt, dass für die Formel an der Wurzel keine Interpretation existieren kann, die diese Formel zu wahr auswertet. Das systematische Expandieren des Tableau anhand einer Formel stellt eine Suche nach einer möglichen Interpretation dar, die die Formel zu wahr auswertet. Geschlossene Pfade zeigen, dass hier eine solche Interpretation zu einen Widerspruch führen würde,

Tableau ohne Unifikation

Wir werden zunächst eine primitive Form des Tableauxkalkül der Prädikatenlogik betrachten, in der eine Instanziierung der Variablen der Formeln auf geschickte Weise erraten werden muss. In einem zweiten Schritt werden wir dann Unifikatoren einführen.

Der Tableauxkalkül funktioniert in der Prädikatenlogik fast genauso wie in der Aussagenlogik. Die Erweiterungsregeln der Aussagenlogik werden eins zu eins übernommen. Hinzu kommen aber noch zwei weitere Regeln, die die Expansion für Knoten mit quantifizierten Formeln ermöglichen.

Definition 2.2.12 (Tableaux ohne Unifikation)
Ein Tableau ist ein Binärbaum, dessen Knoten mit Formeln aus For_S *markiert sind.*

Ein Pfad von der Wurzel zu einem Blatt des Tableau heisst geschlossen, wenn es auf diesem Pfad ein Atom A *gibt, so dass auch* $\neg A$ *auf diesem Pfad liegt.*

Ein Tableau heisst geschlossen, wenn alle Pfade zu einem Blatt geschlossen sind.

An Blätter, an denen ein Tableau geschlossen ist, werden mit dem zusätzlichen Kind $\square$ *gekennzeichnet.*

Natürlich soll ein Tableau auch in der Prädikatenlogik nicht aus beliebigen Formelmarkierungen bestehen. Die Kinder eines Knotens sollen systematisch die Formel der Wurzel dekonstruieren. Hierzu gibt es Erweiterungsregeln.

Definition 2.2.13 (Erweiterungsregeln)
Ein Tableau kann nach vier Arten von Regelschemata erweitert werden:

- **Und-Regeln (auch als α-Regeln bezeichnet)***:*

  $(A \wedge B)$ | A | B $\qquad$ $\neg(A \vee B)$ | $\neg A$ | $\neg B$ $\qquad$ $\neg(A \to B)$ | A | $\neg B$ $\qquad$ $\neg\neg A$ | A

- **Oder-Regeln (auch als β-Regeln bezeichnet)***:*

  $(A \vee B)$: $A \quad B$ $\qquad$ $\neg(A \wedge B)$: $\neg A \quad \neg B$ $\qquad$ $(A \to B)$: $\neg A \quad B$

- **Für-alle-Regeln (auch als γ-Regeln bezeichnet)***:*

  $\forall x A$ | $A\{x \mapsto c\}$ $\qquad$ $\neg\exists x A$ | $\neg A\{x \mapsto c\}$

  Wobei c *eine beliebiger Grundterm ist.*

- **Es-existiert-Regeln (auch als δ-Regeln bezeichnet)***:*

  $\exists x A$ | $A\{x \mapsto c\}$ $\qquad$ $\neg\forall x A$ | $\neg A\{x \mapsto c\}$

  Wobei c *ein neues Konstantensymbol ist oder ein Konstantensymbol, das noch durch keine Es-existiert-Regel für eine Variable ersetzt wurde.*

Ein Tableau kann erweitert werden, indem für einen Knoten, der die Form einer der Wurzeln der obigen Regeln hat, an alle Blätter, die von diesem Knoten erreichbar sind, die entsprechenden Kinder aus der benutzten Regel angehängt werden.

Wir betrachten nun nur noch Tableaux, die durch sukzessive Anwendung der Regeln von einem Wurzelknoten aus aufgebaut wurden.

Die obigen zwei neuen Erweiterungsregeln gehen von sehr einfachen und naheliegenden Überlegungen aus: wenn eine Variable allquantifiziert ist, dann darf für diese Variable nach der Semantik ein beliebiger Wert stehen, d.h. er kann semantisch auf beliebige Elemente der Trägermenge abgebildet werden. Daher erlauben wir die Variable durch einen beliebigen Grundterm (variablenfreien Term) zu ersetzen. Dieser bezeichnet dann in der Sementak ein Element aus der Trägermenge.

Die Regeln für den Existenzquantor gehen von der Überlegung aus: wenn ein Element aus der Trägermenge existieren soll, dann gebe ich hiermit diesem Element einen neuen Namen. Ich erweitere die Signatur um eine Konstante, die genau dieses eine Element der Trägermenge mit der gewünschten Existenzeigenschaft bezeichnet.

Beispiel 2.2.14 *Machen wir ein kleines Beispiel für die einfachen Tableaux ohne Unifikation. In diesem Beispiel soll ein komplettes Tableau zum Beweis der Allgemeingültigkeit der Formel:*

$(\forall\, x\ (\mathrm{P}(x) \to \mathrm{Q}(x)) \to (\forall\, x\, \mathrm{P}(x) \to \forall\, x\, \mathrm{Q}(x)))$

Hierzu wird die Formel negiert und an die Wurzel eines Tableaus gestellt. Systematisch wird das Tableau expandiert. Anschließend wird geschaut ob das Tableau an allen Blättern geschlossen werden kann. Das entstandene Tableau ist in Abbildung 2.7 zu finden.

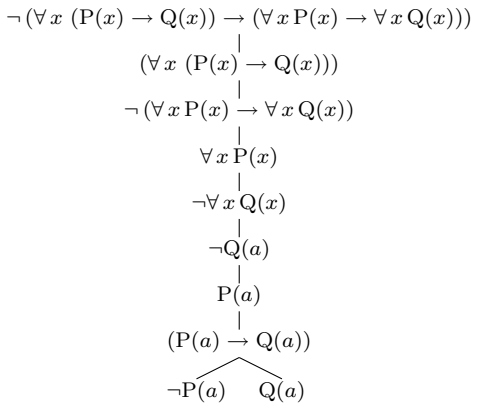

Abbildung 2.7: Geschlossenes Tableau

Unifikation

Wie am obigen Beispiel bereits zu sehen war, gibt es ein unschönes Problem. Man muss sich beim Erweitern der Quantorenknoten bereits direkt für einen variablenfreien Term entscheiden, mit dem irgendwann später der Tableau geschlossen werden kann. Man muss also sehr vorausschauend die Variablen durch bestimmte Terme substituieren. Im Prinzip muss die kluge Ersetzung für eine quantifizierte Varable erraten werden.

Die Verfeinerung des Tableauxkalkül arbeitet mit Formeln, die freie Variablen enthalten. Diese freien Variablen werden als implizit allquantifiziert betrachtet. Zum Schließen eines Pfades braucht nicht ein Atom einmal negiert einmal unnegiert auf dem Pfad gefunden zu werden, sondern es müssen zwei Literate gefunden werden, von denen eines negiert ist und eines unegiert. Wenn dann eine Substitution existiert, die auf die beiden Atome angewendet die Atome syntaktisch gleich macht, dann kann der Tableau geschlossen werden.

Eine solche Substitution, die zwei Ausdrücke syntaktisch gleich macht, wird als Unifikator bezeichnet:

Definition 2.2.15 (Unifikator)
Seien t_1 und t_2 Terme. Eine Substitution σ mit $\sigma(t_1) = \sigma(t_2)$ heisst Unifikator.

Wenn es einen Unifikator gibt, dann gibt es in der regel mehrere Unifikatoren. Man betrachte die zwei Terme, die jeweils nur aus einer Variablen bestehen. $t_1 = x$ und $t_2 = y$. Die einfache Substitution $\sigma_1 = \{x \mapsto y\}$ ist ein Unifikator für diese beiden Terme. Aber auch Substitutionen, die die beiden Variablen durch denselben beliebigen Term ersetzen, also z.B. auch die Substitution $\sigma_2 = \{x \mapsto f(a,b,c), y \mapsto f(a,b,c)\}$. Es leuchtet ein, dass σ_2 wesentlich spezieller ist als σ_1. σ_2 legt sich unnötig auf einen komplexeren Term für die beiden Variablen fest. Von unsere Zwecke wird es zweckdienlich sein, immer einen möglichst allgemeinen Unifikator zu finden. Ein allgemeinster Unifikator ist ein Unifikator, aus dem durch eine weitere spezialisierende Substitution jeder weitere Unifikator gewonnen werden kann.

Definition 2.2.16 (allgemeinster Unifikator (most general unifier MGU))
Ein Unifikator μ für zwei Terme t_1 und t_2 heisst allgemeinster Unifikator, wenn für jeden anderen Unifikator τ gilt: es existiert eine Substitution σ mit $\tau = \sigma \circ \mu$, sprich, es gilt= $\tau(x) = \sigma(\mu(x))$.

