

Logik und formale Methoden

Vorlesungsskript

von
Burkhardt Renz

Wintersemester 2020/21

Burkhardt Renz
Technische Hochschule Mittelhessen
Rev 0.47 – 13. November 2020

© 2020 by Burkhardt Renz



Dieses Dokument ist lizenziert unter einer Creative Commons Namensnennung - Weitergabe unter gleichen Bedingungen 4.0 International Lizenz (siehe <http://creativecommons.org/licenses/by-sa/4.0/>).

Inhaltsverzeichnis

Inhaltsverzeichnis	ii
1 Einleitung	1
1.1 Klassiker der Logik	1
1.2 Mathematische Logik	6
1.3 Logik und Informatik	11
1.4 Programm der Veranstaltung	14
I Aussagenlogik	16
2 Aussagen und Formeln	17
3 Die formale Sprache der Aussagenlogik	20
4 Die Semantik der Aussagenlogik	26
4.1 Modell, Belegung	28
4.2 Wahrheitstafel	30
4.3 Semantische Äquivalenz und Substitution	31
4.4 Boolesche Operatoren und funktionale Vollständigkeit .	32
5 Das Beweissystem des natürlichen Schließens	34
5.1 Schlussregeln	36
5.2 Beispiele für das natürliche Schließen	38
5.3 Beweisstrategien	41
5.4 Eigenschaften der Herleitbarkeit \vdash	43
5.5 Vollständigkeit des natürlichen Schließens	46
6 Normalformen	59
6.1 Negationsnormalform NNF	59
6.2 Konjunktive Normalform CNF	60
6.3 Disjunktive Normalform DNF	62
6.4 Normalformen und Entscheidungsprobleme	62
7 Die Komplexität des Erfüllbarkeitsproblems	64
7.1 Das Erfüllbarkeitsproblem	64

7.2 Komplexität von Algorithmen	65
7.3 Die Komplexität des Erfüllbarkeitsproblems	68
8 Hornlogik	70
9 Erfüllbarkeit und SAT-Solver	73
9.1 DIMACS-Format	73
9.2 Tseitin-Transformation	75
9.3 DPLL und CDLC	76
10 Anwendungen der Aussagenlogik in der Softwaretechnik	83
10.1 Anwendungen von SAT-Techniken	83
10.2 Statische Codeanalyse	84
10.3 Featuremodelle für (Software-)Produktlinien	85
II Prädikatenlogik	89
11 Objekte und Prädikate	90
11.1 Elemente der Sprache der Prädikatenlogik	91
11.2 Prädikate und Relationen	92
12 Die formale Sprache der Prädikatenlogik	93
12.1 Signatur, Terme, Formeln	93
12.2 Freie und gebundene Variablen	95
12.3 Substitution	96
13 Semantik der Prädikatenlogik	98
13.1 Modell/Struktur	98
13.2 Semantische Folgerung und Äquivalenz	100
13.3 Fundamentale Äquivalenzen der Prädikatenlogik	101
14 Natürliches Schließen in der Prädikatenlogik	102
14.1 Schlussregeln	103
14.2 Beispiele	104
14.3 Vollständigkeit des natürlichen Schließens	111
15 Unentscheidbarkeit der Prädikatenlogik	113
15.1 Das Postsche Korrespondenzproblem	114
15.2 Die Unentscheidbarkeit des Gültigkeitsproblems in der Prädikatenlogik	114
15.3 Der Sonderfall endlicher Universen	116
16 Anwendungen der Prädikatenlogik in der Software-technik	120

16.1 Spezifikation und Analyse von Eigenschaften von Pro- grammen	120
16.2 Analyse von Software mit Alloy	122
III Lineare Temporale Logik	131
17 Dynamische Modelle	132
17.1 Konzept der Transitionssysteme	132
17.2 Beispiel eines Programms mit zwei Threads	133
17.3 Temporale Logik	134
18 Die formale Sprache der linearen temporalen Logik (LTL)	136
19 Die Semantik der linearen temporalen Logik (LTL)	138
19.1 Kripke-Struktur	138
19.2 Äquivalenzen von Formeln der LTL	143
19.3 Typische Aussagen in der LTL	144
19.4 Büchi-Automaten	146
19.5 Erfüllbarkeit in der LTL	152
19.6 Auswertung von Formeln in Kripke-Strukturen	161
20 Natürliches Schließen in der LTL	165
21 Anwendungen der LTL in der Softwaretechnik	166
21.1 Model Checking	166
21.2 Zielemodell in der Anforderungsanalyse	166
Literaturverzeichnis	167

Kapitel 1

Einleitung

Die Wissenschaft der Logik befasst sich mit den *Formen* des Denkens unter Absehen vom jeweiligen Inhalt. Es geht um das Argumentieren, das Überzeugen, um *zwingende* Schlussfolgerungen. Es ist die Wissenschaft, bei der sich das Denken gewissermaßen mit sich selbst beschäftigt. Dies kann man auf sehr unterschiedliche, auch fragwürdige Weise tun. In der *formalen Logik* geht es um ein *Kalkül*, bei dem aus gegebenen Prämissen Schlußfolgerungen gezogen werden durch die Manipulation von *Symbolen* — etwas was Computer gut können, ohne auch nur den geringsten Schimmer von der Bedeutung dieser Symbolen zu haben.

1.1 Klassiker der Logik

Ein Beispiel für eine zwingende Schlussfolgerung ist etwa folgende Argumentation über natürliche Zahlen:

$$\begin{array}{c} \text{Wenn } p > 2 \text{ und Primzahl ist, dann ist } p + 1 \text{ keine Primzahl} \\ 7 > 2 \text{ und 7 ist prim} \\ \hline \text{Also: } 7 + 1 = 8 \text{ ist keine Primzahl} \end{array}$$

Kenntnisse über natürliche Zahlen helfen, diese Argumentation zu überprüfen: Wenn die natürliche Zahl p größer als 2 und prim ist, dann muss p ungerade sein, denn sonst wäre p durch 2 teilbar, also keine Primzahl. Ist p ungerade, dann ist $p+1$ gerade, also da größer als 2 garantiert keine Primzahl. Die erste Prämisse der Argumentation ist also zutreffend. Die zweite auch — man kann ja einfach nach den echten Teilern von 7 suchen und findet nur die 1. Beide Prämissen treffen zu, die Schlussfolgerung über die Zahl 8 folglich auch, das *Also* ist gerechtfertigt.

Wenn man die folgende Schlussfolgerung betrachtet, dann hat sie strukturell denselben Aufbau:

Wenn es regnet, ist die Straße nass.
Es regnet.

Also: Die Straße ist nass.

Diese Struktur, nämlich

$$\frac{P \rightarrow Q \\ P}{\text{Also: } Q}$$

nennt man *Modus Ponens*, aus dem Lateinischen *ponere* = stellen, setzen, also der setzende Modus, die Schlussfigur, die die Aussage Q „setzt“.

Wir finden sie auch bei folgender Argumentation —

Wenn die Erde eine Kugel ist, dann ist 7 eine Primzahl.
Die Erde ist eine Kugel.

Also: 7 ist eine Primzahl.

— und rein formal betrachtet sind auch hier beide Prämisse zutreffend, also die Schlussfolgerung zwingend. Aber während bei dem Beispiel mit der nassen Straße ein inhaltlicher Zusammenhang der Aussagen besteht, ist dies in diesem Beispiel offensichtlich nicht der Fall. Die Aussage über die Erde und die Aussage über die Zahl 7 schließen sozusagen *windschief* aneinander vorbei. Anders gesagt: Die rein formale Betrachtung hat auch ihren Preis, reichlich *unsinnige* Aussagen werden als zutreffend erachtet. In der Symbiose von formaler Logik und Informatik spielt dies allerdings keine Rolle, denn die Systeme, die die Informatik baut (und sie selbst) sind *selbst* formale Systeme.

Die Beobachtung, dass man die *formale Struktur* von Argumentationen ohne Kenntnisse der Inhalte betrachten und sie als zwingende *Schlussregeln* sehen kann, hat vielleicht als erster Aristoteles¹ gemacht. Er hat *Syllogismen* (aus dem Altgriechischen für „Zusammenrechnen“, „logischer Schluss“) betrachtet:

Eine Deduktion (*syllogismos*) ist also ein Argument, in welchem sich, wenn etwas gesetzt wurde, etwas anderes als das Gesetzte mit Notwendigkeit durch das Gesetzte ergibt.

– Aristoteles: Topik I 1, 100a25-27

¹ Aristoteles, griechischer Philosoph, 384 - 322 v. Chr.

Die Syllogismen haben immer zwei Prämisse und eine Konklusion. Ein oft zitiertes Beispiel ist:

Alle Menschen sind sterblich.
Alle Griechen sind Menschen.
<i>Also:</i> Alle Griechen sind sterblich.

Dieser Syllogismus wird *Modus Barbara* genannt, weil er von zwei All-Aussagen zu einer Schlussfolgerung führt, die auch eine All-Aussage ist.

Aus der Perspektive von Aristoteles ergeben sich Fragen wie:

- Was sind *gültige* Schlussregeln?
- Was ist ein *Beweis*?
- Wann ist eine Theorie *widerspruchsfrei*?
- ...

Man kann die Syllogismen als eine frühe Variante der Prädikatenlogik mit unären Prädikaten sehen. Erst 1879 hat Gottlob Frege² in seiner „Begriffsschrift“ die Prädikatenlogik formalisiert und damit die Grundlage gelegt für die heutige formale Logik.

Aristoteles betont im obigen Zitat, dass in einem Syllogismus die Schlussfolgerung mit *Notwendigkeit* aus den Prämissen folgt. Dies zeichnet ja gerade die Logik aus, wie in folgendem Zitat auch Goethe³ sie charakterisiert:

Mephistopheles:

Erklärt Euch, eh Ihr weiter geht,
Was wählt Ihr für eine Fakultät?

Schüler:

Ich wünschte recht gelehrt zu werden,
Und möchte gern, was auf der Erden
Und in dem Himmel ist, erfassen,
Die Wissenschaft und die Natur.

Mephistopheles:

Da seid Ihr auf der rechten Spur;
Doch müßt Ihr Euch nicht zerstreuen lassen.

Schüler:

Ich bin dabei mit Seel und Leib;

²Gottlob Frege, deutscher Logiker, Mathematiker und Philosoph, 1848 - 1925.

³Johann Wolfgang von Goethe, deutscher Dichter, 1749 - 1832

Doch freilich würde mir behagen
Ein wenig Freiheit und Zeitvertreib
An schönen Sommerfeiertagen.

Mephistopheles:

Gebraucht der Zeit, sie geht so schnell von hinten,

Doch Ordnung lehrt Euch Zeit gewinnen.

Mein teurer Freund, ich rat Euch drum

Zuerst Collegium Logicum.

Da wird der Geist Euch wohl dressiert,

In spanische Stiefeln eingeschnürt,

Daß er bedächtiger so fortan

Hinschleiche die Gedankenbahn,

Und nicht etwa, die Kreuz und Quer,

Irrlichteliere hin und her.

Dann lehret man Euch manchen Tag,

Daß, was Ihr sonst auf einen Schlag

Getrieben, wie Essen und Trinken frei,

Eins! Zwei! Drei! dazu nötig sei.

Zwar ist's mit der Gedankenfabrik

Wie mit einem Weber-Meisterstück,

Wo ein Tritt tausend Fäden regt,

Die Schifflein herüber hinüber schießen,

Die Fäden ungesehen fließen,

Ein Schlag tausend Verbindungen schlägt.

Der Philosoph, der tritt herein

Und beweist Euch, es müßt so sein:

Das Erst wär so, das Zweite so,

Und drum das Dritt und Vierte so;

Und wenn das Erst und Zweit nicht wär,

Das Dritt und Viert wär nimmermehr.

Das preisen die Schüler allerorten,

Sind aber keine Weber geworden.

Wer will was Lebendigs erkennen und beschreiben,

Sucht erst den Geist heraus zu treiben,

Dann hat er die Teile in seiner Hand,

Fehlt, leider! nur das geistige Band.

– Goethe: Faust - Der Tragödie erster Teil, Vers 1896 ff.

Goethe betont die Strenge, das Normative, das *Zwingende* der Logik — und betrauert das ebenso: ihm fehlt dabei das Lebendige und das „geistige Band“!

Auch Hegel⁴ hat einen Einwand gegen die Sichtweise des formalen Schlie-

⁴Georg Wilhelm Friedrich Hegel, deutscher Philosoph, 1770 - 1831.

ßens:

Es ist überhaupt eine bloß subjektive Reflexion, welche die Beziehung der Terminorum in abgesonderte Prämissen und einen davon verschiedenen Schlußsatz trennt:

Alle Menschen sind sterblich,
Cajus ist ein Mensch,
Also ist er sterblich.

Man wird sogleich von Langeweile befallen, wenn man einen solchen Schluß heranziehen hört; – dies röhrt von jener unnützen Form her, die einen Schein von Verschiedenheit durch die abgesonderten Sätze gibt, der sich in der Sache selbst sogleich auflöst. Das Schließen erscheint vornehmlich durch diese subjektive Gestaltung als ein subjektiver *Notbehelf*, zu dem die Vernunft oder der Verstand da ihre Zuflucht nehme, wo sie nicht *unmittelbar* erkennen könne.

– Hegel: Logik II, Werke Bd. 6, S.358

Formales Schließen in der Form eines Aristotelischen Syllogismus erscheint Hegel als „langweilig“, weil man gewissermaßen als Schlussfolgerung nur herausbekommt, was man bereits sowieso in die Prämissen hineingesteckt hat. Für ihn ist es nur der *Schein* der Verschiedenheit der Aussagen, der die Trivialität des Schlusses ausmacht.

Hegel selbst interessiert in seiner Logik etwas anderes: Er analysiert die Leistungen des Denken im „Allgemeinen“: Was sagt jemand, wenn er sagt, dass *A* die *Bedingung* für *B* ist? Was bedeutet es, *A* als *Grund* für *B* zu bezeichnen? Was macht das *Wesen* eines untersuchten Gegenstandes aus? Allgemein: wie geht das Denken im Begreifen einer Sache vor?

Zwei Perspektiven auf Logik

Ein erstes Fazit dieses schnellen (und etwas eklektizistischen) Blicks auf Klassiker der Logik ist, dass man Logik durchaus unterschiedlich sehen kann:

- **Erste Perspektive** Logik als *Gesetze des Denkens* im Sinne von (begründeten) *Normen* für korrekte Schlussfolgerungen und Argumentationen
- **Zweite Perspektive** Logik als *Gesetze des Denkens* im Sinne von „wie geht Denken?“

Wir werden bei der Betrachtung der Logik in der mathematischen Argumentation noch ein weitere Sichtweise kennenlernen.

1.2 Mathematische Logik



Abbildung 1.1: Papyrusfragment der „Elemente“ des Euklid

Abbildung 1.1 zeigt ein Papyrusfragment der „Elemente“ des Euklid⁵, ein Werk, das nicht nur die Geometrie als Wissenschaft begründet hat, sondern auch die Denk- und Argumentationsweise der Mathematik, die *axiomatische Methode*.

Euklid hat seine Untersuchung der Geometrie begründet auf fünf Postulate (heute würde man dazu Axiome sagen), aus denen er dann mit einigen wenigen Schlussregeln alle seine Sätze über die Geometrie herleitet. Euklid hat außerdem Definitionen der Begriffe Punkt, Gerade usw. seinen Postulaten vorangestellt. Heute werden in der Mathematik solche Grundbegriffe nicht definiert, sondern sie ergeben sich aus ihren Eigenschaften, die durch die Axiome festgelegt sind.

Die Axiome von Euklid sind:

1. Zu jedem Paar von Punkten gibt es genau eine Gerade, die durch diese Punkte geht.
2. Eine beliebige Strecke kann man zu einer Geraden verlängern.
3. Zu jedem Mittelpunkt und Radius kann man einen Kreis ziehen.
4. Alle rechten Winkel sind einander gleich.
5. Zu jeder Geraden und einem Punkt außerhalb dieser Geraden gibt es genau *eine* parallele Gerade durch diesen Punkt.

⁵Euklid von Alexandria, griechischer Mathematiker, 3. Jahrhundert v. Chr.

Was wird man von den Axiomen erwarten?

- *Widerspruchsfreiheit:* Aus den Axiomen soll sich der Widerspruch nicht herleiten lassen, denn sonst würde ja *alles* aus den Axiomen folgen!
- *Unabhängigkeit:* Es soll nicht möglich sein, ein Axiom aus den anderen zu deduzieren. Diese Frage hat sich insbesondere bezüglich des fünften Axioms von Euklid gestellt, dem Parallelenaxiom. Der Versuch, es aus den anderen Axiomen herzuleiten, hat zu interessanten Entdeckungen geführt, die wir gleich kennenlernen werden.
- *Vollständigkeit:* Ein Axiomensystem ist vollständig, wenn man alle Sätze, die man auf Basis der Terme des Systems formulieren kann, entweder beweisen oder widerlegen kann. Euklids Axiomensystem ist übrigens nicht vollständig. David Hilbert⁶ hat 1899 in seiner Schrift „Grundlagen der Geometrie“ ein vollständiges Axiomensystem für die euklidische Geometrie entworfen.

Es ist nicht gelungen, das Parallelenaxiom aus den anderen Axiomen herzuleiten. Beim Versuch dies zu tun, wurde entdeckt, dass man durch Abwandlung des Parallelenaxioms neue interessante, sogenannte nicht-euklidische Geometrien definieren kann.

Euklidische Geometrie (der Ebene) Zu einer Geraden und einem Punkte außerhalb dieser Geraden, gibt es genau *eine* Parallelle. In dieser Geometrie ist die Summe der Winkel eines Dreiecks gerade 180° .

Elliptische Geometrie (auf der Kugeloberfläche) In dieser Geometrie sind die Geraden gerade die Großkreise. Und es gilt: Zu einer Geraden und einem Punkt außerhalb derselben, gibt es *keine* Parallelle, d.h. verschiedene Geraden schneiden sich immer. In dieser Geometrie ist die Summe der Winkel eines Eulerschen Dreiecks immer größer als 180° . Abbildung 1.3 zeigt die Geraden a, b und c und das Eulersche Dreieck, das sie bilden.

Hyperbolische Geometrie In dieser Geometrie gibt es zu einer Geraden und einem Punkt außerhalb dieser Geraden *mindestens zwei* parallele Geraden. Es sind dann tatsächlich unendlich viele. Die Summe der Winkel im Dreieck ist immer kleiner als 180° .

Ein Modell der hyperbolischen Geometrie ist die Geometrie auf der offenen oberen Halbebene der Ebene, das Poincaré'sche Halbebenenmodell.

⁶David Hilbert, deutscher Mathematiker, 1862 - 1943.

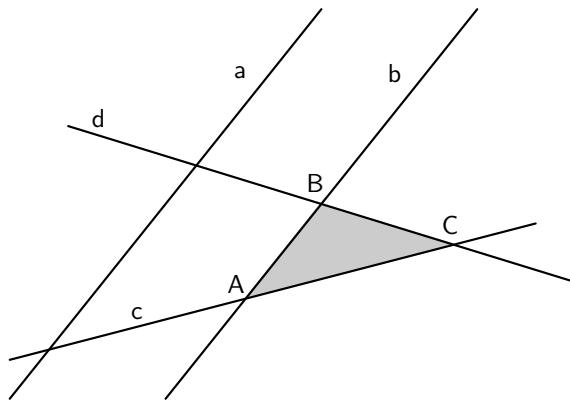


Abbildung 1.2: Euklidische Geometrie

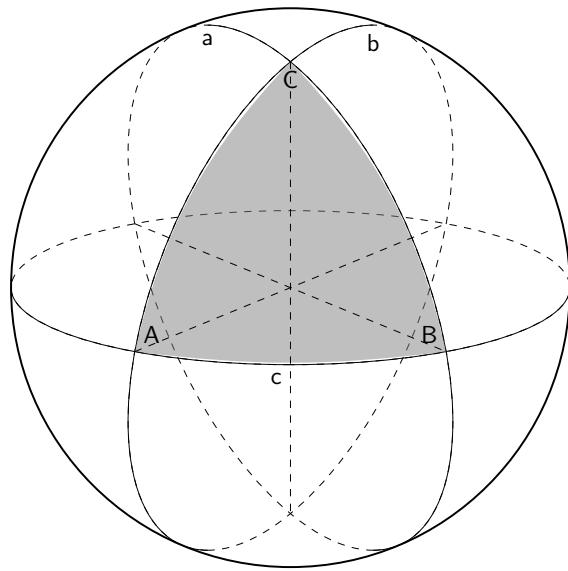


Abbildung 1.3: Elliptische Geometrie

Die Geraden sind Halbkreise oder senkrechte Halbgeraden in der offenen oberen Halbebene. In Abbildung 1.4 sind die Geraden a und b asymptotisch parallel, d.h. sie treffen sich auf der Grenze der Halbebene in einem sogenannten uneigentlichen Punkt, der nicht mehr zum Inneren der Halbebene gehört. a und c sind auch parallel, man sagt: ultraparallel, während b und c nicht parallel sind.

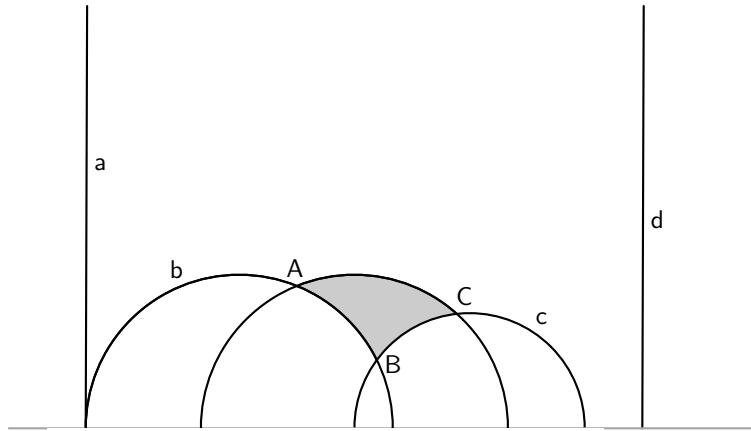


Abbildung 1.4: Hyperbolische Geometrie (Poincarés Halbebenen-Modell)

Theorie und Modell

In der mathematischen Logik hat sich (ausgehend von der Entdeckung nichteuklidischer Geometrien) eine Sprechweise von *Modell* durchgesetzt, die nicht verwechselt werden sollte mit dem Begriff des Modells, wie er zum Beispiel beim Softwaredesign mit der UML verwendet wird.

Im Beispiel der hyperbolischen Geometrie ist der Ausgangspunkt ein Axiomensystem und die Poincaré'sche Halbebene ist eine Struktur, in der sich die Terme des Axiomensystems interpretieren lassen und in der die Axiome zutreffen. Es gibt für das Axiomensystem der hyperbolischen Geometrie auch andere Modelle, zum Beispiel die Sattelfläche. Abbildung 1.5 zeigt ein Dreieck auf der Sattelfläche.

Ein *Modell* einer *Theorie* ist in der mathematischen Logik also eine mit passenden Strukturen versehene Menge, auf die die Axiome und alle daraus ableitbaren Sätze der Theorie zutreffen. Es ist wichtig, diese Denk- und Sprechweise zu kennen, denn in Teil II der Veranstaltung wird ein *Model Finder* vorkommen und in Teil III geht es u.a. um *Model Checking* — beide Bezeichnungen kommen von der Sprechweise der mathematischen Logik.

Nebenbei bemerkt: es mag einem durch diese Sprechweise die Mathematik erscheinen wie ein Spiel mit willkürlich festgelegten Regeln, den

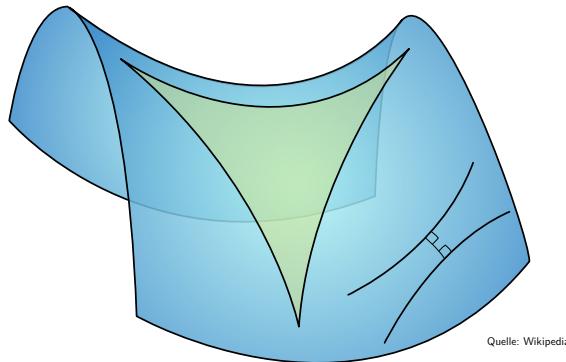


Abbildung 1.5: Hyperbolische Geometrie (Sattelfläche)

Axiomen, für die sich, sofern widerspruchsfrei, schon eine passende Wirklichkeit, ein Modell wird finden lassen. Ganz so ist es nicht. Die Axiome zu finden ist die Aufgabe, einen Sachverhalt auf seinen ganz grundlegenden Kern zu reduzieren. Natürlich kann man auch mit Variationen der Axiome „spielen“, oft führt das zu gar nichts, manchmal eröffnet es neue Einsichten.

Seine Axiomatisierung der Geometrie schien für David Hilbert das Vorbild dafür zu sein, wie Mathematik zu denken habe. Die Mathematik sollte auf die Grundlage eines Axiomensystems gestellt werden, aus der die gesamte Mathematik mit definierten und als korrekt angesehenen Schlussregeln in Beweisen mit endlich vielen Schritten hergeleitet könnte.

Wir erörtern noch kurz, welche berechtigten allgemeinen Forderungen an die Lösung eines mathematischen Problems zu stellen sind: ich meine vor Allem, die, daß es gelingt, die Richtigkeit der Antwort durch eine endliche Anzahl von Schlüssen darzuthun und zwar auf Grund einer endlichen Anzahl von Voraussetzungen, welche in der Problemstellung liegen und die jedesmal genau zu formuliren sind. Diese Forderung der logischen Deduktion mittelst einer endlichen Anzahl von Schlüssen ist nichts anderes als die Forderung der Strenge in der Beweisführung.

– David Hilbert: Vortrag auf dem internationalen Mathematiker-Kongress Paris 1900

Dritte Perspektive auf die Logik

Man kann die bisherigen Erläuterungen so zusammenfassen: Wir betrachten Logik als eine formale Sprache, deren *Syntax* präzise und eindeutig definiert ist. Die *Semantik* der Symbole der Sprache ergibt sich durch Modelle der Sprache. Und ein *Beweissystem* mit Axiomen und Schlussregeln erlaubt es durch rein symbolische Manipulation von Ausdrücken der Sprache neue Formeln herzuleiten.

- **Dritte Perspektive** Logik als *Kalkül* einer formalen Sprache — Syntax, Semantik und Beweissystem.

Und mit dieser dritten Perspektive sind wir nun schon gleich beim Thema Logik und Informatik. Denn mit formalen Sprachen und Umformungen von Ausdrücken kennt sich die Informatik ja bestens aus. Doch bevor wir auf falsche Ideen kommen, lassen wir noch Henri Poincaré⁷ zu Wort kommen:

Wer einer Schachpartie beiwohnt, dem wird es zum Verständnis der Partie nicht genügen, die Regeln über den Lauf der Figuren zu kennen. Was würde ihm nur erlauben zu erkennen, daß jeder Zug den Regeln entsprechend gespielt wurde, und dieser Vorzug hätte sehr wenig Wert. Es wäre jedoch das gleiche, wie es dem Leser eines mathematischen Buches ginge, wenn er nur Logiker wäre. Die Partie verstehen, das ist etwas ganz anderes, das heißt wissen, warum der Spieler mit dieser Figur zieht anstatt mit jener anderen, was er auch hätte tun können, ohne die Spielregeln zu übertreten; das heißt den inneren Grund zu erkennen, der aus dieser Reihe aufeinanderfolgender Züge ein organisches Ganzes macht. Mit viel mehr Grund ist diese Fähigkeit dem Spieler selbst nötig, das heißt dem Erfinder.

— Henri Poincaré: Der Wert der Wissenschaft, S.15

1.3 Logik und Informatik

[Symbolic] Logic and computer science share a symbiotic relationship. Computers provide a concrete setting for the implementation of logic. Logic provides language and methods for the study of theoretical computer science.

— Shawn Hedman: A First Course in Logic, S.xiv

⁷Henri Poincaré, französischer Mathematiker, 1854 - 1912.

Diese enge Beziehung zwischen formaler Logik und Informatik zeigt sich in vielen Teilgebieten der Informatik, etwa:

Digitaltechnik

Kombinatorische Schaltungen implementieren logische Operatoren. In der Digitaltechnik als einem Teilgebiet der technischen Informatik spielt also die Aussagenlogik eine ganz grundlegende Rolle.

Komplexitätstheorie

In diesem Teilgebiet der Informatik geht es um die Komplexität algorithmisch behandelbarer Probleme, insbesondere was die Rechenzeit in Abhängigkeit von der Größe der Eingabe angeht. In der Komplexitätstheorie ist die Frage immer noch ungeklärt, ob effizient verifizierbare Probleme (der Klasse \mathcal{NP}) sich auch in polynomieller Zeit lösen lassen, also auch in der Klasse \mathcal{P} sind. Ein oder vielleicht das Musterbeispiel für ein Problem der Klasse \mathcal{NP} ist das Problem der Erfüllbarkeit einer Formel der Aussagenlogik.

Datenbanken

Relationale Datenbanken kann man sehen als endliche Modelle von Sprachen der Prädikatenlogik. Datenbankabfragen kann man deshalb auch in Formeln der Prädikatenlogik übersetzen. Man kann vereinfacht sagen, dass heutiges SQL als Sprache die Ausdrucksmächtigkeit der Prädikatenlogik mit transitivem Abschluss hat. Die Solidität relationaler Datenbanksysteme ist sicherlich auch darauf zurückzuführen, dass mit der relationalen Algebra, die auf der Prädikatenlogik basiert, eine mathematische Grundlage existiert.

Logik und Softwaretechnik

In dieser Veranstaltung interessieren wir uns aber insbesondere für den Zusammenhang zwischen Logik und Softwaretechnik.

A specification is a written description of what a system is supposed to do. Specifying a system helps us understand it.
It's a good idea to understand a system before building it, so it's a good idea to write a specification of a system before implementing it.

...

Our basic tools for writing specifications is mathematics. Mathematics is nature's way of letting you know how sloppy your writing is. It's hard to be precise in an imprecise language like English or Chinese. In engineering, imprecision

can lead to errors. To avoid errors, science and engineering have adopted mathematics as their language.

– Leslie Lamport: Specifying Systems, S.1f.⁸

Da wir mit einem Computerprogramm letztlich eine formale Beschreibung des Verhaltens einer abstrakten Maschine formulieren, müssen wir in der Softwaretechnik im Grunde den Beweis erbringen, dass diese Beschreibung die Anforderungen an das Programm tatsächlich erfüllt.

A solution of the problem [of providing a software based machine to fulfill a purpose in the real world, i.e. software development] must be based on at least the following descriptions:

- **requirement \mathcal{R}** : a statement of the customer's requirement;
- **domain properties \mathcal{W}** : a description of the given properties of the problem world;
- **specification \mathcal{S}** : a specification of the machine's behaviour at its interface with the problem world; and
- **program \mathcal{P}** : a program describing the machine's internal and external behaviour in a language that the general-purpose computer can interpret.

To show that the problem is solved we must discharge a proof obligation whose form is, roughly:

$$(\mathcal{P} \Rightarrow \mathcal{S}) \wedge ((\mathcal{S} \wedge \mathcal{W}) \Rightarrow \mathcal{R})$$

– Michael Jackson: Where, Exactly, Is Software Development? LNCS 2757, 2003⁹

Man kann das auch etwas plakativ ausdrücken (im folgenden Zitat steht S für Spezifikation, E für Environment — bei Jackson die Domain Properties und R für Requirements):

... the more we realised the key role of the fundamental logic—that $S, E \vdash R$ is truly the $E = mc^2$ of requirements engineering.

– Anthony Hall: $E = mc^2$ Explained, 2010¹⁰

⁸Leslie Lamport, amerikanischer Informatiker. Mehr zum Thema Spezifikation in Leslie Lamperts Vortrag *Thinking for Programmers*, URL: <http://channel9.msdn.com/Events/Build/2014/3-642>.

⁹Michael Jackson (not the singer), britischer Informatiker, geb. 1936.

¹⁰Anthony Hall, britischer Informatiker.

1.4 Programm der Veranstaltung

Viele Fragestellungen in der Softwaretechnik und der Programmierung haben eine Darstellung in der *Aussagenlogik*, die wir im ersten Teil der Veranstaltung behandeln werden. Ein Beispiel ist die Beherrschung der Variabilität in Softwareproduktlinien, für die Featuremodelle eingesetzt werden, bei deren Analyse Techniken des *SAT-Solving*, der Lösung des Erfüllbarkeitsproblems der Aussagenlogik, zum Einsatz kommen.

Man kann sich auch fragen, ob Spezifikationen widerspruchsfrei sind, ob sie gewünschte Eigenschaften tatsächlich erfüllen, oder ob sich Gegenbeispiele finden lassen. Wir werden im zweiten Teil der Veranstaltung nach der Diskussion der *Prädikatenlogik* als Werkzeug die Sprache *Alloy* und den *Alloy Analyzer* kennenlernen, mit dem man leichtgewichtig und interaktiv „Mikromodelle“ von Architekturen, Entwürfen, Software überprüfen kann.

Man kann auch so vorgehen, dass man gebaute oder konzipierte Softwaresysteme logisch exakt beschreibt und dann prüft, ob sie gewünschte Eigenschaften tatsächlich erfüllen. Die Modelle sind häufig dynamische Modelle und die gewünschten Eigenschaften werden als Formeln der *linearen temporalen Logik (LTL)* formuliert. Mit der Technik des *Model Checking* kann man insbesondere in verteilten Systemen beweisbar überprüfen, ob Eigenschaften etwa der Fairness, des wechselseitigen Ausschlusses und Ähnliches erfüllt sind.

Wir werden also drei Logiken behandeln und jeweils einen Blick auf Anwendungen in der Softwareentwicklung werfen:

1. Aussagenlogik
2. Prädikatenlogik
3. Lineare temporale Logik

Dabei wird bei der Diskussion der Grundlagen die oben skizzierte dritte Perspektive auf die Logik verwendet, d.h.

- Die formale Sprache der jeweiligen Logik, also die *Syntax*
- Die *Semantik* der jeweiligen Logik, also Modelle in denen die Ausdrücke der formalen Sprache repräsentiert werden können
- Ein *Beweissystem*, das es erlaubt aus gegebenen Formeln durch Umformungen nach gewissen Regeln andere Formeln herzuleiten, und damit Beweise zu führen. Es gibt verschiedene solche Beweissysteme. Wir werden das Beweissystem des *natürlichen Schließens* in allen drei Logiken verwenden. Das natürliche Schließen kann

auch durch Software unterstützt werden, eine Software, die die jeweiligen Schritte bei der Anwendung der Regeln überprüft. Die *Logic Workbench lwb*¹¹ ist ein solches Werkzeug, das das natürliche Schließen in allen drei Logiken unterstützt.

¹¹Wiki zu Natural Deduction mit lwb auf github.

Teil I

Aussagenlogik

Kapitel 2

Aussagen und Formeln

In der Aussagenlogik werden Aussagen betrachtet, die wahr oder falsch sein können. Solche Aussagen nennt man *wahrheitsdefinite* Aussagen. Es gibt viele andere Formen von Aussagen in natürlichen Sprachen, wie z.B. ironische Äußerungen, Fragen oder Aufforderungen. Die Sprache der Aussagenlogik ist eine *formale* Sprache, in der nur wahrheitsdefinite Aussagen verwendet werden.

Beispiele

$P =$ „Göttingen ist nördlich von Frankfurt“

$Q =$ „6 ist eine Primzahl“

$R =$ „Jede gerade Zahl > 2 ist die Summe zweier Primzahlen“¹

aber nicht:

„Könnten Sie bitte die Türe schließen“, eine Aufforderung

„Wie spät ist es“, eine Frage

„Guten Tag“, ein Gruß

Der *Inhalt* der Aussagen ist für die Aussagenlogik nicht wirklich von Belang. Wir studieren *nicht* den Wahrheitsgehalt von Aussagen, sondern die *Beziehung* der Aussagen.

Wir können atomare Aussagen miteinander verbinden und daraus zusammengesetzte Aussagen, *Formeln* bilden. Die Verbindung wird durch *Junktoren* hergestellt - siehe Tabelle 2.1

¹ Goldbach-Vermutung, benannt nach dem Mathematiker Christian Goldbach (1690 - 1764).

Tabelle 2.1: Junktoren und weitere Symbole

\neg	„nicht“, not	Negation
\wedge	„und“, and	Konjunktion
\vee	„oder“, or	Disjunktion
\rightarrow	„impliziert“, implies	Implikation
\top	„wahr“, true , verum	Wahrheit
\perp	„falsch“, false , absurdum	Widerspruch

Beispiele

- $P \wedge Q$ „Göttingen ist nördlich von Frankfurt *und* 6 ist eine Primzahl“ ①
 $P \vee Q$ „Göttingen ist nördlich von Frankfurt *oder* 6 ist eine Primzahl“ ②
 $P \rightarrow Q$ „Göttingen ist nördlich von Frankfurt *impliziert* 6 ist eine Primzahl“ ③
 $Q \rightarrow P$ „6 ist eine Primzahl *impliziert* Göttingen ist nördlich von Frankfurt“ ④

Bemerkungen

- Die erste Aussage ist falsch. Allerdings muss man bei der Übertragung von Aussagen aus der natürlichen Sprache Vorsicht walten lassen:
 Die folgenden beiden Aussagen haben einen unterschiedlichen Sinn
 „Er ging zur Schule und ihm war langweilig.“
 „Ihm war langweilig und er ging zur Schule.“
 obwohl die Aussagen $P \wedge Q$ und $Q \wedge P$ in der formalen Sprache der Aussagenlogik äquivalent sind.
- Die zweite Aussage ist wahr.
- Die dritte Aussage ist falsch.
- Die vierte Aussage ist wahr – obwohl offensichtlicher Unsinn! Implikationen in der formalen Logik darf man nicht mit *Kausalität* verwechseln. Die Aussage ist wahr, weil in der formalen Logik das Prinzip *ex falso quodlibet* gilt: Aus einer falschen Voraussetzung darf man alles folgern. Warum diese Definition der Implikation sinnvoll ist, werden wir später sehen.
- Es gibt Aussagen, die wahr sind, egal welche Wahrheitswerte die beteiligten atomaren Aussagen haben, wie z.B.
 $((P \rightarrow Q) \rightarrow (\neg P \vee Q)) \wedge ((\neg P \vee Q) \rightarrow (P \rightarrow Q))$
 Solche Aussagen nennt man allgemeingültig oder *Tautologie*.