Wenn ein allgemeinster Unifikator existiert, lässt ein solcher dieser algorithmisch errechnen. Hierzu werden die beiden zu unifizierenden Terme schrittweise auseinander genommen und der Unifikator dabei schrittweise aufgebaut. Der Algorithmus operiert auf einer Menge zu unifizierender Termpaare. Gestartet wird er mit der einelementigen Menge aus dem Termpaar der zu unifizierenden Terme. Die Menge wird erweitert um weitere Terme, wenn zwei Funktionsausdrücke mit gleichen Funktionssymbol zu unifizieren sind. Die Menge wird kleiner, wenn in einem Termpaar der linke Term eine Variable ist. Der Unifikator ist gefunden, wenn die Menge der zu unifizierenden Termpaare leer ist.

Definition 2.2.17 (Unifikationsalgorithmus $U1$:)

Eingabe: *zwei Terme oder Atome s und t:*

Ausgabe: *"nicht unifizierbar" oder einen allgemeinsten Unifikator:*

Zustände: *auf denen der Algorithmus operiert: Eine Menge Γ von Gleichungen.*

Initialzustand: $\Gamma_0 = \{s \stackrel{?}{=} t\}$.

Unifikationsregeln:

$$\frac{f(s_1,\ldots,s_n) \stackrel{?}{=} f(t_1,\ldots,t_n), \Gamma}{s_1 \stackrel{?}{=} t_1 \ , \ \ldots \ , \ s_n \stackrel{?}{=} t_n, \Gamma} \qquad \textit{(Dekomposition)}$$

$$\frac{x \stackrel{?}{=} x, \Gamma}{\Gamma} \qquad \textit{(Tautologie)}$$

$$\frac{x \stackrel{?}{=} t, \Gamma}{x \stackrel{?}{=} t, \{x \mapsto t\}\Gamma} \qquad x \in FV(\Gamma),\ x \notin FV(t) \qquad \textit{(Anwendung)}$$

$$\frac{t \stackrel{?}{=} x, \Gamma}{x \stackrel{?}{=} t, \Gamma} \qquad t \notin V \qquad \textit{(Orientierung)}$$

Abbruchbedingungen:

$$\frac{f(\ldots) \stackrel{?}{=} g(\ldots), \Gamma}{Fail} \qquad \textit{wenn } f \neq g \qquad \textit{(Clash)}$$

$$\frac{x \stackrel{?}{=} t, \Gamma}{Fail} \qquad \textit{wenn } x \in FV(t) \textit{ vorkommt und } t \neq x \qquad \textit{(occurs check Fehler)}$$

Steuerung:

Starte mit $\Gamma = \Gamma_0$, und transformiere Γ solange durch (nichtdeterministische, aber nicht verzweigende) Anwendung der Regeln, bis entweder eine Abbruchbedingung erfüllt ist oder keine Regel mehr anwendbar ist. Falls eine Abbruchbedingung erfüllt ist, terminiere mit "nicht unifizierbar". Falls keine Regel mehr anwendbar ist, hat die Gleichungsmenge die Form $\{x_1 \stackrel{?}{=} t_1, \ldots, x_k \stackrel{?}{=} t_k\}$, wobei keine der Variablen x_i in einem t_j vorkommt; d.h. sie ist in gelöster Form. Das Resultat ist dann $\{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$.

Implementierung Der Unifikationsalgorithmus ist erstaunlich einfach und in wenigen Zeilen implementierbar:

PL1.hs

```
116 occur x  = (elem x).getVars
117
118 uni :: ([(String,Term)],[(Term,Term)]) -> Maybe [(String,Term)]
119 uni (res,[]) = Just res
```

```
120 uni (res,(Fun f1 t1,Fun f2 t2):ts)
121   |f1==f2 = uni (res,(zip t1 t2)++ts)
122   |otherwise = Nothing -- clash
123 uni (res,(f@(Fun _ _),v@(Var _)):ts) = uni (res,(v,f):ts)
124 uni (res,(Var v,t):ts)
125   |occur v t = Nothing -- occur check
126   |otherwise
127     = uni ((v,t):map (\(x,te)->(x,substitute [(v,t)] te)) res
128           ,substitute [(v,t)] ts)
```

Beispiele Die folgenden drei Beispiele illustrieren die Wirkungsweise des Unifikationsalgorithmus:

	$\mathrm{Q}(y, \mathrm{G}(\mathrm{A}(), \mathrm{B}())) = \mathrm{Q}(\mathrm{G}(x, x), y) \quad \{\}$
Dekomposition	$y = \mathrm{G}(x, x) \;\; \mathrm{G}(\mathrm{A}(), \mathrm{B}()) = y \quad \{\}$
Anwendung	$\mathrm{G}(\mathrm{A}(), \mathrm{B}()) = \mathrm{G}(x, x) \quad \{y \mapsto \mathrm{G}(x, x)\}$
Dekomposition	$\mathrm{A}() = x \;\; \mathrm{B}() = x \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$\mathrm{B}() = x \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$x = \mathrm{A}() \;\; x = \mathrm{B}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Anwendung	$\mathrm{A}() = \mathrm{B}() \quad \{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$

Clash: nicht unifizierbar

	$\mathrm{Q}(y, \mathrm{G}(\mathrm{A}(), \mathrm{A}())) = \mathrm{Q}(\mathrm{G}(x, x), y) \quad \{\}$
Dekomposition	$y = \mathrm{G}(x, x) \;\; \mathrm{G}(\mathrm{A}(), \mathrm{A}()) = y \quad \{\}$
Anwendung	$\mathrm{G}(\mathrm{A}(), \mathrm{A}()) = \mathrm{G}(x, x) \quad \{y \mapsto \mathrm{G}(x, x)\}$
Dekomposition	$\mathrm{A}() = x \;\; \mathrm{A}() = x \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$\mathrm{A}() = x \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Orientierung	$x = \mathrm{A}() \;\; x = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(x, x)\}$
Anwendung	$\mathrm{A}() = \mathrm{A}() \quad \{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$
Dekomposition	$\{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$
Unifikator	$\{y \mapsto \mathrm{G}(\mathrm{A}(), \mathrm{A}()), x \mapsto \mathrm{A}()\}$

	$\mathrm{H}(\mathrm{g}(\mathrm{f}(x,u)),\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())),\mathrm{f}(u,\mathrm{f}(x,u))) = \mathrm{H}(\mathrm{g}(v),v,\mathrm{f}(y,z))$	$\{\}$
Dekomposition	$\mathrm{g}(\mathrm{f}(x,u)) = \mathrm{g}(v) \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = v \quad \mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z)$	$\{\}$
Dekomposition	$\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = v \quad \mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z) \quad \mathrm{f}(x,u) = v$	$\{\}$
Orientierung	$\mathrm{f}(u,\mathrm{f}(x,u)) = \mathrm{f}(y,z) \quad \mathrm{f}(x,u) = v \quad v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))$	$\{\}$
Dekomposition	$\mathrm{f}(x,u) = v \quad v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) \quad u = y \quad \mathrm{f}(x,u) = z$	$\{\}$
Orientierung	$v = \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) \quad u = y \quad \mathrm{f}(x,u) = z \quad v = \mathrm{f}(x,u)$	$\{\}$
Anwendung	$u = y \quad \mathrm{f}(x,u) = z \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,u)$	$\{v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\mathrm{f}(x,y) = z \quad \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,y)$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$\mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())) = \mathrm{f}(x,y) \quad z = \mathrm{f}(x,y)$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Dekomposition	$z = \mathrm{f}(x,y) \quad \mathrm{g}(\mathrm{a}()) = x \quad \mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y$	$\{u \mapsto y, v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\mathrm{g}(\mathrm{a}()) = x \quad \mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$\mathrm{f}(\mathrm{a}(),\mathrm{a}()) = y \quad x = \mathrm{g}(\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Orientierung	$x = \mathrm{g}(\mathrm{a}()) \quad y = \mathrm{f}(\mathrm{a}(),\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(x,y), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$y = \mathrm{f}(\mathrm{a}(),\mathrm{a}())$	$\{u \mapsto y, z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),y), x \mapsto \mathrm{g}(\mathrm{a}()), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}()))\}$
Anwendung	$\{u \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), y \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), x \mapsto \mathrm{g}(\mathrm{a}())\}$	
Unifikator	$\{u \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), y \mapsto \mathrm{f}(\mathrm{a}(),\mathrm{a}()), z \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), v \mapsto \mathrm{f}(\mathrm{g}(\mathrm{a}()),\mathrm{f}(\mathrm{a}(),\mathrm{a}())), x \mapsto \mathrm{g}(\mathrm{a}())\}$	

Tableaux mit freien Variablen

Mit dem Begriff der Unifikation haben wir nun ein Werkzeug, um den Tableaukalkül zu verfeinern. Statt nun syntaktische Gleichheit des Prädikats, auf dem ein Pfad geschlossen wird zu verlangen, werden wir jetzt lediglich die Unifizierbarkeit verlangen:

Definition 2.2.18 (Tableaux mit freien Variablen)
Ein Tableau ist ein Binärbaum, dessen Knoten mit Formeln aus For_S markiert sind.