Wir haben in dieser einführenden Diskussion zwei Konzepte verwendet, ohne sie genau zu unterscheiden: Die *Syntax* der Aussagenlogik, die festlegt, welche Formeln wir aus atomaren Aussagen und Junktoren bilden können, sowie die *Semantik* der Aussagenlogik, bei der wir von Wahrheitswerten der atomaren Aussagen reden und aus diesen den Wahrheitswert von Formeln ermitteln können.

In den folgenden drei Kapiteln werden wir diese beiden Konzepte und ihren Zusammenhang auf systematische Weise untersuchen.

Kapitel 3

Die formale Sprache der Aussagenlogik

In der Sprache der Aussagenlogik kombiniert man atomare Aussagen mit Junktoren. Dabei gehen wir von einer gegebenen Menge \mathcal{P} von Aussagensymbolen sowie Junktoren aus. Genau genommen beziehen sich die folgenden Definitionen auf die Wahl dieser Menge und man sollte von *einer* Sprache der Aussagenlogik sprechen.

Definition 3.1 (Alphabet der Aussagenlogik). Das *Alphabet* der Sprache der Aussagenlogik besteht aus

- (i) einer Menge \mathcal{P} von Aussagensymbolen,
- (ii) den Junktoren: $\neg, \wedge, \vee, \rightarrow$
- (iii) der Konstanten: \perp
- (iv) den zusätzlichen Symbolen: $(,)$

Bemerkungen

- Für eine Sprache der Aussagenlogik nennt man die Wahl der Menge der Aussagensymbole sowie der Junktoren usw. auch die *logische Signatur*.
- Viele Autoren geben eine fixe (abzählbare) Menge von Aussagensymbolen vor, etwa $\mathcal{P} = \{P_0, P_1, P_2, \dots\}$ und sprechen dann von *der* Sprache der Aussagenlogik.¹

¹Die Menge der Aussagensymbole muss übrigens nicht unbedingt abzählbar sein, sondern kann eine beliebige Menge sein, siehe [Rau08, S. 4 und Abschnitt 1.5]. Das ist interessant, wenn man den Kompaktheitssatz der Aussagenlogik verwendet, um Aussagen über unendliche Mengen zu beweisen. In dieser Vorlesung werden wir uns damit nicht befassen, weil wir uns auf Anwendungen der formalen Logik in Informatik und Softwaretechnik konzentrieren.

Für Anwendungen der Aussagenlogik in der Informatik und der Softwareentwicklung haben wir jedoch mit endlichen Mengen zu tun und wir möchten den Aussagensymbole auch „sprechende“ Bezeichnungen geben können. Deshalb geht in unsere Definition des Alphabets die Wahl der Menge der Aussagensymbole ein.

- Die Bezeichnung von \neg, \wedge etc. als *Junktoren* soll unterstreichen, dass wir eine formale Sprache definieren. Natürlich aber darf man schon daran denken, dass mit \neg die *Negation* („nicht“), mit \wedge die *Konjunktion* („und“), mit \vee die *Disjunktion* („oder“) und mit \rightarrow die *Implikation* („impliziert“) gemeint sind.
- Die Konstante \perp steht für den *Widerspruch* („falsch“).
- Auch was die Junktoren angeht, treffen wir in der Definition eine Wahl. Wir könnten z.B. noch weitere Junktoren hinzunehmen, wie etwa \leftrightarrow oder auch Junktoren weglassen. Wir werden später sehen, dass sich aus der Semantik der Aussagenlogik ergibt, dass die Menge von Operatoren, die unsere gewählten Junktoren definieren *funktional vollständig* ist, d.h. jede beliebige Boolesche Funktion darstellen kann.

Definition 3.2 (Formeln der Aussagenlogik). Die *Formeln* der Aussagenlogik sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jedes Aussagensymbol ist eine Formel und auch \perp ist eine Formel.
- (ii) Ist φ eine Formel, dann auch $(\neg\varphi)$.
- (iii) Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$

Bemerkungen

- In der Definition der Formeln der Aussagenlogik verwenden wir auch die Implikation, wenn wir z.B. sagen: „Ist φ eine Formel, dann auch $(\neg\varphi)$ “. Man muss also unterscheiden, ob wir *über* die Logik sprechen — man sagt dann auch: wir verwenden die *Metasprache* — oder ob wir eine logische Formel angeben — dann verwenden wir die *Objektsprache*, die wir eben definiert haben.
- Die Symbole φ und ψ sind also *Variablen der Metasprache*, sie stehen als Platzhalter für *Formeln der Objektsprache*.
- Unsere Definition der Menge der Formeln der Aussagenlogik ist eine sogenannte *induktive* Definition.
- Eine Formel, die nur aus einem Aussagensymbol oder \perp besteht, nennt man auch *atomare* Formel oder *Primformel*.

Als *Grammatik* in Backus-Naur-Darstellung² können wir diese induktive Definition der Formeln der Aussagenlogik so ausdrücken:

$$\varphi ::= P \mid \perp \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi)$$

mit Aussagensymbolen $P \in \mathcal{P}$ und (bereits gebildeten) Formeln φ .

Zu einer Formel φ kann man ihren Syntaxbaum bilden:

Definition 3.3. Als *Syntaxbaum* bezeichnen wir einen endlichen Baum, dessen Knoten mit Formeln beschriftet sind und der folgende Eigenschaften hat:

- (i) Die Blätter sind mit Primformeln beschriftet.
- (ii) Ist ein Knoten mit einer Formel der Form $(\neg\varphi)$ beschriftet, dann hat er genau ein Kind, das mit φ beschriftet ist.
- (iii) Ist ein Knoten mit einer Formel der Form $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ oder $(\varphi \rightarrow \psi)$ beschriftet, dann hat er genau zwei Kinder und das linke ist mit φ , das rechte mit ψ beschriftet.

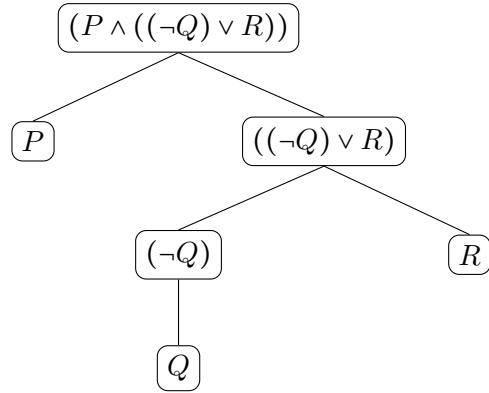
Ein Syntaxbaum repräsentiert die Formel, mit der die Wurzel des Baumes beschriftet ist.

Beispiel 3.1. Gegeben sei die Formel

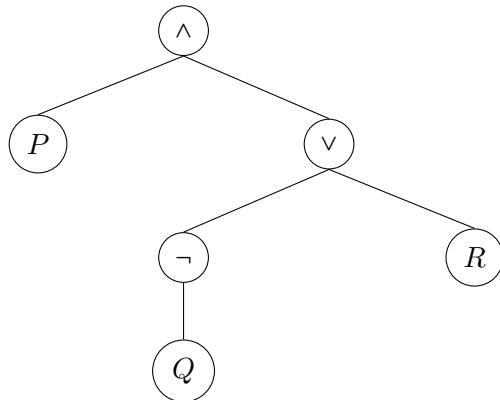
$$(P \wedge ((\neg Q) \vee R))$$

Der Syntaxbaum ist dann

²John W. Backus, amerikanischer Informatiker, 1942 - 2007; Peter Naur, dänischer Informatiker, 1928 - 2016 (Tatsächlich war Naur nicht erfreut über diese Bezeichnung, er hätte „Backus-Normalform“ vorgezogen.)



Wir verwenden oft die abgekürzte Darstellung des Syntaxbaums, in der die Knoten nicht mit den Subformeln bezeichnet werden, sondern mit dem Operator. Nur an den Blättern des Baums kommen dann die Primformeln vor. In unserem Beispiel also



Bemerkung Die Sprechweise „*der* Syntaxbaum einer Formel“ unterstellt, dass es nur *einen* solchen Baum zu einer gegebenen Formel gibt. Mit anderen Worten: die Grammatik aus der Definition der Formeln ist eindeutig.

Satz 3.1. *Jede Formel der Aussagenlogik hat genau einen Syntaxbaum.*

Zum Beweis von Aussagen über Formeln wendet man oft das Prinzip der *strukturellen Induktion* an.

Prinzip der strukturellen Induktion Sei \mathcal{E} eine Eigenschaft. Dann gilt $\mathcal{E}(\varphi)$ für alle Formeln φ , wenn gilt:

- (i) \mathcal{E} gilt für alle Primformeln und \perp .
- (ii) Gilt \mathcal{E} für φ , dann auch für $(\neg\varphi)$.
- (iii) Gilt \mathcal{E} für φ und ψ , dann auch für $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$.

Dieses Prinzip kann man anwenden, um obigen Satz zu beweisen.

Beweisidee für Satz 3.1. Man beweist den Satz in folgenden Schritten:
Zunächst zeigt man: Jede Formel hat eine gerade Anzahl von Klammern und zwar ebenso viele öffnende wie schließende Klammern.

Dann zeigt man: Jedes echte Anfangsstück einer Formel hat mehr öffnende als schließende Klammern.

Mit Hilfe dieser beiden Aussagen kann man dann den Satz beweisen. \square

Definition 3.4 (Subformel). Eine *Subformel* einer Formel φ ist ein Formel, die als Beschriftung an einem Knoten des Syntaxbaums von φ vorkommt.

Definition 3.5 (Rang einer Formel). Der *Rang* $rg(\varphi)$ einer Formel ist definiert durch

- (i) $rg(\varphi) = 0$, wenn φ eine Primformel.
- (ii) $rg((\varphi \square \psi)) = \max(rg(\varphi), rg(\psi)) + 1$, wobei \square für einen der Junktoren $\wedge, \vee, \rightarrow$ steht.
- (iii) $rg((\neg\varphi)) = rg(\varphi) + 1$.

Der Rang einer Formel ist gerade die Tiefe des Syntaxbaums, d.h. die maximale Pfadlänge von der Wurzel zu einem beliebigen Blatt.

Die Eindeutigkeit des Syntaxbaums erlaubt es uns, Konventionen zu vereinbaren, mit denen man Klammern „sparen“ kann. Man vereinbart, dass die Junktoren in folgender Reihenfolge binden, die stärkste Bindung zuerst: $\neg, \wedge, \vee, \rightarrow$. Außerdem sind \wedge und \vee linksassoziativ; \rightarrow ist rechtsassoziativ.

Bemerkung Für die Definition der Syntax der Aussagenlogik haben wir die *Infix-Notation* verwendet, wie das in den Lehrbüchern über formale Logik üblich ist.

Man kann stattdessen die *Präfix-Notation* verwenden und hat dann folgende Grammatik:

$$\varphi ::= P \mid \perp \mid (\neg\varphi) \mid (\wedge \varphi \dots) \mid (\vee \varphi \dots) \mid (\rightarrow \varphi \varphi)$$

In der Präfix-Notation kann man die Junktoren \wedge und \vee als *n-äre* Junktoren definieren.

In der **Logic Workbench (lwb)**, einer in Clojure geschriebenen Bibliothek von Funktionen für die Aussagen-, Prädikaten- und lineare temporale Logik wird Präfix-Notation mit folgenden Junktoren verwendet:

Tabelle 3.1: Junktoren für die Aussagenlogik in lwb

Junktor	Beschreibung	Arität
<code>not</code>	Negation	unär
<code>and</code>	Konjunktion	n-är
<code>or</code>	Disjunktion	n-är
<code>impl</code>	Implikation	binär
<code>equiv</code>	Biimplikation	binär
<code>xor</code>	Exklusives Oder	binär
<code>ite</code>	If-then-else	ternär

Beispiel 3.2. Die Formel

$$(P \wedge ((\neg Q) \vee R))$$

in Infix-Notation wird in der Präfix-Notation der Logic Workbench so geschrieben:

$$(\text{and } P \ (\text{or } (\text{not } Q) \ R))$$

Diese Schreibweise entspricht genau dem Syntaxbaum der Formel.

Kapitel 4

Die Semantik der Aussagenlogik

In der klassischen Aussagenlogik wird jedem Aussagensymbol ein Wahrheitswert aus der Menge $\{T, F\}$ zugeordnet. (T steht für `true` und F für `false`, viele Autoren verwenden auch die Menge $\{1, 0\}$).

Die Bedeutung einer Formel der Aussagenlogik ergibt sich dann daraus, dass man diese Zuordnung von den Aussagensymbolen auf die Formel erweitert, in dem man definiert, welcher Wahrheitswert sich beim Zusammensetzen von Formeln durch die Junktoren ergibt. Auf diese Weise definiert man den Booleschen *Operator* zum jeweiligen Junktor.

Bemerkung Die Festlegung auf genau zwei Wahrheitswerte definiert eine *zweiwertige* Logik. Man kann auch andere Festlegungen treffen wie:

- Łukasiewic¹-Logik mit den Wahrheitswerten $\{1, \frac{1}{2}, 0\}$ oder $\{T, U, F\}$. Łukasiewic² versteht den Wert $\frac{1}{2}$ als „nicht bewiesen, aber auch nicht widerlegt“, manchmal wird der dritte Wert U auch als „unbekannt“ interpretiert, wie zum Beispiel in SQL.
- Zadeh²-Logik mit Wahrheitswerten im Intervall $[0, 1]$; eine Logik mit dieser Semantik wird auch Fuzzy-Logik genannt.

Sei $\mathbb{B} = \{T, F\}$. Die Junktoren $\neg, \wedge, \vee, \rightarrow$ können als *Operatoren* auf der Menge \mathbb{B} , also als Boolesche Operatoren aufgefasst werden, in dem man die Verknüpfungstafeln wie folgt definiert:

¹Jan Łukasiewic², polnischer Philosoph, Mathematiker und Logiker 1878 - 1956.

²Lotfi A. Zadeh, amerikanischer Informatiker, 1921 - 2017

\neg	φ	$\neg\varphi$	
	T	F	
	F	T	

$\neg\varphi$ genau dann **true**, wenn φ **false**.

\wedge	φ	ψ	$\varphi \wedge \psi$	
	T	T	T	
	T	F	F	
	F	T	F	
	F	F	F	

$\varphi \wedge \psi$ ist genau dann **true**, wenn sowohl φ als auch ψ **true** sind.

\vee	φ	ψ	$\varphi \vee \psi$	
	T	T	T	
	T	F	T	
	F	T	T	
	F	F	F	

$\varphi \vee \psi$ ist genau dann **true**, wenn einer der Operanden **true** ist. \vee ist also ein einschließendes oder.

\rightarrow	φ	ψ	$\varphi \rightarrow \psi$	
	T	T	T	
	T	F	F	
	F	T	T	
	F	F	T	

$\varphi \rightarrow \psi$ ist genau dann **true**, wenn φ **false** oder ψ **true** ist.

Bemerkung

Der Operator \rightarrow wird auch als *materiale Implikation*³ bezeichnet. Er behauptet *keinen* kausalen Zusammenhang zwischen der linken Seite, dem *Antezedens* und der rechten Seite, der *Konsequenz* oder dem *Sukzedens*.

Es seien die Aussagen

$$\begin{aligned} P &= \text{„Die Erde umkreist die Sonne“}, \\ P' &= \text{„Die Sonne umkreist die Erde“ und} \\ Q &= \text{„6 ist Primzahl“} \end{aligned}$$

gegeben.

$P \rightarrow Q$ hat den Wahrheitswert F – wie man vielleicht erwarten würde, auch wenn kein inhaltlicher Zusammenhang zwischen den beiden Aussagen besteht.

Hingegen hat $P' \rightarrow Q$ den Wahrheitswert T – denn das Antezedens ist F. Das kann man vielleicht so interpretieren: „Wenn die Sonne um die Erde kreisen würde, dann wäre 6 eine Primzahl“ – und da ja die Sonne nicht um die Erde kreist, können wir über die 6 sagen, was wir wollen.

³nach Bertrand Russell und Alfred North Whitehead: *Principia Mathematica* (1910).

Dahinter steckt ein klassisches logisches Prinzip: *ex falso quodlibet*⁴ – „aus Falschem folgt alles, was beliebt“.

Der für uns wichtigere Grund für die Verwendung der materialen Implikation ist jedoch, dass man mit dieser Definition der Implikation eine *einfache* Logik bekommt – in der zum Beispiel folgender Sachverhalt einfach ausdrückbar ist:

Die Aussage $\forall x \in \mathbb{N} : (x > 3) \rightarrow (x > 1)$ ist T.

Setze ein:

$x = 4$	$(4 > 3) \rightarrow (4 > 1)$	Ergebnis
	T T	T
$x = 2$	$(2 > 3) \rightarrow (2 > 1)$	Ergebnis
	F T	T
$x = 0$	$(0 > 3) \rightarrow (0 > 1)$	Ergebnis
	F F	T

In allen drei Fällen ist die Aussage true.

Die materielle Implikation führt zu sogenannten Paradoxien, z.B.

- (1) $P \rightarrow (Q \rightarrow P)$ ist allgemeingültig
„Wenn P gilt, folgt P aus allem“.
- (2) $\neg P \rightarrow (P \rightarrow Q)$ ist allgemeingültig
„Wenn P nicht gilt, dann folgt alles aus P “.

4.1 Modell, Belegung

Sei \mathcal{P} die Menge der Aussagensymbole und \mathbb{B} die Menge der Wahrheitswerte.

Definition 4.1. Eine Abbildung $v : \mathcal{P} \rightarrow \mathbb{B}$ nennt man ein *Modell* für die Sprache der Aussagenlogik. Man nennt Modelle der Aussagenlogik auch spezifischer *Belegung*, weil durch v jedem Aussagensymbol ein Wahrheitswert zugewiesen wird.

Definition 4.2. Zu einem Modell $v : \mathcal{P} \rightarrow \mathbb{B}$ und einer Formel φ definiert man den Wahrheitswert $\llbracket \varphi \rrbracket_v \in \mathbb{B}$ der Formel induktiv durch

⁴eigentlich „ex falso sequitur quodlibet“.

-
- (i) $\llbracket P \rrbracket_v := v(P)$ für alle $P \in \mathcal{P}$
- (ii) $\llbracket \perp \rrbracket_v := \text{F}$
- (iii) $\llbracket (\neg \varphi) \rrbracket_v := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_v = \text{F} \\ \text{F sonst} \end{cases}$
- (iv) $\llbracket (\varphi \wedge \psi) \rrbracket_v := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_v = \text{T und } \llbracket \psi \rrbracket_v = \text{T} \\ \text{F sonst} \end{cases}$
- (v) $\llbracket (\varphi \vee \psi) \rrbracket_v := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_v = \text{T oder } \llbracket \psi \rrbracket_v = \text{T} \\ \text{F sonst} \end{cases}$
- (vi) $\llbracket (\varphi \rightarrow \psi) \rrbracket_v := \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_v = \text{F oder } \llbracket \psi \rrbracket_v = \text{T} \\ \text{F sonst} \end{cases}$

Bemerkung Sei φ eine Formel und v_1, v_2 seien Modelle mit $v_1(P) = v_2(P)$ für alle Aussagensymbole P in φ . Dann gilt:

$$\llbracket \varphi \rrbracket_{v_1} = \llbracket \varphi \rrbracket_{v_2}.$$

Man kann sich zu einer gegebenen Belegung v fragen, welchen Wahrheitswert eine Formel φ hat. Diese Frage ist einfach zu beantworten, indem man die Formel auswertet. Spannender ist die „umgekehrte“ Fragestellung: Gegeben ein Formel φ , gibt es dann eine Belegung, in der die Formel T ist, und wenn ja, welche? Diese ist die Frage nach der *Erfüllbarkeit* der Formel.

Definition 4.3 (Erfüllbarkeit). Eine Formel φ heißt *erfüllbar*, wenn es ein Modell v gibt mit $\llbracket \varphi \rrbracket_v = \text{T}$.

Definition 4.4 (Falsifizierbarkeit). Eine Formel φ heißt *falsifizierbar*, wenn es ein Modell v gibt mit $\llbracket \varphi \rrbracket_v = \text{F}$.

Definition 4.5 (Allgemeingültigkeit). Eine Formel φ heißt *allgemeingültig*, wenn für alle Modelle v gilt: $\llbracket \varphi \rrbracket_v = \text{T}$.

Man schreibt dann $\models \varphi$ und nennt φ eine *Tautologie*.

Definition 4.6 (Unerfüllbarkeit). Eine Formel φ heißt *unerfüllbar*, wenn für alle Modelle v gilt: $\llbracket \varphi \rrbracket_v = \text{F}$.

Man schreibt dann $\not\models \varphi$ und nennt φ eine *Kontradiktion*.

Etwas salopp kann man diese Definitionen so auffassen:

- Eine Formel ist erfüllbar, wenn es eine „Welt“ gibt, in der sie wahr ist.
- Eine Formel ist falsifizierbar, wenn es eine „Welt“ gibt, in der sie nicht wahr ist.
- Eine Formel ist allgemeingültig, wenn sie in jeder möglichen „Welt“ wahr ist.
- Eine Formel ist unerfüllbar, wenn sie in keiner aller möglichen „Welt“ wahr ist.

Satz 4.1 (Dualitätsprinzip). *Eine Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.*

Beweis. Sei φ allgemeingültig, d.h. für alle Belegungen v gilt $\llbracket \varphi \rrbracket_v = T$, d.h. aber auch dass für alle v gilt $\llbracket \neg\varphi \rrbracket_v = F$, d.h. $\neg\varphi$ ist unerfüllbar. Und da $\varphi \equiv \neg\neg\varphi$ gilt auch die Umkehrung. \square

				erfüllbar
φ	ψ	$\neg\psi$	$\neg\varphi$	
allgemeingültig	erfüllbar, nicht allgemeingültig		unerfüllbar	
				falsifizierbar

Abbildung 4.1: Dualitätsprinzip

4.2 Wahrheitstafel

Um zu prüfen, ob eine Formel allgemeingültig oder erfüllbar ist, kann man die Wahrheitstafel verwenden. Im Prinzip listet man in der Wahrheitstafel alle möglichen Belegungen der Aussagensymbole in der Formel auf und berechnet in jeder dieser „Welten“ den Wahrheitswert der Formel. Man geht dabei so vor:

1. Man ermittelt die Menge der Aussagensymbole der Formel. Für jede mögliche Belegung eines Symbols mit einem Wahrheitswert bildet man eine Zeile der Wahrheitstafel. Hat die Formel n verschiedene Aussagensymbole, hat die Wahrheitstafel also 2^n Zeilen. Jedes Symbol bekommt eine Spalte der Wahrheitstabelle.

2. Man bildet eine weitere Spalte der Wahrheitstafel, die mit der Formel beschriftet ist. In dieser Spalte überträgt man nun für jedes Symbol die Belegung der jeweiligen Zeile. Dann geht man entsprechend des zugehörigen Syntaxbaums von unten nach oben und ermittelt sukzessive die Wahrheitswerte der Subformeln, bis man beim Wahrheitswert der Formel angelangt ist.

Satz 4.2. *Die Methode des Aufstellens der Wahrheitstafel ist ein Entscheidungsverfahren für das Erfüllbarkeitsproblem und für das Gültigkeitsproblem.*

Beweis. Eine Formel φ ist *erfüllbar*, wenn es in der Wahrheitstafel *mindestens eine* Zeile mit dem Ergebnis T gibt.

Eine Formel φ ist *allgemeingültig*, wenn in der Wahrheitstafel *alle* Zeilen das Ergebnis T haben. \square

4.3 Semantische Äquivalenz und Substitution

Definition 4.7 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind *semantisch äquivalent*, geschrieben $\varphi \equiv \psi$, wenn für alle Belegungen v gilt: $\llbracket \varphi \rrbracket_v = \llbracket \psi \rrbracket_v$.

Definition 4.8 (Logische Konsequenz). Sei Γ eine Menge von Formeln. Eine Formel φ heißt *logische Konsequenz* von Γ , geschrieben $\Gamma \vDash \varphi$, wenn für alle Modelle v gilt:

Ist $\llbracket \gamma \rrbracket_v = T$ für alle $\gamma \in \Gamma$, dann ist auch $\llbracket \varphi \rrbracket_v = T$.

Definition 4.9 (Substitution). Sei φ eine Subformel von ψ und φ' eine beliebige Formel. Dann ist $\psi[\varphi'/\varphi]$ (lese: ψ mit φ' an Stelle von φ) die Formel, die man erhält, wenn man jedes Vorkommen von φ in ψ durch φ' ersetzt (substituiert).

Satz 4.3 (Substitutionssatz). *Sei φ eine Subformel von ψ und φ' eine semantisch äquivalente Formel, d.h. $\varphi \equiv \varphi'$, dann gilt:*

$$\psi \equiv \psi[\varphi'/\varphi].$$

\square

Wichtige semantische Äquivalenzen

Assoziativität

$$\begin{aligned} (\varphi \vee \psi) \vee \chi &\equiv \varphi \vee (\psi \vee \chi) \\ (\varphi \wedge \psi) \wedge \chi &\equiv \varphi \wedge (\psi \wedge \chi) \end{aligned}$$

Kommutativität

$$\begin{aligned} \varphi \vee \psi &\equiv \psi \vee \varphi \\ \varphi \wedge \psi &\equiv \psi \wedge \varphi \end{aligned}$$

Distributivität

$$\begin{aligned}\varphi \vee (\psi \wedge \chi) &\equiv (\varphi \vee \psi) \wedge (\varphi \vee \chi) \\ \varphi \wedge (\psi \vee \chi) &\equiv (\varphi \wedge \psi) \vee (\varphi \wedge \chi)\end{aligned}$$

De Morgans Gesetze

$$\begin{aligned}\neg(\varphi \vee \psi) &\equiv \neg\psi \wedge \neg\varphi \\ \neg(\varphi \wedge \psi) &\equiv \neg\psi \vee \neg\varphi\end{aligned}$$

Idempotenz

$$\begin{aligned}\varphi \vee \varphi &\equiv \varphi \\ \varphi \wedge \varphi &\equiv \varphi\end{aligned}$$

Doppelte Negation

$$\neg\neg\varphi \equiv \varphi$$

Komplement

$$\begin{aligned}\varphi \wedge \neg\varphi &\equiv \perp \\ \varphi \vee \neg\varphi &\equiv \top\end{aligned}$$

Identitätsgesetze

$$\begin{aligned}\varphi \wedge \top &\equiv \varphi \\ \varphi \vee \perp &\equiv \varphi\end{aligned}$$

Absorption

$$\begin{aligned}\varphi \wedge (\varphi \vee \psi) &\equiv \varphi \\ \varphi \vee (\varphi \wedge \psi) &\equiv \varphi\end{aligned}$$

Bemerkung Betrachtet man die Menge der Formeln der Aussagenlogik und bildet die Menge der Äquivalenzklassen bezüglich der semantischen Äquivalenz (\equiv), dann sagen obige Aussagen unter anderem, dass diese Menge eine *Boolesche Algebra* ist.

4.4 Boolesche Operatoren und funktionale Vollständigkeit

Die Anzahl der unären Operatoren auf \mathbb{B} ist 4, die der binären Operatoren ist 16. Allgemein gilt:

Satz 4.4. Für $n \in \mathbb{N}$ gibt es 2^{2^n} n -äre Boolesche Operatoren.

Beweis. Hat der Operator n Argumente, dann gibt es 2^n n -Tupel von möglichen verschiedenen Werten für die Argumente. Jede dieser Kombinationen kann als Ergebnis des Operators einen der beiden Wahrheitswerte \top oder \perp haben, also gibt es 2^{2^n} Möglichkeiten. \square

Übung:

Erstellen Sie Tabellen für alle möglichen unären und binären Operatoren, finden Sie in der Literatur die gängigen Bezeichnungen und Symbole für die Operatoren.

Offenbar kann man Operatoren durch andere Operatoren ausdrücken, z.B.

$$\varphi \leftrightarrow \psi \equiv (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$

oder

$$\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$$

oder

$$\varphi \wedge \psi \equiv \neg(\neg\varphi \vee \neg\psi)$$

Definition 4.10. Eine Menge Boolescher Operatoren heißt *funktional vollständig*, wenn man jeden beliebigen Operator durch diese Operatoren logisch äquivalent ausdrücken kann.

Satz 4.5. Für jeden n -ären Booleschen Operator gibt es eine äquivalente Formel, die nur die Operatoren \neg und \vee hat. D.h. $\{\neg, \vee\}$ ist funktional vollständig.

Übung:

Beweisen Sie diesen Satz. Hinweis: Sehen Sie in das Skript von R. Stärk.

Kapitel 5

Das Beweissystem des natürlichen Schließens

Mit der Wahrheitstafel gibt es eine einfache Möglichkeit, festzustellen, ob eine Formel der Aussagenlogik erfüllbar ist. Man stellt die Wahrheitstafel auf und prüft, ob es eine Zeile mit dem Ergebnis T gibt.

Es gibt jedoch Beispiele, an denen man leicht sieht, dass diese Methode ihre Probleme hat. Nehmen wir etwa als Beispiel die folgende Formel:

$$(P_1 \wedge (P_1 \rightarrow P_2) \wedge \cdots \wedge (P_{n-1} \rightarrow P_n)) \rightarrow P_n$$

Diese Formel hat n Atome, die Wahrheitstafel also 2^n Zeilen.

Wir können jedoch auf eine andere Weise zu einer Lösung kommen. Wenn wir akzeptieren, dass die Schlussregel (man nennt sie *Modus ponens*)

$$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi}$$

erlaubt ist, dann kann man so argumentieren:

P_1 und $P_1 \rightarrow P_2$ sind gegeben, also gilt P_2 .

P_2 und $P_2 \rightarrow P_3$ sind gegeben, also gilt P_3 .

...

P_{n-1} und $P_{n-1} \rightarrow P_n$ sind gegeben, also gilt P_n .

Und aus dieser Argumentation folgt, dass es sich bei der gegebenen Formel um eine Tautologie handelt.

In diesem Beispiel haben wir eine Schlussregel angewandt. Wichtig ist dabei zu sehen, dass sich damit die Perspektive gewandelt hat. In der Ermittlung der Wahrheitstafel wird die *Semantik* der Aussagenlogik verwendet: für jedes mögliche Modell, jede mögliche Belegung wird der Wahrheitswert der Formel ermittelt. In der Argumentation mit der obigen Schlussregel wird kein Bezug auf die Semantik genommen, sondern die Schlussregel wird verwendet, um Transformation an den Symbolen vorzunehmen, die die Formel ausmachen.

Sei $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ eine Menge von Formeln und φ eine Formel, dann gibt es zwei Sichten auf die Frage, ob φ aus Γ folgt:

- *Semantische Sicht*

$$\Gamma \vDash \varphi$$

In allen Modellen (allen „möglichen Welten“), in denen $\gamma_1, \gamma_2, \dots, \gamma_n$ gelten, ist auch φ wahr.

- *Syntaktische Sicht*

$$\Gamma \vdash \varphi$$

Man kann aus den gegebenen Formeln $\gamma_1, \gamma_2, \dots, \gamma_n$ die Formel φ herleiten, indem man ausschließlich die Regeln des Beweissystems verwendet.

Es gibt für die Aussagenlogik (und für die Prädikatenlogik) verschiedene solcher Beweissysteme. Wir werden uns in der Veranstaltung mit dem *natürlichen Schließen* (auch *natürliche Deduktion*) befassen.

Das Kalkül des natürlichen Schließen wurde 1934 von Gerhard Gentzen¹ und unabhängig von ihm von Stanisław Jaśkowski² entwickelt.

Die Bezeichnung „Natürliches Schließen“ röhrt daher, dass die Regeln des Kalküls das „natürliche“ Argumentieren von Mathematikern formalisieren.

¹Gerhard Gentzen (1909–1945), deutscher Mathematiker und Logiker, [Gen35].

²Stanisław Jaśkowski (1906–1965), polnischer Logiker.

„Mein erster Gesichtspunkt war folgender: Die Formalisierung des logischen Schließens, wie sie insbesondere durch Frege, Russell und Hilbert entwickelt worden ist, entfernt sich ziemlich weit von der Art des Schließens, wie sie in Wirklichkeit bei mathematischen Beweisen geübt wird. [...] Ich wollte nun zunächst einmal einen Formalismus aufstellen, der dem wirklichen Schließen möglichst nahekommt. So ergab sich ein ‚Kalkül des natürlichen Schließens‘“ [Gen35, S. 176].

5.1 Schlussregeln

Für die Herleitung von Formeln gibt es im natürlichen Schließen pro logischem Symbol zwei Regeln:

- eine, die das Symbol einführt (*Introduction*, abgekürzt durch i) und
- eine zweite, die das Symbol entfernt (*Elimination* auch: Auflösung), abgekürzt durch e).

Jede Regel gibt an, was *gegeben* sein muss (oberhalb des Strichs), damit die Umformung gemacht werden darf, also was sich aus dem Gegebenen *ergibt* (unterhalb des Strichs).

5.1.1 Konjunktion

Die Regeln für die Konjunktion sind:

	<i>Einführung</i>	<i>Elimination</i>
\wedge	$\frac{\varphi \quad \psi}{\varphi \wedge \psi} \text{ } \wedge i$	$\frac{\varphi \wedge \psi}{\varphi} \text{ } \wedge e_1 \quad \frac{\varphi \wedge \psi}{\psi} \text{ } \wedge e_2$

Die Konjunktion kann man einführen, wenn man Herleitungen für die beiden Formeln der Konjunktion bereits hat.

Für die Elimination der Konjunktion gibt es zwei Subregeln: Eine Herleitung der Gesamtformel der Konjunktion kann man sowohl als Herleitung der linken Teilformel als auch der rechten Teilformel nehmen.

5.1.2 Disjunktion

Die Regeln für die Disjunktion sind:

	<i>Einführung</i>	<i>Elimination</i>
\vee	$\frac{\varphi}{\varphi \vee \psi} \quad \text{vi}_1$ $\frac{\psi}{\varphi \vee \psi} \quad \text{vi}_2$	$\frac{\begin{array}{ c c } \hline \varphi & \psi \\ \vdots & \vdots \\ \chi & \chi \\ \hline \end{array}}{\chi} \quad \text{ve}$

Wenn man eine Herleitung für φ hat, hat man auch eine Herleitung für $\varphi \vee \psi$, ebenso darf man die Herleitung von ψ als Beweis für $\varphi \vee \psi$ nehmen.

Will man die Disjunktion entfernen und dabei χ herleiten, muss man für jede Teilformel der Disjunktion eine Herleitung von χ finden. Diese Regel entspricht also der Beweistechnik der Fallunterscheidung.

5.1.3 Implikation

Die Regeln für die Implikation sind:

	<i>Einführung</i>	<i>Elimination</i>
\rightarrow	$\frac{\begin{array}{ c } \hline \varphi \\ \vdots \\ \psi \\ \hline \end{array}}{\varphi \rightarrow \psi} \quad \rightarrow i$	$\frac{\varphi \quad \varphi \rightarrow \psi}{\psi} \quad \rightarrow e, MP$

Die Implikation leitet man her, indem man die Hypothese als gegeben annimmt und dann daraus die Folgerung herleitet. In der Regel wird in der Box oberhalb des Strichs angegeben, dass φ nur *innerhalb* der Box als gegeben angenommen werden darf. Die senkrechten Punkte : markieren die Beweisverpflichtung, nämlich dass sie durch einen Beweis ersetzt werden müssen, der ψ aus φ herleitet.

Die Implikation kann man entfernen, wenn man die Hypothese φ bewiesen hat und ebenso, dass $\varphi \rightarrow \psi$ gilt. Dann hat man ψ bewiesen. Diese Schlussfigur ist schon seit der Antike geläufig und wird als *Modus ponens* bezeichnet, deshalb auch die Abkürzung MP.

5.1.4 Negation

	<i>Einführung</i>	<i>Elimination</i>
\neg	$\begin{array}{c} \varphi \\ \vdots \\ \perp \end{array}$	$\frac{\varphi \quad \neg\varphi}{\psi} \neg e$

Will man beweisen, dass $\neg\varphi$ gilt — also die Negation einführen —, nimmt man an, dass φ bewiesen ist und führt diesen Beweis dann fort, bis man den Widerspruch \perp hergeleitet hat. Daraus ergibt sich, dass $\neg\varphi$ bewiesen ist.

5.1.5 Widerspruchsbeweis, EFQ

	<i>Einführung</i>	<i>Elimination</i>
RAA, \perp	$\begin{array}{c} \neg\varphi \\ \vdots \\ \perp \end{array}$	$\frac{\perp}{\varphi} \perp e, EFQ$

Man kann eine Formel φ in der klassischen Logik auch durch einen Widerspruchsbeweis herleiten³. Man nimmt das Gegenteil von φ an und führt diese Annahme zum Widerspruch. Die Regel sagt dann, dass nach diesem Widerspruchsbeweis φ bewiesen ist.