Ein Pfad heiße nun geschlossen, wenn es auf dem Pfad zwei Literale A und $\neg A'$ gibt, so dass ein Unifiator für A und A' existiert.

An Blätter, an denen ein Tableau geschlossen ist, werden mit dem zusätzlichen Kind $\Box$ gekennzeichnet.

Die Erweiterungsregeln in Tableaux mit freien Variablen unterscheiden sich nun in den Regeln für die Quantoren. Für allquantifizierte Formeln werden neue freie Variablen eingeführt, für existenzquantifizierte neue Terme, die von den freien Variablen abhängen:

Definition 2.2.19 (Erweiterungsregeln)
Ein Tableau mit freien Variablen kann nach vier Arten von Regelschemata erweitert werden:

- **Und-Regeln (auch als α-Regeln bezeichnet)**:

$$\begin{array}{cccc} (A \wedge B) & \neg(A \vee B) & \neg(A \to B) & \neg\neg A \\ | & | & | & | \\ A & \neg A & A & A \\ | & | & | & \\ B & \neg B & \neg B & \end{array}$$

- **Oder-Regeln (auch als β-Regeln bezeichnet)**:

$$\begin{array}{ccc} (A \vee B) & \neg(A \wedge B) & (A \to B) \\ \diagup\ \diagdown & \diagup\ \diagdown & \diagup\ \diagdown \\ A \quad B & \neg A \quad \neg B & \neg A \quad B \end{array}$$

- **Für-alle-Regeln (auch als γ-Regeln bezeichnet)**:

$$\begin{array}{cc} \forall x A & \neg\exists x A \\ | & | \\ A\{x \mapsto y\} & \neg A\{x \mapsto y\} \end{array}$$

Wobei y eine auf dem Pfad neue Variable ist.

- **Es-existiert-Regeln (auch als δ-Regeln bezeichnet)**:

$$\begin{array}{cc} \exists x A & \neg\forall x A \\ | & | \\ A\{x \mapsto f(x_1, \ldots, x_n)\} & \neg A\{x \mapsto f(x_1, \ldots, x_n)\} \end{array}$$

Wobei f ein neues Funktionssymbol ist und $\{x_1, \ldots, x_n\}$ die Menge der freien Variablen von A.

In den Regel für existenzquantifizierten wird ein Term mit einem neuen Funktionssymbol eingeführt. Diese neue Funktion hat so viele Parameter, wie es freie Variablen in der Formel gibt. Sie wird als Skolemfunktion bezeichnet.

Skolemisierung Die Elimination von Existenzquantoren ist die sogenannte Skolemisierung (Nach Thoralf Skolem).

Die Idee dabei ist, wenn eine Formel die Existenz eines bestimmten Individuums durch einen Existenzquantor ausdrückt, für dieses Individuum dann einen Namen in Form einer Konstanten einzuführen. Betrachte man hierzu einmal die Aussage, dass es jemanden gibt, der das Individuum mit Namen Elvis liebt:

$$\exists x : \mathit{liebt}(x, \text{Elvis})$$

Die Idee der Skolemisierung ist, zu sagen, na wenn so ein Individuum existiert, dann geben wir ihm halt in Form einer Konstanten einen Namen. Laut der Semantik wird diese Konstante dan auf ein Element der nichtleeren Trägermenge abgebildet, und für dieses Element dann die

nachfolgende Teilformel als wahr interpretiert. Geben wir in dem Beispiel also dem x einen Namen:

$$\mathit{liebt}(\text{Priscilla}, \text{Elvis})$$

Ganz so einfach geht es im Allgemeinen allerdings nicht. Wenn vor einem Existenzquantor Allquantoren stehen, so wird es nicht ausreichen, nur eine neue Konstante einzuführen. Betrachte man dazu die etwas allgemeinere Aussage, das sich für jeden jemand findet, der ihn liebt:

$$\forall y : \exists x : \mathit{liebt}(x, y)$$

Würden wir darin den Existenzquantor durch Einführung einer neuen Konstanten eliminieren, also zur Formel:

$$\forall x : \mathit{liebt}(\text{Priscilla}, x)$$

So erhalten wir die Aussage, dass ein bestimmtes Individuum (Priscilla) alle anderen liebt. Was man machen muss, ist nicht eine neue Konstante einführen, sondern eine Funktion, die das versprochene Individuum in Abhängigkeit der zuvor allquantifizierten Variablen berechnet. In unserem Beispiel also:

$$\forall x : \mathit{liebt}(\text{derLiebende}(x), x)$$

Die Funktion *derLiebende* berechnet für jedes Element das Element, das nach der ursprünglichen Aussage existieren soll.

Im nächsten Theorem sei $G[x_1, \ldots, x_n, y]$ eine beliebige Formel, die die Variablensymbole $x_1, \ldots, x_n, y$ frei enthält und $G[x_1, \ldots, x_n, t]$ eine Variante von F, in der alle Vorkommnisse von y durch t ersetzt sind.

Satz 2.2.20 Skolemisierung
Eine Formel $F = \forall x_1 \ldots x_n : \exists y : G[x_1, \ldots, x_n, y]$ ist (un-)erfüllbar gdw. $F' = \forall x_1 \ldots x_n : G[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]$ (un-)erfüllbar ist, wobei f ein n-stelliges Funktionssymbol ist, das nicht in G vorkommt.

Beispiel 2.2.21 Skolemisierung

$$\begin{array}{rll} \exists x : P(x) & \text{wird zu} & P(a) \\ \forall x : \exists y : Q(f(y, y), x, y) & \text{wird zu} & \forall x : Q(f(g(x), g(x)), x, g(x)) \\ \forall x, y : \exists z : x + z = y & \text{wird zu} & \forall x, y : x + h(x, y) = y. \end{array}$$

Beispiel 2.2.22 *Skolemisierung erhält i.a. nicht die Allgemeingültigkeit (Falsifizierbarkeit):*

$\forall x : P(x) \vee \neg\forall x : P(x)$ *ist eine Tautologie*

$\forall x : P(x) \vee \exists x : \neg P(x)$ *ist äquivalent zu*

$\forall x : P(x) \vee \neg P(a)$ *nach Skolemisierung.*

Eine Interpretation, die die skolemisierte Formel falsifiziert kann man konstruieren wie folgt: Die Trägermenge ist $\{a, b\}$. Es gelte $P(a)$ und $\neg P(b)$. Die Formel ist aber noch erfüllbar.

Skolemisierung ist eine Operation, die nicht lokal innerhalb von Formeln verwendet werden darf, sondern nur global, d.h. wenn die ganze Formel eine bestimmte Form hat. Zudem bleibt bei dieser Operation nur die Unerfüllbarkeit der ganzen Klausel erhalten.

Beispiel 2.2.23 *Machen wir ein kleines Beispiel für die einfachen Tableaux mit Unifikation, in dem es zu einer Skolemfunktion kommt. In diesem Beispiel soll ein komplettes Tableau zum Beweis der Allgemeingültigkeit der Formel:*

$(\exists\, w \forall\, x\, \mathrm{R}(x, w, g(x, w)) \rightarrow \exists\, w \forall\, x \exists\, y\, \mathrm{R}(x, w, y))$

erstellt werden. Hierzu wird die Formel negiert und an die Wurzel eines Tableaus gestellt. Systematisch wird das Tableau expandiert. Anschließend wird geschaut ob das Tableau an allen Blättern geschlossen werden kann. Das entstandene Tableau ist in Abbildung 2.8 zu finden.

$$\neg\,(\exists\, w \forall\, x\, \mathrm{R}(x, w, g(x, w)) \rightarrow \exists\, w \forall\, x \exists\, y\, \mathrm{R}(x, w, y))$$
$$|$$
$$\exists\, w \forall\, x\, \mathrm{R}(x, w, g(x, w))$$
$$|$$
$$\neg\exists\, w \forall\, x \exists\, y\, \mathrm{R}(x, w, y)$$
$$|$$
$$\forall\, x\, \mathrm{R}(x, a, g(x, a))$$
$$|$$
$$\neg\forall\, x \exists\, y\, \mathrm{R}(x, v_1, y)$$
$$|$$
$$\neg\exists\, y\, \mathrm{R}(b(v_1), v_1, y)$$
$$|$$
$$\mathrm{R}(v_2, a, g(v_2, a))$$
$$|$$
$$\neg\mathrm{R}(b(v_1), v_1, v_3)$$

Abbildung 2.8: Geschlossenes Tableau mit Skolemfunktion

Der Unifikator $\{v_2 \mapsto b(a), v_1 \mapsto a, v_3 \mapsto g(b(a), a)\}$ schließt den Tableau.

2.2.6 Resolutionskalkül

Der Resolutionskalkül funktioniert für die Prädikatenlogik in analoger Weise wie auch schon für die Aussagenlogik. Es wird ein Widerlegungsbeweis geführt, die abzuleitende Formel wird also zunächst negiert.