Dass aus dem Widerspruch jede beliebige Aussage folgt, wird auch als *Ex falso quodlibet* oder genauer *Ex falso sequitur quodlibet* bezeichnet.

5.2 Beispiele für das natürliche Schließen

5.2.1 Gentzens Beispiel

Als erstes Beispiel nehmen wir ein Beispiel aus der Arbeit von Gerhard Gentzen, mit der er das natürliche Schließen motiviert [Gen35].

Bewiesen werden soll die Formel

$$(X \vee (Y \wedge Z)) \rightarrow ((X \vee Y) \wedge (X \vee Z))$$

³In der *intuitionistischen* Logik ist diese Schlussregel nicht erlaubt.

Im folgenden Beweis geben die Angaben rechts die jeweils verwendete Regel an und die Zeile (oder Zeilen), auf die sie angewandt wurden.

1.	$(X \vee (Y \wedge Z))$	angenommen
2.	X	angenommen
3.	$(X \vee Y)$	$\vee i_1$ 2
4.	$(X \vee Z)$	$\vee i_1$ 2
5.	$((X \vee Y) \wedge (X \vee Z))$	$\wedge i$ 3, 4
6.	$(Y \wedge Z)$	angenommen
7.	Y	$\wedge e_1$ 6
8.	$(X \vee Y)$	$\vee i_2$ 7
9.	Z	$\wedge e_2$ 6
10.	$(X \vee Z)$	$\vee i_2$ 9
11.	$((X \vee Y) \wedge (X \vee Z))$	$\wedge i$ 8, 10
12.	$((X \vee Y) \wedge (X \vee Z))$	$\vee e$ 1, 2-5, 6- 11
13.	$(X \vee (Y \wedge Z)) \rightarrow ((X \vee Y) \wedge (X \vee Z))$	$\rightarrow i$ 1-12

5.2.2 Abgeleitete Regeln der Aussagenlogik

Hat man eine Herleitung $P \vdash Q$, dann kann man sie in anderen Herleitungen wie eine Regel verwenden. Zwar führen wir den Beweis im natürlichen Schließen mit bestimmten Symbolen aus, da jedoch der Beweis selbst ganz unabhängig von der Wahl der speziellen Symbole wie etwa P ist, gilt er für *jedes* solche Symbol und kann deshalb selbst wie eine Regel verwendet werden.

Auf dem Merkblatt für die Regeln des natürlichen Schließens <https://esb-dev.github.io/mat/rules.pdf> sind vier solcher abgeleiteter Regeln aufgeführt, die im Folgenden bewiesen werden.

$$\frac{\varphi}{\neg\neg\varphi} \quad \neg\neg i$$

$$\frac{\neg\neg\varphi}{\varphi} \quad \neg\neg e$$

$$\frac{\varphi \rightarrow \psi \quad \neg\psi}{\neg\varphi} \quad MT$$

$$\frac{}{\varphi \vee \neg\varphi} \quad TND$$

Beweis für die Einführung der doppelten Negation:

1. P gegeben
2. $\neg P$ angenommen
3. \perp $\neg e$ 2, 1
4. $\neg\neg P$ $\neg i$ 2-3

Beweis für die Elimination der doppelten Negation:

1. $\neg\neg P$ gegeben
2. $\neg P$ angenommen
3. \perp $\neg e$ 1, 2
4. P RAA 2-3

Beweis für Modus Tollens

1. $P \rightarrow Q$ gegeben
2. $\neg Q$ gegeben
3. P angenommen
4. Q $\rightarrow e$ 1, 3
5. \perp $\neg e$ 2, 4
6. $\neg P$ $\neg i$ 3-5

Beweis für *Tertium Non Datur*

1. $\neg(P \vee \neg P)$ angenommen
2. P angenommen
3. $P \vee \neg P$ $\vee i_1$ 2
4. \perp $\neg e$ 1, 3
5. $\neg P$ $\neg i$ 2-4
6. $P \vee \neg P$ $\vee i_2$ 5
7. \perp $\neg e$ 1, 6
8. $P \vee \neg P$ RAA 1-7

Die Beweise für diese abgeleiteten Regel haben wir mit *konkreten* Formeln, in diesem Fall Primformeln wie P und Q , durchgeführt. Sie als Regel zu verwenden bedeutet jedoch, dass sie für *jede* beliebige Formel gelten, weshalb ja auch in der Formulierung der Regel die Symbole der Metasprache φ und ψ verwendet wurden. Da aber in den Beweisen selbst keinerlei spezifische Eigenschaften von P oder Q verwendet wurden, können wir die Beweise für beliebige Formeln abstrahieren — und somit wie Regeln verwenden.

Dies ist in der Logic WorkBench generell der Fall: ein im natürlichen Schließen geführter Beweis kann gespeichert werden (mit der Funktion `export` und dann in anderen Beweisen verwendet werden (siehe [Wiki zum Natürlichen Schließen in lwb](#)).

5.3 Beweisstrategien

In diesem Abschnitt wird an einem Beispiel erläutert, welche Strategien man beim Entwickeln einer Herleitung im natürlichen Schließen verwenden kann.

Als Beispiel nehmen wir:

$$P \rightarrow (Q \vee R) \vdash Q \vee (\neg P \vee R)$$

Der erste Schritt besteht immer darin, dass man die Voraussetzungen als gegeben hinschreibt und dann eine Lücke lässt, der das zu zeigende Ziel folgt. Die Lücke stellt die Beweisverpflichtung dar.

In unserem Beispiel führt dieser erste Schritt zu:

1. $P \rightarrow (Q \vee R)$ gegeben
2. :
3. $Q \vee (\neg P \vee R)$

Im weiteren Vorgehen untersucht man, welche Regeln man anwenden kann. Dazu gibt es zwei Möglichkeiten:

- Bei den Zeilen oberhalb der Beweisverpflichtung gibt es eine Formel, für deren Haupt-Junktor eine Regel zur *Elimination* anwendbar ist. Dann kann man diese Regel *vorwärts* verwenden und erhält eine neue Zeile oberhalb der Beweisverpflichtung — oder man schließt den Beweis ab.
- Bei den Zeilen unterhalb der Beweisverpflichtung gibt es eine Formel, für deren Haupt-Junktor eine Regel zur *Einführung* angewandt werden kann. Dann kann man diese Regel *rückwärts* anwenden und erhält eine neue Zeile (oder einen Block).

In unserem Beispiel ist der Haupt-Junktor der Voraussetzung die Implikation. Um sie auflösen zu können, haben wir als Regel den Modus Ponens, der allerdings nur angewandt werden kann, wenn nicht nur $P \rightarrow (Q \vee R)$ gegeben ist, sondern auch P .

Betrachten wir also das Ziel. Dort ist der Haupt-Junktor die Disjunktion. Diese können wir einführen, wenn eine der Seiten gegeben ist. Das ist aber auch nicht der Fall.

In dieser Situation muss man zu anderen Waffen greifen. Wir können immer die Regel TND vorwärts und die Regel RAA rückwärts anwenden.

In unserem Beispiel scheint TND eine gute Wahl zu sein.

Dann haben wir folgende Situation:

1. $P \rightarrow (Q \vee R)$ gegeben
2. $P \vee \neg P$ TND
3. :
4. $Q \vee (\neg P \vee R)$

In den folgenden Schritten hilft das Anwenden der Regeln für die Elimination von Junktoren bzw. der Einführung von Junktoren.

Wir verwenden die Regel zur Elimination der Disjunktion:

1. $P \rightarrow (Q \vee R)$ gegeben
2. $P \vee \neg P$ TND
3.

P	angenommen
:	
$Q \vee (\neg P \vee R)$	
4.

$\neg P$	angenommen
:	
$Q \vee (\neg P \vee R)$	
5. $Q \vee (\neg P \vee R)$ $\vee e$

Es ist jetzt nicht mehr schwierig, die beiden Beweisverpflichtungen zu erfüllen:

1.	$P \rightarrow (Q \vee R)$	gegeben
2.	$P \vee \neg P$	TND
3.	P	angenommen
4.	$Q \vee R$	MP 1, 3
5.	Q	angenommen
6.	$Q \vee (\neg P \vee R)$	$\vee i_1 5$
7.	R	angenommen
8.	$\neg P \vee R$	$\vee i_2 7$
9.	$Q \vee (\neg P \vee R)$	$\vee i_2 8$
10.	$Q \vee (\neg P \vee R)$	$\vee e 4, 5-6, 7-9$
11.	$\neg P$	angenommen
12.	$\neg P \vee R$	$\vee i_1 11$
13.	$Q \vee (\neg P \vee R)$	$\vee i_2 12$
14.	$Q \vee (\neg P \vee R)$	$\vee e 2, 3-10, 11-13$

Man mag versucht sein, die genannten Strategie *schematisch* anzuwenden und einfach auszuprobieren, welche Regel anwendbar ist. Dies kann in einfachen Fällen zu einer Herleitung führen, trägt jedoch wenig zum *Verständnis* des bewiesenen Sachverhalts bei. Deshalb zum Schluß die

Goldene Regel

Man muss sich die Aussage klarmachen, die man beweisen möchte und einen Beweis formulieren, ohne direkt an die Regeln zu denken, sondern eher wie die *Idee* des Beweises aussehen kann. Erst dann setzt man diese Idee in die Anwendung der Regeln um.

Werkzeuge wie die Logic Workbench eignen sich dann dafür zu überprüfen, ob die Umsetzung der Idee in einzelne Schritte korrekt durchgeführt wurde. Man vermeidet damit Flüchtigkeitsfehler.

5.4 Eigenschaften der Herleitbarkeit \vdash

In diesem Abschnitt betrachten wir Eigenschaften, die die Herleitbarkeit \vdash mittels des natürlichen Schließens hat.

Im Folgenden seien stets Γ, Γ' , Δ Mengen von Formeln der Aussagenlogik sowie φ und ψ einzelne Formeln der Aussagenlogik.

Lemma 5.1 (Monotonie).

$$\Gamma \vdash \varphi \Rightarrow \Gamma \cup \Delta \vdash \varphi$$

Beweis. Monotonie ist eine offensichtliche Eigenschaft von Herleitungen: Zu einer gegebenen Herleitung von φ aus Γ kann man weitere Voraussetzungen nach Belieben hinzufügen, sie werden ja für die Herleitung gar nicht notwendigerweise benötigt. \square

Lemma 5.2.

- (a) $\varphi \in \Gamma \Rightarrow \Gamma \vdash \varphi$
- (b) $\Gamma \vdash \varphi$ und $\Gamma' \vdash \psi \Rightarrow \Gamma \cup \Gamma' \vdash \varphi \wedge \psi$
- (c) $\Gamma \vdash \varphi \wedge \psi \Rightarrow \Gamma \vdash \varphi$ und $\Gamma \vdash \psi$
- (d) $\Gamma \cup \{\varphi\} \vdash \psi \Rightarrow \Gamma \vdash \varphi \rightarrow \psi$
- (e) $\Gamma \vdash \varphi$ und $\Gamma' \vdash \varphi \rightarrow \psi \Rightarrow \Gamma \cup \Gamma' \vdash \psi$
- (f) $\Gamma \vdash \perp \Rightarrow \Gamma \vdash \varphi$
- (g) $\Gamma \cup \{\neg\varphi\} \vdash \perp \Rightarrow \Gamma \vdash \varphi$

Beweis.

- (a) Wenn $\varphi \in \Gamma$ gilt, besteht die Herleitung von φ trivialerweise aus der Wiederholung von φ .
- (b) Wegen Monotonie gilt $\Gamma \vdash \varphi \Rightarrow \Gamma \cup \Gamma' \vdash \varphi$ sowie $\Gamma' \vdash \psi \Rightarrow \Gamma \cup \Gamma' \vdash \psi$, also wegen der Regel $\wedge i$ auch $\Gamma \cup \Gamma' \vdash \varphi \wedge \psi$.
- (c) Wenn man aus der Herleitung von $\varphi \wedge \psi$ mit der Regel $\wedge e_1$ bzw. der Regel $\wedge e_2$ die Konjunktion eliminiert, erhält man $\Gamma \vdash \varphi$ bzw. $\Gamma \vdash \psi$.
- (d) Die Aussage gilt wegen der Regel $\rightarrow i$.
- (e) Wegen Monotonie gilt unter den gegebenen Voraussetzung $\Gamma \cup \Gamma' \vdash \varphi$ sowie $\Gamma \cup \Gamma' \vdash \varphi \rightarrow \psi$, also $\Gamma \cup \Gamma' \vdash \psi$ wegen der Regel $\rightarrow e$, dem Modus Ponens.
- (f) Die Aussage gilt wegen der Regel EFQ.
- (g) Eine Herleitung für $\Gamma \vdash \varphi$ ergibt sich durch die Regel RAA, nach deren Anwendung man die gegebene Herleitung einsetzt und den Widerspruch erhält.

\square

Satz 5.1 (Endlichkeitssatz). *Gilt $\Gamma \vdash \varphi$, dann gibt es eine endliche Teilmenge $\Gamma' \subseteq \Gamma$, für die $\Gamma' \vdash \varphi$ gilt.*

Beweis. Wenn $\varphi \in \Gamma$ ist, dann reicht $\Gamma' = \{\varphi\}$ für die Herleitung aus. Für die Induktion über die Länge von Herleitungen setzen wir voraus, dass die Aussage nun für alle Herleitungen der Länge $n - 1$ gelte.

Für eine Herleitung der Länge n , betrachten wir die letzte angewandte Regel. Sie braucht nur endlich viele vorausgesetzte Aussagen, also folgt die Behauptung durch natürliche Induktion. \square

Definition 5.1. (Konsistenz) Eine Menge Γ von Formeln der Aussagenlogik ist *konsistent*, wenn $\Gamma \not\vdash \perp$, *inkonsistent* andernfalls.

Lemma 5.3. Folgende Aussagen sind äquivalent:

- (i) Γ ist konsistent
- (ii) Es gibt kein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$
- (iii) Es gibt mindestens eine Formel φ mit $\Gamma \vdash \neg\varphi$

Das Lemma lässt sich auch so formulieren:

Folgende Aussagen sind äquivalent:

- (i') Γ ist inkonsistent
- (ii') Es gibt ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$
- (iii') Für alle φ gilt $\Gamma \vdash \varphi$

Beweis.

- (i') \Rightarrow (iii') Sei φ eine beliebige Formel, da Γ inkonsistent ist, gilt $\Gamma \vdash \varphi$ nach Lemma 5.2 (f).
- (iii') \Rightarrow (ii') Da alle Formeln herleitbar sind, gibt es ein φ mit $\Gamma \vdash \varphi$ sowie $\Gamma \vdash \neg\varphi$.
- (ii') \Rightarrow (i') Es gibt nach Voraussetzung ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$, d.h. $\Gamma \vdash \varphi \wedge \neg\varphi$, also wegen der Regel $\neg e$ auch $\Gamma \vdash \perp$.

\square

Lemma 5.4.

- (i) $\Gamma \vdash \varphi \Leftrightarrow \Gamma \cup \{\neg\varphi\} \vdash \perp$
- (ii) $\Gamma \vdash \neg\varphi \Leftrightarrow \Gamma \cup \{\varphi\} \vdash \perp$

Beweis.

- (i) \Rightarrow) Wenn $\Gamma \vdash \varphi$ gilt und zusätzlich die Voraussetzung $\neg\varphi$, kann man mit der Regel $\neg e$ den Widerspruch herleiten.
 \Leftarrow) siehe Lemma 5.2 (g).

- (ii) \Rightarrow) Wenn $\Gamma \vdash \neg\varphi$ gilt und zusätzlich die Voraussetzung φ , kann man mit der Regel $\neg e$ den Widerspruch herleiten.
 \Leftarrow) Eine Herleitung für $\Gamma \vdash \neg\varphi$ ergibt sich durch die Regel $\neg i$, nach deren Anwendung man die gegebene Herleitung einsetzt und den Widerspruch erhält.

□

Definition 5.2. (Maximal konsistente Formelmenge) Eine Formelmenge Γ ist *maximal konsistent*, wenn sie konsistent ist, aber jede echte Obermenge $\Gamma' \supset \Gamma$ inkonsistent ist.

Maximal konsistente Mengen sind im Prinzip folgende Mengen von Aussagen: Gegeben sei eine Belegung v und $\Gamma = \{\varphi \mid \llbracket \varphi \rrbracket_v = T\}$, dann ist diese Menge konsistent und sogar maximal konsistent.

Lemma 5.5. Eine maximal konsistente Menge Γ ist abgeschlossen bezüglich Herleitbarkeit, d.h.

$$\Gamma \vdash \varphi \Rightarrow \varphi \in \Gamma \text{ für alle } \varphi.$$

Beweis. Es gelte für eine maximal konsistente Menge Γ und eine Formel $\varphi: \Gamma \vdash \varphi$. Angenommen $\varphi \notin \Gamma$, dann ist $\Gamma \cup \{\varphi\}$ inkonsistent, weil Γ maximal konsistent ist. Also gilt nach Lemma 5.4 $\Gamma \vdash \neg\varphi$. D.h. es gibt ein φ mit $\Gamma \vdash \varphi$ und $\Gamma \vdash \neg\varphi$, d.h. Γ ist inkonsistent. □

Lemma 5.6. Eine maximal konsistente Menge Γ ist negationstreu, d.h. für eine beliebige Formel φ gilt

$$\text{entweder } \Gamma \vdash \varphi \text{ oder } \Gamma \vdash \neg\varphi.$$

Beweis. Gilt $\Gamma \vdash \varphi$, dann kann wegen der Konsistenz von Γ nicht $\Gamma \vdash \neg\varphi$ gelten.

Gilt $\Gamma \not\vdash \varphi$, dann ist $\Gamma \cup \{\neg\varphi\}$ konsistent (Lemma 5.2 (g)). Da Γ maximal konsistent ist, folgt dann $\neg\varphi \in \Gamma$, und somit $\Gamma \vdash \neg\varphi$. □

5.5 Vollständigkeit des natürlichen Schließens

Mit dem Beweissystem des natürlichen Schließens haben wir eine weiteren Zugang zur Aussagenlogik neben der semantischen Sicht, wie bereits oben erwähnt:

Sei $\Gamma = \{\gamma_1, \gamma_2, \dots, \gamma_n\}$ eine Menge von Formeln und φ eine Formel, dann gibt es zwei Sichten auf die Frage, ob φ aus Γ folgt:

- In der *semantische Sicht* betrachten wir die Frage der *logischen Konsequenz*: $\Gamma \vDash \varphi$.
- In der *syntaktischen Perspektive* stellt sich die Frage, ob man für eine Formel eine *Herleitung* findet: $\Gamma \vdash \varphi$.

Nun stellt sich natürlich die Frage, ob die beiden Zugänge gleichwertig sind. Im Grunde handelt es sich im zwei Fragen:

- Ist das Beweissystem *korrekt*? Anders gesagt: wir können aus wahren Voraussetzungen nur wahre Schlüsse ziehen.

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

- Ist das Beweissystem *vollständig*? Ist es möglich mit den Regeln auch jede in der Semantik zutreffende Schlussfolgerung herzuleiten, also die Umkehrung:

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

Die erste Frage, die nach der *Korrektheit* des natürlichen Schließens, ist die einfachere Angelegenheit, schließlich haben wir die Regeln ja so ausgewählt, dass sie Wahrheit erhalten. Die zweite Frage, die nach der *Vollständigkeit* des Beweissystems, ist schwieriger einzusehen, weil ja nicht auf Anhieb erkennbar ist, wie wir aus der semantischen Sicht auf die Existenz einer Herleitung schließen können, oder sie sogar konstruieren können.

Es sei noch bemerkt, dass es neben dem natürlichen Schließen noch andere Beweissysteme für die Aussagenlogik gibt, zum Beispiel das Hilbert-Kalkül, Gentzens Sequenzenkalkül oder die Resolution. Für alle diese Beweissysteme kann man die Vollständigkeit zeigen. Vollständigkeit ist also eine Eigenschaft der Logik selbst: es gibt die Möglichkeit ein Kalkül zu finden, mit denen durch rein symbolische Manipulation von Formeln jede semantisch wahre Aussage gezeigt werden kann.⁴

Zum Begriff *Vollständigkeit* Wir hatten in der Einleitung 1.2 schon den Begriff der Vollständigkeit und auch in 4.4 war von „funktionaler Vollständigkeit“ die Rede. Man muss diese drei Bedeutungen auseinanderhalten⁵:

⁴“However, completeness should be understood not as a statement about these specific rules, but as a statement about the logic itself. Completeness asserts the existence of a list of rules that allows us to deduce every consequence from any set of formulas of the logic.” [Hed04, S.44]

⁵Der Wikipedia-Artikel über **Vollständigkeit in der Logik** beschreibt die drei Bedeutungen des Begriffs recht gut.

- Eine Menge von Booleschen Operatoren heißt *funktional vollständig*, wenn man jede Boolesche Funktion $f : \mathbb{B}^n \rightarrow \mathbb{B}, n \geq 1$ durch diese Operatoren darstellen kann, siehe 4.4.
- Eine durch ein Axiomensystem definierte Theorie heißt *vollständig*, wenn jede geschlossene Formel oder ihre Negation in der Theorie enthalten ist, siehe 1.2.
- Ein Beweissystem, Kalkül für eine Logik heißt *vollständig*, wenn jedes richtige Theorem auch mit den Regeln des Kalküls hergeleitet werden kann. Dieser Begriff von Vollständigkeit ist das Thema in diesem Abschnitt. Man sieht leicht, dass damit nicht Vollständigkeit im Sinne einer vollständigen Theorie gemeint ist, denn die Formel P kann in der Aussagenlogik offensichtlich weder als wahr noch als falsch bewiesen werden.

5.5.1 Korrektheit des natürlichen Schließens für die Aussagenlogik

Satz 5.2 (Korrektheit des natürlichen Schließens).

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

(Informeller) Beweis:

Zunächst muss man präzise sagen, was in Herleitungen erlaubt ist.

Dazu muss man unterscheiden zwischen Regeln, die man einfach dadurch anwenden kann, indem man syntaktisch die Ausdrücke über dem Strich der Regel durch den Ausdruck unter dem Strich ersetzt. Ein Beispiel ist die Einführung von \wedge aus zwei gegebenen Formeln.

Andere Regeln sind so aufgebaut, dass sie selbst wieder eine Beweisverpflichtung enthalten, wie etwa die Einführung von $\varphi \rightarrow \psi$. Für diesen Schritt wird angenommen, dass φ bereits hergeleitet wurde und man zeigt unter dieser Annahme, dass man dann ψ herleiten kann. In diesem Fall darf die Annahme φ nicht außerhalb des Bereichs der Beweisverpflichtung verwendet werden.

Um es mit Richard Bornat auszudrücken [Bor05, Definition 5.2, Seite 56] auszudrücken:

In a box-and-line proof

1. every line must be justified either as a premise or by use of a rule appealing to *previous* lines or boxes;
2. if an appealed-to line is inside a box, then that box must also enclose the justified line.

Wenn eine Herleitung diese Bedingungen erfüllt, dann wird sie wahre Aussagen in wahre Aussagen überführen, wenn jede der Regeln des natürlichen Schließens das tut. Man muss also für jede Regel überprüfen, ob sie Wahrheit erhält.

Zwei Beispiele für die Argumentation, die Korrektheit der restlichen Regeln kann man ähnlich überprüfen:

Die Regel $\frac{\varphi \quad \psi}{\varphi \wedge \psi}$ erhält Wahrheit, denn sind die Voraussetzungen wahr, dann auch die Folgerung. Das ist gerade die semantische Definition von \wedge .

$$\boxed{\begin{array}{c} \varphi \\ \vdots \\ \psi \end{array}}$$

Die Regel $\frac{\varphi \rightarrow \psi}{\varphi \rightarrow \psi} \rightarrow i$ Wenn wir annehmen, dass $\llbracket \varphi \rrbracket = T$ und es gelingt unter Anwendung unserer wahrheitserhaltender Regeln zu zeigen, dass dann auch ψ wahr ist, dann bedeutet das, dass $\llbracket \varphi \rightarrow \psi \rrbracket = T$ ist, wie man an der Wahrheitstafel von \rightarrow sieht. \square

5.5.2 Vollständigkeit des natürlichen Schließens für die Aussagenlogik

Es geht darum, dass jede wahre Aussage auch hergeleitet werden kann. Wir haben also alle Regen, die dafür erforderlich sind. Ordentlicher ausgedrückt:

Satz 5.3 (Vollständigkeit des natürlichen Schließens).

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

Man kann diesen Satz durch einen Widerspruchsbeweis zeigen. In Büchern über mathematische Logik wird gerne diese Art des Beweises verwandt.

Eine andere Möglichkeit des Beweises besteht darin, dass man aus der gegebenen wahren Aussage eine Herleitung des natürlichen Schließens *konstruiert*. Man zeigt also, dass man ein Programm schreiben kann, das die Herleitung ausgibt und wie es dies tun müsste. Dies ist eine Art des Beweises, die einem Informatiker naheligt, so wird der Beweis etwa in [HR04] geführt.

Wir wollen beide Beweise kennenlernen, in diesem Abschnitt zunächst den indirekten Beweis.

Wir benötigen zwei wichtige Sätze, um den Widerspruchsbeweis führen zu können:

Satz 5.4 (Satz von Lindenbaum). *Jede konsistente Menge Γ kann zu einer maximal konsistenten Menge $\Gamma^* \supseteq \Gamma$ erweitert werden.*

Beweis. Wir setzen für den Beweis voraus, dass unsere Sprache abzählbar viele Formeln enthält⁶: $\varphi_0, \varphi_1, \varphi_2, \dots$. Nun kann man eine aufsteigende Folge von Formelmengen definieren, deren Vereinigung maximal konsistent ist.

Die Konstruktion geht so:

$$\begin{aligned}\Gamma_0 &= \Gamma \\ \Gamma_{n+1} &= \begin{cases} \Gamma_n \cup \{\varphi_n\} & \text{falls } \Gamma_n \cup \{\varphi_n\} \text{ konsistent} \\ \Gamma_n & \text{sonst} \end{cases} \\ \Gamma^* &= \bigcup \{\Gamma_n \mid n \geq 0\}\end{aligned}$$

Es ist nun zu zeigen, dass Γ^* konsistent und maximal konsistent ist:

Jede der Formelmengen Γ_n ist konsistent, so wurden sie ja konstruiert.

Nehmen wir nun an, ihre Vereinigung Γ^* wäre nicht konsistent. Dann könnte man aus Γ^* den Widerspruch herleiten. Für diese Herleitung genügen endlich viele der Aussagen aus Γ^* , d.h. aber, dass es ein n gibt, so dass alle die für die Herleitung benötigten Aussagen in Γ_n sind, also $\Gamma_n \vdash \perp$, aber Γ_n ist konsistent. Die Annahme ist also widerlegt, Γ^* ist konsistent.

Es bleibt zu zeigen, dass Γ^* maximal konsistent ist. Nehmen wir eine konsistente Menge von Aussagen $\Delta \supseteq \Gamma^*$. Ist $\varphi \in \Delta$, dann ist φ eine der Formeln φ_i , und demzufolge in Γ_{i+1} enthalten, also in Γ^* , d.h. $\Delta = \Gamma^*$. \square

Satz 5.5 (Modellexistenzsatz). *Eine konsistente Menge Γ ist erfüllbar.*

Beweis. Wir können wegen dem Satz von Lindenbaum annehmen, dass Γ eine maximale konsistente Menge ist. Wir definieren eine Belegung, indem wir auf den Primformeln festlegen

$$v(P_i) = \begin{cases} T & \text{für } P_i \in \Gamma \\ F & \text{sonst} \end{cases}$$

⁶Man kann den Beweis auch für überabzählbare Mengen führen, siehe [Rau08, Lemma 4.3]

Nun muss man also zeigen, dass für eine beliebige Formel φ gilt:

$$v(\varphi) = \text{T} \Leftrightarrow \varphi \in \Gamma$$

Dazu kann man strukturelle Induktion über den Aufbau der Formeln machen. Da wir wissen, dass $\{\wedge, \neg\}$ funktional vollständig ist, genügt es die Induktion für diese beiden Junktoren zu machen.

1. Für Primformeln gilt die Behauptung per Definition der Belegung.
2. Für $\varphi = \psi \wedge \chi$:
 $v(\varphi) = \text{T} \Leftrightarrow v(\psi) = v(\chi) = \text{T}$ also durch die Induktionsvoraussetzung: $\psi, \chi \in \Gamma$, also auch $\varphi \in \Gamma$ wegen der Maximalität von Γ .
Umgekehrt: Maximale konsistente Mengen sind abgeschlossen bezüglich Herleitbarkeit (Lemma 5.5), d.h. $\psi \wedge \chi \in \Gamma \Leftrightarrow \psi, \chi \in \Gamma$, also $v(\varphi) = \text{T}$.
3. Für $\varphi = \neg\psi$:
 $v(\varphi) = \text{T} \Leftrightarrow v(\psi) = \text{F}$ also durch die Induktionsvoraussetzung: $\psi \notin \Gamma$ und $\varphi \in \Gamma$.
Umgekehrt: $\varphi \in \Gamma$ ergibt $\psi \notin \Gamma$, also $v(\psi) = \text{F}$, also $v(\varphi) = \text{T}$.

Aus der strukturellen Induktion folgt die Aussage, wir haben ein Modell konstruiert. \square

Nun können wir den Vollständigkeitssatz beweisen:

Beweis des Vollständigkeitssatzes. Zu zeigen ist, dass $\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$ gilt. Wir nehmen das Gegenteil an, dass also zwar $\Gamma \vDash \varphi$, aber $\Gamma \not\vdash \varphi$ der Fall ist.

Wenn $\Gamma \not\vdash \varphi$, dann ist $\Gamma \cup \{\neg\varphi\}$ konsistent. Nach dem Satz von Lindenbaum gibt es eine maximale konsistente Erweiterung Γ^* für $\Gamma \cup \{\neg\varphi\}$. Nach dem Modellexistenzsatz gibt es ein Modell für Γ^* , das alle diese Formeln wahr macht. Also ist dann auch $\neg\varphi$ wahr, das bedeutet aber, dass $\Gamma \not\vDash \varphi$ gelten würde, was der Voraussetzung widerspricht. \square

5.5.3 Ein konstruktiver Beweis für den Vollständigkeitssatz in der Aussagenlogik

Für den konstruktiven Beweis setzen wir voraus, dass die Menge Γ endlich ist und es gilt $\Gamma \vDash \varphi$. Es ist dann zu zeigen, dass $\Gamma \vdash \varphi$ gilt, indem wir angeben, wie man eine Herleitung erstellen kann.

Zunächst reduzieren wir die Fragestellung auf den Fall einer allgemeingültigen Formel, d.h. auf

$$\vDash \varphi \Rightarrow \vdash \varphi$$

Mit Lemma 5.7 wird aus dem allgemeinen Fall $\Gamma \vDash \varphi$ eine allgemeingültige Formel. Wenn wir für diese zeigen, dass es eine Herleitung gibt, können wir mit Lemma 5.8 eine Herleitung $\Gamma \vdash \varphi$ finden.

Lemma 5.7. Für Aussagen $\gamma_1, \gamma_2, \dots, \gamma_n$ und φ gilt:

$$\gamma_1, \gamma_2, \dots, \gamma_n \vDash \varphi \Rightarrow \vDash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi)$$

Beweis. Wir betrachten den Syntaxbaum der Formel $\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$ in Abb. 5.1.

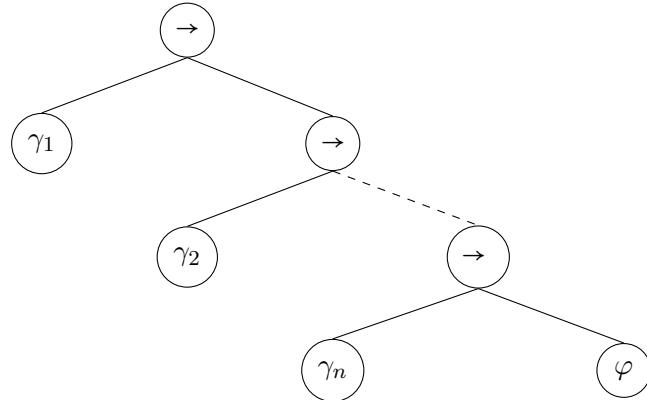


Abbildung 5.1: Syntaxbaum für Lemma 5.7

Angenommen $\not\vDash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi)$, d.h. es gibt eine Belegung bei der die Aussage F ergibt. Dann muss die Wurzel des Baums F sein. Das ist nur möglich, wenn $\gamma_1 T$ ist und die zweite Implikation F — und so weiter. Also ergibt sich, dass alle der γ s T sind, aber φF . Das steht aber im Widerspruch zur Voraussetzung $\gamma_1, \gamma_2, \dots, \gamma_n \vDash \varphi$. \square

Lemma 5.8. Für Aussagen $\gamma_1, \gamma_2, \dots, \gamma_n$ und φ gilt:

$$\vdash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi) \Rightarrow \gamma_1, \gamma_2, \dots, \gamma_n \vdash \varphi$$

Beweis. Nach Voraussetzung gibt es eine Herleitung für $\vdash (\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi)$. Wir müssen einen Beweis konstruieren für $\gamma_1, \gamma_2, \dots, \gamma_n \vdash \varphi$.

Wir gehen so vor:

1.	γ_1	gegeben
2.	γ_2	gegeben
:	:	:
n.	γ_n	gegeben
:	:	Beweis aus der Voraussetzung
m.	$\gamma_1 \rightarrow \gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	
m+1.	$\gamma_2 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	$\rightarrow e 1, m$
m+2.	$\gamma_3 \rightarrow \dots \rightarrow \gamma_n \rightarrow \varphi$	$\rightarrow e 2, m+1$
:	:	:
m+n.	φ	$\rightarrow e n, m+n-1$

□

Mit diesen beiden Lemmas haben wir Aufgabe darauf reduziert, zu zeigen, dass

$$\vDash \varphi \Rightarrow \vdash \varphi$$

gilt. Die Idee des Beweise besteht darin, aus der Wahrheitstafel der Formel φ die Herleitung zu konstruieren.

Lemma 5.9. φ sei eine Formel und $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n$ die Literale der Wahrheitstafel für φ . Dann gilt:

1. Wenn der Wert der Zeile T ist, dann gibt es eine Herleitung $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \varphi$
2. Wenn der Wert der Zeile F ist, dann gibt es eine Herleitung $\hat{P}_1, \hat{P}_2, \dots, \hat{P}_n \vdash \neg\varphi$

Beweis. Der Beweis geht durch strukturelle Induktion über den Formelaufbau:

Wenn die Formel atomar ist, etwa P , dann muss man zeigen, dass $P \vdash P$ und $\neg P \vdash \neg P$. Diese Beweise sind einfach die Übernahme der gegebenen Aussage in die Schlussfolgerung.

Wenn die Formel nicht atomar ist, dann hat sie eine (bei \neg als Hauptjunktor) oder zwei Subformeln. Nach Induktionsvoraussetzung können wir voraussetzen, dass das Lemma für diese Subformeln bereits gilt. Das bedeutet, dass wir für jeden Junktor einen Beweis für alle Fälle seiner zugeordneten Wahrheitstafel finden müssen.

Falls $\varphi = \neg\varphi_1$:

Wenn $\varphi \equiv T$ ist, brauchen wir einen Beweis $\neg\varphi_1 \vdash \neg\varphi_1$, was automatisch gilt.

Wenn $\varphi \equiv F$ ist, brauchen wir einen Beweis $\varphi_1 \vdash \neg\neg\varphi_1$:

1. φ_1 gegeben
2. $\neg\varphi_1$ angenommen
3. \perp $\neg e$ 2, 1
4. $\neg\neg\varphi_1$ $\neg i$ 2-3

Falls $\varphi = \varphi_1 \rightarrow \varphi_2$:

Wenn $\varphi \equiv T$ müssen wir drei Fälle betrachten, nämlich φ_1 ist F oder φ_2 ist T:

1. $\neg\varphi_1$ gegeben
2. φ_2 gegeben
3. φ_1 angenommen
4. φ_2 übernommen 2
5. $\varphi_1 \rightarrow \varphi_2$) $\rightarrow i$ 3-4

1. $\neg\varphi_1$ gegeben
2. $\neg\varphi_2$ gegeben
3. φ_1 angenommen
4. \perp $\neg e$ 1, 3
5. φ_2 EFQ 4
6. $\varphi_1 \rightarrow \varphi_2$ $\rightarrow i$ 3-5

1. φ_1 gegeben
2. φ_2 gegeben
3. φ_1 angenommen
4. φ_2 übernommen 2
5. $\varphi_1 \rightarrow \varphi_2$ $\rightarrow i$ 3-4

Bleibt für die Implikation noch der Fall, dass φ zu F auswertet:

1. φ_1 gegeben
2. $\neg\varphi_2$ gegeben
3. $\varphi_1 \rightarrow \varphi_2$ angenommen
4. φ_2 $\rightarrow e$ 3, 1
5. \perp $\neg e$ 2, 4
6. $\neg(\varphi_1 \rightarrow \varphi_2)$ $\neg i$ 3-5

Falls $\varphi = \varphi_1 \wedge \varphi_2$:

Wenn φ zu T auswertet, haben wir einen Fall zu betrachten:

1. φ_1 gegeben
2. φ_2 gegeben
3. $\varphi_1 \wedge \varphi_2$ $\wedge i$ 1, 2

Ist $\varphi \equiv F$ sind drei Fälle zu zeigen:

1. φ_1 gegeben
2. $\neg\varphi_2$ gegeben
3. $\varphi_1 \wedge \varphi_2$ angenommen
4. φ_2 $\wedge e_2$ 3
5. \perp $\neg e$ 2, 4
6. $\neg(\varphi_1 \wedge \varphi_2)$ $\neg i$ 3-5

Der Fall $\neg\varphi_1$ und φ_2 ist symmetrisch zum eben gezeigten Fall.