Während der Tableaukalkül schrittweise die zu beweisene Formel in die einzel bestandteile auseinandernimmt, besteht der Resolutionskalkül wieder aus zwei schritten. Zunächst wird die Formel in die Klauselform transformiert.

Klauselnormalform

Der eigentliche Kalkül rechnet dann wieder auf der Klauselform. Hier muss allerdings zunächst der Begriff der Klauselnormalform für die Prädikatenlogik erweitert werden.

Definition 2.2.24 (Klauselnormalform)
Eine Formel der folgenden Form ist in Klauselnormalform:

$$\begin{aligned} \forall x_1 : \ldots \forall x_n : \quad & (L_{1,1} \vee \ldots \vee L_{1,n_1}) \\ \wedge \quad & (L_{2,1} \vee \ldots \vee L_{2,n_2}) \\ \wedge \quad & \\ \ldots \quad & \\ \wedge \quad & (L_{k,1} \vee \ldots \vee L_{1,n_k}) \end{aligned}$$

Alle Variablen sind dabei durch Allquantoren gebunden, die alle am Anfang der Formel stehen. Die Formel ist dann eine Konjunktion von Disjunktionen (Klauseln) von Literalen.

Wie man sieht muss man auf irgendeine Weise, um die Klauselnormalform herzustellen, die Existenzquantoren eliminieren. Wir können zwar Formel der Form $\exists x : F$ äquivalent als $\neg\forall x : \neg F$ aber dann steht aber ein Negationszeichen nicht direkt vor einem Prädikat, was der Klauselnormalform widerspricht. Es ist ein anderer Trick notwendig, um Existenzquantoren zu eliminieren, die Skolemisierung. Diese wurde auch schon im Tableaukalkül angewendet, um Existenzquantoren zu eleminieren,

Mit der Skolemisierung steht somit der wichtigste Schritt zur Transformation einer prädikatenlogischen Formel in Klauselnormalform zur Verfügung. Insgesamt kann diese Transformation durch nachfolgenden siebenschrittigen Algorithmus erreicht werden. Drei der Schritte sind schon aus der Transformation einer Formel in Klauselform in der Aussagenlogik bekannt.

Definition 2.2.25 (Transformation in Klauselnormalform (unter Erhaltung der Unerfüllbarkeit))
Folgende Prozedur wandelt jede prädikatenlogische Formel in Klauselform (CNF) um:

1. *Elimination von $\rightarrow$:* $F \rightarrow G$ wird zu $\neg F \vee G$
2. *Negation ganz nach innen schieben:*

$\neg\neg F$	wird zu	F
$\neg(F \wedge G)$	wird zu	$\neg F \vee \neg G$
$\neg(F \vee G)$	wird zu	$\neg F \wedge \neg G$
$\neg\forall x : F$	wird zu	$\exists x : \neg F$
$\neg\exists x : F$	wird zu	$\forall x : \neg F$

3. *Skopus von Quantoren minimieren, d.h. Quantoren so weit wie möglich nach innen schieben*

$$\forall x : (F \wedge G) \quad \text{wird zu} \quad (\forall x : F) \wedge G \qquad \textit{falls } x \textit{ nicht frei in } G$$

$$\forall x : (F \vee G) \quad \text{wird zu} \quad (\forall x : F) \vee G \qquad \textit{falls } x \textit{ nicht frei in } G$$

$$\exists x : (F \wedge G) \quad \text{wird zu} \quad (\exists x : F) \wedge G \qquad \textit{falls } x \textit{ nicht frei in } G$$

$$\exists x : (F \vee G) \quad \text{wird zu} \quad (\exists x : F) \vee G \qquad \textit{falls } x \textit{ nicht frei in } G$$

$$\forall x : (F \wedge G) \quad \text{wird zu} \quad \forall x : F \wedge \forall x : G$$

$$\exists x : (F \vee G) \quad \text{wird zu} \quad \exists x : F \vee \exists x : G$$

4. *Alle gebundenen Variablen sind systematisch umzubenennen, um Namenskonflikte aufzulösen.*

5. *Existenzquantoren werden durch Skolemisierung eliminiert*

6. *Allquantoren löschen (alle Variablen werden als allquantifiziert angenommen.*

7. *Distributivität (und Assoziativität, Kommutativität) iterativ anwenden, um $\wedge$ nach außen zu schieben ("Ausmultiplikation"). $F \vee (G \wedge H)$* wird zu *$(F \vee G) \wedge (F \vee H)$ (Das duale Distributivgesetz würde eine disjunktive Normalform ergeben.)*

Das Resultat dieser Prozedur ist eine Konjunktion von Disjunktionen (Klauseln) von Literalen:

$$\begin{aligned} & (L_{1,1} \vee \ldots \vee L_{1,n_1}) \\ \wedge \quad & (L_{2,1} \vee \ldots \vee L_{2,n_2}) \\ \wedge \quad & \\ \ldots \quad & \\ \wedge \quad & (L_{k,1} \vee \ldots \vee L_{1,n_k}) \end{aligned}$$

oder in Mengenschreibweise:

$$\begin{aligned} & \{\{L_{1,1}, \ldots, L_{1,n_1}\}, \\ & \{L_{2,1}, \ldots, L_{2,n_2}\}, \\ & \ldots \\ & \{L_{k,1}, \ldots, L_{1,n_k}\}\} \end{aligned}$$

Beispiele

Zwei Beispiele sollen illustrieren, wie eine prädikatenlogische Formel schrittweise in Klauselform gebracht wird.

Beispiel 2.2.26 *Betrachten wir zunächst folgende Formel über Tierliebe:*

$$(\forall x.((\forall y.(\mathrm{Tier}(y) \rightarrow \mathrm{Liebt}(x,y))) \rightarrow (\exists z.\mathrm{Liebt}(z,x))))$$

Implikationen eliminiert:

$$(\forall x.(\neg(\forall y.(\neg\mathrm{Tier}(y) \vee \mathrm{Liebt}(x,y))) \vee (\exists z.\mathrm{Liebt}(z,x))))$$

Negationen nach innen:

$$(\forall x.((\exists y.(\mathrm{Tier}(y) \wedge \neg\mathrm{Liebt}(x,y))) \vee (\exists z.\mathrm{Liebt}(z,x))))$$

Quantorenskopus minimiert:

$$(\forall x.((\exists y.(\mathrm{Tier}(y) \wedge \neg\mathrm{Liebt}(x,y))) \vee (\exists z.\mathrm{Liebt}(z,x))))$$

Variablen umbenennen:

$$(\forall x_1.((\exists x_2.(\mathrm{Tier}(x_2) \wedge \neg\mathrm{Liebt}(x_1,x_2))) \vee (\exists x_3.\mathrm{Liebt}(x_3,x_1))))$$

Skolemisieren:

$$(\forall x_1.((\mathrm{Tier}(\mathrm{s}_1(x_1)) \wedge \neg\mathrm{Liebt}(x_1,\mathrm{s}_1(x_1))) \vee \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1)))$$

Allquantoren löschen:

$$((\mathrm{Tier}(\mathrm{s}_1(x_1)) \wedge \neg\mathrm{Liebt}(x_1,\mathrm{s}_1(x_1))) \vee \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1))$$

Und-/Oder- Ausmultiplizieren:

$$((\mathrm{Tier}(\mathrm{s}_1(x_1)) \vee \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1)) \wedge (\neg\mathrm{Liebt}(x_1,\mathrm{s}_1(x_1)) \vee \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1)))$$

Klauseln:

$$\{\{\mathrm{Tier}(\mathrm{s}_1(x_1)), \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1)\},$$
$$\{\neg\mathrm{Liebt}(x_1,\mathrm{s}_1(x_1)), \mathrm{Liebt}(\mathrm{s}_2(x_1),x_1)\}$$

Beispiel 2.2.27 *Als zweites Beispiel soll die folgende Formel in eine Klauselnormalform gebracht werden:*

$$((\forall x.((\forall y.\mathrm{Q}(x,y)) \rightarrow \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.(\neg(\exists y.\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z),z))))$$

Implikationen eliminiert:

$$((\forall x.(\neg(\forall y.\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.(\neg(\exists y.\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z),z))))$$

Negationen nach innen:

$$((\forall x.((\exists y.\neg\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee (\exists z.((\forall y.\neg\mathrm{P}(y)) \wedge \mathrm{Q}(\mathrm{f}(z),z))))$$

Quantorenskopus minimiert:

$$((\forall x.((\exists y.\neg\mathrm{Q}(x,y)) \vee \mathrm{P}(\mathrm{f}(x)))) \vee ((\forall y.\neg\mathrm{P}(y)) \wedge (\exists z.\mathrm{Q}(\mathrm{f}(z),z))))$$

Variablen umbenennen:

$$((\forall x_1.((\exists x_2.\neg\mathrm{Q}(x_1,x_2)) \vee \mathrm{P}(\mathrm{f}(x_1)))) \vee ((\forall x_3.\neg\mathrm{P}(x_3)) \wedge (\exists x_4.\mathrm{Q}(\mathrm{f}(x_4),x_4))))$$