Für den Fall $\neg\varphi_1$ und $\neg\varphi_2$ kann man obigen Beweis verwenden, denn die Aussage in Zeile 1 haben wir für die Herleitung ja garnicht verwendet, sie funktioniert also auch, wenn statt $\varphi_1 \not\models_1$ gegeben ist.

Falls $\varphi = \varphi_1 \vee \varphi_2$:

Ist $\varphi \equiv T$ haben wir drei Fälle zu betrachten:

1. φ_1 gegeben
2. φ_2 gegeben
3. $\varphi_1 \vee \varphi_2$ $\vee i_1$ 1

Diese Herleitung geht auch, wenn $\neg\varphi_2$ vorausgesetzt wird. Und nehmen wir den Fall, dass $\neg\varphi_1$ gilt, gibt es die symmetrische Herleitung mit der Regel $\vee i_2$.

Ist schließlich $\varphi \equiv F$ können wir folgende Herleitung nehmen:

1.	$\neg\varphi_1$	gegeben
2.	$\neg\varphi_2$	gegeben
3.	$\varphi_1 \vee \varphi_2$	angenommen
4.	φ_1	angenommen
5.	\perp	$\neg e 1, 4$
6.	φ_2	angenommen
7.	\perp	$\neg e 2, 6$
8.	\perp	$\vee e 3, 4-5], 6-7$
9.	$\neg(\varphi_1 \vee \varphi_2)$	$\neg i 3-8$

□

Nun können wir den konstruktiven Beweis des Vollständigkeitssatzes abschließen, in dem wir zeigen, dass

$$\vDash \varphi \Rightarrow \vdash \varphi$$

gilt.

Beweis. Da $\vDash \varphi$ gilt, hat die Formel φ eine Wahrheitstafel, in der alle Zeilen zu T auswerten. Seien P_1, P_2, \dots, P_n die Aussagensymbole in φ . Die Wahrheitstafel hat dann 2^n Zeilen.

Wir beginnen die Konstruktion der Herleitung, indem wir mit der Regel TND als erste Zeile der Herleitung $P_1 \vee P_2$ einführen. Der nächste Schritt besteht dann darin, dass wir zwei Unterbeweise für die Auflösung des \vee der ersten Zeile aufmachen. In jeder dieser beiden Boxen wenden wir nun erneut die Regel TND mit P_2 , gefolgt von $\vee e$.

Nach zwei Schritten sieht die Herleitung so aus:

1.	$P_1 \vee \neg P_1$	TND
2.	P_1	angenommen
3.	$P_2 \vee \neg P_2$	TND
4.	P_2	angenommen
5.	:	
6.	φ	
7.	$\neg P_2$	angenommen
8.	:	
9.	φ	
10.	φ	ve
11.	$\neg P_1$	angenommen
12.	$P_2 \vee \neg P_2$	TND
13.	P_2	angenommen
14.	:	
15.	φ	
16.	$\neg P_2$	angenommen
17.	:	
18.	φ	
19.	φ	ve
20.	φ	ve

Wenn man das Verfahren abwechselnd die Regel TND und die Auflösung von \vee bis zu P_n fortsetzt, dann hat man 2^n Boxen mit Beweisverpflichtungen. Jede der Boxen entspricht genau einer Zeile der Wahrheitstafel und hat als Voraussetzungen die entsprechenden Literale \hat{P}_i .

In jeder Box können wir nach Lemma 5.9 eine Herleitung rekursiv konstruieren. Diese Herleitungen fügen wir an Stelle der Beweisverpflichtungen ein — und so erhalten wir eine Herleitung für φ . \square

Bemerkung In Lemma 5.9 und im Beweis eben wurden alle Regeln zur Einführung und Auflösung von $\wedge, \vee, \rightarrow, \neg$ verwendet, außerdem EFQ und TND. Dies zeigt, dass diese Regeln für das natürliche Schließen in der klassischen Aussagenlogik ausreichend sind.

5.5.4 Kompaktheitssatz

Der Vollständigkeitssatz hat eine wichtige Folgerung, den Endlichkeitsatz der Erfüllbarkeit oder den *Kompaktheitssatz*:

Satz 5.6 (Kompaktheitssatz). *Eine Menge Γ von Aussagen ist erfüllbar, wenn jede endliche Teilmenge von Γ erfüllbar ist.*

Beweis. Wir zeigen die Kontraposition: Wenn Γ nicht erfüllbar ist, dann gibt es eine endliche Teilmenge von Γ , die nicht erfüllbar ist.

Nehmen wir also an, dass Γ nicht erfüllbar ist. Nach dem Vollständigkeitssatz gilt dann $\Gamma \vdash \perp$. Ein solcher Beweis kommt aber mit einer endlichen Menge von Voraussetzungen Γ' aus. Das bedeutet, dass es eine endliche Teilmenge von Γ gibt, eben Γ' , die nicht erfüllbar ist. \square

Kapitel 6

Normalformen

6.1 Negationsnormalform NNF

Wir beobachten zunächst, dass man jede Formel äquivalent so umformen kann, dass sie keine Implikationen mehr enthält. Dazu verwendet man die Äquivalenz $\varphi \rightarrow \psi \equiv \neg\varphi \vee \psi$.

Definition 6.1 (Literal). Ein *Literal* ist eine Primaussage P oder ihre Negation $\neg P$.

Wir formulieren einen Algorithmus für die Elimination der Implikation aus aussagenlogischen Formeln:

```
function IMPL_FREE( $\varphi$ ) {
    // pre: beliebige Formel  $\varphi$ 
    // post: äquivalente Umformung von  $\varphi$ , die kein  $\rightarrow$  mehr enthält
    case {
         $\varphi$  ist Literal:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\varphi_1$ :
            return  $\neg\text{IMPL\_FREE}(\varphi_1)$ ;
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return  $\text{IMPL\_FREE}(\varphi_1) \wedge \text{IMPL\_FREE}(\varphi_2)$ ;
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return  $\text{IMPL\_FREE}(\varphi_1) \vee \text{IMPL\_FREE}(\varphi_2)$ ;
         $\varphi$  hat die Form  $\varphi_1 \rightarrow \varphi_2$ :
            return  $\neg\text{IMPL\_FREE}(\varphi_1) \vee \text{IMPL\_FREE}(\varphi_2)$ ;
    }
}
```

Definition 6.2 (Negationsnormalform NNF). Eine Formel φ ohne Implikation ist in der *Negationsnormalform NNF*, wenn jede Negation direkt vor einer Primaussage steht.

Beispiele

$\neg\neg P$	nicht NNF	$\neg P$	NNF
$\neg(P \wedge Q)$	nicht NNF	$\neg P \vee \neg Q$	NNF
$\neg(P \vee Q)$	nicht NNF	$\neg P \wedge \neg Q$	NNF

Wir formulieren einen Algorithmus, der eine Formel in die Negationsnormalform bringt:

```
function NNF( $\varphi$ ) {
    // pre:  $\varphi$  hat keine Implikationen
    // post: äquivalente Umformung von  $\varphi$  in NNF
    case {
         $\varphi$  ist Literal:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\neg\varphi_1$ :
            return NNF( $\varphi_1$ );
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return NNF( $\varphi_1$ )  $\wedge$  NNF( $\varphi_2$ );
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return NNF( $\varphi_1$ )  $\vee$  NNF( $\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \wedge \varphi_2)$ :
            return NNF( $\neg\varphi_1$ )  $\vee$  NNF( $\neg\varphi_2$ );
         $\varphi$  hat die Form  $\neg(\varphi_1 \vee \varphi_2)$ :
            return NNF( $\neg\varphi_1$ )  $\wedge$  NNF( $\neg\varphi_2$ );
    }
}
```

6.2 Konjunktive Normalform CNF

Definition 6.3 (Klausel). Eine Formel der Form $\varphi = \hat{P}_1 \vee \hat{P}_2 \vee \dots \vee \hat{P}_n$ mit Literalen \hat{P}_i nennt man eine *Klausel*.

Beispiele:

$P \vee \neg Q \vee \neg R$ ist eine Klausel
 $\neg P_1 \vee \neg P_2 \vee \dots \vee \neg P_n \vee Q$ ist eine Klausel,
sie ist äquivalent zu
 $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$.

Bemerkung: Oft stellt man Klauseln als Mengen von Literalen dar, wobei mit \bar{P} die Negation $\neg P$ einer Primaussage bezeichnet wird. Der Grund dafür besteht darin, dass logischen Operationen mit Formeln in konjunktiver Normalform und Klauseln mengentheoretischen Operationen auf Klauselmengen und Klauseln entsprechen.

Beispiele: Korrespondierend zu obigen Beispielen

$$\begin{aligned} &\{P, \bar{Q}, \bar{R}\} \\ &\{\bar{P}_1, \bar{P}_2, \dots, \bar{P}_n, Q\} \end{aligned}$$

Definition 6.4 (Konjunktive Normalform CNF). Eine Formel φ ist in der *konjunktiven Normalform CNF*, wenn sie die Form $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ hat mit lauter Klauseln φ_i .

Wir formulieren einen Algorithmus, der eine Formel in die konjunktive Normalform bringt:

```
function DISTR( $\varphi_1, \varphi_2$ ) {
    // pre:  $\varphi_1$  und  $\varphi_2$  sind in CNF
    // post: berechnet CNF für  $\varphi_1 \vee \varphi_2$ 
    case {
         $\varphi_1$  hat die Form  $\varphi_{11} \wedge \varphi_{12}$ :
            return DISTR( $\varphi_{11}, \varphi_2$ )  $\wedge$  DISTR( $\varphi_{12}, \varphi_2$ );
         $\varphi_2$  hat die Form  $\varphi_{21} \wedge \varphi_{22}$ :
            return DISTR( $\varphi_1, \varphi_{21}$ )  $\wedge$  DISTR( $\varphi_1, \varphi_{22}$ );
        default:
            return  $\varphi_1 \vee \varphi_2$ ;
    }
}
```

Mit Hilfe der Hilfsfunktion DISTR ist es nun einfach, einen Algorithmus für das Erzeugen der CNF zu formulieren:

```
function CNF( $\varphi$ ) {
    // pre:  $\varphi$  ist in NNF
    // post: eine zu  $\varphi$  äquivalente Formel in CNF
    case {
         $\varphi$  ist Literal:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\varphi_1 \wedge \varphi_2$ :
            return CNF( $\varphi_1$ )  $\wedge$  CNF( $\varphi_2$ );
         $\varphi$  hat die Form  $\varphi_1 \vee \varphi_2$ :
            return DISTR(CNF( $\varphi_1$ ), CNF( $\varphi_2$ ));
    }
}
```

}
}

6.3 Disjunktive Normalform DNF

Definition 6.5 (Monom). Eine Formel der Form $\psi = \hat{P}_1 \wedge \hat{P}_2 \wedge \dots \wedge \hat{P}_n$ mit Literalen \hat{P}_i bezeichnet man als *Monom*.

Definition 6.6 (Disjunktive Normalform DNF). Eine Formel ψ ist in der *disjunktiven Normalform DNF*, wenn sie die Form $\psi_1 \vee \psi_2 \vee \dots \vee \psi_n$ hat mit lauter Monomen ψ_i .

Beobachtung: („Dualität“ von CNF und DNF)

Ist die Formel φ in CNF, dann ist $\neg\varphi$ modulo simpler Transformationen in DNF.

Ist φ in CNF, dann hat φ die Form $\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n$ mit Klauseln φ_i .

Betrachte nun $\neg\varphi$, also $\neg(\varphi_1 \wedge \varphi_2 \wedge \dots \wedge \varphi_n)$.

Nach De Morgan ist dies äquivalent zu $\neg\varphi_1 \vee \neg\varphi_2 \vee \dots \vee \neg\varphi_n$.

Wendet man nun De Morgan auch auf die Klauseln φ_i an und eliminiert doppelte Negation, dann erhält man Monome und die Formel $\neg\varphi$ ist in DNF.

6.4 Normalformen und Entscheidungsprobleme

Ein *komplementäres* Paar von Literalen ist eine Primaussage samt seiner Negation, also z.B. P und $\neg P$.

6.4.1 CNF und Gültigkeit

Satz 6.1 (CNF und Gültigkeit). *Sei φ in CNF. Dann gilt:*

φ ist allgemeingültig \Leftrightarrow Jede Klausel enthält ein komplementäres Paar von Literalen

Beweis. Wenn φ allgemeingültig ist, dann könnte man sich als kürzeste Darstellung in CNF \top vorstellen, \top gehört aber nicht zum Alphabet unsere Sprache. Eine Darstellung der Formel in CNF enthält also Klauseln mit Aussagensymbolen aus φ . Diese Klauseln sind mit \wedge verbunden, müssen also alle zu \top auswerten, damit die Formel wahr wird. Eine Klausel kann aber nur zu \top auswerten, wenn sie ein komplementäres Paar von Literalen enthält. \square

6.4.2 DNF und Erfüllbarkeit

Satz 6.2 (DNF und Erfüllbarkeit). *Sei ψ in DNF. Dann gilt:*

ψ ist unerfüllbar \Leftrightarrow Jedes Monom enthält ein komplementäres Paar von Literalen

Beweis. Die kürzeste Darstellung von ψ ist \perp . Angenommen die Kontradiktion ψ ist als Disjunktion von Monomen dargestellt. Dann müssen diese alle ebenfalls Kontradiktionen sein. Ein Monom ist genau dann unerfüllbar, wenn sie ein komplementäres Paar von Literalen enthält. \square

Übung. Vorausgesetzt φ ist eine Tautologie in konjunktiver Normalform CNF, wie kann man dann daraus eine Herleitung von φ nach den Regeln des natürlichen Schließens *konstruieren*? Diese Übung ergibt einen Teil eines anderen Beweises der Vollständigkeit des natürlichen Schließens (siehe 5.5) in der Aussagenlogik.

Kapitel 7

Die Komplexität des Erfüllbarkeitsproblems

7.1 Das Erfüllbarkeitsproblem

7.1.1 Entscheidungsfragen der Aussagenlogik

In der Aussagenlogik kann man sich zu einer gegebenen Formel φ folgende Fragen stellen:

- Ist φ *allgemeingültig*?

Diese Frage nennt man auch das Gültigkeitsproblem. Die Formel φ ist allgemeingültig, wenn sie für jede beliebige Belegung der Aussagensymbole wahr ist. Ist eine Formel φ allgemeingültig, nennt man φ auch eine *Tautologie*.

- Ist φ *unerfüllbar*?

Diese Frage nennt man auch das Unerfüllbarkeitsproblem. Die Formel φ ist unerfüllbar, wenn es keine Belegung der Aussagensymbole gibt, die sie wahr macht. Ist eine Formel φ unerfüllbar, sagt man auch φ ist der *Widerspruch* oder eine *Kontradiktion*.

- Ist φ *erfüllbar*?

Diese Frage nennt man auch das Erfüllbarkeitsproblem. Die Formel φ ist erfüllbar, wenn es eine Belegung gibt, unter der die Formel wahr ist. In der Regel ist man dann natürlich auch interessiert daran, ein solches Modell zu finden.

- Ist φ *falsifizierbar*?

Diese Frage nennt man auch das Widerlegungsproblem. Die Formel φ ist falsifizierbar oder widerlegbar, wenn es eine Belegung gibt, unter der die Formel falsch ist. Auch dann möchte man üblicherweise wissen, für welches Modell die Formel falsch ist.

Diese Fragen hängen miteinander zusammen. Denn aus den Definitionen ergibt sich unmittelbar:

- Eine Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.
- Eine Formel φ ist genau dann falsifizierbar, wenn $\neg\varphi$ erfüllbar ist.

Also genügt es, die Frage zu betrachten, ob eine Formel *erfüllbar* ist. Wenn ja, unter welcher Belegung?

7.1.2 Das Erfüllbarkeitsproblem

Die Frage, ob eine Formel erfüllbar ist, wird als das *Erfüllbarkeitsproblem*, auch *SAT-Problem* (*SAT* = *satisfiability*) oder ganz kurz *SAT*, bezeichnet.

Es ist nicht schwierig, das Erfüllbarkeitsproblem zu lösen. Hat man eine Formel φ , dann stellt man die Wahrheitstafel für die Formel auf und sieht nach, ob es eine Zeile der Wahrheitstafel gibt, in der die Formel wahr ist.

Dieses Vorgehen ist jedoch nur für Formeln mit wenig Atomen geeignet. Hat φ n Atome, dann hat die Wahrheitstafel 2^n Zeilen. Hat also zum Beispiel eine Formel 100 Atome und die Untersuchung einer Zeile der Wahrheitstafel dauert 10^{-10} Sekunden, dann dauert die Untersuchung der gesamten Wahrheitstafel $2^{100} \times 10^{-10}$ Sekunden, also etwa 4×10^{12} Jahre. „Das ist länger als die Zeit, die seit der Entstehung des Universums vergangen ist.“¹

7.2 Komplexität von Algorithmen

Ein *Algorithmus* ist ein Verfahren, das in endlich vielen Schritten zur Lösung eines Problems führt. Mit dieser informellen Definition eines Algorithmus wollen wir im Folgenden arbeiten.²

Die *Komplexität* eines Algorithmus wird gemessen durch die Menge an Ressourcen (Zeit oder Speicher), die für die Durchführung des Algorithmus im Prinzip benötigt wird. Wir werden die Laufzeit eines Algorithmus im Folgenden als seine Komplexität betrachten.

¹Die beispielhafte Berechnung der Ineffizienz der Entscheidung des Erfüllbarkeitsproblems durch die Wahrheitstafel ist aus: Uwe Schöning: *Das SAT-Problem* in: Informatik Spektrum Band 33 Heft 5 Oktober 2010

²Präziseres findet man in: [Sip13, Chap. 3.3]

7.2.1 Arten von Algorithmen

Definition 7.1. Ein Algorithmus ist *deterministisch*, wenn die Berechnung und somit das Ergebnis vollständig durch die Eingabe bestimmt wird. Ein deterministischer Algorithmus ist genau dann *korrekt*, wenn das Ergebnis zur gegebenen Eingabe korrekt ist.

Beispiel 7.1. Der Algorithmus mittels der Wahrheitstafel die Erfüllbarkeit einer Formel zu berechnen ist ein deterministischer Algorithmus für das Erfüllbarkeitsproblem.

Definition 7.2. Ein Algorithmus ist *nichtdeterministisch*, wenn seine Schritte mehrere mögliche Folgeschritte haben, deren Wahl nicht durch die Eingabe und bisherige Berechnung bestimmt wird. Zu einer Eingabe sind also mehrere verschiedene Berechnungsergebnisse möglich. Ein nichtdeterministischer Algorithmus heißt *korrekt*, wenn unter den möglichen Ergebnissen wenigstens eines korrekt ist.

Beispiel 7.2. Ein nichtdeterministischer Algorithmus für das Erfüllbarkeitsproblem ist leicht zu definieren. Der erste Schritt besteht darin, eine Belegung der Aussagensymbole zu raten. Im zweiten Schritt wird die Formel berechnet.

Dieser Algorithmus ist sogar ein korrekter nichtdeterministischer Algorithmus. Wenn die Formel erfüllbar ist, dann gibt es eine erfüllende Belegung. Rät der Algorithmus mal richtig, dann entscheidet er das Erfüllbarkeitsproblem korrekt.

7.2.2 Laufzeit von Algorithmen

Nun definieren wir Maße für die Komplexität von Algorithmen:

Definition 7.3. Ein Algorithmus heißt *polynomiell*, wenn seine Laufzeit nach oben durch ein Polynom in n (n ist die Größe der Eingabe) beschränkt ist. Algorithmen, die in polynomieller Zeit ein korrektes Ergebnis liefern, werden oft auch als *effiziente* Algorithmen bezeichnet.

Beispiel 7.3. Unser nichtdeterministischer Algorithmus durch Erraten das Erfüllbarkeitsproblem zu lösen ist polynomiell: Das Erraten einer Belegung ist linear bezüglich der Größe der Eingabe, d.h. der Anzahl n der Atome und das Überprüfen, ob das Erratene zutrifft ist offensichtlich effizient durchführbar.

Definition 7.4. Ein Algorithmus läuft in *exponentieller* Zeit, wenn die Laufzeit nach unten durch eine Funktion 2^{cn} für die Größe der Eingabe n und ein $c > 0$ beschränkt ist.

Beispiel 7.4. Die Methode mit der Wahrheitstafel die Erfüllbarkeit einer Formel zu entscheiden ist exponentiell: Die Größe der Eingabe ist die Zahl n der Atome der Formel. Im schlechtesten Fall muss man zur Entscheidung alle 2^n Zeilen der Wahrheitstafel konstruieren.

7.2.3 Klassen von Problemen und \mathcal{NP} -Vollständigkeit

Definition 7.5. Die Klasse \mathcal{P} bezeichnet die Probleme, die in polynomieller Zeit durch einen deterministischen Algorithmus gelöst werden können.

Beispiel 7.5. Es ist nicht bekannt, ob das Erfüllbarkeitsproblem zur Klasse \mathcal{P} gehört. Es wird vermutet, dass dies nicht der Fall ist, wie wir gleich genauer diskutieren werden.

Definition 7.6. Die Klasse \mathcal{NP} bezeichnet die Probleme, die ein nicht-deterministischer Algorithmus in polynomieller Zeit lösen kann.

Beispiel 7.6. Das Erfüllbarkeitsproblem gehört zur Klasse \mathcal{NP} . Man sagt auch: SAT ist \mathcal{NP} .

Man kann auch so ausdrücken, dass ein Problem in \mathcal{NP} ist: eine *Lösung* des Problems ist in polynomieller Zeit *überprüfbar*, auch wenn sie möglicherweise nicht in polynomieller Zeit *gefunden* werden kann.

Vermutung: Ist $\mathcal{P} = \mathcal{NP}$? Dies ist eine grundlegende Frage der Informatik, die ungelöst ist.³ Allgemein wird vermutet, dass gilt:

$$\mathcal{P} \neq \mathcal{NP}.$$
⁴

Zur genaueren Bestimmung der Komplexität des Erfüllbarkeitsproblems benötigen wir noch weitere Definitionen:

Definition 7.7. Ein Problem P ist \mathcal{NP} -schwierig, wenn sich jedes Problem Q in \mathcal{NP} deterministisch in polynomieller Zeit auf P reduzieren lässt.

³Das Clay Mathematics Institute <http://www.claymath.org> hat dieses Problem als eines der 7 Millenniums-Probleme ausgewählt – neben u.a. der Riemann-Vermutung oder der (mittlerweile von Grigori Perelman gelösten) Poincaré-Vermutung.

⁴Donald E. Knuth sieht das anders: „... almost everybody who has studied the subject thinks that satisfiability cannot be decided in polynomial time. The author of this book, however, suspects that $N^{O(1)}$ -step algorithms do exist, yet that they're unknowable. Almost all polynomial time algorithms are so complicated that they lie beyond human comprehension, and could never be programmed for an actual computer in the real world. Existence is different from embodiment.“ [Knu15, S. 1]

Definition 7.8. Ein Problem P ist \mathcal{NP} -vollständig, wenn es in \mathcal{NP} und \mathcal{NP} -schwierig ist.

7.3 Die Komplexität des Erfüllbarkeitsproblems

Nun stehen alle Definitionen zur Verfügung, um den Satz zu formulieren, der die Komplexität des Erfüllbarkeitsproblems bestimmt:

Satz 7.1 (Cook 1971, Levin 1973). *Das Erfüllbarkeitsproblem ist \mathcal{NP} -vollständig.⁵*

Wie sieht es mit dem komplementären Problem der Nichterfüllbarkeit bzw. der Allgemeingültigkeit aus? Dieses Problem ist insofern schwieriger, als man ja zeigen muss, dass es kein Modell gibt, bzw. dass die Aussage in allen Belegungen wahr ist. Hier hilft „Raten“ nicht mehr. Genauer sagt man:

Definition 7.9. Ein Problem ist in der Klasse $Co\text{-}\mathcal{NP}$, wenn das komplementäre Problem in \mathcal{NP} ist.

Beispiel: Das Nichterfüllbarkeitsproblem ist in $Co\text{-}\mathcal{NP}$.

Satz 7.2. $Co\text{-}\mathcal{NP} = \mathcal{NP}$ genau dann, wenn Nichterfüllbarkeit in \mathcal{NP} ist.

Vermutung: Es ist nicht bekannt, ob es einen nichtdeterministischen polynomiellen Algorithmus für das Nichterfüllbarkeitsproblem gibt. Man nimmt vielmehr an, dass gilt:

$$Co\text{-}\mathcal{NP} \neq \mathcal{NP}$$

Die Ergebnisse über die Komplexität des Erfüllbarkeitsproblems können zu der Annahme verleiten, dass es keinen „effizienten“ Algorithmus für SAT geben kann und deshalb Probleme, die man als aussagenlogische Formeln formulieren kann, nur gelöst werden können, wenn man wenige aussagenlogischen Atome in der Formel hat.

⁵Stephen A. Cook, amerikanischer Informatiker, heute Professor für Informatik in Toronto. Er erhielt 1982 für diesen Satz den Turing-Award. Leonid Levin, ukrainischer Informatiker, hat die Theorie der \mathcal{NP} -Vollständigkeit und den Satz über das Erfüllbarkeitsproblem 1973 entwickelt. Seine Ergebnisse waren im Westen zunächst nicht bekannt. 1978 emigrierte er in die USA.

Dies ist jedoch keineswegs der Fall: „Zum Glück gibt es und gab es Algorithmenentwickler, Theoretiker und Praktiker, die sich von diesem Negativergebnis [Satz von Cook und Levin] nicht abschrecken ließen. Dies hat in den vergangenen Jahren dazu geführt, dass die ‚SAT-Solver‘-Technologie immer weiter vorangeschritten ist [und] dass diese das SAT-Problem lösenden Verfahren heutzutage mit Formeln, die Tausende von Variablen enthalten, in Sekundenbruchteilen fertig werden.“⁶

⁶So schreibt Uwe Schöning im bereits zitierten Artikel. Ein solcher SAT-Solver ist **Sat4j** – siehe <http://www.sat4j.org/>.

Kapitel 8

Hornlogik

Betrachtet man spezielle Klassen von Formeln, dann findet man oft effiziente Algorithmen für die Entscheidung des Erfüllbarkeitsproblems. Eine wichtige solche Klasse sind die Hornformeln¹.

Definition 8.1 (Hornklausel). Eine Klausel $\hat{P}_1 \vee \hat{P}_2 \vee \dots \vee \hat{P}_n$ heißt *Hornklausel*, wenn höchstens eines der Literale \hat{P}_i positiv ist.

Definition 8.2 (Hornformel). Eine Formel φ in CNF heißt *Hornformel*, wenn jede Klausel eine Hornklausel ist.

Definition 8.3 (Arten von Hornklauseln).

- Besteht eine Hornklausel nur aus einem positiven Literal, dann heißt sie *Tatsachenklausel*, kurz *Tatsache*.
- Hat eine Hornklausel ein positives Literal und mindestens ein negatives Literal, dann heißt sie *Prozedurklausel* oder *Regel*.
- Hat die Hornklausel kein positives Literal, dann heißt sie *Zielklausel* oder *Frageklausel*, kurz *Ziel*.

Beispiel 8.1 (Hornklauseln und Hornformel).

- $\{P\}$ ist eine Tatsachenklausel,
- $\{\neg P, Q\}$ und $\{\neg P, \neg Q, R\}$ sind Regeln und
- $\{\neg P, \neg R\}$ ist eine Zielklausel.

Die Hornformel in diesem Beispiel ist also:

$$P \wedge (\neg P \vee Q) \wedge (\neg P \vee \neg Q \vee R) \wedge (\neg P \vee \neg R)$$

Wir können Hornformel auch (alternativ) definieren, indem wir eine Grammatik angeben:

¹Alfred Horn, amerikanischer Mathematiker, 1918 - 2001

Definition 8.4 (Hornformel). Eine Hornformel ist eine Formel φ der Aussagenlogik, die folgender Grammatik genügt:

$$\begin{aligned} T &:= \top \rightarrow P \text{ für Aussagensymbole } P \text{ (Tatsache)} \\ P &:= P \mid P \wedge P \text{ für Aussagensymbole } P \\ Z &:= P \mid P \rightarrow \perp \text{ (Ziel)} \\ R &:= P \rightarrow Q \text{ für Aussagensymbole } Q \text{ (Regel)} \\ H &:= \varphi \wedge T \mid \varphi \wedge Z \mid \varphi \wedge R \end{aligned}$$

Die Bezeichnungen *Tatsache*, *Regel*, *Ziel* ergeben sich aus folgenden Überlegungen:

1. Jede Hornklausel ist entweder eine Tatsache, eine Regel oder ein Ziel.
2. Eine Tatsache muss wahr sein, d.h. $P \equiv \top \rightarrow P$.
3. Eine Prozedurklausel (Regel) $\neg P_1 \vee \dots \vee \neg P_n \vee Q$ ist äquivalent zu $P_1 \wedge \dots \wedge P_n \rightarrow Q$.
4. Eine Zielklausel $\neg P_1 \vee \dots \vee \neg P_n$ ist äquivalent zu $\neg(P_1 \wedge \dots \wedge P_n)$, also $P_1 \wedge \dots \wedge P_n \rightarrow \perp$.

Beispiel 8.2 (Hornformel). Die Hornformel aus Beispiel 8.1 ausgedrückt mit der Grammatik 8.4:

$$(\top \rightarrow P) \wedge (P \rightarrow Q) \wedge (P \wedge Q \rightarrow R) \wedge (P \wedge R \rightarrow \perp)$$

In der Hornlogik wird typischerweise ein Widerlegungsverfahren verwendet. In der Hornformel wird die angestrebte Aussage als *nicht* wahr angenommen (deshalb ist die Form der Zielklausel so wie oben definiert). Kann man nun die Nichterfüllbarkeit der Hornformel zeigen, sieht man, dass die Zielklausel zutrifft.

Wir formulieren einen Algorithmus für die Entscheidung der Erfüllbarkeit von Hornformeln (den sogenannten Markierungsalgorithmus):

```
function HORN( $\varphi$ ) {
    // pre:  $\varphi$  ist eine Hornformel
    // post: entscheidet die Frage, ob  $\varphi$  erfüllbar ist
    markiere alle  $\top$  in  $\varphi$ 
    while ( es gibt  $P_1 \wedge P_2 \wedge \dots \wedge P_n \rightarrow Q$  mit
             $P_1, P_2, \dots, P_n$  markiert, aber  $Q$  nicht) {
        markiere  $Q$ 
```

```

    }
    if (  $\perp$  ist markiert ){
        return „unerfüllbar“
    } else {
        return „erfüllbar“
    }
}

```

Eigenschaften dieses Algorithmus

1. Er entscheidet, ob die Hornformel φ erfüllbar ist.
 2. Ist φ erfüllbar, dann können wir eine erfüllende Belegung ablesen, nämlich
- $$v(P_i) = \begin{cases} \text{T falls } P_i \text{ markiert ist} \\ \text{F sonst} \end{cases}$$
3. Der Algorithmus endet nach spätestens $n+2$ Markierungsschritten, wenn n die Zahl der Atome in φ ist.

Beispiel 8.3 (Markierungsalgorithmus).

Gegeben sei die Aussage P , d.h. $\top \rightarrow P$, eine *Tatsache*.

Ferner sei bekannt, dass folgende *Regeln* gelten: $P \rightarrow Q$ und $P \wedge Q \rightarrow R$. Man möchte nun in dieser Situation wissen, ob $P \wedge R$ gilt.

In der Hornlogik klärt man die Frage durch ein *Widerlegungsverfahren*: Man nimmt an, dass $P \wedge R$ nicht gilt. Durch diese Annahme entsteht folgende Hornformel:

$$(\top \rightarrow P) \wedge (P \rightarrow Q) \wedge (P \wedge Q \rightarrow R) \wedge (P \wedge R \rightarrow \perp)$$

Wenn gezeigt werden kann, dass diese Formel *unerfüllbar* ist, dann hat man gezeigt, dass in der gegebenen Situation $P \wedge R$ gilt. Dies zeigt man nun mit Hilfe des Markierungsalgorithmus:

$$\frac{(\top \rightarrow P) \wedge (P \rightarrow Q) \wedge (P \wedge Q \rightarrow R) \wedge (P \wedge R \rightarrow \perp)}{\begin{array}{cccccccccc} * & & & & & & & & & \\ * & * & * & & * & & & & & * \\ * & * & * & * & * & * & & & & * \\ * & * & * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * & * & * \end{array}}$$

Der Algorithmus endet in diesem Beispiel, wenn \perp markiert ist. Das bedeutet, dass die Formel unerfüllbar ist, d.h. dass in der gegebenen Situation $P \wedge R$ gilt.

Kapitel 9

Erfüllbarkeit und SAT-Solver

Die Entscheidungsfragen der Aussagenlogik lassen sich alle auf das Erfüllbarkeitsproblem zurückspielen.

Ein Programm, das für eine Formel der Aussagenlogik das Erfüllbarkeitsproblem löst (und eine erfüllende Belegung ermittelt), heißt *SAT-Solver*.

Die Kunst in der Entwicklung von SAT-Solvern besteht darin, Algorithmen zu finden, die für große Klassen von Formeln das Problem effizient entscheiden, obgleich es \mathcal{NP} -vollständig ist.¹

Typischerweise setzen SAT-Solver voraus, dass eine Formel in CNF vorliegt. Das Eingabeformat ist üblicherweise DIMACS (vorgeschlagen vom Center for Discrete Mathematics and Theoretical Computer Science <http://dimacs.rutgers.edu/>).

9.1 DIMACS-Format

SAT-Solver verwenden eine einfache Variante des DIMACS-Formats. Eine Formel in CNF wird in einer ASCII-Datei gespeichert, die in drei Sektionen aufgebaut ist:

Zuerst kommen optionale *Kommentarzeilen*, die durch ein kleines c an der ersten Position gekennzeichnet sind.

¹Donald E. Knuth: „The story of satisfiability is the tale of a triumph of software engineering, blended with rich doses of beautiful mathematics. Thanks to elegant new data structures and other techniques, modern SAT solvers are able to deal routinely with practical problems that involve many thousands of variables, although such problems were regarded as hopeless just a few years ago.“[Knu15, S. iv]

Darauf folgt die *Präambel*, die aus einer Zeile der Form

p cnf v c

besteht, wobei v für die Zahl der Atome der Formel² und c für die Zahl der Klauseln steht.

Danach folgen die *Klauseln*. Die Literale werden durch Integers codiert. Dabei steht die positive Zahl für ein Atom, die negative Zahl für seine Negation. Jede Klausel nimmt eine Zeile der Datei ein, sie enthält die Literale getrennt durch Leerzeichen und wird abgeschlossen durch eine 0.

Beispiel φ sei die folgende Formel in CNF:

$$\begin{aligned}
 & (P_1 \vee P_2 \vee P_3) \wedge \\
 & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\
 & (P_1 \vee \neg P_5) \wedge \\
 & (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\
 & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\
 & (P_4 \vee P_6) \wedge \\
 & (P_4 \vee \neg P_6) \wedge \\
 & (P_2 \vee \neg P_4) \wedge \\
 & (\neg P_3 \vee \neg P_4)
 \end{aligned}$$

Sie wird im DIMACS-Format so geschrieben:

```

c Beispiel DIMACS Vorlesung LfM
p cnf 6 9
1 2 3 0
1 -2 -3 0
1 -5 0
-2 -3 -5 0
-1 -2 3 0
4 6 0
4 -6 0
2 -4 0
-3 -4 0

```

Doch stop! Wir haben gesehen, dass die Umformung einer beliebigen Formel in eine äquivalente Formel in CNF im schlechtesten Fall eine exponentielle Laufzeit (in Bezug auf die Zahl der Atome der Formel) haben kann. Führt das nicht schon von vorneherein dazu, dass die Entscheidung der Erfüllbarkeit *nicht* effizient sein kann, noch ehe der SAT-

²Sieht man die aussagenlogische Formel als eine Boolesche Funktion spricht man auch von *Variablen*, deshalb der Buchstabe v.

Solver überhaupt startet, weil CNF als Eingabeform erwartet wird? Dies ist nicht der Fall:

9.2 Tseitin-Transformation

Definition 9.1 (Erfüllbarkeitsäquivalenz). Zwei Formeln der Aussagenlogik φ und ψ heißen *erfüllbarkeitsäquivalent*, wenn gilt:

$$\varphi \text{ erfüllbar} \Leftrightarrow \psi \text{ erfüllbar.}$$

Die *Tseitin*³-Transformation besteht nun darin, eine beliebige Formel φ in eine *erfüllbarkeitsäquivalente* Formel in CNF umzuformen. Die Tseitin-Transformation ist linear in der Zahl der Atome der Formel.

```
function TSEITIN( $\varphi$ ) {
// post: Eine erfüllbarkeitsäquivalente Formel  $\varphi'$  in CNF
```

1. Führe für jede Subformel ψ von φ , die kein Atom ist, ein neues Atom T_ψ hinzu.
2. Setze $\varphi' = T_\varphi$.
3. Durchlaufe den Syntaxbaum von φ und füge φ' je nach Form der Subformel an einem inneren Knoten die Formel in CNF nach Tabelle 9.1 verbunden mit \wedge hinzu. (Atome werden analog zu den anderen Subformeln behandelt.)