Skolemisieren:

$$((\forall x_1.(\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1)))) \vee ((\forall x_3.\neg\mathrm{P}(x_3)) \wedge \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()),\mathrm{s}_2())))$$

Allquantoren löschen:

$$((\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee (\neg\mathrm{P}(x_3) \wedge \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()),\mathrm{s}_2())))$$

Und-/Oder- Ausmultiplizieren:

$$(((\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee \neg\mathrm{P}(x_3)) \wedge ((\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1)) \vee \mathrm{P}(\mathrm{f}(x_1))) \vee \mathrm{Q}(\mathrm{f}(\mathrm{s}_2()),\mathrm{s}_2())))$$

Klauseln:

$$\{\{\mathrm{Q}(\mathrm{f}(\mathrm{s}_2()),\mathrm{s}_2()),\mathrm{P}(\mathrm{f}(x_1)),\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1))\},$$

$$\{\neg\mathrm{P}(x_3),\mathrm{P}(\mathrm{f}(x_1)),\neg\mathrm{Q}(x_1,\mathrm{s}_1(x_1))\}$$

Resolution

Mit der Klauselform liegen uns die Formeln jetzt in der von der Resolution benutzten Form vor. Betrachten wir hierzu zunächst ein einfaches Beispiel:

Beispiel 2.2.28 *Seien die folgenden bekannten Fakten gegeben:*

$$\forall x : \text{Mensch}(x) \rightarrow \text{sterblich}(x)$$
$$\forall x : \text{Mensch}(x) \rightarrow \text{Mensch}(\text{Vater}(x))$$
$$\text{Mensch}(\text{Elvis})$$

Es soll hieraus bewiesen werden:

$$\text{sterblich}(\text{Vater}(\text{Elvis}))$$

Wir negieren die zu beweisende Aussage und transformieren in Klauselform:

$$\{\neg\text{Mensch}(x_1), \text{sterblich}(x_1)\} \tag{2.28}$$
$$\{\neg\text{Mensch}(x_2), \text{sterblich}(\text{Vater}(x_2))\} \tag{2.29}$$
$$\{\text{Mensch}(\text{Elvis})(x_2)\} \tag{2.30}$$
$$\{\neg\text{sterblich}(\text{Vater}(\text{Elvis}))\} \tag{2.31}$$

Nun würden wir gerne auf dieser Klauselmenge Resolventen bilden. Hier würden sich z.B. Klausel 2.28 und 2.31 anbieten. Hier taucht das Prädikat *sterblich* einmal negiert und einmal unnegiert auf. Allerdings sind die Argumente des Prädikats recht unterschiedlich: einmal hanelt es sich um eine Variabel und einmal um den Grundterm *Vater(Elvis)*.

Die Variablen in den Klauseln sind alle implizit allquantifiert, also steht eine Klausel, die Variablen enthält für alle Klauseln, in der diese Variablen durch Grundterme ersetzt wurden. In unserem Beispiel also:

$$\{\neg\text{Mensch}(\text{Elvis}), \text{sterblich}(\text{Elvis})\} \tag{2.32}$$
$$\{\neg\text{Mensch}(\text{Vater}(\text{Elvis})), \text{sterblich}(\text{Vater}(\text{Elvis}))\} \tag{2.33}$$
$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Elvis}))), \text{sterblich}(\text{Vater}(\text{vater}(\text{Elvis})))\} \tag{2.34}$$
$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis})))), \text{sterblich}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis}))))\} \tag{2.35}$$
$$\{\neg\text{Mensch}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis}))))), \text{sterblich}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Vater}(\text{Elvis})))))\} \tag{2.36}$$
$$\vdots$$

Klauseln mit Variablen stehen also für abzählbar undendlich viele Grundklauseln. Wir könnten jetzt Klausel 2.28 mit 2.33 resolvieren. Und damit hat man im Prinzip schon ein Beweisverfahren für die Prädikatenlogik. Man kann über einen Iterator die durch die prädikatenlogischen Klauseln definierten unendlich vielen Grundklauseln aufzählen. Dann lässt sich, da die unendlich viel Klauseln ja abzählbar sind, ein Iterator aller schreiben, der alle möglichen Resoventen findet. Wir erhalten dann einen unendlichen großen Suchbaum, in dem wir aber erschöpfend

nach der leeren Klausel, die den Widerspruch repräsentiert suchen können. Wenn also die leer Klausel per Resolution hergeleitet werden kann, werden wir sie finden. Was passiert hingegen, wenn es keinen Beweis gibt? Dann werden wir unter Umständen immer tiefer in dem unendlichen Suchbaum suchen. Die Suche terminiert nicht. Unser Beweisverfahren findet also ntweder einen Beweis oder terminiert nicht. Dieses sind gerade die wichtigsten Eigenschaften der Prädikatenlogik in Bezug auf die Berechenbarkeit:

Es gilt die Unentscheidbarkeit der Prädikatenlogik:

Satz 2.2.29 *Es ist unentscheidbar, ob eine geschlossene Formel der Prädikatenlogik allgemeingültig ist.*

Einen Beweis geben wir nicht. Der Beweis besteht darin, ein Verfahren anzugeben, das jeder Turingmaschine M eine prädikatenlogische Formel zuordnet, die genau dann ein Satz ist, wenn diese Turingmaschine auf dem leeren Band terminiert. Hierbei nimmt man TM, die nur mit einem Endzustand terminieren können. Da das Halteproblem für Turingmaschinen unentscheidbar ist, hat man damit einen Beweis für den Satz.

Satz 2.2.30 *Die Menge der allgemeingültigen Formeln der Prädikatenlogik ist rekursiv aufzählbar.*

Als Schlußfolgerung kann man sagen, dass es kein Deduktionssystem gibt (Algorithmus), das bei eingegebener Formel nach endlicher Zeit entscheiden kann, ob die Formel ein Satz ist oder nicht. Allerdings gibt es einen Algorithmus, der für jede Formel, die ein Satz ist, auch terminiert und diese als Satz erkennt.

Genau einen solchen Algorithmus haben wir oben beschrieben.

Resolvieren

Die eigentliche Resolutionsregel entspricht der Resolutionsregel der Aussagenlogik mit Hinzunahme der Unifikation.

Definition 2.2.31 (Resolutionsschritt, Resolvente)
Seien C_1 und C_2 zwei Klauseln, so dass es ein Atom $A \in C_1$ und ein Literal $\neg A' \in C_2$ gibt, so dass A und A' unifizierbar sind. Sei μ ein allgemeinster Unifikator for a und A'. Dann kann in einem Resolutionsschritt aus C_1 und C_2 die Resolvente C_3 gebildet werden mit:
$C_3 = \{\mu(x)|x \in C_1, x \neq A\} \cup \{\mu(x)|x \in C_2, x \neq \neg A\}$

Faktorisierung

Betrachten wir einmal folgende Formel:

$$\forall\, x \forall\, y(P(x) \vee P(y)) \wedge \forall\, u \forall\, v(\neg P(u) \vee \neg P(v)))$$

Es gibt keine Interpretation, die diese Formel erfüllt. Es handelt sich also um eine unerfüllbare Formel. Wir würden erwarten, dass per Resolution aus den Klauseln für diese Formel die leere Klausel abgeleitet werden kann. Die Formel entspricht folgenden zwei Klauseln:

$$\{P(x), P(y)\}, \{\neg P(u), \neg P(v)\}$$

Die Resolutionsregel liefert, selbst bei belie- biger Wiederholung, immer nur Klausel der Form:

$$\{P(y), \neg P(v)\}$$

jedoch nie die leere Klausel.

Somit kann der Resolutionskalkül mit der bisher vorgestellten Regel der Resolution noch nicht Widerlegungsvollständig sein. Es wird eine zusätzliche Regel benötigt, die Faktorisierung.

Definition 2.2.32 (Faktorisierung)
Seien C_1 eine Klausel und σ eine Substitution, die mindestens zwei Literale der Klausel C_1 unifiziert. Dann kann in einem Faktoriesierungsschritt aus C_1 die Klausel C_2 gebildet werden mit:
$C_2 = \{\sigma(x) | x \in C_1\}$

Durch die Faktorisierung wird die Anzahl der Literale in einer Klausel durch Anwendung einer Substitution verringert. Eine Klausel entspricht einer Mengel, in der es keine Auftreten von doppelten Elementen gibt.

Im obigen motivierenden Beispiel lassen sich beide Klauseln jeweils zu einelementigen Klauseln faktorisieren. Diese lassen sich dann in einem Reslutionsschritt zur leeren Klausel resolvieren.

Satz 2.2.33 *Der Resolutionskalkül der Prädikatenlogik mit den Regeln für Resolventenbildung und Faktorisierung ist widerlegungsvollständig, d.h. aus einer unerfüllbaren Klauselmenge, lässt sich die leere Klausel herleiten.*

Kapitel 3

Berechenbarkeit

Nachdem in den vorangegangenen Kapiteln verschiedene logische Kalküle kennengelernt wurden, soll in diesem Kapitel der Begriff der Berechenbarkeit mit Hilfe eines Kalküls spezifiziert werden. In den vorangegangen bachelorvorlesungen wurde die Menge der berechenbaren Funktionen über ein Maschinenmodell, der Turing-Maschine eingeführt. Ungefähr zeitgleich zu Turing hat Alonzo Church mit dem Lambda-Kalkül eine Definition der berechenbaren Funktionen gegeben. Hierbei wird über einen Kalkül ein Weg bewählt, der sehr stark die den Formalismen und Vorgehensweisen in der Prädikatenlogik ähnelt. Das interessante und zunächst gar nicht selbstverständliche Ergebnis ist, dass beide Defintionen genau die gleiche Menge berechenbarer Funktionen definieren.

3.1 Der Lambda-Kalkül

Dieses Kapitel wird nur die grobe Stichpunkte der Definition des Lambda-Kalküls enthalten. Die Leser sind auf das kurze Skript von Tobias Nipkow[Nip98] verwiesen. Dafür enthält dieses Kapitel eine komplette Implementierung des Lambda-Kalküls mit der als Beispielpplikation tatsächlich die Fakultätsfunktion (für sehr kleine Zahlen) berechnet werden kann.

Die Definition der Syntax des Lambda-Kalküls erfolgt wie in der Prädikatenlogik auf strukturell Induktive Weise:

Definition 3.1.1 (λ-Terme)
Sei gegeben eine abzählbare Menge $\mathcal{V} = \{x_1, x_2, \dots\}$ von Variablensymbolen. Dann sei Λ die Menge der λ-Terme definiert als die kleinste Menge von Wörtern mit der folgenden Eigenschaft:

- $\mathcal{V} \subseteq \Lambda$
- *Wenn $s, t \in \Lambda$ dann ist auch:* $(s\ t) \in \Lambda$.
- *Wenn $s \in \Lambda$ und $x \in \mathcal{V}$ dann ist auch* $\lambda x.s \in \Lambda$.

Diese Definition mündet wieder direkt in einen entsprechenden Haskelldatentypen:

Lambda.hs
```
1 {-# OPTIONS -fglasgow-exts #-}
2
```

```
3  module Lambda where
4  import ToLatex
5
6  data Lambda =
7    Var String
8   |App Lambda Lambda
9   |L String Lambda
10   deriving (Show,Eq)
```

Um die Lambda-Terme gut in gedruckter Form darzustellen werden sie als Instanz der Typklasse `ToLatex` definiert:

Lambda.hs

```
11 instance ToLatex Lambda where
12   toLatex (Var x) = x
13   toLatex (App e1 e2) = "("++toLatex e1++"\\and ~"++toLatex e2++")"
14   toLatex (L x e) = "\n\\lambda "++x++".\\and "++toLatex e
15
16 instance ToLatex [Lambda] where
17   toLatex es =
18    "\n "
19     ++concatWith "\n\n "
20         ["\\["++toLatex e++"\\]"|e<-es]
21     ++"\n\\normalsize "
```

Es folgen ein paar erste Beispiel

Lambda.hs

```
22 app [x] = x
23 app (x1:x2:xs) = app ((App x1 x2):xs)
24
25 concatWith _ []=""
26 concatWith _ [x]=x
27 concatWith sep (x:xs)=x++sep++concatWith sep xs
28
29 id = L "x" (Var "x")
30 true = L "x" (L "y" (Var "x"))
31 false = L "x" (L "y" (Var "y"))
32 wenn = L "c"$L"a_1"$L "a_2"$
33          App (App (Var "c") (Var "a_1")) (Var "a_2")
34
35 w1 = App (App (App wenn true) (Var "x")) (Var "y")
36 w2 = App (App (App wenn false)(Var "x")) (Var "y")
```

Der Lambda-Kalkül kennt Variablen, wie schon aus der Prädikatenlogik bekannt. In der Prädikatenlogik binden die Quantoren Variablennamen, im Lambda-Kalkül bindet das Symbol λ die Variablen. Variablen, die durch kein λ-Symbol gebunden sind, gelten als frei.

Definition 3.1.2 (frei Variablen)
Die Menge der freien Variablen eines λ-Terms sei definiert durch folgende Funktion FV:

- $FV(x) = \{x\}$ *für* $x \in \mathcal{V}$
- $FV((s\ t)) = FV(s) \cup FV(f)$ *mit* $s, t \in \Lambda$

- $FV(\lambda x.t) = FV(t)\{x\}$

Die entsprechende Haskellimplementierung, die die Menge der freien Variablen eines Lambda-Terms definiert, lässt sich durch die folgenden drei Zeilen ausdrücken:

Lambda.hs
```
37 freeVars (Var x) = [x]
38 freeVars (App e1 e2) = freeVars e1 ++ freeVars e2
39 freeVars (L x  e) = [y|y<-freeVars e,y/=x]
```

Entscheidend für den Lambda-Kalkül ähnlich wie auch in der Prädikatenlogik ist, die Substitution von freien Variablen:

Definition 3.1.3 (Substitution)
Eine Substition $\sigma = [x \to t]$ mit $x \in \mathcal{V}$ und $t \in \Lambda$ sei eine Abbildung $\Lambda \to \Lambda$, die in einem Term s die freien Auftreten von x durch t ersetzt. Wir schreiben $s[x \to t]$. Eine Substition sei rekursiv definiert mit:

- $x[x \to t] = t$
- $y[x \to t] = y$ *für* $x \neq y$
- $(s_1\ s_2)[x \to] = (s_1[x \to t]\ s_2[x \to t])$
- $(\lambda x.s)[x \to t] = \lambda x.s$
- $(\lambda y.s)[x \to t] = \lambda x.(s[x \to t])$, *falls* $x \neq y$ *und* $y \notin FV(t)$
- $(\lambda y.s)[x \to t] = \lambda z.(s[y \to z][x \to t])$, *falls* $x \neq y$ *und* $z \notin FV(t) \cup FV(s)$

Die letzten beiden Punkte der Definition sollen verhindern, dass durch die Anwendung einer Substitution eine Variable sozusagen ausversehen gebunden wird. Hierfür soll die Umbenennung einer Variabel im letzten Punkt sorgen.

Man beachte, dass nach dieser Definition eine Substitution im Lambda-Kalkül genau eine Variabel substituiert. Anders als in der Prädikatenlogik, in der beliebig viele Variablen substituiert wurden.

Es folgt die Haskellimplementierung der Substitution:

Lambda.hs
```
40 sub x1 arg v@(Var x2)
41  |x1==x2 = arg
42  |otherwise = v
43 sub x arg (App e1 e2) = App (sub x arg e1)(sub x arg e2)
44 sub x1 arg (L x2 body)
45  |x1==x2 = (L x2 body)
46  |x2 `elem` (freeVars arg)
47          = L newVar (sub x1 arg (sub x2 (Var newVar) body))
48  |otherwise = (L x2 (sub x1 arg body))
49    where
50      (newVar:_) = freshVars (App body arg)
51
52 allVars = ["x_{"++show n++"}"|n<-[1,2..]]
53 freshVars e = [v|v<-allVars,not (v `elem` freeVars e)]
```

Mit diesen Begriffen lässt sich der Ableitungsschritt für den Lambda-Kalkül definieren. Er wird als β-Reduktion bezeichnet. Die β-Reduktion erzeugt aus einem λ-Term einen neuen Term:

Definition 3.1.4 (β-Reduktion)
Die β-Reduktion im Zeichen $\rightarrow_\beta$ sei die kleinste Relation auf Λ mit folgenden Eigenschaften:

- $(\lambda x.s\ t) \rightarrow_\beta$.
- *Wenn* $s_1 \rightarrow_\beta S_2$ *dann auch* $(s_1\ t) \rightarrow_\beta (s_2\ t)$ *für alle* $t \in \Lambda$
- *Wenn* $s_1 \rightarrow_\beta S_2$ *dann auch* $(t\ s_1) \rightarrow_\beta (t\ s_2)$ *für alle* $t \in \Lambda$
- *Wenn* $s_1 \rightarrow_\beta S_2$ *dann auch* $\lambda x.s_1 \rightarrow_\beta \lambda x.s_2$

Der Subterm eines *lambda*-terms, an dem eine β-Reduktion möglich ist, wird als Redex (kurz für *reducable expression*) bezeichnet.

Im Lambda-Kalkül wird nun per Anwendung der Reduktionsregel der Wert eines Lambda-Ausdruckes ausgerechnet. Ein Lambda-Term, auf dem kein Reduktionsschritt mehr angewendet werden kann, der also keinen Redex mehr enthält, wird als Normalform bezeichnet. Die Normalform eines Lambda-Terms stellt quasi das Ergebnis ocder den Wert des Terms dar.