}

Tabelle 9.1: Regeln für die Tseitin-Transformation

$\varphi = \neg\varphi_1$	$(\neg T_\varphi \vee \neg T_{\varphi_1}) \wedge (T_\varphi \vee T_{\varphi_1})$
$\varphi = \varphi_1 \wedge \varphi_2$	$(\neg T_\varphi \vee T_{\varphi_1}) \wedge (\neg T_\varphi \vee T_{\varphi_2}) \wedge (T_\varphi \vee \neg T_{\varphi_1} \vee \neg T_{\varphi_2})$
$\varphi = \varphi_1 \vee \varphi_2$	$(T_\varphi \vee \neg T_{\varphi_1}) \wedge (T_\varphi \vee \neg T_{\varphi_2}) \wedge (\neg T_\varphi \vee T_{\varphi_1} \vee T_{\varphi_2})$
$\varphi = \varphi_1 \rightarrow \varphi_2$	$(T_\varphi \vee T_{\varphi_1}) \wedge (T_\varphi \vee \neg T_{\varphi_2}) \wedge (\neg T_\varphi \vee \neg T_{\varphi_1} \vee T_{\varphi_2})$

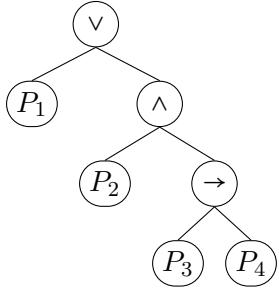
Die Tseitin-Transformation φ' einer Formel φ hat folgende Eigenschaften:

1. Die Menge der Atome von φ ist eine Teilmenge der Atome von φ' .
2. Wenn v eine erfüllende Belegung von φ ist, dann gibt es eine Erweiterung von v auf die Atome von φ' , so dass diese Belegung φ' erfüllt.
3. Ist v' eine erfüllende Belegung von φ' , dann erfüllt sie auch φ .

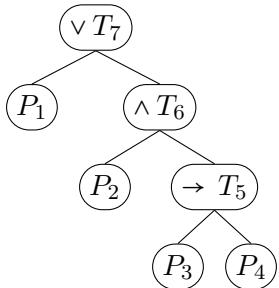
³nach Grigori S. Zeitin, russischer Mathematiker, geb. 1936 [Tse83]

9.2.1 Beispiel

Betrachten wir die Formel $\varphi = P_1 \vee (P_2 \wedge (P_3 \rightarrow P_4))$.



Im Schritt 1 legen wir pro Knoten, der nicht Blatt ist, ein neues Atom fest.



Im Schritt 2 erstellen wir φ' :

$$\begin{aligned}
 & T_7 \wedge \\
 & (T_7 \vee \neg P_1) \wedge (T_7 \vee \neg T_6) \wedge (\neg T_7 \vee P_1 \vee T_6) \wedge \\
 & (\neg T_6 \vee P_2) \wedge (\neg T_6 \vee T_5) \wedge (T_6 \vee \neg P_2 \vee \neg T_5) \wedge \\
 & (T_5 \vee P_3) \wedge (T_5 \vee \neg P_4) \wedge (\neg T_5 \vee \neg P_3 \vee P_4)
 \end{aligned}$$

9.3 DPLL und CDLC

Definition 9.2 (Wert-Propagation). Sei φ eine Formel in CNF und \hat{P} ein Literal. Dann bezeichnet $\varphi[\top/\hat{P}]$ die Formel, die dadurch entsteht, dass \hat{P} durch den Wert \top ersetzt wird.

Durch die Wert-Propagation wird die gegebene Formel folgendermaßen vereinfacht:

- Jede Klausel der Formel, die \hat{P} enthält, fällt weg. Denn sie wird durch die Wertzuweisung von \top an \hat{P} wahr.
- In jeder Klausel der Formel, in der $\neg\hat{P}$ vorkommt, entfällt dieser Ausdruck, da er \perp ist und deshalb die Klausel nicht wahr machen kann.

Der Basis-Algorithmus von SAT-Solvern basiert wesentlich auf der Wert-Propagation.

Definition 9.3 (Einzelklausel). Eine Klausel einer Formel in CNF heißt *Einzelklausel*, wenn sie aus genau einem Literal besteht.

Definition 9.4 (Reines Literal). Ein Literal in einer Formel in CNF heißt *reines Literal*, wenn es in allen Klauseln nur als P oder $\neg P$ vorkommt.

Viele SAT-Solver verwenden (stark weiterentwickelte) Varianten des folgenden Algorithmus von Martin Davis⁴, Hilary Putnam⁵, George Logemann⁶ und Donald W. Loveland⁷ ([DP60], [DLL62]).

```
boolean function DPLL( $\varphi, \mathcal{M}$ ) {
    // pre:  $\varphi$  eine Formel in CNF,  $\mathcal{M}$  eine partielle Belegung
    // return: true falls  $\varphi$  erfüllbar ist
    // post: Eine Belegung  $\mathcal{M} = \{\hat{P}_1, \hat{P}_2, \dots\}$  der Aussagensymbole
    // von  $\varphi$ , falls  $\varphi$  erfüllbar ist
    // modifies:  $\mathcal{M}$ 

    case {
         $\varphi = \top$ :
            return true;
         $\varphi = \perp$ :
            return false;
         $\varphi$  hat eine Einzelklausel ( $\hat{P}$ ):
            return DPLL( $\varphi[\top/\hat{P}], \mathcal{M} \cup \hat{P}$ );
         $\varphi$  hat ein reines Literal  $\hat{P}$ :
            return DPLL( $\varphi[\top/\hat{P}], \mathcal{M} \cup \hat{P}$ );
        otherwise:
            wähle zu einem Atom  $Q$  in der Formel mit  $\hat{Q} \notin \mathcal{M}$ 
            return DPLL( $\varphi[\top/Q], \mathcal{M} \cup Q$ )  $\vee$  DPLL( $\varphi[\perp/Q], \mathcal{M} \cup \neg Q$ );
    }
}
```

⁴Martin Davis, amerikanischer Logiker und Informatiker, geb. 1928

⁵Hilary Putnam, amerikanischer Philosoph, 1926 - 2016

⁶George Logemann, amerikanischer Mathematiker, 1938 - 2012

⁷Donald W. Loveland, amerikanischer Informatiker, geb. 1934

9.3.1 Idee des Algorithmus

1. Propagation von Einzelklauseln

Besteht eine Klausel nur aus einem Literal, also P oder $\neg P$, dann muss $P = \text{true}$ bzw. $\neg P = \text{true}$ sein, damit die Formel in CNF erfüllbar wird.

Folge: Man kann alle Klauseln weglassen, in denen \hat{P} vorkommt, und außerdem $\neg\hat{P}$ in allen Klauseln, in denen es vorkommt.

2. Elimination von reinen Literalen

Wenn in der Formel nur \hat{P} , niemals aber $\neg\hat{P}$ vorkommt, dann kann man $\hat{P} = \text{true}$ setzen. Denn dadurch fallen alle Klauseln weg, die \hat{P} enthalten, d.h. die Wahl kann niemals zu einem Widerspruch führen, weil ja weder P noch $\neg P$ noch vorkommt.

Folge: Man kann alle Klauseln weglassen, in denen \hat{P} vorkommt, denn sie sind dann erfüllt.

3. Rekursion (*Backtracking*)

Wenn keine der genannten Möglichkeiten bestehen, muss man ein Aussagensymbol Q der Formel φ wählen. Es gilt dann natürlich: φ erfüllbar $\Leftrightarrow \varphi \wedge Q$ erfüllbar oder $\varphi \wedge \neg Q$ erfüllbar.

9.3.2 Beispiele

Wir betrachten zunächst das einfache Beispiel der Formel, an der wir den Markierungsalgorithmus für Horn-Formeln demonstriert haben:

$$\begin{aligned} & P \wedge \\ & (\neg P \vee Q) \wedge \\ & (\neg P \vee \neg Q \vee R) \wedge \\ & (\neg P \vee \neg R) \end{aligned}$$

Im ersten Schritt setzt man $P \text{ true}$, da die erste Klausel eine Einzelklausel ist. Die Propagation dieses Werts führt dazu, dass diese Klausel selbst wegfällt und in allen anderen Klauseln $\neg P$ gestrichen werden kann. Man erhält also:

$$\begin{aligned} & Q \wedge \\ & (\neg Q \vee R) \wedge \\ & \neg R \end{aligned}$$

Nun ist Q auf **true** zu setzen und dieser Wert zu propagieren, also folgt:

$$\begin{array}{c} R \wedge \\ \neg R \end{array}$$

Und diese beiden Einzelklauseln bilden den Widerspruch, d.h. die Formel ist *unerfüllbar*.

Nach diesem einfachen Beispiel, das gezeigt hat, dass der Markierungsalgorithmus im Grunde mit der Idee der Propagation von Einzelklauseln arbeitet, folgt das Beispiel von 9.1:

φ sei wieder die folgende Formel in CNF:

$$\begin{aligned} & (P_1 \vee P_2 \vee P_3) \wedge \\ & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\ & (P_1 \vee \neg P_5) \wedge \\ & (\neg P_2 \vee \neg P_3 \vee \neg P_5) \wedge \\ & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 1 Es gibt keine Einzelklausel, aber $\neg P_5$ ist ein reines Literal, d.h. wir brauchen für P_5 nur den Fall $\neg P_5 = \text{true}$ zu betrachten.

D.h. $\mathcal{M} = \{\neg P_5\}$ und $\varphi[\top/\neg P_5]$ enthält die Klauseln mit $\neg p_5$ nicht mehr, also

$$\begin{aligned} & (P_1 \vee P_2 \vee P_3) \wedge \\ & (P_1 \vee \neg P_2 \vee \neg P_3) \wedge \\ & (\neg P_1 \vee \neg P_2 \vee P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 2 Probiere $P_1 = \text{true}$, also $\mathcal{M} = \{\neg P_5, P_1\}$. Die Klauseln mit P_1 sind **true** und in den Klauseln mit $\neg P_1$ kann man $\neg P_1$ weglassen. Dann bleibt

$$\begin{aligned} & (\neg P_2 \vee P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 3 Probiere $P_2 = \text{true}$, also $\mathcal{M} = \{\neg P_5, P_1, P_2\}$. Die Klauseln mit P_2 sind **true** und in den Klauseln mit $\neg P_2$ kann man $\neg P_2$ weglassen. Dann bleibt

$$\begin{aligned} & (P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

(P_3) ist nun eine Einzelklausel, d.h. P_3 muss **true** sein, also $\mathcal{M} = \{\neg P_5, P_1, P_2, P_3\}$. Es bleibt:

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_4) \end{aligned}$$

$(\neg P_4)$ ist nun auch Einzelklausel, und es bleibt:

$$\begin{aligned} & (P_6) \wedge \\ & (\neg P_6) \end{aligned}$$

Dies ist aber der Widerspruch.

Schritt 4 Probiere $P_2 = \text{false}$, also $\mathcal{M} = \{\neg P_5, P_1, \neg P_2\}$. Dann bleibt

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

$(\neg P_4)$ ist jetzt Einzelklausel, d.h. P_4 muss **false** sein, d.h.

$$\begin{aligned} & (P_6) \wedge \\ & (\neg P_6) \wedge \\ & (\neg P_3) \end{aligned}$$

Erneut der Widerspruch.

Schritt 5 Probiere $P_1 = \text{false}$, also $\mathcal{M} = \{\neg P_5, \neg P_1\}$. Dann bleibt:

$$\begin{aligned} & (P_2 \vee P_3) \wedge \\ & (\neg P_2 \vee \neg P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (P_2 \vee \neg P_4) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

Schritt 5 Probiere $P_2 = \text{true}$, also $\mathcal{M} = \{\neg P_5, \neg P_1, P_2\}$. Dann bleibt:

$$\begin{aligned} & (\neg P_3) \wedge \\ & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \wedge \\ & (\neg P_3 \vee \neg P_4) \end{aligned}$$

$(\neg P_3)$ ist jetzt Einzelklausel, d.h. P_3 muss **false** sein, also $\mathcal{M} = \{\neg P_5, \neg P_1, P_2, \neg P_3\}$. Dann bleibt:

$$\begin{aligned} & (P_4 \vee P_6) \wedge \\ & (P_4 \vee \neg P_6) \end{aligned}$$

P_4 ist jetzt reines Literal, d.h. P_4 muss **true** sein, dann bleibt nichts mehr übrig, alle Klauseln sind erfüllt. D.h. die Wahl der Belegung für P_6 ist beliebig.

Ergebnis $\mathcal{M} = \{\neg P_5, \neg P_1, P_2, \neg P_3, P_4, P_6\}$ ist eine erfüllende Belegung.

9.3.3 Von DPLL zu CDLC

Der vorgestellte DPLL-Algorithmus gibt die Idee wieder, heutige SAT-Solver verwenden verbesserte Algorithmen mit folgenden Eigenschaften:

1. Oft wird keine Analyse bezüglich reiner Literale gemacht, weil diese Analyse aufwändig ist.
2. Stattdessen wird im Fall eines *Konflikts* genau analysiert, wodurch er entstanden ist und eine neue Klausel hinzugefügt, die dazu führt, dass der Algorithmus einmal gefundene Abhängigkeiten zwischen Atomen nicht „vergisst“ und deshalb erneut auswerten muss. Die Idee kann man an unserem Beispiel oben sehen. Wir haben in Schritt 1 P_1 als `true` und in Schritt 2 P_2 als `true` gewählt. Diese beiden Entscheidungen haben zum Widerspruch geführt. Also muss gelten: $\neg p_1 \vee \neg p_2$. Diese Klausel können wir nun unserer ursprünglichen Formel hinzufügen. Der Algorithmus hat aus dem Konflikt „gelernt“.
Klausel-lernende SAT-Solver werden auch CDLC-Solver (*Conflict Driven Clause Learning*) genannt.
3. Diese Analyse kann noch weitergehen: Man merkt sich in welchem Level der Analyse man eine Entscheidung getroffen hat und macht bei einem Konflikt nicht einfach *Backtracking*, sondern *Backjumping*.
4. Außerdem werden Heuristiken eingesetzt, welche Variable beim Backjumping „ausprobiert“ wird. Ein Beispiel wäre etwa: DLCS (*Dynamic Largest Combined Sum*) – man wählt die Variable, die positiv oder negativ am häufigsten in der Formel vorkommt.
Würde man diese Strategie in unserem Beispiel verwenden, müsste man in Schritt 2 mit $P_2 = \text{true}$ starten, danach P_3 mit `true` probieren, was zum Konflikt führt. Doch $P_3 = \text{false}$ führt zu einer erfüllenden Belegung. Wir wären also etwas schneller.

Wie aktuelle SAT-Solver arbeiten wird in [KS06] und [Knu15] beschrieben.

Kapitel 10

Anwendungen der Aussagenlogik in der Softwaretechnik

In der Programmierung spielt die Aussagenlogik eine prominenten Rolle, wenn immer Bedingungen zu formulieren sind, also etwa bei bedingten Anweisungen oder Fallunterscheidungen. In diesem Kapitel werfen wir einen kurzen Blick auf einige andere Anwendungen der Aussagenlogik und von SAT-Solvern in der Softwaretechnik.

10.1 Anwendungen von SAT-Techniken

Einige Beispiele für den Einsatz von SAT-Solvern, um nicht-triviale Fragestellungen zu lösen.

- 2003 – Validierung von Produktkonfigurationen in der Automobilindustrie, Carsten Sinz et al, siehe [SKK03].
- 2009 – Formale Verifizierung des Intel Core 7 Prozessors, Kaivola et al, siehe [KGN⁺09]
- 2010 – Verifikation von Windows 7 Gerätetreibern mit einem SMT-Solver, De Moura, Bjørner, siehe [dMB10]. SMT (SAT modulo Theory) baut auf dem Konzept von SAT-Solvern auf. Microsoft Research hat einen SMT-Solver namens Z3 entwickelt, siehe <https://github.com/Z3Prover/z3/wiki>.
- 2014 – Analyse der Terminierung von Programmen, Jürgen Giesl et al, siehe [GBE⁺14].
- 2016/18 – Management von Abhängigkeiten innerhalb des Ökosystems der Eclipse Plattform, Le Berre, Rapicault, siehe [BR18].
- ...

10.2 Statische Codeanalyse

Das folgende Beispiel ist aus [KS06, Example 2.2], von mir zu einer kleinen Geschichte ausgebaut.

Bei einem Codereview richtet sich das Auge des Reviewers auf folgendes Codestück:

```
if (!a && !b) h();
else if (!a) g();
else f();
```

Der Entwickler versichert, dass der Code getestet wurde und in alle Testfällen das gewünschte Ergebnis zur Folge hatte. Gleichwohl ist der Reviewer der Auffassung, dass dieses Codestück doch einfacher und damit für Menschen verständlicher geschrieben werden können müsste. Etwas unwillig erklärt sich der Entwickler dazu bereit und kommt am kommenden Tag wieder.

Er bringt zwei Fassungen mit:

Version 1:

```
if (a) h();
else if (b) g();
else f();
```

Version 2:

```
if (a) f();
else if (b) g();
else h();
```

und richtet milde lächelnd die Frage an den Reviewer: „Und welche der Versionen sollen wir nun nehmen?“

Der Reviewer freilich kennt seine Aussagenlogik und übersetzt die ursprüngliche Fassung sowie die beiden neuen Versionen indem er die Terme der Bedingungen sowie die Funktionen zu Aussagensymbolen macht:

Original:

```
if ( $\neg a \wedge \neg b$ ) then  $h$ 
else if ( $\neg a$ ) then  $g$ 
else  $f$ 
```

Version 1 (2 analog):

```
if  $a$  then  $h$ 
else if  $b$  then  $g$ 
else  $f$ 
```

Nun muss man nur noch **if-then-else** in eine Formel transformieren:

$$\text{if } x \text{ then } y \text{ else } z \rightsquigarrow (x \wedge y) \vee (\neg x \wedge z)$$

Und schon hat man drei Formeln:

$$\begin{aligned}\varphi_{orig} &\hat{=} ((\neg a \wedge \neg b) \wedge h) \vee (\neg(\neg a \wedge \neg b) \wedge (\neg a \wedge g)) \vee (a \wedge f)) \\ \varphi_{v_1} &\hat{=} (a \wedge h) \vee (\neg a \wedge ((b \wedge g) \vee (\neg b \wedge f))) \\ \varphi_{v_2} &\hat{=} (a \wedge f) \vee (\neg a \wedge ((b \wedge g) \vee (\neg b \wedge h)))\end{aligned}$$

Jetzt kann man einen SAT-Solver verwenden, um zu überprüfen, ob $\varphi_{orig} \leftrightarrow \varphi_{v_1}$ oder $\varphi_{orig} \leftrightarrow \varphi_{v_2}$ gilt.

Mit der Logic Workbench kann man sich die Sache noch einfacher machen, weil es in ihr den dreistelligen Operator `ite` gibt:

```
; Originaler Code
(def orig '(ite (and (not a) (not b)) h (ite (not a) g f)))
; Version 1
(def vers1 '(ite a h (ite b g f)))
; Version 2
(def vers2 '(ite a f (ite b g h)))

; Was ist richtig?
(valid? (list 'equiv orig vers1))
; => false
(valid? (list 'equiv orig vers2))
; => true
```

10.3 Featuremodelle für (Software-)Produktlinien

In der Softwareentwicklung kommt es nicht selten vor, dass verwandte Anwendungen entwickelt werden, die Gemeinsamkeiten, aber auch Unterschiede haben. Wenn man dies auf *systematische* Weise tut, dann spricht man von einer *Softwarereproduktlinie*.

Die systematische Analyse von Gemeinsamkeit und Variabilität innerhalb einer Softwarereproduktlinie wird mittels *Feature-Modellierung* durchgeführt. Die Eigenschaften der verschiedenen Produkte in einer Produktlinie werden als *Features* bezeichnet.

Die *Features* werden im Feature-Modell in einem Baum angeordnet, wobei verschiedene Arten von Kanten zwischen Super- und Subfeatures die Variabilität beschreiben. Außerdem gibt es Integritätsbedingungen, die über Zweige des Baums hinweggehen, sogenannte *Cross Tree Constraints* (CTCs). Der Feature-Baum zusammen mit den CTCs ergibt das Feature-Modell.

Eine Auswahl von Features, die alle Bedingungen im Feature-Modell erfüllt, wird als eine gültige *Konfiguration* bezeichnet. Können noch weitere Features gewählt werden, spricht man von einer *partiellen* Konfiguration, besteht keine Wahlmöglichkeit mehr von einer *vollständigen* Konfiguration.

Typischerweise werden in Feature-Modelle folgende vier Arten der Beziehungen zwischen Super- und Subfeatures unterschieden:

- ① Ein Subfeature ist *obligatorisch*, d.h. wenn das Superfeature in einer Konfiguration gewählt wird, dann ist das Subfeature automatisch auch gewählt.
- ② Ein Subfeature ist *optional*, d.h. wenn das Superfeature in einer Konfiguration gewählt wird, dann kann das Subfeature gewählt werden oder auch nicht.
- ③ Eine Gruppe von Subfeatures wird als *Oder-Gruppe* dargestellt, wenn die Wahl des Superfeatures erzwingt, dass mindestens eines der Subfeatures gewählt werden muss.
- ④ Eine Gruppe von Subfeatures wird als *Alternativ-Gruppe* dargestellt, wenn die Wahl des Superfeatures erzwingt, dass genau eines der Subfeatures gewählt werden muss.

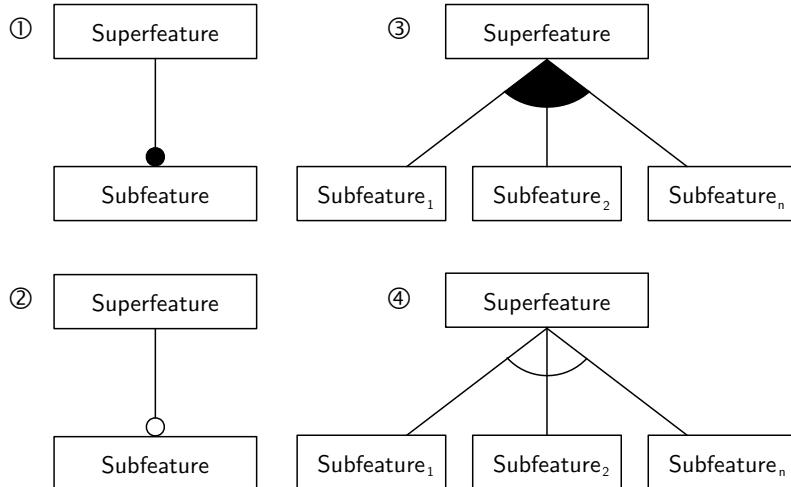


Abbildung 10.1: Beziehungen im Feature-Baum

Abbildung 10.1 stellt dar, wie diese vier Beziehungen im Feature-Diagramm graphisch dargestellt werden.

Die Integritätsbedingungen zwischen beliebigen Features unabhängig von der Baumstruktur, die CTCs, werden als aussagenlogische Formeln ausgedrückt, wobei die Features die Aussagensymbole sind. (Das setzt voraus, dass die Bezeichnungen der Features im Baum eindeutig sind.)

Die Featuremodellierung wurde als Bestandteil der *Feature Oriented Domain Analysis* (FODA) 1990 von Kyo C. Kang et al. beim Software Engineering Institute SEI eingeführt [KCH⁺90]. Seither wurden viele Varianten des Feature-Modells entwickelt, einen Überblick findet man

in [MH07]. Ich verwende die Notation, die in der FeatureIDE, einem Werkzeug der Feature-Modellierung, eingesetzt wird, siehe [ABKS13] und [MTS⁺17].

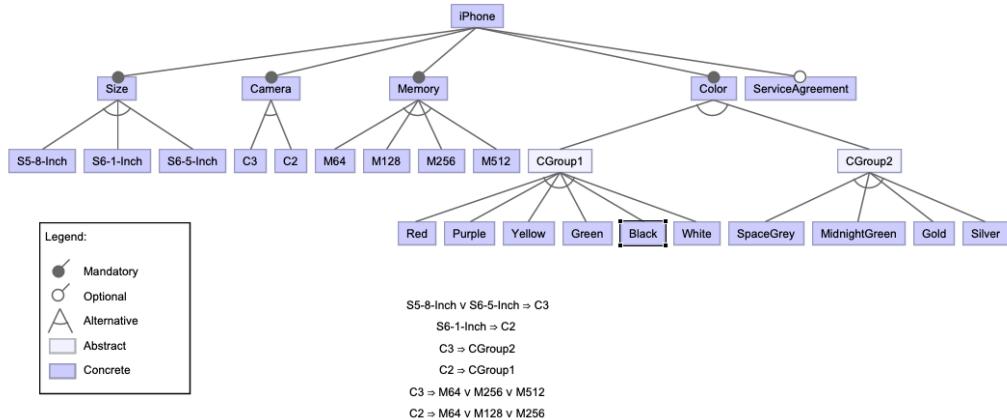


Abbildung 10.2: Feature-Modell für das iPhone

Abbildung 10.2 zeigt ein Feature-Modell für die Varianten des iPhones 2019. Das Modell wurde in FeatureIDE (siehe <https://featureide.github.io> und [MTS⁺17]) erstellt. In der Legende werden Features nach *abstract* und *concrete* unterschieden. Abstrakte Features dienen der Strukturierung des Feature-Baums, sie sind als solche nicht im Produkt konkret auffindbar. Die Aussagen unterhalb des Baums sind die CTCs.

Was nun hat das Feature-Modell mit der Aussagenlogik zu tun? Wir nehmen die Features als Aussagensymbole und verstehen zugeordnete Wahrheitswerte als Festlegungen, ob das Feature für die zu treffende Konfiguration gewählt wurde (also T ist) oder nicht (also F ist).

Das Feature-Modell entspricht auf diese Weise einer Formel der Aussagenlogik, die als Konjunktion folgender Teilformel gebildet wird:

- Jede mögliche Konfiguration soll ja tatsächlich ein Produkt definieren, dessen Varianten im Modell beschrieben werden. Das bedeutet, dass die Wurzel *Root* des Baums T sein muss, d.h. *Root* ist eine Teilformel:

$$Root$$

- Bei obligatorischen Subfeatures besteht eine logische Äquivalenz zum Superfeature: Ist das Superfeature *Super T*, dann muss auch das Subfeature *oblSub T* sein. Und wenn das Subfeature T ist, dann

ist das nur möglich, wenn das Superfeature gewählt ist, also

$$oblSub \leftrightarrow Super$$

- Beim optionalen Subfeature $optSub$ ist dieses bei Wahl des Super nicht zwingend erforderlich, also

$$optSub \rightarrow Super$$

- Bei einer Oder-Gruppe von Subfeatures $oSub_1, oSub_2, \dots, oSub_n$ gilt

$$oSub_1 \vee oSub_2 \vee \dots \vee oSub_n \leftrightarrow Super$$

- Bei einer Alternativ-Gruppe von Subfeatures $aSub_1, aSub_2, \dots, aSub_n$ muss exakt eines der Subfeatures der Wahl des Superfeatures entsprechen:

$$(aSub_1 \vee aSub_2 \vee \dots \vee aSub_n) \wedge \bigwedge_{i < j} (\neg aSub_i \vee \neg aSub_j) \leftrightarrow Super$$

- Cross Tree Constraints werden der Formel einfach hinzufügt.

Auf diese Weise (zuerst beschrieben in [Bat05]) entspricht ein Feature-Modell gerade einer Formel der Aussagenlogik und die erfüllenden Belegungen der Formel sind gerade die möglichen vollständigen Konfigurationen von Varianten.

Das bereits erwähnte Werkzeug FeatureIDE verwendet diese Korrespondenz und setzt den SAT-Solver SAT4J ein, um Feature-Modelle zu analysieren und Modellierer zu unterstützen.

Teil II

Prädikatenlogik

Kapitel 11

Objekte und Prädikate

Die Ausdruckskraft der Aussagenlogik ist beschränkt. Wir werden folgenden Schluss sicherlich für richtig halten:

Alle Quadratzahlen $\neq 0$ sind positiv (P)

16 ist eine Quadratzahl (Q)

Also folgt: 16 ist positiv (R)

Übertragen wir diesen Schluss etwas naiv in die Aussagenlogik, so ergibt sich $P \wedge Q \rightarrow R$ — und es gibt keinerlei Grund anzunehmen, dass diese Aussage zutrifft.

Dies liegt daran, dass wir in der Aussagenlogik den Zusammenhang zwischen der ersten und der zweiten Aussage nicht erfassen können, nämlich, dass die 16 ein Exemplar von der Sorte *Quadratzahl* ist.

Also: Wir müssen unterscheiden können zwischen

Objekten, den Dingen eines Universums, einer „Welt“, wie z.B. Zahlen, Strings, Werten, Objekten usw. und

Prädikaten, Aussagen über die Objekte, die wahr oder falsch sein können.

Beispiel Ist etwa unser Universum die Welt der ganzen Zahlen \mathbb{Z} , dann könnten wir folgende Prädikate haben

$Sq(x)$ bedeutet „ x ist eine Quadratzahl $\neq 0$ “

$Pos(x)$ bedeutet „ x ist positiv“

Dann können wir das einleitende Beispiel so ausdrücken

$$\forall x(Sq(x) \rightarrow Pos(x)) \wedge Sq(16) \rightarrow Pos(16)$$

11.1 Elemente der Sprache der Prädikatenlogik

Gegeben sei stets eine Menge \mathbb{U} , das Universum, auch genannt „Miniwelt“. In der Regel wird in der Literatur vorausgesetzt, dass das Universum nicht leer ist, d.h. $\mathbb{U} \neq \emptyset$ ¹

Wir verwenden zusätzlich zu den Junktoren der Aussagenlogik

Variablen

$x, y, z \dots$

Variablen sind Platzhalter für beliebige Objekte des Universums, z.B. steht in $Sq(x)$ die Variable x für ein Element des Universums, also in unserem Beispiel für eine Zahl in \mathbb{Z} .

Konstanten

$c, d, e \dots$

Konstanten sind bestimmte, benannte Elemente des Universums, z.B. 16, die Zahl $16 \in \mathbb{Z}$.

Funktionen

$f, g, h \dots$

Funktionen operieren auf den Elementen des Universums und ergeben wieder Elemente des Universums, z.B. $plus(16, 9)$ ergibt das Element $25 \in \mathbb{Z}$ mit der naheliegenden Definition von $plus$.

Prädikate

$P, Q, R \dots$

Prädikate sind Boolesche Funktionen, die Aussagen über Elemente der Welt machen. Die Wertemenge eines Prädikats ist also in $\mathbb{B} = \{\text{T}, \text{F}\}$. In unserem Beispiel ist Sq ein Prädikat der Arität 1, $Sq(x)$ ist genau dann wahr, wenn $x \in \mathbb{Z}$ eine Quadratzahl ist.

Gleichheit

$=$

Gleichheit ist ein spezielles Prädikat, das wir in die Sprache von vorneherein aufnehmen.²

¹Ist das Universum leer, dann ist jede Aussage der Form $\exists x \dots$ immer falsch und eine der Form $\forall x \dots$ immer wahr – diese Fälle möchte man gerne vermeiden.

²Man kann auch Prädikatenlogik ohne Gleichheit machen, für Anwendungen ist es jedoch nützlich, die Gleichheit als *logisches* Symbol aufzufassen.

Quantoren

$\forall x\varphi, \exists y\varphi$

Quantoren machen Aussagen über *alle* Elemente des Universums oder darüber, ob ein Element des Universums mit einer bestimmten Eigenschaft *existiert*.

11.2 Prädikate und Relationen

Es besteht eine wichtige grundlegende Beziehung zwischen *Prädikaten und Relationen*:

Sei R eine n -wertige Relation über \mathbb{U} , d.h. $R \subset \mathbb{U}^n$. Dann kann man diese Relation repräsentieren durch die Boolesche Funktion $r : \mathbb{U}^n \rightarrow \mathbb{B}$ definiert durch

$$r(a_1, \dots, a_n) = \text{T} \text{ genau dann, wenn } (a_1, \dots, a_n) \in R$$

Ist etwa $Sq \subset \mathbb{Z} = \{x / \exists y \text{ mit } x = y^2 \text{ und } x > 0\}$, also

$$Sq = \{1, 4, 9, 16, 25, \dots\}$$

dann entspricht diese einstellige Relation gerade dem oben verwendeten Prädikat Sq .

Offenbar sind also Prädikate und Relationen austauschbare Konzepte, deshalb spricht man bei der Prädikatenlogik erster Ordnung auch manchmal von *relationaler Logik*.

Bemerkung Man könnte in unserem Beispiel auch eine binäre Relation $Sq \subset \mathbb{N}^2$ auf folgende Weise definieren:

Sei $square : \mathbb{N} \rightarrow \mathbb{N}$ die Funktion, die einem $x \in \mathbb{N}$ sein Quadrat als Funktionswert zuordnet, also $x \mapsto x^2$. Diese Funktion kann man als binäre Relation auffassen, nämlich $Sq = \{(x, x^2) / x \in \mathbb{N}\}$. Ein Tupel (x, y) ist also genau dann in Sq , wenn y das Quadrat von x ist. Auf diese Weise entspricht jeder n -stellige Funktion eine $n + 1$ -stellige Relation.

Kapitel 12

Die formale Sprache der Prädikatenlogik

12.1 Signatur, Terme, Formeln

In der Sprache der Prädikatenlogik verwendet man Funktions- und Konstantensymbole sowie Prädikatssymbole. Genau genommen bezieht man sich auf eine bestimmte Wahl dieser Symbole und spricht dann von *einer Sprache* \mathcal{L} der Prädikatenlogik.

Definition 12.1 (Signatur). Die *Signatur* einer Sprache \mathcal{L} besteht aus einer Menge von Prädikatssymbolen $\{P_1, \dots, P_n\}$, von Funktionssymbolen $\{f_1, \dots, f_m\}$ und von Konstantensymbolen $\{c_1, \dots, c_k\}$.

Man schreibt die Signatur so:

$$\{P_1^{r_1}, \dots, P_n^{r_n}; f_1^{a_1}, \dots, f_m^{a_m}; c_1, \dots, c_k\}$$

wobei die r und die a die Arität der Prädikatssymbole bzw. Funktionssymbole bezeichnet.

Bemerkungen

- Die Signatur besteht aus den „nicht-logischen“ Symbolen der Sprache \mathcal{L} .
- Manche Autoren definieren die Signatur dadurch, dass sie nur die Arität von Prädikatssymbolen, Funktionssymbolen und die Zahl der Konstanten angeben. Denn „Name ist Schall und Rauch“¹. In dieser Sicht wird die Signatur so angegeben:

$$< r_1, \dots, r_n; a_1, \dots, a_m; k >$$

¹J.W. von Goethe, Faust I, Marthens Garten

wobei r_i die Arität eines Prädikatensymbols, a_i die Arität eines Funktionssymbols und k die Zahl der Konstanten ist.

- Man kann Konstantensymbole auch als Funktionssymbole der Arität 0 auffassen.
- Prädikatssymbole der Arität 0 kann man als Aussagensymbole auffassen. Oder anders: Wenn in der Signatur nur Prädikatssymbole der Arität 0 vorkommen und keine anderen Symbole und wir uns auf die Junktoren $\neg, \wedge, \vee, \rightarrow$ beschränken, dann erhalten wir gerade die Definition einer Sprache der Aussagenlogik, wie in Teil I.

Im Folgenden sei eine Signatur gegeben, sowie eine Menge von Variablen $\{x, y, \dots\}$.

Definition 12.2 (Terme). Die *Terme* sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jede Variable ist ein Term.
- (ii) Jedes Konstantensymbol ist ein Term.
- (iii) Sind t_1, \dots, t_n Terme und f ein Funktionssymbol mit der Arität n , dann ist auch $f(t_1, \dots, t_n)$ ein Term.

Als *Grammatik* in Backus-Naur-Darstellung können wir diese Regeln so ausdrücken:

$$t ::= x \mid c \mid f \mid f(t, \dots, t)$$

mit Variablen x , Konstantensymbolen c und Funktionssymbolen f .

Definition 12.3 (Formeln). Die *Formeln* sind die Zeichenketten, die durch folgende Regeln generiert werden können:

- (i) \perp ist eine Formel (genannt: der Widerspruch) und \top ist eine Formel (genannt: die Wahrheit).
- (ii) Ist P ein Prädikatssymbol der Arität 0, dann ist P eine Formel.
Ist P ein Prädikatssymbol der Arität $n \geq 1$ und sind t_1, \dots, t_n Terme, dann ist $P(t_1, \dots, t_n)$ eine Formel.
Sind t_1 und t_2 Terme, dann ist $(t_1 = t_2)$ eine Formel.
- (iii) Ist φ eine Formel, dann auch $(\neg\varphi)$.
Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$.
- (iv) Ist φ eine Formel und x eine Variable, dann sind $\forall x \varphi$ und $\exists x \varphi$ Formeln.

In Backus-Naur-Darstellung:

$$\begin{aligned}\varphi ::= & \perp \mid \top \mid P(t, \dots, t) \mid (t = t) \mid \\ & (\neg\varphi), \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & \forall x \varphi \mid \exists x \varphi\end{aligned}$$

mit Termen t , Variablen x und Prädikatssymbolen P .

Bemerkungen

- Die Argumente von Prädikatssymbolen dürfen *nur* Terme sein, keine anderen Prädikate – wir definieren hier nämlich die Prädikatenlogik *erster* Stufe.
- In unserer Definition der Sprachen der Prädikatenlogik, haben wir die Gleichheit $=$ als *logisches* Symbol mit aufgenommen. In vielen Büchern wird unterschieden zwischen der Prädikatenlogik und der Prädikatenlogik mit Gleichheit.