Beispiel 3.1.5 *Eine Reduktion kleine Beispielreduktion eines Lambda-Terms:*

$((\lambda c.\lambda a_1.\lambda a_2.((c\ a_1)\ a_2)\ \lambda x.\lambda y.x)\ x)\ y)$
$((\lambda a_1.\lambda a_2.((\lambda x.\lambda y.x\ a_1)\ a_2)\ x)\ y)$
$(\lambda a_2.((\lambda x_1.\lambda y.x_1\ x)\ a_2)\ y)$
$((\lambda x_1.\lambda x_2.x_1\ x)\ y)$
$(\lambda x_2.x\ y)$
x

Die β-Reduktion entspricht einem Funktionsaufruf einer Funktion, wie man ihn auch aus herkömmlichen Programmiersprachen kennt. Auch dort wird der formale Parameter durch das konkrete Argument ersetzt.

Mit der folgenden Implementierung in Haskell ist der Lambda-Kalkül bereits vollständig implementiert:

Lambda.hs

```
54 reduce (App (L x body) arg) = (True,sub x arg body)
55 reduce (App e1 e2)
56  |res1      = (res1,App e1' e2)
57  |otherwise = (res2,App e1  e2')
58   where
59     (res1,e1') = reduce e1
60     (res2,e2') = reduce e2
61 reduce (L x body) = (res, L x body')
62   where
63      (res,body') = reduce body
64 reduce v@(Var x) = (False,v)
65
66 reduction e
67  |reduced = e:reduction e1
68  |otherwise = [e]
69  where
70    (reduced,e1)= reduce e
```

3.1.1 Konfluenz

Der aufmerksame Leser wird festgestellt haben, dass es sein kann, dass auf ein und denselben Term auf verschiedene Weise an verschiedenen Stellen die β-Reduktion durchgeführt werden kann. Jetzt ergeben sich interessante Fragen:

- Wenn ich mich für einen Redex entscheide, kann ich dann in eine Sackgasse geraten?
- Kann es sein dass ich so ungeschickt immer den Redex wähle, dass die Reduktion nicht terminiert, obwohl durch andere Wahl auf eine Normalform reduziert werden kann?
- Gibt es überhaupt eine eindeutige Normalform?

Diese Fragestelungen münden alle in die Frage der Konfluenz eines Termersetzungssystems. Die entsprechenden Begriffe, der Konfuenz, lokalen Konfluenz und Karo- (Diamant-)eigenschaft, sowie die Reduktionsstrategien der normalen und der applikativen Reihenfolge entnehme der Leser dem Nipkow-Skript.

3.1.2 Church Numerale

Wir sind mit dem Lambda-Kalkül angetreten, um einen Begriff der berechenbaren Funktionen zu definieren. Es soll sich dabei um Funktionen auf den natürlichen Zahlen handeln. Bisher ist noch unklar, in welcher Weise Lambda-Terme in der Lage sein sollen, natürliche Zahlen zu kodieren. Church gibt Lambda-Terme an, die die natürlichen Zahlen repräsentieren:

$$\begin{aligned} 0 &= \lambda f.\lambda x.x \\ 1 &= \lambda f.\lambda x.(f\ x) \\ 2 &= \lambda f.\lambda x.(f\ (f\ x)) \\ 3 &= \lambda f.\lambda x.(f\ (f\ (f\ x))) \end{aligned}$$

Es folgen die entsprechenden Terme als Haskellimplementierung:

Lambda.hs

```
71 n0 = L "f"$L "x" (Var "x")
72 n1 = L "f"$L "x" (App (Var "f")(Var "x"))
73 n2 = L "f"$L "x" (App (Var "f")(App (Var "f") (Var "x")))
74 n3 = L "f"$L "x" (App (Var "f")(App (Var "f")(App (Var "f") (Var "x"))))
75
76 n n= L "f"$L "x" $numberApp n
77
78 numberApp 0 = Var "x"
79 numberApp n = App (Var "f")(numberApp (n-1))
```

Die Definition der einzelnen natürlichen Zahlen als Lambda-Terme ist zunächst ziemlich willkürlich und kann nur gerechtfertigt werden, wenn diese Terme im Zusammenspiel mit bestimmten Operationen das gewünschte Verhalten bringen. Wir erwarten Lambda-Terme, die es ermöglichen zwei Zahlen zu addieren oder zu multiplizieren, Vorgänger und Nachfolger zu ermitteln, oder zu testen, ob ein bestimmter Lambda-Term die Null darstellt.

Church gibt in seiner Codierung entsprechende Lambda-Terme an:

Der folgende Ausdruck kodiert die Addition:

$$\lambda n.\lambda m.\lambda f.\lambda x.((n\ f)\ ((m\ f)\ x))$$

Man betrachte die folgende Reduktion der Addition der 1 mit 1:

$$((\lambda n.\lambda m.\lambda f.\lambda x.((n\ f)\ ((m\ f)\ x))\ \lambda f.\lambda x.(f\ x))\ \lambda f.\lambda x.(f\ x))$$

$$(\lambda m.\lambda f.\lambda x.((\lambda f.\lambda x.(f\ x)\ f)\ ((m\ f)\ x))\ \lambda f.\lambda x.(f\ x))$$

$$\lambda f.\lambda x.((\lambda f.\lambda x.(f\ x)\ f)\ ((\lambda f.\lambda x.(f\ x)\ f)\ x))$$

$$\lambda f.\lambda x.(\lambda x.(f\ x)\ ((\lambda f.\lambda x.(f\ x)\ f)\ x))$$

$$\lambda f.\lambda x.(f\ ((\lambda f.\lambda x.(f\ x)\ f)\ x))$$

$$\lambda f.\lambda x.(f\ (\lambda x.(f\ x)\ x))$$

$$\lambda f.\lambda x.(f\ (f\ x))$$

Die schließlich erzielte Normalform ist die Darstellung der Zahl 2.

Der folgende Term kodiert die Multiplikation:

$$\lambda n.\lambda m.\lambda f.(n\ (m\ f))$$

Der nächste Term die Vorgängerfunktion (also -1).

$$\lambda n.\lambda f.\lambda x.(((n\ \lambda g.\lambda h.(h\ (g\ f)))\ \lambda u.x)\ \lambda u.u)$$

Der nachfolgende Term die Nachfolgerfunktion (also +1).

$$\lambda n.\lambda f.\lambda x.(f\ ((n\ f)\ x))$$

Und schließlicxh lässt sich mit folgenden Term testen, ob das Argument die Zahl 0 ist.

$$\lambda n.((n\ \lambda x.\lambda x.\lambda y.y)\ \lambda x.\lambda y.x)$$

Hierbei sind die Bool‘schen-Werte für wsahr und falsch und die Fallunterscheidung aufgrund dieser Werte zu definieren. Der aufmerksame Leser wird dieses schon in den ersten Beispielen im Haskellcode gefunden haben.

Es folgen die eben codierten Operationen als Haskellcode:

Lambda.hs

```
80 isZero=L"n"$ App (App (Var "n")(L "x" false)) true
81
82 succc = L "n"$L"f"$L"x"$
83          App (Var "f")(App (App (Var "n")(Var "f" ))(Var "x"))
84
85 predd =
86  L"n"$L"f"$L"x"$
87   App (App (App (Var "n")
88                 (L"g"$L"h"$App (Var "h")(App (Var"g")(Var"f"))))
89              (L"u"$Var "x"))
90       (L"u"$Var "u")
91
92 add =
93  L "n"$L"m"$L "f"$L"x"$
94   App (App (Var "n") (Var "f"))
95       (App (App (Var "m")(Var "f"))(Var "x"))
96
97 mult = L "n"$L"m"$L "f"$ App (Var "n")(App (Var "m")(Var "f"))
```

3.1.3 Rekursion

Nachdem nun einfache Funktionen auf den natürlichen Zahlen definiert sind, ist der entscheidene Trick auf dem Weg zur Darstellung aller berechenbaren Funktionen, auch eine rekursive Berechnung darzustellen. Der Lambda-Kalkül hat nicht wie gängige Programmiersprachen die Möglichkeit eine Funktion rekursiv zu definieren. Es wird also eine nichtrekursive Darstellung für ansonsten rekursiv definierte Funktionen benötigt.

In gängigen Programmiersprachen wird eine rekursive Funktion definiert durch:

$$f(x) = e$$

Dabei taucht im Ausdruck e in der Regel das Argument x wieder auf, aber auch das gerade definierte Funktionssymbol f.

Im lambda-Kalkül wird der formale Parameter x durch ein λ-Symbol gebunden:

$$f = \lambda x.e$$

In e taucht so immer noch das Funktionssymbol f auf, das im Lambda-Kalkül nicht durch die obige Definition rekursiv definiert werden darf.

Wenn im Funktionsrumpf von der Funktion selbst gesprochen werden soll, so tun wir eimal so, als wenn uns jemand die Funktion als Argument übergibt. Der entsprechende Lambda-Ausdruck wäre also:

$$\lambda f.\lambda x.e$$

Gib mir mich selbst als Argument, dann weiß ich wie ich funktioniere. es führt also zur Definition der rekursiven Funktion:

$$f = (\lambda f.\lambda x.ef)$$

Damit ist f also ein Fixpunkt des Ausdrucks: $\lambda f.\lambda x.e$.