Man könnte denken, dass man die Gleichheit einfach dadurch einführen kann, dass man ein Prädikatsymbol für die Gleichheit definiert und dabei verlangt, dass in jedem Modell der Sprache dieses gerade die Identitätsrelation über dem Universum des Modells ist.
(Vorgriff – siehe Abschnitt 13.1) [Hof11, S.114f]

Bindungsregeln

$\neg, \forall x, \exists x$ binden am stärksten, dann
 \wedge und \vee (linksassoziativ) und schließlich
 \rightarrow (rechtsassoziativ.)

Diese Bindungsregeln erlauben uns, sparsamer (und dadurch besser lesbar) mit Klammern umzugehen, ohne die Grammatik mehrdeutig zu machen.

12.2 Freie und gebundene Variablen

Sei φ eine Formel der Prädikatenlogik. Ein Vorkommen der Variablen x in φ heißt *frei*, wenn x im Syntaxbaum nicht Abkömmling eines Quantorknotens $\forall x$ oder $\exists x$ ist. Andernfalls heißt das Vorkommen *gebunden*.

Die Namen von gebundenen Variablen spielen keine Rolle, deshalb werden Formeln, die sich nur durch den Namen gebundener Variablen unterscheiden, identifiziert. $\forall x P(x, y)$ ist also dieselbe Formel wie $\forall z P(z, y)$, nicht jedoch $\forall x P(x, z)$, da die Bedeutung der freien Variablen vom Kontext abhängt.

Definition 12.4 (Satz). Eine Formel φ , die keine freien Variablen hat, heißt *Satz* oder *geschlossene Formel*.

Definition 12.5 (Grundterm). Ein Term t , der keine Variablen hat, heißt *geschlossener Term* oder *Grundterm*.

Bemerkung

Hat eine Formel freie Variablen, dann macht man sie zu einem Satz, in dem man sich alle freien Variablen durch den Allquantor gebunden denkt.

12.3 Substitution

Variablen sind Platzhalter für Terme. Substitution besteht darin, solche Platzhalter durch Terme zu ersetzen.

Ein Ersetzen von Variablen durch Terme ist nur möglich für Variablen, die *nicht* durch einen Quantor gebunden sind. Gebundene Variablen stehen für ein bestimmtes Element des Universums oder für alle Elemente des Universums, können somit nicht durch den Term ersetzt werden.

Definition 12.6 (Substitution). Gegeben eine Variable x und ein Term t . Die Formel $\varphi[t/x]$ ist die Formel, die aus φ entsteht, in dem jedes *freie* Vorkommen von x durch t ersetzt wird.

Dabei darf der eingesetzte Term t keine Variablen enthalten, die durch die Substitution in den Geltungsbereich eines Quantors kommen würden.

Die einschränkende Bemerkung zur Substitution sei an einem Beispiel erläutert:

Haben wir die Formel $\forall xP(x, y)$ und wollen y durch den Term $f(x)$ ersetzen, dann ergäbe sich durch einfaches Einsetzen $\forall xP(x, f(x))$ und plötzlich ist die Variable x im Term $f(x)$ in den Bereich des Allquantors gekommen. Dies ist nicht erlaubt.

Richtig wäre es, zunächst die gebundene Variable umzubenennen, also etwa $\forall zP(z, y)$, wodurch sich die Formeln nicht ändert. Jetzt ist die Substitution möglich:

$(\forall xP(x, y))[f(x)/y]$ ergibt $\forall zP(z, f(x))$, nicht aber $\forall xP(x, f(x))$.

Man sagt, dass eine Term t frei ist für eine Variable x in einer Formel φ , wenn t keine Variable enthält, die beim Einsetzen für x in den Geltungsbereich eines Quantors käme.

Bei einer Substitution muss man also erst prüfen, ob der zu substituierende Term frei für die Variable in der Formel ist. Ist dies nicht der Fall, muss man gebundene Variablen in der Formel so umtaufen, dass keine „Kollision“ auftritt.

Bemerkung In der Logic Workbench (lwb) hat man als Junktoren die Junktoren aus der Aussagenlogik, siehe Tabelle 3.1, darüber hinaus:

- das ausgezeichnete Symbol $=$ für die Gleichheit,
- den Allquantor, z.B. `(forall [x y] (and (P x) (Q y)))` mit unären Prädikaten P und Q sowie
- den Existenzquantor, z.B. `(exists [x] (= (f x) (g x)))` mit den unären Funktion f und g .

Kapitel 13

Semantik der Prädikatenlogik

13.1 Modell/Struktur

Definition 13.1 (Modell/Struktur). Ein *Modell* für die Sprache \mathcal{L} ist ein Paar $\mathcal{M} = \langle \mathbb{U}, I \rangle$ mit einer nicht-leeren Menge \mathbb{U} und einer Funktion I , die jedem Symbol in \mathcal{L} eine Interpretation zuordnet nach folgenden Regeln:

- (i) Ist P ein 0-äres Prädikatensymbol, dann ist $I(P)$ ein Wahrheitswert.
- (ii) Ist P ein n -äres Prädikatensymbol für $n > 0$, dann ist $I(P) \subseteq \mathbb{U}^n$ eine n -äre Relation über \mathbb{U} .
- (iii) Ist c ein Konstantensymbol, dann ist $I(c) \in \mathbb{U}$, ein Element von \mathbb{U} .
- (iv) Ist f ein Funktionssymbol mit Arität n , dann ist $I(f) : \mathbb{U}^n \rightarrow \mathbb{U}$ eine Funktion.

Man nennt Modelle der Prädikatenlogik auch spezifischer *Strukturen*.

Die Menge \mathbb{U} nennt man auch das Universum. Die Interpretation schreibt man auch oft so:

- $P^{\mathcal{M}}$ für die Prädikate über \mathbb{U}
- $c^{\mathcal{M}}$ für die Elemente zu den Konstantensymbolen und
- $f^{\mathcal{M}}$ für die Funktionen zu den Funktionssymbolen

Definition 13.2 (Variablenbelegung). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} . Eine *Variablenbelegung* in \mathcal{M} ist eine Funktion l , die jeder Variablen x einen Wert $l(x) \in \mathbb{U}$ zuordnet.

Man schreibt $l[x \mapsto a]$ für die Variablenbelegung, die x auf a abbildet und alle anderen Variablen auf $l(y)$, d.h.

$$l[x \mapsto a](y) = \begin{cases} a & \text{falls } y = x. \\ l(y) & \text{falls } y \neq x. \end{cases}$$

Definition 13.3 (Interpretation der Terme). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} und l eine Variablenbelegung. Für einen Term t von \mathcal{L} definiert man die Interpretation $I(t)$ bezüglich \mathcal{M} und der Variablenbelegung l induktiv über die Länge des Terms durch

- (i) $I(x) := l(x)$ für die Variablen x ,
- (ii) $I(c) := I(c)$ für die Konstanten c , und
- (iii) $I(f(t_1, \dots, t_n)) := I(f)(I(t_1), \dots, I(t_n))$ für die Funktionen f .

Definition 13.4 (Interpretation der Formeln). Sei $\mathcal{M} = \langle \mathbb{U}, I \rangle$ eine Struktur für \mathcal{L} und l eine Variablenbelegung, dann ist \mathcal{M} ein Modell für eine Formel φ , geschrieben

$$\mathcal{M} \models_l \varphi \quad \text{für die Formel } \varphi,$$

falls $\llbracket \varphi \rrbracket_l^{\mathcal{M}} = \top$. Dabei wird $\llbracket \varphi \rrbracket_l^{\mathcal{M}}$ induktiv definiert über den strukturellen Aufbau von φ :

(i)	$\llbracket P(t_1, \dots, t_n) \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } (I(t_1), \dots, I(t_n)) \in P^{\mathcal{M}} \\ \text{F sonst} \\ \text{bzw. } I(P) \text{ falls } P\text{-är} \end{cases}$
(ii)	$\llbracket s = t \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } I(s) = I(t) \\ \text{F sonst} \end{cases}$
(iii)	$\llbracket \neg \varphi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{F} \\ \text{F sonst} \end{cases}$
(iv)	$\llbracket \varphi \wedge \psi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{T} \text{ und } \llbracket \psi \rrbracket_l^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
(v)	$\llbracket \varphi \vee \psi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{T} \text{ oder } \llbracket \psi \rrbracket_l^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
(vi)	$\llbracket \varphi \rightarrow \psi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls } \llbracket \varphi \rrbracket_l^{\mathcal{M}} = \text{F} \text{ oder } \llbracket \psi \rrbracket_l^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
(vii)	$\llbracket \forall x \varphi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls für alle } a \in \mathbb{U} \text{ gilt: } \llbracket \varphi \rrbracket_{l[x \mapsto a]}^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
(viii)	$\llbracket \exists x \varphi \rrbracket_l^{\mathcal{M}}$	$:= \begin{cases} \text{T falls es existiert ein } a \in \mathbb{U} \text{ mit: } \llbracket \varphi \rrbracket_{l[x \mapsto a]}^{\mathcal{M}} = \text{T} \\ \text{F sonst} \end{cases}$
(ix)	$\llbracket \top \rrbracket_l^{\mathcal{M}}$	$:= \text{T}$
(x)	$\llbracket \perp \rrbracket_l^{\mathcal{M}}$	$:= \text{F}$

13.2 Semantische Folgerung und Äquivalenz

Definition 13.5 (Semantische Folgerung). Sei Γ eine Menge prädikatenlogischer Formeln und φ eine prädikatenlogische Formel. Man sagt:

$\Gamma \models \varphi$ d.h. φ folgt semantisch aus Γ , genau dann wenn jedes Modell für Γ auch ein Modell für φ ist.

Besteht Γ nur aus einer Formel, sage ψ , dann schreibt man auch $\psi \models \varphi$.

Definition 13.6 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind semantisch äquivalent, geschrieben $\varphi \equiv \psi$, genau dann, wenn $\varphi \models \psi$ und $\psi \models \varphi$ gilt.

In der Prädikatenlogik können wir nun dieselben Definitionen wie in der Aussagenlogik machen:

Definition 13.7 (Erfüllbarkeit). Eine prädikatenlogische Formel φ heißt *erfüllbar*, wenn sie ein Modell hat.

Definition 13.8 (Falsifizierbarkeit). Eine prädikatenlogische Formel φ heißt *falsifizierbar*, wenn es ein Modell \mathcal{M} gibt mit $\mathcal{M} \not\models \varphi$.

Definition 13.9 (Allgemeingültigkeit). Eine prädikatenlogische Formel φ heißt *allgemeingültig*, wenn sie in jedem Modell wahr ist.

Man schreibt dann $\models \varphi$ und nennt φ eine *Tautologie*.

Definition 13.10 (Unerfüllbarkeit). Eine prädikatenlogische Formel φ heißt *unerfüllbar*, wenn es kein Modell für sie gibt.

Man schreibt dann $\not\models \varphi$ und nennt φ eine *Kontradiktion*.

Auch in der Prädikatenlogik gilt das Dualitätsprinzip:

Satz 13.1 (Dualitätsprinzip). Eine prädikatenlogische Formel φ ist genau dann allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist.

13.3 Fundamentale Äquivalenzen der Prädikatenlogik

Satz 13.2. φ , ψ und χ seien Formeln der Prädikatenlogik, wobei in χ die Variable x nicht vorkommt.

Es gelten folgende (semantische) Äquivalenzen:

$$\begin{aligned}
 \neg(\forall x \varphi) &\equiv \exists x (\neg\varphi) \\
 \neg(\exists x \varphi) &\equiv \forall x (\neg\varphi) \\
 \forall x \varphi \wedge \forall x \psi &\equiv \forall x (\varphi \wedge \psi) \\
 \exists x \varphi \vee \exists x \psi &\equiv \exists x (\varphi \vee \psi) \\
 \forall x (\forall y \varphi) &\equiv \forall y (\forall x \varphi) \\
 \exists x (\exists y \varphi) &\equiv \exists y (\exists x \varphi) \\
 (\forall x \varphi) \wedge \chi &\equiv \forall x (\varphi \wedge \chi) \\
 (\forall x \varphi) \vee \chi &\equiv \forall x (\varphi \vee \chi) \\
 (\exists x \varphi) \wedge \chi &\equiv \exists x (\varphi \wedge \chi) \\
 (\exists x \varphi) \vee \chi &\equiv \exists x (\varphi \vee \chi)
 \end{aligned}$$

Kapitel 14

Natürliches Schließen in der Prädikatenlogik

In Kapitel 5 wurde das Beweissystems des natürlichen Schließens nach Gerhard Gentzen für die Aussagenlogik eingeführt. Es beruht auf Regeln, wie Formeln (syntaktisch) umgeformt werden dürfen, um aus gegebenen Aussagen Schlussfolgerungen herzuleiten.

Ein wichtiges Ergebnis war dabei die Vollständigkeit des natürlichen Schließens als Beweissystem für die Aussagenlogik, d.h. dass für eine Menge Γ von gegebenen Formeln und eine Formel φ der Aussagenlogik gilt:

$$\Gamma \vdash \varphi \Leftrightarrow \Gamma \vDash \varphi$$

Die „syntaktische Sicht“ und die „semantische Sicht“ sind also äquivalent und wir können zwischen ihnen wechseln, je nachdem welche für eine konkrete Fragestellung besser geeignet ist.

Das Beweissystem des natürlichen Schließens hat Gerhard Gentzen für die Prädikatenlogik (mit Gleichheit) eingeführt. In diesem Kapitel werden die Regeln für die Quantoren und die Gleichheit vorgestellt.

Der Satz über die Vollständigkeit des natürlichen Schließens gilt auch für die Prädikatenlogik. Dies wurde zuerst von Kurt Gödel¹ in seiner Dissertation 1929 für das formale System gezeigt, das Bertrand Russell² und Alfred North Whitehead³ in ihrem Buch *Principia Mathematica* verwendet haben und das dieselbe Ausdrucks Kraft wie die Prädikatenlogik hat.⁴

¹Kurt Gödel (1906–1978), österreichisch-amerikanischer Logiker.

²Bertrand Russell (1872–1970), britischer Philosoph, Mathematiker und Logiker.

³Alfred North Whitehead (1861–1947), britischer Philosoph und Mathematiker.

⁴„Der Hauptgegenstand der folgenden Untersuchungen ist der Beweis der Vollständigkeit des in Russell, *Principia mathematica*, P. I, Nr. 1 und Nr. 10, und ähnlich in Hilbert-Ackermann, *Grundzüge der theoretischen Logik* (zitiert als H. A.), Ill, § 5,

Gerhard Gentzen hat ihn für sein Beweissystem des Sequenzenkalküls bewiesen. Man muss den Vollständigkeitssatz so begreifen, dass er nicht für ein spezielles Beweissystem gilt, sondern eine Eigenschaft der Logik selbst ist, vorausgesetzt natürlich, dass das Beweissystem „vernünftig“ definiert ist.

14.1 Schlussregeln

14.1.1 Allquantor

	<i>Einführung</i>	<i>Elimination</i>
\forall	$\frac{x_0 \quad \vdots \quad \varphi[x_0/x]}{\forall x\varphi}$ $\forall x i$	$\frac{\forall x\varphi}{\varphi[t/x]} \quad \forall x e$

Um den Allquantor einzuführen, hat man folgende Beweisverpflichtung: Gegeben sei ein beliebiges Objekt x_0 des Universums. Man muss dann zeigen, dass die Formel φ mit x_0 an Stelle der Variablen x gilt (dies schreibt man kurz als $\varphi[x_0/x]$). Dabei darf in dieser Herleitung keinerlei *spezielle* Eigenschaft von x_0 vorkommen, denn x_0 steht ja für ein *beliebiges* Objekt des Universums. Man sagt auch, dass x_0 ein *frisches* beliebiges Objekt ist, sein Name darf somit nicht außerhalb der Box vorkommen.

Die Entfernung des Allquantors ist ein naheliegender Schritt: Wenn φ für alle x gilt, dann kann man ein beliebiges konkretes t des Universums an Stelle von x in die Formel φ einsetzen.

14.1.2 Existenzquantor

	<i>Einführung</i>	<i>Elimination</i>
\exists	$\frac{\varphi[t/x]}{\exists x\varphi} \quad \exists x i$	$\frac{\exists x\varphi \quad \begin{array}{ c c }\hline x_0 & \varphi[x_0/x] \\ \vdots & \\ \chi & \end{array}}{\chi} \quad \exists x e$

Den Existenzquantor kann man einführen, indem man einen *Zeugen* vorweist: Gilt φ mit t an Stelle von x , dann gibt es offenbar ein x für das φ gilt, nämlich eben t .

angegebenen Axiomensystems des sogenannten engeren Funktionenkalküls.“ [Gö29]

Will man den Existenzquantor entfernen, muss man ein Objekt x_0 nehmen, das φ an Stelle von x erfüllt. Solch ein Objekt existiert, weil ja $\exists x \varphi$ gilt. Nun hat man die Beweisverpflichtung zu zeigen, dass daraus χ herleitbar ist. In dieser Herleitung darf man keine spezielle Aussage über x_0 verwenden, außer $\varphi[x_0/x]$.

14.1.3 Gleichheit

	<i>Einführung</i>	<i>Elimination</i>
=	$t = t$ = i, ID	$t_1 = t_2$ $\varphi[t_1/x]$ $\varphi[t_2/x]$ = e, SUB

Die Regel ID besagt, dass ein Symbol, das für ein Objekt steht, dieses eindeutig bestimmt. Dies ist gewissermaßen die Charakteristik der Gleichheit.

Die Entfernung der Gleichheit besteht darin, dass wenn t_1 und t_2 gleich sind, man in einer Formel φ t_1 durch t_2 ersetzen kann. Dies klingt wie selbstverständlich, muss aber mit Vorsicht gehandhabt werden. Es sind nur gültige Substitutionen erlaubt: In allen Substitutionen $\varphi[t/x]$ muss t frei für x in der Formel φ sein, d.h. keine freie Variable y in t gelangt durch das Einsetzen von x in φ in den Bereich eines Quantors $\forall y$ oder $\exists y$.

14.2 Beispiele

Gentzen zeigt in [Gen35, S. 183] an drei Beispielen, wie das natürliche Schließen geht. Das erste Beispiel für eine Formel der Aussagenlogik wurde in Abschnitt 5.2 verwendet.

Für die Prädikatenlogik verwendet Gentzen die beiden folgenden Beispiele:

Beispiel: Vertauschen von Quantoren

Herleitung für

$$\exists x \forall y F(x, y) \rightarrow \forall y \exists x F(x, y)$$

1.	$\exists x \forall y F(x, y)$	angenommen
2.	a	angenommen
3.	$\forall y F(a, y)$	angenommen
4.	b	beliebig
5.	$F(a, b)$	$\forall e 3, 4$
6.	$\exists x F(x, b)$	$\exists i 2, 5$
7.	$\forall y \exists x F(x, y)$	$\forall i 4-6$
8.	$\forall y \exists x F(x, y)$	$\exists e 1, 2-7$
9.	$\exists x \forall y F(x, y) \rightarrow \forall y \exists x F(x, y)$	$\rightarrow i 1-8$

Negation und Quantoren

Als weiteres Beispiel folgt ein Beweis für

$$\neg \exists x G(x) \rightarrow \forall y \neg G(y)$$

Der Beweis folgt wieder der Argumentation von Gentzen in [Gen35, S. 183]:

1.	$\neg \exists x G(x)$	angenommen
2.	a	beliebig
3.	$G(a)$	angenommen
4.	$\exists x G(x)$	$\exists i 2, 3$
5.	\perp	$\neg e 1, 4$
6.	$\neg G(a)$	$\neg i 3-5$
7.	$\forall y \neg G(y)$	$\forall i 2-6$
8.	$\neg \exists x G(x) \rightarrow \forall y \neg G(y)$	$\rightarrow i 1-7$

Die verallgemeinerten Gesetze von De Morgan

In diesem Abschnitt verwenden wir das natürliche Schließen in der Logic WorkBench lwb, um die verallgemeinerten Gesetze von De Morgan herzuleiten.

Es geht um folgende Biimplikationen:

$$\begin{aligned} \vdash \neg \forall x \varphi &\leftrightarrow \exists x \neg \varphi \\ \vdash \neg \exists x \varphi &\leftrightarrow \forall x \neg \varphi \end{aligned}$$

Da die Regeln für das natürliche Schließen nur eine Regel für die Einführung bzw. Auflösung der Implikation, nicht jedoch der Biimplikation enthalten, müssen wir je Gesetz beide Richtungen herleiten.

Tatsächlich beweisen wir in lwb die Gesetze für unäre Prädikate. Da die Bezeichnung der Variablen und der Prädikate in der Herleitung aber keine Rolle spielt, kann man die Gesetze in lwb für beliebige Variablen und Prädikate anwenden (siehe auch [HR04, Section 2.3]).

- $\vdash \neg \forall x \varphi \rightarrow \exists x \neg \varphi$

Die Herleitung hat in lwb die folgenden Schritte:

```

1 (proof '(not (forall [x] (P x))) '(exists [x] (not (P x))))
2 (step-b :raa 3)
3 (step-b :not-e 4 1)
4 (step-b :forall-i 4)
5 (swap '?1 :i)
6 (step-b :raa 5)
7 (step-b :not-e 6 2)
8 (step-b :exists-i 6 3)

```

In Schritt 1 wird die Beweisverpflichtung formuliert.

```
(proof '(not (forall [x] (P x))) '(exists [x] (not (P x))))
-----
1: (not (forall [x] (P x))) :premise
2: ...
3: (exists [x] (not (P x)))
-----
```

Wir beginnen mit Schritt 2 einen Widerspruchsbeweis mit der Regel RAA, die in lwb als :raa geschrieben wird und rückwärts (**step-b**) auf Zeile 3 angewandt wird.

```
(step-b :raa 3)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
3: | ...
4: | contradiction
-----
5: (exists [x] (not (P x))) :raa [[2 4]]
-----
```

Der dritte Schritt besteht darin, dass wir mit Zeile 4 und 1 die Regel für die Auflösung von \neg anwenden, was uns in folgende Situation bringt:

```
(step-b :not-e 4 1)
-----
```

```

1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
3: | ...
4: | (forall [x] (P x))
5: | contradiction :not-e [1 4]
-----
6: (exists [x] (not (P x))) :raa [[2 5]]
-----
```

Die Beweisverpflichtung besteht nun darin, den Allquantor in Zeile 4 einzuführen, dies bereiten wir im 4. Schritt vor:

```

(step-b :forall-i 4)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| -----
3: | | (actual ?1) :assumption
4: | | ...
5: | | (P ?1)
| -----
6: | (forall [x] (P x)) :forall-i [[3 5]]
7: | contradiction :not-e [1 6]
-----
8: (exists [x] (not (P x))) :raa [[2 7]]
-----
```

Für die Einführung des Allquantors brauchen wir ein beliebiges Objekt des Universums. In Zeile 3 wurde uns ein solches beliebiges Objekt bereitgestellt, es hat noch keinen Namen, sondern wird markiert mit ?1. In Schritt 5 ersetzen wir ?1 durch einen Namen für das Objekt des Universums, wir nennen es :i.

```

swap '?1 :i)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| -----
3: | | (actual :i) :assumption
4: | | ...
5: | | (P :i)
| -----
6: | (forall [x] (P x)) :forall-i [[3 5]]
7: | contradiction :not-e [1 6]
-----
8: (exists [x] (not (P x))) :raa [[2 7]]
-----
```

Also müssen wir nun für das beliebige $:i$ zeigen, dass $(P :i)$ gilt. Und noch einmal die Regel für den Widerspruchsbeweis im 6. Schritt:

```
(step-b :raa 5)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| -----
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | ...
6: | | | contradiction
| |
7: | | (P :i) :raa [[4 6]]
| -----
8: | (forall [x] (P x)) :forall-i [[3 7]]
9: | contradiction :not-e [1 8]
-----
10: (exists [x] (not (P x))) :raa [[2 9]]
-----
```

Wir kommen dem Ziel näher, denn in Zeile 2 der Herleitung haben wir ja die Aussage, dass kein Objekt x existiert für das $\neg P(x)$ gilt.
Also Schritt 7:

```
(step-b :not-e 6 2)
-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| -----
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | ...
6: | | | (exists [x] (not (P x))) :not-e [2 6]
7: | | | contradiction :not-e [2 6]
| |
8: | | (P :i) :raa [[4 7]]
| -----
9: | (forall [x] (P x)) :forall-i [[3 8]]
10: | contradiction :not-e [1 9]
-----
11: (exists [x] (not (P x))) :raa [[2 10]]
-----
```

Nun bleibt nur noch in Schritt 8 die Einführung des Existenzquants durch die entsprechende Regel abzusichern:

```

-----
1: (not (forall [x] (P x))) :premise
-----
2: | (not (exists [x] (not (P x)))) :assumption
| -----
3: | | (actual :i) :assumption
| |
4: | | | (not (P :i)) :assumption
5: | | | (exists [x] (not (P x))) :exists-i [3 4]
6: | | | contradiction :not-e [2 5]
| |
7: | | (P :i) :raa [[4 6]]
| -----
8: | (forall [x] (P x)) :forall-i [[3 7]]
9: | contradiction :not-e [1 8]
-----
10: (exists [x] (not (P x))) :raa [[2 9]]
-----
```

Für die drei weiteren Herleitung geben wir jeweils die Beweisschritte und das Ergebnis an:

- $\vdash \exists x \neg \varphi \rightarrow \neg \forall x \varphi$

Die Schritte der Herleitung:

```

1 (proof '(exists [x] (not (P x))) '(not (forall [x] (P x))))
2 (step-b :not-i 3)
3 (step-f :exists-e 1 4)
4 (swap '?1 :i)
5 (step-f :forall-e 2 3)
6 (step-f :not-e 4 5)
```

Das Ergebnis:

```

-----
1: (exists [x] (not (P x))) :premise
-----
2: | (forall [x] (P x)) :assumption
| -----
3: | | (actual :i) :assumption
4: | | (not (P :i)) :assumption
5: | | (P :i) :forall-e [2 3]
6: | | contradiction :not-e [4 5]
| -----
7: | contradiction :exists-e [1 [3 6]]
-----
8: (not (forall [x] (P x))) :not-i [[2 7]]
-----
```

- $\vdash \neg \exists x \varphi \rightarrow \forall x \neg \varphi$

Die Schritte der Herleitung:

```

1 (proof '(not (exists [x] (P x))) '(forall [x] (not (P x))))
2 (step-b :forall-i 3)
3 (swap '?1 :i)
4 (step-b :not-i 4)
5 (step-b :not-e 5 1)
6 (step-b :exists-i 5 2)

```

Das Ergebnis:

```

-----
1: (not (exists [x] (P x))) :premise
-----
2: | (actual :i) :assumption
| -----
3: | | (P :i) :assumption
4: | | (exists [x] (P x)) :exists-i [2 3]
5: | | contradiction :not-e [1 4]
| -----
6: | (not (P :i)) :not-i [[3 5]]
-----
7: (forall [x] (not (P x))) :forall-i [[2 6]]
-----
```

- $\vdash \forall x \neg \varphi \rightarrow \neg \exists x \varphi$

Die Schritte der Herleitung:

```

1 (proof '(forall [x] (not (P x))) '(not (exists [x] (P x))))
2 (step-b :not-i 3)
3 (step-f :exists-e 2 4)
4 (swap '?1 :i)
5 (step-f :forall-e 1 3)
6 (step-f :not-e 5 4)

```

Das Ergebnis:

```

-----
1: (forall [x] (not (P x))) :premise
-----
2: | (exists [x] (P x)) :assumption
| -----
3: | | (actual :i) :assumption
4: | | (P :i) :assumption
5: | | (not (P :i)) :forall-e [1 3]
6: | | contradiction :not-e [5 4]
| -----
7: | contradiction :exists-e [2 [3 6]]
-----
8: (not (exists [x] (P x))) :not-i [[2 7]]
-----
```

14.3 Vollständigkeit des natürlichen Schließens

Wie beim natürlichen Schließen in der Aussagenlogik stellen sich mit den zusätzlichen Regeln für die Prädikatenlogik mit Gleichheit die zwei Fragen:

- Sind die Regeln wahrheitserhaltend, leiten sie aus wahren Formeln auch immer nur wahre Formeln her? Dies ist die Frage nach der *Korrektheit* des Beweissystems:

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

- Sind alle Regeln vorhanden, die es gestatten jeden zutreffenden Zusammenhang auch tatsächlich herzuleiten? Dies ist die Frage nach der *Vollständigkeit* des Beweissystems:

$$\Gamma \vDash \varphi \Rightarrow \Gamma \vdash \varphi$$

14.3.1 Die Korrektheit des natürlichen Schließens in der Prädikatenlogik

Satz 14.1 (Korrektheit). *Gegeben eine Menge Γ von geschlossenen Formeln sowie eine geschlossene Formel φ . Dann gilt:*

$$\Gamma \vdash \varphi \Rightarrow \Gamma \vDash \varphi$$

Beweisskizze. Der Beweis geht durch Induktion über die Länge der gegebenen Herleitung. Man kann also voraussetzen, dass die Aussage für kürzere Herleitungen zutrifft. Also muss man nur zeigen, dass der jeweils letzte Schritt wahrheitserhaltend ist.

Dazu muss man sich davon überzeugen, dass dieses für jede der Regeln des natürlichen Schließens zutrifft, denn jede könnte ja die letzte der in der Herleitung verwendeten Regeln sein.

Was die Regeln für \neg, \wedge usw. angeht, haben wir dies schon beim Beweis der Korrektheit des natürlichen Schließens für die Aussagenlogik gesehen. Also muss man nur noch die Regeln für die Quantoren und die Gleichheit untersuchen. Nun sind die Regeln aber eben so konstruiert, dass dies der Fall ist (für einen systematischen und detaillierten Beweis dafür siehe [vD13, Lemma 3.8.2]). \square

14.3.2 Die Vollständigkeit des natürlichen Schließens in der Prädikatenlogik

In der Aussagenlogik ist es uns gelungen einen Beweis für eine Tautologie zu konstruieren, in dem wir in der Wahrheitstafel alle Modelle für die

Formel betrachtet haben und durch die fortwährende Anwendung des Regel TND eine Situation geschaffen haben, wo wir aus jeder Zeile der Wahrheitstafel, also aus jedem möglichen Modell die Formel durch die Regeln des natürlichen Schließens herzuleiten hatten.

Ein solches Vorgehen ist in der Prädikatenlogik nicht möglich, weil wir ja unmöglich alle Modelle hinschreiben könnten. Also müssen wir den Weg des indirekten Beweises gehen und wie im Fall der Aussagenlogik nehmen wir für den Beweis von $\Gamma \vDash \varphi \rightarrow \Gamma \vdash \varphi$ an, dass die Folgerung nicht richtig ist. Wir wissen dann, dass $\Gamma \cup \{\neg\varphi\}$ konsistent ist. Können wir nun aus dieser konsistenten Menge von Formeln ein Modell herleiten, dann haben wir gezeigt, dass $\Gamma \vDash \neg\varphi$, was der Voraussetzung widerspricht. Ganz analog zum indirekten Beweis des Vollständigkeitsatz in der Aussagenlogik benötigen wir also den Modellexistenzsatz, der besagt, dass eine konsistente Menge von Formeln ein Modell hat.

Für den Beweis wird die sogenannte Henkin-Konstruktion verwendet nach Leon Henkin⁵. In Lehrbüchern findet man viele Varianten des Beweise, die alle die Henkin-Konstruktion in der einen oder anderen Form verwenden.

Auf Basis der Henkin-Konstruktion kann man zusammen mit dem Satz von Lindenbaum aus den syntaktischen Symbolen der Sprache selbst ein Modell konstruieren, für das man dann zeigen kann, dass in ihm $\neg\varphi$ gelten muss, was den Widerspruch liefert.

In der Vorlesung werden wir die Vollständigkeit des natürlichen Schließens für die Prädikatenlogik nicht beweisen. Wer sich für den Beweis interessiert: Ich finde die Argumentation von Richard Kaye in [Kay07, Section 4.12] recht gut nachvollziehbar.

⁵Leon Henkin (1921–2006), amerikanischer Logiker.

Kapitel 15

Unentscheidbarkeit der Prädikatenlogik

Eine der Entscheidungsfragen in der Logik ist die nach der Allgemeingültigkeit einer Formel. Wir haben in der Aussagenlogik gesehen, dass diese Frage *entscheidbar* ist: es gibt einen Algorithmus, der für eine *beliebige* Formel der Aussagenlogik ermitteln kann, ob sie allgemeingültig ist. Wir können als Algorithmus das Aufstellen der Wahrheitstafel nehmen.

In der Prädikatenlogik ist die Sache anders: Das Gültigkeitsproblem in der Prädikatenlogik ist *nicht entscheidbar*. Präziser: es ist *semi-entscheidbar*, was bedeutet, dass es einen Algorithmus gibt, der für eine allgemeingültige Formel ein positives Ergebnis ergibt, bei einer nicht allgemeingültigen Formel jedoch möglicherweise nicht terminiert.

Wir werden in diesem Kapitel die Unentscheidbarkeit des Gültigkeitsproblems der Prädikatenlogik zeigen, in dem wir die Fragestellung zurückführen auf ein anderes Entscheidungsproblem, das Postsche Korrespondenzproblem¹, von dem man weiß, dass es unentscheidbar ist [Sip13, Chap 5.2].

Aus der Unentscheidbarkeit des Gültigkeitsproblems für die Prädikatenlogik folgt auch die Unentscheidbarkeit des Erfüllbarkeitsproblems. Mit der Vollständigkeit des natürlichen Schließens ergibt sich damit auch, dass es kein Programm geben kann, das für jede beliebige Fragestellung $\Gamma \vdash \varphi?$ eine Herleitung automatisch erstellen kann.

¹nach **Emil Leon Post** (1897–1954), polnisch-amerikanischer Mathematiker und Logiker

15.1 Das Postsche Korrespondenzproblem

Definition 15.1 (Postsches Korrespondenzproblem).

Gegeben eine endliche Folge von Paaren $(s_1, t_1), (s_2, t_2), \dots, (s_k, t_k)$ von Strings über dem Alphabet $\{0, 1\}$ positiver Länge, gibt es dann eine Folge von Indices i_1, i_2, \dots, i_n mit $n \geq 1$, so dass die Konkatenation der Strings $s_{i_1} s_{i_2} \dots s_{i_n}$ und $t_{i_1} t_{i_2} \dots t_{i_n}$ identisch sind.

Man kann sich die Paare von Strings als Dominosteine vorstellen. Zum Beispiel könnten wir folgende Steine haben:

$$\left[\begin{array}{c} 1 \\ 101 \end{array} \right] \quad \left[\begin{array}{c} 10 \\ 00 \end{array} \right] \quad \left[\begin{array}{c} 011 \\ 11 \end{array} \right]$$

Man beachte, dass man die Steine mehrfach verwenden kann. Gesucht ist also eine Folge von Dominosteinen, so dass der String oben identisch mit dem String unten ist. In unserem Beispiel gibt es folgende Lösung:

$$\begin{aligned} & \left[\begin{array}{c} 1 \\ 101 \end{array} \right] \left[\begin{array}{c} 011 \\ 11 \end{array} \right] \left[\begin{array}{c} 10 \\ 00 \end{array} \right] \left[\begin{array}{c} 011 \\ 11 \end{array} \right] \\ & \left[\begin{array}{c} 1 \cdot 011 \cdot 10 \cdot 011 \\ 101 \cdot 11 \cdot 00 \cdot 11 \end{array} \right] \end{aligned}$$

15.2 Die Unentscheidbarkeit des Gültigkeitsproblems in der Prädikatenlogik

Die Reduktion der Frage nach der Entscheidbarkeit des Gültigkeitsproblems in der Prädikatenlogik auf das Postsche Korrespondenzproblem besteht darin, dass wir letzteres durch eine Formel so ausdrücken, dass $\models \varphi$ genau dann gilt, wenn das durch φ kodierte Korrespondenzproblem eine Lösung hat.

Wir starten mit einem (beliebigen) Korrespondenzproblem C

$$\left[\begin{array}{c} s_1 \\ t_1 \end{array} \right] \left[\begin{array}{c} s_2 \\ t_2 \end{array} \right] \dots \left[\begin{array}{c} s_k \\ t_k \end{array} \right]$$

und wollen nun das Problem als eine Formel der Prädikatenlogik darstellen.

Für die Kodierung in der Prädikatenlogik brauchen wir eine Sprache \mathcal{P} mit geeigneter Signatur:

- Wir nehmen eine Konstante e , die für den leeren String steht.

- Ferner sehen wir für unsere Sprache zwei unäre Funktionen f_0 und f_1 vor. Die Idee dahinter ist, dass die Funktion f_0 für das Anfügen der 0 und f_1 der 1 zu einem String ist. Zum Beispiel: $f_1(e)$ ergibt den String 1, $f_0(f_1(e))$ ergibt den String 10 Achtung: umgekehrte Reihenfolge, deshalb vereinbaren wir, dass für einen binären String s die Funktion $f_s(e)$ dadurch definiert ist, dass man die Buchstaben von s nacheinander auf e anwendet.
- Schließlich soll unsere Sprache noch ein binäres Prädikat P haben. Hier ist die Idee, dass das Prädikat zwei Terme s und t als Argumente hat und T wird, wenn es eine Folge von Indices (i_1, i_2, \dots, i_n) gibt und die beiden Terme erfüllen gerade die Anforderung des Postschen Korrespondenzproblems.