Hierbei ist entscheidend einen Fixpunktoperator zur Verfügung zu haben, mit der folgenden Fixpunkteigenschaft:

$$(fixt) = (t(fixt))$$

Denn so kann die rekursive Funktion degfiniert werden als:

$$f = (fix(\lambda f.\lambda x.e))$$

Denn es ergibt sich für einen konkreten Argumentausdruck e_2 bei der Anwednung dieser Funktion auf das Argument:

$$\begin{array}{ll} & ((fix\ \lambda f.\lambda x.e)\ e_2) \\ \rightarrow_\beta & ((\lambda f.\lambda x.e\ (fix\ \lambda f.\lambda x.e))\ e_2) \\ \rightarrow_\beta & \lambda x.e[f \mapsto (fix(\lambda f.\lambda x.e))]e_2 \end{array}$$

Es ist also genau das rekursive Verhalten zu beobachten. Das Funktionssymbol f wird in der Definition wieder durch die komplette Definition ersetzt.

Und tatsächlich gibt es lambda-Terme, die das an den Fixpunktoperator fix gewünschte Verhalten zeigen. Einer ist der soganennte Y-Operator, mit folgender Definition:

$$Y := \lambda h.(\lambda x.(h\ (x\ x))\ \lambda x.(h\ (x\ x)))$$

Jetzt können wir beliebige rekursive Funktionen im Lambda-Kalkül definieren. In der Haskellimplementierung soll zum Abschluss die Implementierung der Fakultät als Lambda-Term stehen. Hierzu wird zunächst der Y-Operator implementiert:

Lambda.hs

```
98  rec =
99   L"h"$
100   App (L"x"$ App (Var "h")
101                  (App(Var"x")(Var"x")))
102       (L"x"$ App (Var "h")
103                  (App(Var"x")(Var"x")))
```

Dann folgt die Definition der Fakultät als Lambda-Ausdruck, wobei angenommen wird, dass die Funktion erst noch als Parameter h übergeben wird. Der entsprchende Lambda-Ausdruck ist:

$$\begin{array}{l} \lambda h.\lambda n.(((\lambda c.\lambda a_1.\lambda a_2.((c\ a_1)\ a_2)\ (\lambda n.((n\ \lambda x.\lambda x.\lambda y.y)\ \lambda x.\lambda y.x)\ n))\ \lambda f.\lambda x.(f\ x)) \\ ((\lambda n.\lambda m.\lambda f.(n\ (m\ f))\ n)\ (h\ (\lambda n.\lambda f.\lambda x.(((n\ \lambda g.\lambda h.(h\ (g\ f)))\ \lambda u.x)\ \lambda u.u)\ n)))) \end{array}$$

Mit den bisher schon definierten Lambda-Ausdrücken lässt er sich in Haskell wie folgt definieren:

Lambda.hs
```
104 facAux =
105  L"h"$L"n"$
106   app [wenn
107       ,App isZero (Var "n")
108       ,n 1
109       ,app [mult,Var "n",App (Var "h") (App predd (Var "n"))]]
```

Die Fakultät ist schließlich der Fixpunkt dieser Funktion:

Lambda.hs
```
110 factorial = App rec facAux
```

Dieses mündet also in folgenden Lambda-Ausdruck, der die Fakultätsfunktion vollständig und rekursionsfrei definiert:

$$(\lambda h.(\lambda x.(h\ (x\ x))\ \lambda x.(h\ (x\ x)))\ \lambda h.\lambda n.(((\lambda c.\lambda a_1.\lambda a_2.((c\ a_1)\ a_2)\ (\lambda n.((n\ \lambda x.\lambda x.\lambda y.y)\ \lambda x.\lambda y.x)\ n))\ \lambda f.\lambda x.(f\ x))\ ((\lambda n.\lambda m.\lambda f.(n\ (m\ f))\ n)\ (h\ (\lambda n.\lambda f.\lambda x.(((n\ \lambda g.\lambda h.(h\ (g\ f)))\ \lambda u.x)\ \lambda u.u)\ n)))))$$

Dieser Ausdruck auf eine der entsprechenden Lambda-Terme, die die natürlichen Zahlen repräsentieren, anggewendet, reduziert zu dem Lambda-Term, der die Fakultät des Arguments darstellt.

Anhang A

Nützliche Hilfsprogramme

ToLatex.hs

```
111 module ToLatex where
112
113 import System.Cmd
114 import System.Posix.Process
115
116 class ToLatex a where
117     toLatex:: a -> String
118
119 latexStart
120  = "\\documentclass[10pt,a4paper,oneside]{article}\n\
121    \\\usepackage[german]{babel}\n\
122    \\\setlength{\\parindent}{0pt}\n\
123    \\\setlength{\\parskip}{1ex plus 0.5ex minus 0.2ex}\n\
124    \\\usepackage{qtree}\n\
125    \\\usepackage{amssymb}\n\
126    \\\begin{document}\n"
127
128 latexEnd = "\n\\end{document}"
129
130 pdfLatex file = rawSystem "pdflatex" [file]
131
132 concatWith sep [] = ""
133 concatWith sep [a] = a
134 concatWith sep (x:xs) = x++sep++concatWith sep xs
135
136
137 showPDF file  = do
138   ex <- rawSystem "acroread" [file]
139   forkProcess $return ()
140
141 latex  = latexFile "test"
142
143 latexF  title xs = latexFile "test" title ("$"++toLatex xs++"$")
144
145 latexFile fileName title content = do
146   writeFile (fileName++".tex")
```

```
147     (latexStart++"\\title{"++title
148                 ++"}\n\\maketitle\n\n"++content++latexEnd)
149   pdfLatex (fileName++".tex")
150   showPDF (fileName++".pdf")
```

LatexPropositions.hs

```
1  module LatexPropositions
2    (module PropositionalLogic, module LatexPropositions) where
3
4  import PropositionalLogic
5  import ToLatex
6
7
8  latexAll = latex "alle Formeln" $toLatex$take 500$allformulae ["A","B","C"]
9
10 m3 taut =  content
11  where
12   f1 = Not taut
13   f2 = noArrows f1
14   f3 = notInside f2
15   f4 = clauselForm f3
16   f5 = mkClauses f4
17   content =
18         "Als Tautologie zu beweisende Formel: \\\\\n$"
19      ++ toLatex taut++"$\\\\\n"
20      ++ "Negat der Formel:\\\\\n$"++toLatex f1++"$\\\\\n"
21      ++ "Elemination der Pfeiloperatoren:\\\\\n$"++toLatex f2++"$\\\\\n"
22      ++ "Negation direkt vor die Atome:\\\\\n$"++toLatex f3++"$\\\\\n"
23      ++ "Klauselform:\\\\\n$"++toLatex f4++"$\\\\\n"
24      ++ "Klauselmenge:\\\\\n$"++toLatex f5++"$\\\\\n"
25      ++ "Beweis:\\\\\n"++(showProof$snd$head$proofs f5)
26
27 m4 ts= latex "Resolutionsbeweise von Tautologien der Aussagenlogik"
28        $foldr1 (\x y -> x++"\\eject\n"++y) (map m3 ts)
29
30 m5 = m4 ts
31
32 natweis fileName f
33   = latexFile
34       fileName
35       ("Ableitungskette f\\\"ur: $"++toLatex f++"$") -- "
36       (concat$take 100000 ["$"++toLatex f1++"$\n\n"|f1<-beweis f])
```

Main.hs

```
1  module Main where
2  import LatexPropositions
3  import PropositionalLogic
4
5  main=do
6   natweis "f00" f00
7   natweis "f01" f01
8   natweis "f02" f02
9   natweis "f03" f03
10  natweis "f04" f04
```

```
11 natweis "f05" f05
12 natweis "f06" f06
13 natweis "f07" f07
14 natweis "f08" f08
15 natweis "f09" f09
16 natweis "f10" f10
17 natweis "f11" f11
18 natweis "f12" f12
19 natweis "f13" f13
20 natweis "f14" f14
```

Klassenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Nip98] Tobias Nipkow. Lambda-Kalkül. Skript zur Vorlesung, TU München, 1998. www4.informatik.tu-muenchen.de/lehre/vorlesungen/lambda/SS98/lambda.ps.gz.

[Sch08] Peter Schmitt. Formale Systeme. Skript zur Vorlesung, Uni Karlsruhe, 2008. `i12www.ira.uka.de/~pschmitt/FormSys/FSSkript.pdf`.

[Smu68] Raymond M. Smullyan. *First-order logic [by] Raymond M. Smullyan.* Springer-Verlag, Berlin, New York [etc.], 1968.

[Spe86] Volker Sperschneider. Logik. Handschriftliches Vorlesungskript der Universität Karlsruhe, 1986.

[SS06] Manfred Schmidt-Schauß. Einführung in die künstliche intelligenz. Skript zur Vorlesung, 2006. `http://www.ki.informatik.uni-frankfurt.de/lehre/SS2006/KI/main.html`.