Die Sprache \mathcal{P} hat also die Signatur $\{P^2; f_0^1, f_1^1; e\}$.

In dieser Sprache formulieren wir nun eine Formel, die das Postsche Korrespondenzproblem kodiert:

$$\begin{aligned}\varphi_1 &\stackrel{\text{def}}{=} \bigwedge_{i=1}^k P(f_{s_i}(e), f_{t_i}(e)) \\ \varphi_2 &\stackrel{\text{def}}{=} \forall x \forall y \left(P(x, y) \rightarrow \bigwedge_{i=1}^k P(f_{s_i}(x), f_{t_i}(y)) \right) \\ \varphi_3 &\stackrel{\text{def}}{=} \exists z P(z, z) \\ \varphi &\stackrel{\text{def}}{=} \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3\end{aligned}$$

Nun ist zu zeigen:

$\vdash \varphi \Leftrightarrow$ Das Postsche Korrespondenzproblem C hat eine Lösung.

Beweis.

Wir nehmen an, dass $\models \varphi$ gilt. Das bedeutet, dass die Formel in allen Modellen der Sprache gilt. Wir müssen also nur ein Modell finden, in dem man die Lösung der Korrespondenzproblems C aus dem Zutreffen der Formel einfach ablesen kann.

Das Universum \mathbb{U} sei die Menge aller endlichen binären Strings. Die Interpretation der Konstanten e sei der leere String. Die beiden Funktionen sind definiert durch Anhängen ihres Index an ihr Argument. Schließlich sei noch die Interpretation von P definiert:

$$\begin{aligned}P^{\mathcal{M}} &\stackrel{\text{def}}{=} \{(s, t) \mid \text{es gibt eine Folge von Indices } (i_1, i_2, \dots, i_n) \text{ so dass} \\ &\quad s = s_{i_1} s_{i_2} \cdots s_{i_n} \text{ und } t = t_{i_1} t_{i_2} \cdots t_{i_n}\}\end{aligned}$$

wobei die s_i und die t_i die Vorgaben aus dem Korrespondenzproblem C sind.

Da φ allgemeingültig ist nach Voraussetzung, gilt auch $\mathcal{M} \models \varphi$. Es gilt auch $\mathcal{M} \models \varphi_2$. Warum? Wenn $P(s, t)$ gilt, dann kann man für jedes $i = 1, 2, \dots, k$ die Folge der Indices einfach verlängern. Außerdem gilt auch $\mathcal{M} \models \varphi_1$, denn $f_{s_i}(e)$ ist ja gerade s_i , für t analog.

Also gilt $\mathcal{M} \models \varphi_1 \wedge \varphi_2 \rightarrow \varphi_3$ sowie $\mathcal{M} \models \varphi_1 \wedge \varphi_2$, also $\mathcal{M} \models \varphi_3$. Das bedeutet aber gerade, dass das Korrespondenzproblem C eine Lösung hat.

Nun ist die umgekehrte Richtung zu zeigen. Wir nehmen also an, dass C eine Lösung hat und müssen zeigen, dass in einem beliebiges Modell \mathcal{M} zu unserer Sprache die Formel φ wahr wird.

Wir machen eine Fallunterscheidung:

Angenommen $\mathcal{M} \not\models \varphi_1 \wedge \varphi_2$, dann gilt $\models \varphi$.

Angenommen als, dass gilt $\mathcal{M} \models \varphi_1 \wedge \varphi_2$. In diesem Fall ist zu zeigen, dass dann auch $\mathcal{M} \models \varphi_3$ gilt. Die Idee ist nun, dass man endliche binäre Strings im Universum von \mathcal{M} , d.h. man definiert eine Abbildung ι induktiv durch

$$\begin{aligned}\iota(e) &\stackrel{\text{def}}{=} e^{\mathcal{M}} \\ \iota(s0) &\stackrel{\text{def}}{=} f_0^{\mathcal{M}}(\iota(s)) \\ \iota(s1) &\stackrel{\text{def}}{=} f_1^{\mathcal{M}}(\iota(s))\end{aligned}$$

Da es nach Voraussetzung eine Lösung $s = s_{i_1}s_{i_2}\dots s_{i_n}$ und $t = t_{i_1}t_{i_2}\dots t_{i_n}$ von C gibt, sind s und t gleich, das bedeutet abder auch ihre Interpretationen sind gleich: $\iota(s) = \iota(t)$ und damit existiert ein Element z des Universums von \mathcal{M} mit $P(z, z)$, nämlich $\iota(s)$. Also $\mathcal{M} \models \varphi_3$ \square

15.3 Der Sonderfall endlicher Universen

Im vorherigen Abschnitt haben wir gesehen, dass die Prädikatenlogik unentscheidbar ist. Die Situation sieht anders aus, wenn wir zu einer gegebenen Sprache \mathcal{L} nur Modelle mit endlichen Universen einer vorgegebenen Größe n betrachten.

Zunächst beobachtet man, dass bei einem Universum $\mathbb{U} = \{u_1, u_2, \dots, u_n\}$

die Quantoren durch \wedge bzw. \vee ausgedrückt werden können:

$$\begin{aligned}\forall x \varphi(x) &= u_1 \wedge u_2 \wedge \dots \wedge u_n \\ \exists x \varphi(x) &= u_1 \vee u_2 \vee \dots \vee u_n\end{aligned}$$

Was Prädikate angeht, können sie in einem endlichen Universum jeweils durch endlich viele Symbole ausgedrückt werden. Ein einstelliges Prädikat P etwa durch $P_{u_1}, P_{u_2}, \dots, P_{u_n}$, ein zweistelliges Prädikat durch $P_{(u_1, u_1)}, P_{(u_1, u_2)}, \dots, P_{(u_n, u_n)}$ usw.

Auch Funktionen könne durch explizite Wertetabellen dargestellt werden, die angeben, zu welchen Elementen des Universums sie auswerten.

Das bedeutet, dass wir jedes der von den Prädikaten herrührenden Symbole als atomare Aussagen auffassen können. Auf diese Weise ist es möglich jede Formel zu einer Formel in der Aussagenlogik zu machen.

Und folglich können wir auch das Gültigkeitsproblem in diesem Sonderfall lösen:

Sei φ eine Formel über einem *endlichen* Universum. Dann wandeln wir φ um in eine Formel φ' der Aussagenlogik und prüfen die Erfüllbarkeit von $\neg\varphi'$. Ist $\neg\varphi'$ unerfüllbar, dann ist φ' allgemeingültig und demzufolge φ in allen Universen der fixen Größe n wahr.

Aus dieser Überlegung kann man ein Widerlegungsverfahren für das Gültigkeitsproblem im allgemeinen Fall machen:

Sei φ eine beliebige Formel der Prädikatenlogik. φ ist allgemeingültig, wenn $\neg\varphi$ unerfüllbar ist. Für $n = 1, 2, 3, \dots$ prüfe man $\neg\varphi$ für die Größe n des Universums durch die Frage nach der Erfüllbarkeit der korrespondierenden Formel in der Aussagenlogik. Wenn sich erweist, dass in einem dieser Universen die Formel erfüllbar ist, dann ist sie nicht allgemeingültig.

Ob eine Formel allgemeingültig ist, können wir auf diese Weise jedoch nicht feststellen – so viel Zeit haben wir nicht.

Ein Beispiel für Erfüllbarkeit in endlichen Universen

In diesem Abschnitt wollen wir die Logic Workbench als *Model Finder* in der Prädikatenlogik verwenden. Wir machen uns auf die Suche nach einer Gruppe, die nicht kommutativ ist.

Wir legen die Sprache fest, in der wir arbeiten wollen, in dem wir eine Signatur definieren:

```
(def grp-sig
  {'unit [:func 0]
   'op   [:func 2]
   'inv  [:func 1]})
```

Die binäre Funktion **op** ist die Operation der Gruppe, **inv** ist die Inversenbildung und die Konstante **unit** ist das neutrale Element der Gruppe.

In dieser Sprache kann man die Axiome der Gruppentheorie formulieren:

```
;; Assoziativität
(def grp-ass
  '(forall [x y z] (= (op x (op y z)) (op (op x y) z))))
;; Neutrales Element
(def grp-unit
  '(forall [x] (= (op x unit) x)))
;; Inverses Element
(def grp-inv
  '(forall [x] (= (op x (inv x)) unit)))
```

Darüberhinaus formulieren wir die Eigenschaft der Kommutativität:

```
(def grp-comm '(forall [x y] (= (op x y) (op y x))))
```

Auf der Suche nach einer Gruppe, die nicht kommutativ ist, brauchen wir ein Modell, das die Gruppenaxiome erfüllt, jedoch nicht das Kommutativgesetz:

```
(def grp
  (list 'and grp-ass grp-unit grp-inv))
(def grp-na
  (list 'and grp (list 'not grp-comm)))
```

Nun können wir in der Logic Workbench die Funktion **sat** mit Angabe der Größe des Universums dazu verwenden, um Modelle zu generieren, die die Formel **grp-na** erfüllen, also gerade nicht-kommutative Gruppen sind.

Wir probieren einfach beginnend mit einem Universum mit 2 Elementen aus und zählen alle gefundenen Modelle:

```
(count (sat grp-na grp-sig 2 :all))
(count (sat grp-na grp-sig 3 :all))
(count (sat grp-na grp-sig 4 :all))
(count (sat grp-na grp-sig 5 :all))
; => 0
```

Das Ergebnis ist 0 für Universen mit weniger als 6 Elementen. Also probieren wir:

(sat grp-na grp-sig 6)

Wir bekommen folgendes Ergebnis

```
{unit #{{:unit}},
op #{{:e2 :e5 :e3] [:e5 :e1 :e2] [:unit :unit :unit]
      [:unit :e5 :e5] [:e5 :e5 :unit] [:e2 :e4 :e1]
      [:e4 :e5 :e1] [:e5 :unit :e5] [:e3 :e4 :e5]
      [:e1 :e2 :e5] [:e3 :e1 :unit] [:e1 :e1 :e3]
      [:unit :e4 :e4] [:e1 :unit :e1] [:unit :e2 :e2]
      [:e3 :e3 :e1] [:e2 :e2 :unit] [:e2 :unit :e2]
      [:e3 :e5 :e2] [:e4 :e3 :e2] [:e3 :e2 :e4]
      [:e2 :e3 :e5] [:e4 :unit :e4] [:e2 :e1 :e4]
      [:e1 :e5 :e4] [:e4 :e1 :e5] [:e4 :e2 :e3]
      [:unit :e1 :e1] [:unit :e3 :e3] [:e4 :e4 :unit]
      [:e1 :e3 :unit] [:e1 :e4 :e2] [:e3 :unit :e3]
      [:e5 :e3 :e4] [:e5 :e4 :e3] [:e5 :e2 :e1]},
inv #{{:e5 :e5] [:e2 :e2] [:unit :unit] [:e1 :e3] [:e3 :e1] [:e4 :e4]},
:univ #{{:e1 :e2 :e3 unit :e4 :e5}}}
```

Die Relation `op` ist gerade die Multiplikationstabelle der Gruppe und wir können direkt ablesen, dass zum Beispiel `:e3 op :e5 = :e2`, aber `:e5 op :e3 = :e4` ist.

Es handelt sich um die symmetrische Gruppe S_3 , siehe [Group Explorer](#).

Kapitel 16

Anwendungen der Prädikatenlogik in der Softwaretechnik

16.1 Spezifikation und Analyse von Eigenschaften von Programmen

Eine wichtige Anwendung der Prädikatenlogik in der Softwaretechnik besteht in der Spezifikation von Funktionen und der Möglichkeit der systematischen Analyse der Spezifikationen und ihrer Implementierungen.

Dies kann man durchaus auf sehr unterschiedlichen Leveln der Präzision und Formalität tun. Um mit einem sehr einfachen Beispiel zu beginnen: Man kann Mehrdeutigkeiten vermeiden.

Wenn in einer Spezifikation zum Beispiel steht: „Alle Elemente des Arrays A der Länge n sind entweder 0 oder 1“, dann kann man diesen Satz ganz unterschiedlich auffassen:

$$\forall i ((0 \leq i \wedge i < n) \rightarrow (A[i] = 0 \vee A[i] = 1))$$

oder

$$\forall i ((0 \leq i \wedge i < n) \rightarrow A[i] = 0) \vee \forall i ((0 \leq i \wedge i < n) \rightarrow A[i] = 1)$$

Die Formulierung mit Hilfe der Prädikatenlogik stiftet Klarheit, und wenn man die Syntax etwas anreichert, werden die Bedingungen noch besser lesbar:

$$\forall i \in [0, n] : A[i] = 0 \vee A[i] = 1$$

$$\forall i \in [0, n] : A[i] = 0 \vee \forall i \in [0, n] : A[i] = 1$$

Ein bisschen Formalismus hilft Missverständnisse zu vermeiden.

*

Man kann die Anwendung der Prädikatenlogik natürlich weit konsequenter anwenden. Eine naheliegende Möglichkeit besteht darin für eine Funktion (oder ein Programm) zwei Prädikate zu formulieren:

- Eine *Vorbedingung* φ spezifiziert, was vor Ausführen der Funktion zutrifft.
- Die *Nachbedingung* ψ spezifiziert, was nach dem Ende der Funktion zutrifft.

Mit dieser Idee kann man über die Korrektheit von Programmen in imperativen Sprachen mit Mitteln der Prädikatenlogik argumentieren. Dieses Vorgehen wird als *Hoare-Kalkül*¹ bezeichnet.

Das Hoare-Tripel beschreibt wie ein Programmsegment P den Zustand der Berechnung so ändert, dass vorher die Vorbedingung φ gilt und hinterher die Nachbedingung ψ :

$$\langle\varphi\rangle P \langle\psi\rangle$$

Im Hoare-Kalkül geht es darum, zu *beweisen*, dass das Programmsegment P tatsächlich die Eigenschaft hat, dass $\varphi \rightarrow \psi$ gilt. Außerdem ist zu untersuchen, ob die Berechnung *terminiert*. Ist beides der Fall, dann spricht man von *totaler Korrektheit*.

Dieser Abschnitt soll nur auf diese Technik der Programmverifikation hinweisen, mehr findet man in [HR04, Kapitel 4].

*

Es gibt noch einen anderen Ansatz für die Verwendung der Prädikatenlogik in der Softwaretechnik, der hier erwähnt werden soll. Man kann mit dem Beweissystem *Coq* sogenannte zertifizierte Programme entwickeln. Im ersten Kapitel von [BC04] wird als Beispiel gezeigt, wie man zwei Prädikate formulieren kann, mit denen man beschreiben kann, was ein Sortierprogramm erreichen soll:

Gegeben eine Liste L von ganzen Zahlen. Sei $\sigma(L)$ die Formel, die beschreibt, dass L sortiert ist und sei \equiv die Relation, die für zwei Listen L und L' ausdrückt, dass sie die gleichen Elemente enthalten. Die Spezifikation eines Sortieralgorithmus ist dann der Typ einer Funktion, die jede

¹nach C.A.R. „Tony“ Hoare britscher Informatiker, geb. 1934

Liste L auf eine Liste L' abbildet, die folgende Formel erfüllt:

$$\sigma(L') \wedge L' \equiv L$$

Die Aufgabe ein zertifiziertes Sortierprogramm zu entwickeln ist dann identisch mit der Aufgabe einen Term des eben beschriebenen Typs zu finden. Und eben dieses kann man interaktiv in *Coq* tun und anschließend daraus Programmcode in den Sprachen OCaml, Haskell oder Scheme generieren.

Dieser kurze Hinweis soll als „Eyecatcher“ genügen. Wir werden uns im folgenden Abschnitt mit einer „leichtgewichtigen“ formalen Methode beschäftigen.

16.2 Analyse von Software mit Alloy

In der Prädikatenlogik kann man sich (wie in der Aussagenlogik auch) zwei Fragen stellen, zwei Konzepte betrachten:

1. Gegeben sei ein Modell \mathcal{M} , also gewissermaßen eine vorgegebene *Welt*, und eine Formel φ der Prädikatenlogik, gilt dann die Formel φ in \mathcal{M} , d.h. gilt $\mathcal{M} \models \varphi$?
2. Gegeben eine Mengen Γ von Formeln und eine Formel φ , folgt dann φ aus Γ , d.h. gilt $\Gamma \vdash \varphi$ bzw. $\Gamma \models \varphi$?

Überträgt man diese beiden Fragestellung auf das Spezifizieren und Entwickeln von Software, dann führen die Konzepte zu:

1. *Model Checking*. Gegeben sei eine Implementierung ($\hat{=}$ Modell \mathcal{M}), erfüllt diese dann eine verlangte Anforderung ausgedrückt in der Formel φ ? Bei diesem Ansatz brauchen wir also ein Modell, d.h. eine Menge Details im Modell, die vielleicht gar nicht aus den Anforderungen an die Software stammen.
2. *Allgemeingültigkeit*. Der Ausgangspunkt ist nicht ein Modell, sondern eine Spezifikation des zu untersuchenden Systems, ausgedrückt in der Formelmenge Γ . Dann kann man Fragen, ob in *allen* Implementierungen, die diese Spezifikation erfüllen, auch die Anforderung φ garantiert ist. Bei diesem Ansatz benötigt man also nicht Details eines Modells, hat aber andererseits den Nachteil, dass diese Frage – wie wir gesehen haben – nicht entscheidbar ist.

16.2.1 Die Idee hinter Alloy

Die Idee hinter der leichtgewichtigen formalen Methode Alloy von Daniel Jackson² besteht darin, die beiden Ansätze zu kombinieren:

1. Ausgangspunkt ist die Spezifikation Γ . Begrenzt man nun die Größe des Universums, so ist es möglich, Modelle \mathcal{M} zu finden, die dieser Spezifikation genügen. Man macht also gewissermaßen *Model Finding*.
2. Nun verwendet man diese Modelle um zu checken, ob sie die Formel φ erfüllen.

Diese Kombination kann man verwenden, um zu zeigen, dass eine Spezifikation Γ die Anforderung φ *nicht* erfüllt: Man sucht nach Modellen für die Formelmenge $\Gamma \cup \{\neg\varphi\}$. Wenn man ein (kleines) Modell dafür findet, dann hat man ein Gegenbeispiel. Und kann dann untersuchen, woran es liegt, dass $\Gamma \not\models \varphi$ gilt.

Wenn man kein Gegenbeispiel findet, dann kann man natürlich nicht sicher sein, dass $\Gamma \models \varphi$ gilt, weil für das Generieren der Modell eine maximale Größe des Universums vorgegeben werden muss. Es könnte ja sein, dass in einem größeren Universum doch ein Modell existiert.

Daniel Jackson geht von der sogenannten *Small Scope Hypothesis*³ aus, die besagt, dass viele (Denk-)Fehler schon im Kleinen auftauchen und mit Alloy sichtbar gemacht werden können. Unsere Erfahrung als Softwareentwickler bestätigt dies: es ist oft so, dass wenn ein Fehler in einem Stück Software erstmal gefunden ist, sich einfache Beispiele bilden lassen ihn zu demonstrieren.⁴

16.2.2 Beispiele für Analysen mit Alloy

Auf der [Webseite zu Alloy](#) findet man eine Vielzahl von Analysen, die mit Alloy gemacht wurden. Darunter:

- Kritische Systeme in der *Medizintechnik*. Eine Gruppe an der University of Washington hat mit Alloy ein System für die Neutronen-

²Daniel Jackson, Professor für Informatik am Massachusetts Institute of Technology (MIT).

³“Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or small instance makes no difference. So if the analysis considers all small instances, most flaws will be revealed. This observation, which I call the small scope hypothesis, is the fundamental premise that underlies Alloy’s analysis.” [Jac12, S.15]

⁴Das ist gewissermaßen das $\mathcal{P} \neq \mathcal{NP}$ der Softwareentwicklung: Fehler finden ist extrem schwer, hinterher zu sehen, worin der Fehler besteht, oft überraschend einfach.

Strahlentherapie systematisch untersucht und dabei mehrere kritische Fehler gefunden, die vor dem Einsatz des Systems korrigiert werden konnten, siehe [University of Washington PLSE Neutrons](#).

- *Netzwerk-Protokolle.* Pamela Zave⁵ hat die Spezifikation für Chord, eine Peer-to-Peer verteilte Hashtabelle, mit Alloy und Spin analysiert und dabei verschiedene Fehler entdeckt [[Zav12](#)].
- *Programmverifikation.* Die Idee hierbei besteht darin, aus Java-Code, der mit JML (Java Modeling Language) annotiert ist, also mit Klasseninvarianten, Vorbedingungen, Nachbedingungen und Schleifen-Invarianten formal spezifiziert ist, automatisch Spezifikationen in Alloy zu erzeugen, die man dann für die Verifikation von Programmeigenschaften verwenden kann. Greg Dennis aus der Arbeitsgruppe von Daniel Jackson hat das Tool [Forge](#) entwickelt, mit dem solche Analysen möglich sind. Forge wurde verwendet, um Implementierungen von Listen in Java zu untersuchen. Dabei wurden sowohl Fehler in den JML-Spezifikationen als auch in den Implementierungen gefunden⁶. Außerdem wurde Forge eingesetzt, um eine Wahlsoftware in den Niederlanden zu analysieren. Auch bei dieser Analyse konnten Fehler in Spezifikationen und in der Implementierung aufgedeckt werden⁷.
- *IT-Sicherheit.* Eine Gruppe an der University of California Berkeley und an der Stanford University hat verschiedene Sicherheitsmechanismen für das Web analysiert: WebAuth, HTML5 forms und as Cross-Origin Resource Sharing Protokoll u.a. Dabei wurden bekannte Schwachstellen bestätigt und drei weitere entdeckt⁸

16.2.3 Die Sprache Alloy und das Werkzeug Alloy Analyzer

Alloy ist nicht auf die Untersuchung vordefinierter Artefakte der Softwareentwicklung festgelegt. Man kann Ideen, Spezifikationen, Entwürfe, Implementierung usw. mit Alloy untersuchen. In Abbildung 16.1 zeigt die Ziffer ① dass zunächst aus dem zu untersuchenden Gegenstand eine Spezifikation in Alloy erstellt werden muss. Dies betrifft sowohl die Struktur der Situation wie auch die Dynamik, also Veränderungen an der Situation. Was die Dynamik angeht, ist zu beachten, dass man nicht

⁵Pamela Zave, amerikanische Informatikerin, lange Zeit in der Forschung und Entwicklung bei AT&T tätig, seit 2017 Forscherin an der Princeton University.

⁶[Modular Verification of Code with SAT](#) von Greg Dennis, Felix Chang und Daniel Jackson.

⁷[Bounded Verification of Voting Software](#) von Greg Dennis, Kuat Yessenov und Daniel Jackson.

⁸[Towards a Formal Foundation of WebSecurity](#) von Devdatta Akhawe, Adam Arth, Peifung E. Lam, John Mitchell und Dawn Song.

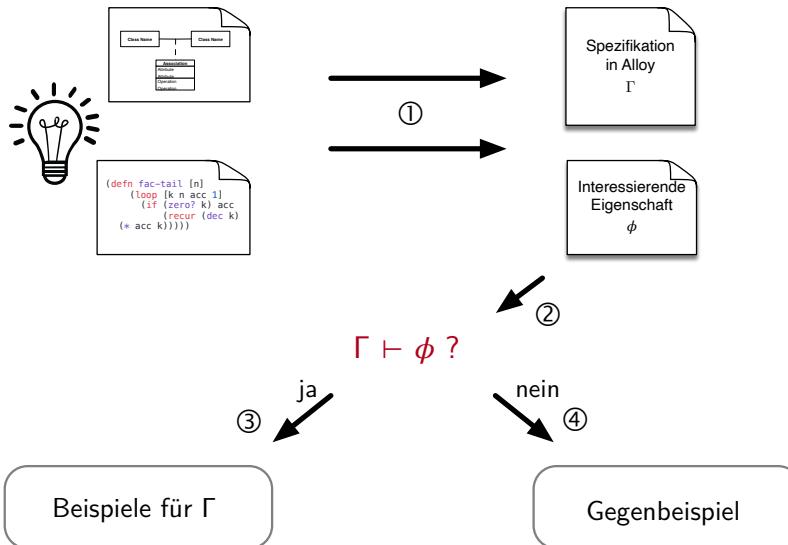


Abbildung 16.1: Verwendung von Alloy

nur spezifizieren muss, was sich ändert, sondern auch was im veränderten Zustand gleich geblieben ist, die sogenannten *frame conditions*.

Den Entwurf der Spezifikation macht man interaktiv im *Alloy Analyzer* ②, indem man schrittweise die Spezifikation erstellt und immer wieder den Alloy Analyzer Modelle für die Spezifikation erzeugen lässt. Dies ist eine hoch interessante, sehr agile Erfahrung. Typischerweise wird man feststellen, dass bei der Spezifikation Bedingungen zunächst vergessen werden, etwa weil man sie für selbstverständlich hält. Oder man erhält Beispiele, an denen man erkennt, dass es Spezialfälle gibt, die man nicht berücksichtigt hat.

Mit dem Kommando `run` kann man zur Spezifikation Γ Beispiele (siehe ③) erzeugen lassen zu einer vorgegebenen Maximalzahl von Elementen der definierten Sigs (Default ist 3). Die Formel φ formuliert man in Alloy als `assert` und das Kommando `check` sucht nach einem Gegenbeispiel. Wird dies gefunden wird es (④) angezeigt und kann analysiert werden.

16.2.4 Die Sprache Alloy

Wir haben die Korrespondenz von Prädikaten und Relationen in Abschnitt 11.2 diskutiert. Alloy ist eine Sprache, in der man Relationen deklarieren kann, also eine Sprache der relationalen Logik.

Für die Elemente der Sprache mag hier ein Hinweis auf den Text *Kurze Einführung in Alloy*, den ich gemeinsam mit Nils Asmussen verfasst habe.

16.2.5 Der Alloy Analyzer — unter der Haube

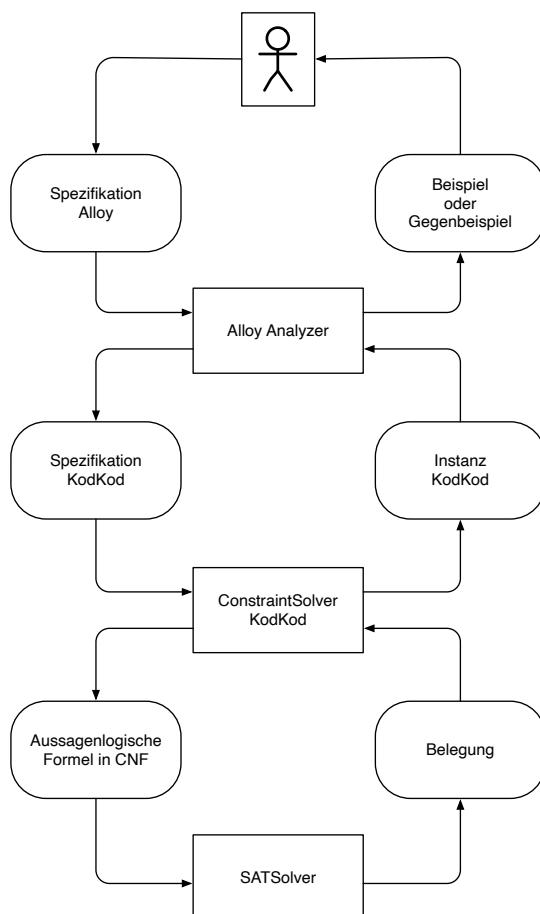


Abbildung 16.2: Arbeitsweise des Alloy Analyzers

Die grundlegende Idee für die Arbeitsweise des Alloy Analyzers ergibt sich aus der Tatsache, dass man für endliche Universen eine Formel der Prädikatenlogik in eine Formel der Aussagenlogik transformieren kann, wie in Abschnitt 15.3 diskutiert. Tatsächlich wird diese Transformation nicht direkt vom Alloy Analyzer durchgeführt, sondern, wie in Abbildung 16.2 dargestellt, vom *Kodkod Constraint Solver*⁹. Der Alloy

⁹Der *Kodkod Constraint Solver* wurde von *Emina Torlak* im Rahmen ihrer Promotion bei Daniel Jackson entwickelt.

Analyzer übersetzt die Spezifikation in Alloy in eine Spezifikation in Kodkod. Dieses Werkzeug leistet nun die eigentliche Transformation der Spezifikation in die Aussagenlogik und verwendet dann SAT-Solver, als Default SAT4J, um die Erfüllbarkeit der Formel zu prüfen. Wird eine Belegung gefunden, wird daraus eine Instanz von Relationen in Kodkod gebildet und diese vom Alloy Analyzer in ein Beispiel oder Gegenbeispiel (je nach Fragestellung) in Alloy zurückinterpretiert. Besonders wichtig ist das sogenannte *Symmetry Breaking*: Permutationen der Bezeichnung von Atome in Formeln führen zu Lösungen, die im Prinzip bis auf die Benennung der Atome identisch sind. Kodkod setzt deshalb automatisch spezielle zusätzliche Bedingungen ein, die solche symmetrischen Modelle ausschließen.

16.2.6 Ein (kleines) Beispiel

Michael Jackson demonstriert die Möglichkeiten von Alloy am Beispiel einer Idee zu einem E-Mail-Programm [Jac06]. Dieses Beispiel wollen wir in diesem Abschnitt nachvollziehen.

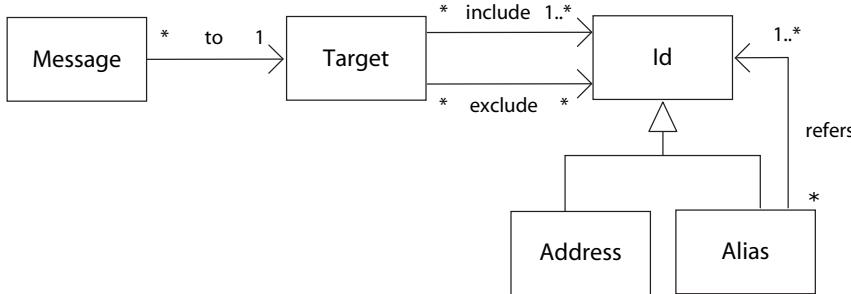


Abbildung 16.3: Klassendiagramm E-Mail-Programm

Das Klassendiagramm in Abbildung 16.3 visualisiert das Konzept der Adresseierung von E-Mails: Jede Nachricht (*Message*) hat genau ein Ziel (*Target*), nämlich die Menge der Adressaten. Diese können in Gruppen eingeteilt werden. Die Konstruktion der 3 Klassen *Id*, *Alias* und *Address* zeigt, dass *Ids* entweder eine Adresse sein können oder ein Alias, der für eine Menge weiterer *Ids* steht, die selbst wieder Adressen oder Aliase sein können.

Die Eigenschaft, die wir mit Alloy analysieren wollen, besteht darin, dass im geplanten E-Mail-Programm es sowohl die Möglichkeit geben soll, *Ids* dem Ziel hinzuzufügen (*include*), als auch auszuschließen (*exclude*). Dieses Feature könnte für Anwender sinnvoll sein, wenn jemand zum Beispiel eine Mitteilung an alle Kolleg:innen verschicken möchte, jedoch bestimmte Personen ausschließen möchte.

Nun stellt sich die Frage, wie werden die wirklichen Adressaten ermittelt: Spielt es eine Rolle, ob man die Differenzmenge zwischen den eingeschlossenen und den ausgeschlossenen Adressen *vor* oder *nach* der Auflösung der Alias-Referenzen bilden muss. Und genau diese Frage wollen wir nun mit Alloy untersuchen.

Zunächst spezifizieren wir die im Klassendiagramm dargestellte Struktur in Alloy:

```

1 module lfm/email
2
3 sig Message {
4   to: Target
5 }
6 sig Target {
7   include: some Id,
8   exclude: set Id
9 }
10 abstract sig Id {}
11 sig Address extends Id {}
12 sig Alias extends Id {
13   refers: set Id
14 }
15
16 run {
17   #Alias = 3
18 }
```

In Zeile 3 der Spezifikation wird eine *Signature* definiert. man kann damit die Vorstellung verbinden, dass damit ein Typ definiert wird. Die *Signatures* **Message** und **Target** kommen in der Relation **to** vor. Dabei beinhaltet die Syntax, dass es zu jedem Element in **Message** genau ein Element von **Target** in der Relation **to** steht.

In Zeile 7 wird festgelegt, dass in **include** zu einem **Target** mehrere Ids, aber mindestens eine, gehören. In Zeile 8 wird eine Menge von Ids zugeordnet, die auch leer sein könnte.

Ein abstrakte *Signature* wie **Id** in Zeile 10 bedeutet eine Partitionierung des Typs in die beiden Typen **Address** sowie **Alias**.

Wie man sieht, ist es recht geradlinig, wie man das Klassendiagramm in die Spezifikation transformiert. Das liegt auch daran, dass Daniel Jackson beim Entwurf der Sprache ihr ein objekt-orientiertes Flair gegeben hat.

in Zeile 16 formulieren wir ein Kommando an den Alloy Analyzer: er soll Modelle zur Spezifikation erstellen und zwar solche, bei denen der Typ

Alias genau drei Elemente hat. Auf diese Weise können wir prüfen, ob unsere Spezifikation auch dem entspricht, wie wir uns die Struktur der Ids vorstellen.

Und siehe da: Es gibt Beispiele, die gar nicht so ausschauen, wie wir dachten. Es kommen Beispiele vor, bei denen ein Alias gar nicht auf eine wirkliche Adresse verweist und es kommen Zyklen im Graph der Adressen vor. Das ist ein Beispiel dafür, dass das Klassendiagramm nicht alle Integritätsbedingungen ausdrückt, die eigentlich gemeint sind. Wir erweitern die Spezifikation und sorgen dafür, dass keine Zyklen auftreten können. Für unsere Fragestellung ist es unerheblich, dass es vorkommen kann, dass ein Alias gar keine Referenz hat, deshalb brauchen wir die Spezifikation bezüglich dieser Frage nicht zu erweitern. Das ist ein Beispiel dafür, dass man Alloy als „leichtgewichtig“ bezeichnen kann.

```
20 fact AliasingIsAcyclic {
21   no a: Alias | a in a.^refers
22 }
```

In Zeile 20 wird spezifiziert, dass es keine Zyklen geben darf. `a.^refers` bezeichnet in Alloy den transitiven Abschluss der binären Relation `refers`. Ein *Fact* ist eine Bedingung die immer gelten muss, hier also, dass kein Alias in seinem transitiven Abschluss enthalten sein darf, die Struktur also keine Zyklen haben darf.

Nun können wir an die eigentliche Fragestellung gehen. Dazu definieren wir in Alloy zwei Funktionen:

```
24 fun diffThenRefers(t: Target): set Id {
25   t.(include - exclude).*refers - Alias
26 }
27 fun refersThenDiff(t: Target): set Id {
28   (t.include.*refers - t.exclude.*refers) - Alias
29 }
```

Die erste Funktion hat ein `Target` als Argument und ergibt die Menge der Ids, die man erhält, wenn man erst die Differenz bildet und dann dereferenziert. Dabei wird der Operator `.`, der sogenannte *Dot Join* verwendet, sowie der Operator `*`, der für den reflexiven transitiven Abschluss steht. Bei der zweiten Funktion wird erst dereferenziert und dann die Differenz gebildet.

Nun können wir den Alloy Analyzer prüfen lassen, ob die beiden Vorgehensweisen übereinstimmen. Dazu verwenden wir das Kommando `check`, mit dem wir den Analyzer auf die Suche nach einem Gegenbeispiel schicken:

```
35 assert OrderIrrelevant{
```

```

36   all t: Target | diffThenRefers[t] = refersThenDiff[t]
37 }
38 check OrderIrrelevant for 3 but 1 Target

```

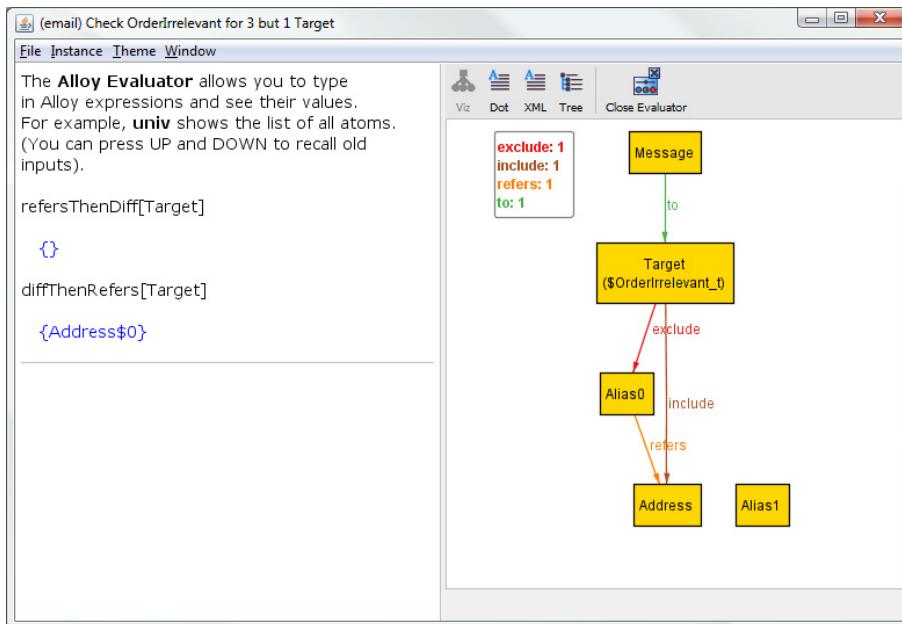


Abbildung 16.4: Alloy Analyzer Überprüfung des E-Mail-Programms

Ein mögliches Ergebnis der Analyse ist in Abbildung 16.4 dargestellt. Dabei wird auch der *Alloy Evaluator* verwendet: Im linken Teil des Fens-ters kann man Ausdrücke eingeben, die im aktuell im rechten Teil ange-zeigten Modell ausgewertet werden. Und nicht nur am Modell kann man so ablesen, dass die beiden Konzepte nicht übereinstimmen, man sieht explizit im linken Teil, dass nach der ersten Vorgehensweise jemand die E-Mail bekommen hätte, der eigentlich ausgeschlossen sein sollte, indirekt über ein Alias.

Doch wir erhalten nicht nur das Ergebnis, dass sicherlich die Vorgehens-weise zuerst zu dereferenzieren und dann die Differenz zu bilden, das gewünschte Verhalten ist, sondern auch, dass es dabei vorkommen kann, dass die Adressatenmenge leer ist, was man also bei der Entwicklung des E-Mail-Programms berücksichtigen muss.

Teil III

Lineare Temporale Logik

Kapitel 17

Dynamische Modelle

Bisher sind die beiden Logiken von ihrem Ansatz her eher statisch. Sie erlauben es, eine bestimmte Situation deklarativ zu einem bestimmten Zeitpunkt zu beschreiben. Softwaressysteme haben jedoch eine zeitliche Dimension: ihr Zustand zu einem Zeitpunkt ändert sich und führt zu einem anderen Zustand in einem späteren Zeitpunkt.

Natürlich kann man diese zeitliche Veränderung des Zustands auch in der Prädikatenlogik formulieren. In Alloy tut man dies zum Beispiel dadurch, dass man eine Signatur `State` oder `Time` einführt und sie als Komponente von Relationen zu all den Signaturen verwendet, deren Zustand sich ändern kann. Dieses Vorgehen wird als eine der Techniken, Dynamik in Alloy zu spezifizieren, beschrieben im Technischen Bericht von Nils Asmussen: [Ansätze zur Modellierung von Dynamik mit Alloy](#).

Man kann aber auch Modelle verwenden, in denen die Dimension der Zeit direkt und explizit vorgesehen ist: Transitionssysteme.

17.1 Konzept der Transitionssysteme

Die Grundidee:

1. Ein System befindet sich zu einem bestimmten Zeitpunkt in einem bestimmten *Zustand*.
2. Es treten Ereignisse auf, die *Transitionen* (Zustandsübergänge) auslösen, die zu einem anderen, späteren Zustand führen.

Man kann unterscheiden zwischen

1. *diskreten* und *kontinuierlichen* Transitionssystemen. In diskreten Systemen geht man von einer diskreten linearen Zeit aus: t_1, t_2, t_3, \dots . Solche Systeme werden wir verwenden.

2. *deterministischen* und *nicht deterministischen* Transitionssystemen. In deterministischen Systemen führt ein bestimmtes Ereignis dazu, dass das System in genau einen bestimmten Folgezustand wechselt.

Am Beispiel einer Ampelschaltung in Abbildung 17.1 kann jeder Zustand durch die Belegung der aussagenlogischen Atome *Rot*, *Gelb*, *Grün* beschrieben werden:

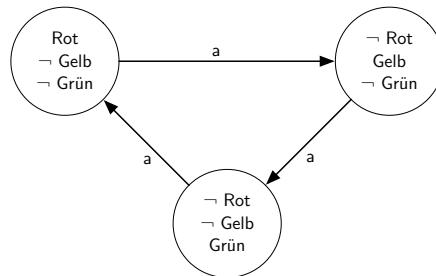


Abbildung 17.1: Transitionssystem Ampel

Geht man von einer fixen Menge von aussagenlogischen Atomen aus, wird man die Zustände des Transitionssystems oft nur mit den Atomen beschriften, die in diesem Zustand mit T belegt sind.

17.2 Beispiel eines Programms mit zwei Threads

Ein Entwickler möchte zwei Threads *T1* und *\T2* synchronisieren, in dem er zwei Variablen *t1gesperrt* und *t2gesperrt* verwendet. Dadurch möchte er sicherstellen, dass sich die beiden Threads in ihren Aktionen nicht in die Quere kommen.

Hier der Pseudocode des Konzepts:

```

volatile boolean t1gesperrt = false;
volatile boolean t2gesperrt = false;

// Thread T1:
run() {
T1.1  t1gesperrt = true;
T1.2  istT2Frei();
      // hier sind die kritischen Aktionen von T1
T1.3  t1gesperrt = false;
T1.e  }
  
```

```
// Thread T2:
run() {
T2.1   t2gesperrt = true;
T2.2   istT1Frei();
        // hier sind die kritischen Aktionen von T2
T2.3   t2gesperrt = false;
T2.e }
```

Die Funktion `istT1Frei()` ist folgendermaßen implementiert, `istT2Frei()` analog.

```
void ist T1Frei() {
    forever {
        if (t1gesperrt == false) return;
        sleep(10);
    }
}
```

Um dieses Konzept in ein Transitionssystem umzusetzen, vereinbaren wir die Notation in Abbildung 17.2.

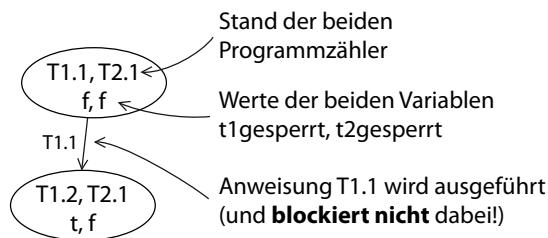


Abbildung 17.2: Notation für das Beispiel eines Programms mit zwei Threads

Nun kann man das Transitionssystem für dieses Programm darstellen in Abbildung 17.3. Man sieht offenbar in dem Graphen alle möglichen Berechnungspfade. Und ebenso ist mühelos sichtbar, dass wir einen Zustand haben, der ein „schwarzes Loch“ darstellt und in dem beide Threads keinen Fortschritt mehr machen können; sie sind verklemmt.

17.3 Temporale Logik

Um Transitionssysteme analysieren zu können, kommt *temporale Logik* ins Spiel. Man will Aussagen über die Zustände in den verschiedenen Berechnungspfaden von Programmen machen können. Ein solcher Pfad ist dann eine Folge von Zuständen. Die diskrete Zeit ist einfach die lineare Ordnung der Zustände entlang des Pfads. Man möchte dann Aussagen

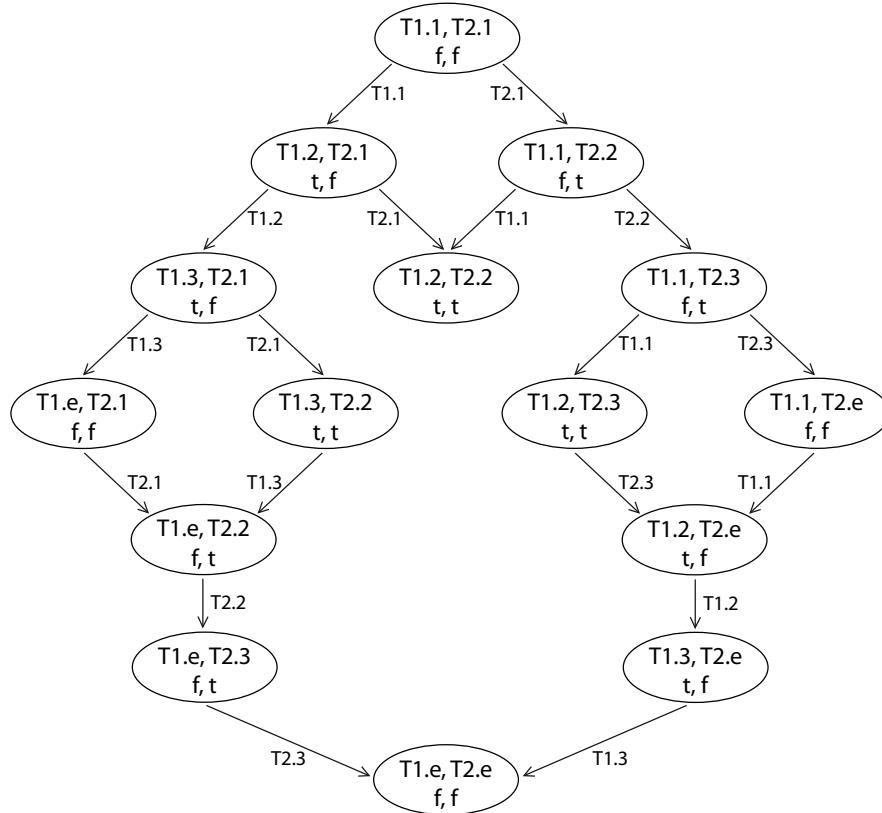


Abbildung 17.3: Transitionssystem für das Beispiel

machen können, wie „irgendwann gilt ...“ oder „es kann niemals sei, dass ...“.

Es gibt einen ganzen Zoo temporaler Logiken:

- *Lineare temporale Logik (LTL)* erlaubt Aussagen, die sich auf *alle* möglichen Pfade in einem Transitionssystem beziehen.
- *Computational Tree Logic (CTL)* hat Quantoren für die Pfade, d.h. man kann Formeln bilden wie „Für alle Pfade gilt ...“ oder „Es gibt einen Pfad, auf dem ...“.
- *CTL**: *LTL* und *CTL* haben unterschiedliche Ausdrucksstärke, *CTL** ist eine temporale Logik, die beider Ausdrucksstärke kombiniert.

Wir werden in diesem Teil der Veranstaltung *LTL* betrachten.

Kapitel 18

Die formale Sprache der linearen temporalen Logik (LTL)

In der Sprache der linearen temporalen Logik (LTL) erweitert man die formale Sprache der Aussagenlogik durch weitere Junktoren, mit denen temporale Eigenschaften formuliert werden können.

Definition 18.1 (Alphabet der LTL). Das *Alphabet* der Sprache der linearen temporalen Logik (LTL) besteht aus

- (i) einer Menge \mathcal{P} von Aussagensymbolen,
- (ii) den (aussagenlogischen) Junktoren: $\neg, \wedge, \vee, \rightarrow$
- (iii) den (temporalen) Junktoren: \circ, \mathcal{U}
- (iv) der Konstanten: \perp
- (v) den zusätzlichen Symbolen: $(,)$

Bemerkungen

- Wie im Falle der Aussagenlogik, definieren wir *eine* Sprache der linearen temporalen Logik durch die Vorgabe der Menge \mathcal{P} und der eben definierten Junktoren.
- Die formale Sprache der LTL ist eine Erweiterung der Aussagenlogik, in der zwei neue Junktoren vorkommen:
 - \circ steht für „zum nächsten Zeitpunkt“ (*next*)
 - \mathcal{U} steht für „bis“ (*until*)

Definition 18.2 (Formeln der LTL). Die *Formeln* der linearen temporalen Logik sind Zeichenketten, die nach folgenden Regeln gebildet werden:

- (i) Jedes Symbol $P \in \mathcal{P}$ ist eine Formel und auch \perp ist eine Formel.
- (ii) Ist φ eine Formel, dann auch $\neg\varphi$.
- (iii) Sind φ und ψ Formeln, dann auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\varphi \rightarrow \psi)$
- (iv) Ist φ eine Formel, dann auch $\circ \varphi$.
- (v) Sind φ und ψ Formeln, dann auch $(\varphi \mathcal{U} \psi)$

Als *Grammatik* in Backus-Naur-Darstellung können wir diese induktive Definition der Formeln der LTL so ausdrücken:

$$\varphi ::= P \mid \perp \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \circ \varphi \mid (\varphi \mathcal{U} \varphi)$$

mit Variablen $P \in \mathcal{P}$ und (bereits gebildeten) Formeln φ .

Die Präzedenz der Junktoren wird folgendermaßen definiert: Die unären Junktoren \neg und \circ binden stärker als die binären, sie selbst binden gleich stark. Binäre Junktoren binden in folgender Reihenfolge, die stärkste Bindung zuerst: $\mathcal{U}, \wedge, \vee, \rightarrow$. Außerdem sind \wedge und \vee linksassoziativ, \mathcal{U} und \rightarrow sind rechtsassoziativ.

Bemerkung

In Formeln der LTL werden wir oft vier weitere Junktoren verwenden, die folgendermaßen definiert werden:

$$\begin{aligned}\diamond \varphi &\stackrel{\text{def}}{=} \neg \perp \mathcal{U} \varphi \\ \Box \varphi &\stackrel{\text{def}}{=} \neg \diamond \neg \varphi \\ \varphi \mathcal{W} \psi &\stackrel{\text{def}}{=} (\varphi \mathcal{U} \psi) \vee \Box \varphi \\ \varphi \mathcal{R} \psi &\stackrel{\text{def}}{=} \neg (\neg \varphi \mathcal{U} \neg \psi)\end{aligned}$$

Wir lesen sie so:

- \diamond steht für „irgendwann“ (*eventually*),
- \Box steht für „immer“ (*always*),
- \mathcal{W} steht für „sofern nicht“ (*unless, weak until*) und
- \mathcal{R} steht für „löst ab“ (*release*).

\diamond und \Box haben dieselbe Bindungspräzedenz wie \circ und \mathcal{W} und \mathcal{R} die von \mathcal{U} .

Kapitel 19

Die Semantik der linearen temporalen Logik (LTL)

19.1 Kripke-Struktur

Modelle in der temporalen Logik enthalten ein implizites Konzept einer diskreten Zeit: Man denkt sich die „Welt“ des Modells als bestehend aus Zuständen, in denen gewissen Aussagen wahr sind und einer Übergangsrelation der Zustände. Jeder Zustandsübergang entspricht dann gerade einem Zeitschritt. Präziser definiert man die Kripke-Struktur¹:

Definition 19.1 (Kripke-Struktur). Eine *Kripke-Struktur* \mathcal{K} ist ein Tupel (S, s_0, \rightarrow, L) bestehend aus

- einer Menge von Zuständen S ,
- einem ausgezeichneten Startzustand $s_0 \in S$,
- einer Übergangsrelation $\rightarrow \subseteq S \times S$, die jedem Zustand s einen Folgezustand s' zuordnet (d.h. $\forall s \exists s' \text{ mit } s \rightarrow s'$) und
- einer Beschriftungsfunktion $L : S \rightarrow \mathbb{P}(\mathcal{P})$ von S in die Potenzmenge von \mathcal{P} , die jedem Zustand eine Menge von (wahren) Aussagenatomen zuordnet.

Bemerkungen

1. Man könnte in die Definition der Kripke-Struktur auch die Wahl von \mathcal{P} explizit aufnehmen. In den Beispielen, die wir betrachten, ergibt sich die Menge der Atome aus der Beschriftungsfunktion.
2. Die Beschriftungsfunktion L ordnet jedem Zustand die in diesem Zustand wahren Aussagen aus \mathcal{P} zu. Dies kann man auch so sehen: L ordnet jedem Zustand s eine Belegung $v_s : \mathcal{P} \rightarrow \mathbb{B}$ zu.

¹Saul A. Kripke (* 1940), amerikanischer Logiker.

3. Eine Kripke-Struktur kann man als gerichteten Graphen sehen, in dem die Zustände S die Knoten sind und die Übergangsrelation gerade die gerichteten Kanten. Zudem wird jeder Zustand mit den in ihm gültigen Aussagen gemäß der Beschriftungsfunktion L markiert. Der Startzustand wird durch einen eingehenden Pfeil ohne Startknoten markiert.
4. Manchmal zeichnet man in einer Kripke-Struktur keinen Startzustand aus, man bezeichnet sie dann als Übergangssystem (siehe [HR04, Abschnitt 3.2]).
5. Manche Autoren lassen in Kripke-Strukturen auch mehrere Startzustände zu.
6. Wenn in einem konkreten System die Übergangsrelation nicht die Eigenschaft hat, dass es zu jedem Zustand einen Folgezustand gibt, kann man den Graph um einen Zustand erweitern, der einen Übergang auf sich selbst hat und hat damit die Definition einer Kripke-Struktur erfüllt.

Beispiele In Abb. 19.1 und 19.2 werden die Graphen zu zwei Beispielen dargestellt.

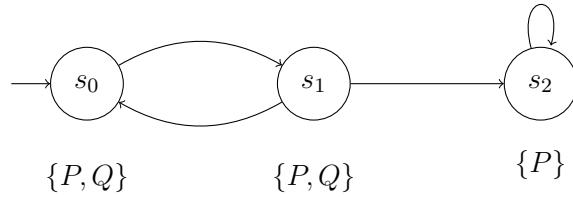


Abbildung 19.1: Beispiel einer Kripke-Struktur

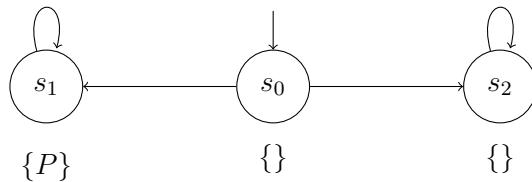


Abbildung 19.2: Beispiel für die Semantik der Negation

Definition 19.2 (Pfad und Berechnung). Sei $\mathcal{K} = (S, s_0, \rightarrow, L)$ eine Kripke-Struktur.

Ein *Pfad* π ist eine unendliche Folge s_1, s_2, s_3, \dots von Zuständen $s_i \in S$ mit $s_i \rightarrow s_{i+1}$ für alle $i \geq 1$. Man schreibt einen Pfad gerne so:

$$\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$$

Hat man einen Pfad $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \dots$ gegeben, dann bezeichnet man mit π^i den Pfad, der im i -ten Zustand von π beginnt, also z.B. $\pi^2 = s_2 \rightarrow s_3 \rightarrow s_4 \dots$

Eine *Berechnung* ist ein Pfad, der mit dem Startzustand $s_0 \in S$ beginnt.

Nun haben wir alle Notation, um Semantiken der LTL definieren zu können:

Definition 19.3 (Semantik der LTL für einen Pfad). Sei \mathcal{K} eine Kripke-Struktur und $\pi = s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$ ein Pfad. Für eine Formel φ der linearen temporalen Logik definiert man

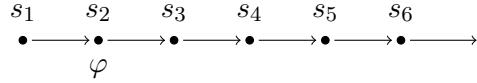
$$\pi \models \varphi,$$

falls $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T$. Dabei wird $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}}$ induktiv definiert über den strukturellen Aufbau von φ

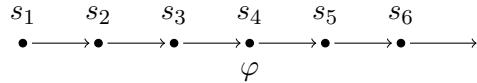
- (i) $\llbracket \perp \rrbracket_{\pi}^{\mathcal{K}} := F$
- (ii) $\llbracket P \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } P \in L(s_1) \\ F & \text{sonst} \end{cases}$
- (iii) $\llbracket \neg \varphi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = F \\ F & \text{sonst} \end{cases}$
- (iv) $\llbracket \varphi \wedge \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T \text{ und } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (v) $\llbracket \varphi \vee \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = T \text{ oder } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (vi) $\llbracket \varphi \rightarrow \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = F \text{ oder } \llbracket \psi \rrbracket_{\pi}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (vii) $\llbracket \circ \varphi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \llbracket \varphi \rrbracket_{\pi^2}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$
- (viii) $\llbracket \varphi U \psi \rrbracket_{\pi}^{\mathcal{K}} := \begin{cases} T & \text{falls } \exists i \geq 1 \text{ mit } \llbracket \psi \rrbracket_{\pi^i}^{\mathcal{K}} = T \\ & \text{und } \forall j=1, \dots, i-1 \llbracket \varphi \rrbracket_{\pi^j}^{\mathcal{K}} = T \\ F & \text{sonst} \end{cases}$

Veranschaulichung der Semantik der temporalen Operatoren der LTL

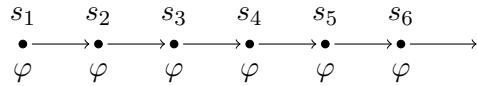
- $\Diamond \varphi$ bedeutet, dass φ im nächsten Zustand gilt:



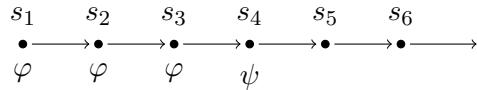
- $\Diamond \varphi$ bedeutet, dass φ irgendwann auf dem Pfad gilt:



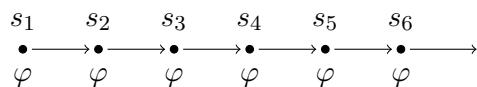
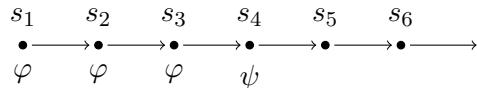
- $\Box \varphi$ bedeutet, dass φ immer auf dem Pfad gilt:



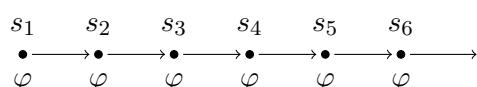
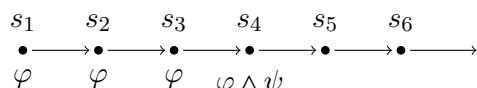
- $\varphi \mathcal{U} \psi$ bedeutet, dass ψ irgendwann auf dem Pfad gilt, und dass bis dahin auf jeden Fall φ wahr ist:



- $\varphi \mathcal{W} \psi$ bedeutet, dass φ gilt bis ψ gilt oder für immer, wenn ein solcher Zustand nicht existiert:



- $\varphi \mathcal{R} \psi$ bedeutet, dass φ gilt einschließlich dem ersten Zustand, in dem ψ gilt oder immer, wenn ein solcher Zustand nicht existiert:



Definition 19.4 (Semantik für Pfade, Zustände und Strukturen). Sei \mathcal{K} eine Kripke-Struktur. Es gilt dann

- Sei π ein *Pfad* über \mathcal{K} . Dann sagt man, dass π eine Formel φ erfüllt, geschrieben $\pi \models \varphi$, wenn gilt $\llbracket \varphi \rrbracket_{\pi}^{\mathcal{K}} = \top$.
- Sei s ein *Zustand* von \mathcal{K} . Dann sagt man, dass s eine Formel φ erfüllt, geschrieben $s \models \varphi$, wenn für alle Pfade π , die mit s beginnen, gilt $\pi \models \varphi$.
- Man sagt, dass die *Kripke-Struktur* \mathcal{K} eine Formel φ erfüllt, geschrieben $\mathcal{K} \models \varphi$, wenn für den Startzustand s_0 gilt: $s_0 \models \varphi$.

Beispiele Wenn wir zunächst das obige Beispiel 19.1 betrachten, ist leicht zu sehen, dass gilt:

- $\mathcal{K} \models \Box P$
denn in allen Zuständen ist P true.
- $s_0 \models \Diamond(P \vee Q)$
denn in s_1 gilt $P \vee Q$.
- $s_0 \models \Diamond(P \wedge Q)$
denn in s_1 gilt $P \wedge Q$.
- $s_1 \not\models \Diamond(P \wedge Q)$
denn zwar gilt in s_0 gilt $P \wedge Q$, nicht aber in s_2 .
- $\mathcal{K} \models \Box(\neg Q \rightarrow \Box(P \wedge \neg Q))$
denn der einzige Zustand mit $\neg Q$ ist s_2 , ab dann gilt aber immer $P \wedge \neg Q$.

An Beispiel 19.2 kann man sehen, dass obgleich für Pfade gilt $\pi \models \varphi \Leftrightarrow \pi \not\models \neg\varphi$ gilt, dies für Kripke-Strukturen nicht der Fall ist:

- $\mathcal{K} \not\models \Diamond P$
denn auf dem Pfad $s_0 \rightarrow s_2 \rightarrow s_2 \rightarrow \dots$ ist P niemals true.
- $\mathcal{K} \not\models \neg\Diamond P$
denn auf dem Pfad $s_0 \rightarrow s_1 \rightarrow s_1 \rightarrow \dots$ ist P schließlich true.

Bemerkung Die Semantik der linearen temporalen Logik unterscheidet sich grundlegend je nach Definition. Betrachtet man die Semantik der LTL für Pfade, dann gilt der Satz vom ausgeschlossenen Dritten, d.h. $\models_{\pi} \varphi \vee \neg\varphi$ für eine beliebige Formel φ der LTL. Für die Semantik der LTL für Kripke-Strukturen ist dies nicht der Fall. Obiges Beispiel zeigt, dass es Kripke-Strukturen geben kann, in denen weder φ noch $\neg\varphi$ gilt. Das liegt daran, dass es in einer Kripke-Struktur Pfade π_1 und π_2 geben kann, für die $\pi_1 \models \varphi$ und $\pi_2 \models \neg\varphi$ gilt.²

²Dass der Satz vom ausgeschlossenen Dritten (*Tertium non datur*) *nicht* richtig ist, gilt auch für die Kripke-Semantik der intuitionistischen Logik. Diese Semantik

Für die Definition von Erfüllbarkeit, Allgemeingültigkeit und semantischer Äquivalenz wird die Pfad-Semantik zugrundegelegt.

Definition 19.5 (Erfüllbarkeit, Allgemeingültigkeit).

- Eine Formel φ heißt *erfüllbar*, wenn es eine Kripke-Struktur gibt mit einem Pfad π so dass gilt $\pi \models \varphi$.
- Eine Formel φ heißt *allgemeingültig*, wenn für alle Pfade in allen Kripke-Strukturen gilt: $\pi \models \varphi$.

Definition 19.6 (Semantische Äquivalenz). Zwei Formeln φ und ψ sind *semantisch äquivalent*, geschrieben $\varphi \equiv \psi$, wenn für alle Pfade π gilt: $\pi \models \varphi \Leftrightarrow \pi \models \psi$.

19.2 Äquivalenzen von Formeln der LTL

- Dualität

$$\begin{aligned}\neg \circ \varphi &\equiv \circ \neg \varphi \\ \neg \diamond \varphi &\equiv \square \neg \varphi \\ \neg \square \varphi &\equiv \diamond \neg \varphi\end{aligned}$$

- Idempotenz

$$\begin{aligned}\diamond \diamond \varphi &\equiv \diamond \varphi \\ \square \square \varphi &\equiv \square \varphi \\ \varphi \mathcal{U} (\varphi \mathcal{U} \psi) &\equiv \varphi \mathcal{U} \psi \\ (\varphi \mathcal{U} \psi) \mathcal{U} \psi &\equiv \varphi \mathcal{U} \psi\end{aligned}$$

- Absorption

$$\begin{aligned}\diamond \square \diamond \varphi &\equiv \square \diamond \varphi \\ \square \diamond \square \varphi &\equiv \diamond \square \varphi\end{aligned}$$

wird über Kripke-Strukturen definiert, die zusätzlich zu unserer Definition die Eigenschaft der *Monotonie* haben. Diese Eigenschaft bedeutet, dass in jedem Folgezustand s' eines beliebigen Zustands s gilt: $L(s) \subseteq L(s')$. D.h. also, dass mit jedem Zustandsübergang mehr atomare Aussagen wahr werden.

Die Semantik der Junktoren der intuitionistischen Aussagenlogik werden dann in den Begriffen der LTL so definiert:

- $P \wedge Q \stackrel{\text{def}}{=} \mathcal{K} \models P \wedge Q$
- $P \vee Q \stackrel{\text{def}}{=} \mathcal{K} \models P \vee Q$
- $\neg P \stackrel{\text{def}}{=} \mathcal{K} \models \neg \diamond P$
- $P \rightarrow Q \stackrel{\text{def}}{=} \mathcal{K} \models \square(P \rightarrow Q)$

Siehe [Bor05, Chap. 9] sowie [vD13, Section 6.3].

- Expansion

$$\begin{aligned}\varphi \mathcal{U} \psi &\equiv \psi \vee (\varphi \wedge \circ(\varphi \mathcal{U} \psi)) \\ \diamond \varphi &\equiv \varphi \vee \circ \diamond \varphi \\ \square \varphi &\equiv \varphi \wedge \circ \square \varphi\end{aligned}$$

- Distributiv-Gesetze

$$\begin{aligned}\diamond(\varphi \vee \psi) &\equiv \diamond \varphi \vee \diamond \psi \\ \square(\varphi \wedge \psi) &\equiv \square \varphi \wedge \square \psi \\ \circ(\varphi \mathcal{U} \psi) &\equiv \circ \varphi \mathcal{U} \circ \psi\end{aligned}$$

19.3 Typische Aussagen in der LTL

Die lineare temporale Logik eignet sich besonders gut dafür, Eigenschaften von Programmen, insbesondere mit Nebenläufigkeit, auszudrücken. In diesem Abschnitt sollen die wichtigsten dieser Eigenschaften vorgestellt und als Formel in der LTL ausgedrückt werden.

Als Beispiel für die Formulierung der Eigenschaften werden wir eine Ampelschaltung an einer einspurigen Baustelle verwenden, wir verwenden zwei Ampeln, die sich verhalten wie die Ampel in Abbildung 17.1. Als zweites Beispiel stellen wir uns ein Programm mit einem kritischen Abschnitt (*critical section*) vor, in dem wechselseitiger Ausschluss zweier Transaktionen, Prozesse oder Threads erforderlich ist.

19.3.1 Sicherheitseigenschaft

Eine *Sicherheitseigenschaft* φ ist erfüllt, wenn sie in jedem Zustand jedes Berechnungspfads eines Programms erfüllt ist:

$$\square \varphi$$

Oder mit Negation informell ausgedrückt: „etwas Unerwünschtes darf niemals passieren“, also $\neg \diamond \neg P$.

Für das Beispiel der beiden Ampeln A und B sind Sicherheitseigenschaften:

$$\begin{aligned}\square \neg(Rot_A \wedge Grün_A) \\ \square \neg(Grün_A \wedge Grün_B)\end{aligned}$$

Wenn wir Symbole $Crit_1$ und $Crit_2$ für den Aufenthalt zweier Prozesse in kritischen Abschnitt $Crit$ nehmen, dann wird die Sicherheitseigenschaft des wechselseitigen Ausschlusses so formuliert:

$$\square(\neg Crit_1 \vee \neg Crit_2)$$

19.3.2 Lebendigkeitseigenschaft

Eine *Lebendigkeitseigenschaft* φ ist erfüllt, wenn sie schließlich eintritt.

$$\Diamond \varphi$$

Oder informell ausgedrückt: „Etwas Erwünschtes wird passieren“.

Lebendigkeitseigenschaften an unseren Beispielen:

$$\begin{aligned} \Diamond Grün_A \wedge \Diamond Grün_B \\ \Diamond Crit_1 \wedge \Diamond Crit_2 \end{aligned}$$

19.3.3 Fairness

Während man Sicherheits- und Lebendigkeitseigenschaften offensichtlich auch für eine einzelnen Prozess (ohne Nebenläufigkeit) formulieren kann, geht es bei der Fairness darum, dass es fair dabei zugeht, wenn mehrere Prozesse darum konkurrieren, als nächstes einen Schritt ausführen zu können.

Im Beispiel des wechselseitigen Ausschlusses zweier Prozesse nehmen wir an, dass binäre Semaphoren verwendet werden und wir geben der Aktion, bei der ein Prozess darauf wartet, den kritischen Abschnitt zu betreten, das Aussagensymbol $Wait_1$ bzw. $Wait_2$ (siehe [?, Abschnitt 2.2]).

Im Allgemeinen soll ψ die Formel bezeichnen, die ausdrückt, dass versucht wird φ zu erreichen.

Man unterscheidet drei Formen der Fairness:

Unbedingte Fairness

Unbedingte Fairness ist gegeben, wenn φ immer wieder erfüllt ist:

$$\square \Diamond \varphi$$

Oder informell ausgedrückt: „Etwas kommt immer wieder dran“.

In unseren Beispielen:

$$\begin{aligned}\square \diamond Grün_A \wedge \square \diamond Grün_B \\ \square \diamond Crit_1 \wedge \square \diamond Crit_2\end{aligned}$$

Schwache Fairness

Schwache Fairness ist gegeben, wenn schließlich ψ immer gilt, dass dann auch φ immer wieder erfüllt ist:

$$\diamond \square \psi \rightarrow \square \diamond \varphi$$

Oder informell ausgedrückt: „Wenn schließlich etwas immerzu versucht wird, dann wird es unendlich oft gelingen“.

Im Beispiel des wechselseitigen Ausschlusses:

$$(\diamond \square Wait_1 \rightarrow \square \diamond Crit_1) \wedge (\diamond \square Wait_2 \rightarrow \square \diamond Crit_2)$$

Starke Fairness

Starke Fairness ist gegeben, wenn immer wieder ψ gilt, dass dann auch φ immer wieder erfüllt ist:

$$\square \diamond \psi \rightarrow \square \diamond \varphi$$

Oder informell ausgedrückt: „Wenn etwas wieder mal versucht wird, dann wird es unendlich oft gelingen“.

Im Beispiel des wechselseitigen Ausschlusses:

$$(\square \diamond Wait_1 \rightarrow \square \diamond Crit_1) \wedge (\diamond \square Wait_2 \rightarrow \square \diamond Crit_2)$$

19.4 Büchi-Automaten

Formeln der linearen temporalen Logik und Kripke-Strukturen hängen eng zusammen mit Büchi³-Automaten. Deshalb soll es in diesem Abschnitt um Büchi-Automaten gehen.

19.4.1 Automaten und unendliche Wörter

Büchi-Automaten sind endliche Automaten, bei denen es darum geht, dass sie *unendliche* Folgen von Symbolen erkennen. Dabei verwendet man für die Akzeptanz für einen unendlichen Lauf auf dem endlichen Automaten die sogenannte *Büchi-Bedingung*, die besagt, dass der Lauf akzeptiert wird, wenn er unendlich oft akzeptierende Zustände durchläuft.

³nach **Juliis Richard Büchi** (1924–1984), Schweizer Logiker und Mathematiker.

Definition 19.7. Ein *Büchi-Automat* ist ein Tupel $\mathcal{A} = (\Sigma, Q, \Delta, q_I, F)$ mit

- Σ ist ein endliches *Alphabet*,
- Q ist eine endliche Menge von *Zuständen*,
- $\Delta \subseteq Q \times \Sigma \times Q$ ist die *Transitionsrelation*,
- q_I ist ein ausgezeichneter *Initialzustand* und
- $F \subseteq Q$ ist eine ausgezeichnete Menge *akzeptierender Zustände*.

Sei v ein unendliche Folge von Symbolen des Alphabets, d.h. $v = a_0a_1a_2\dots$ mit $a_i \in \Sigma$. Man schreibt dann $v \in \Sigma^\omega$ und bezeichnet die Folge als ω -Wort. Ein *Lauf* von \mathcal{A} auf einem Wort v ist eine unendliche Folge von Zuständen $\rho = q_0q_1q_2\dots$ beginnend mit dem Initialzustand q_I und die Transitionsrelation respektierend, d.h. für alle Paare von aufeinanderfolgenden Zuständen gilt $(q_i, a_i, q_{i+1}) \in \Delta$.

Mit $Inf(\rho)$ bezeichnet man die Menge der Zustände, die unendlich oft im Lauf ρ vorkommen. Weil der Lauf unendlich ist, der Automat aber nur endlich viele Zustände hat, muss $Inf(\rho) \neq \emptyset$ gelten.

Ein Lauf ρ ist *akzeptierend*, wenn $Inf(\rho) \cup F \neq \emptyset$ ist, also wenn mindestens ein akzeptierender Zustand unendlich oft durchlaufen wird.

Die Sprache $L(\mathcal{A})$ des Büchi-Automaten \mathcal{A} ist die Menge aller ω -Wörtern, für die es einen akzeptierenden Lauf gibt.

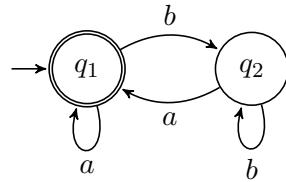


Abbildung 19.3: Beispiel eines Büchi-Automaten

Beispiel 19.1. Abbildung 19.3 zeigt einen Büchi-Automaten, der alle Worte über dem Alphabet $\{a, b\}$ akzeptiert, die unendliche viele as enthalten, wie etwa das Wort $(ab)^\omega$, die unendliche alternierende Sequenz der beiden Buchstaben. Die Sprache, die der Automat akzeptiert, wird durch den ω -regulären Ausdruck $(b^*a)^\omega$ beschrieben.

Zu einer Kripke-Struktur kann man einen Büchi-Automaten auf kanonische Weise finden. Die Zustände sind gerade die Zustände der Kripke-Struktur und die Übergänge in einen Zustand sind genau mit den Teilmengen der Potenzmenge der Atome in der Kripke-Struktur markiert,

die in diesem Zustand gelten müssen. Da die Semantik der LTL in einer Kripke-Struktur so definiert ist, dass eine Formel auf allen Berechnungspfaden gelten muss, sind im korrespondierenden Büchi-Automaten alle Zustände akzeptierend sind. Präzise:

Sei \mathcal{K} die Kripke-Struktur (S, s_o, \rightarrow, L) dann korrespondiert dazu der Büchi-Automat \mathcal{A} mit

- dem Alphabet $\Sigma = \mathbb{P}((P))$, der Potenzmenge der Menge der Aussagensymbole P der Kripke-Struktur,
- der Menge der Zustände $S \cap \{\iota\}$, wobei ι ein zusätzlicher Zustand ist, der für den Eintritt in den Startzustand der Kripkestruktur steht,
- der Transitionsrelation Δ mit

$$(s, \alpha, s') \in \Delta \text{ für } s, s' \in S \Leftrightarrow (s, s') \in \rightarrow \text{ und } \alpha = L(s)$$

$$(\iota, \alpha, s) \in \Delta \Leftrightarrow s = s_o \text{ und } \alpha = L(s_o)$$

- dem Initialzustand $q_I = \iota$ und
- alle Zuständen $F = S \cap \{\iota\}$ als akzeptierende Zustände.

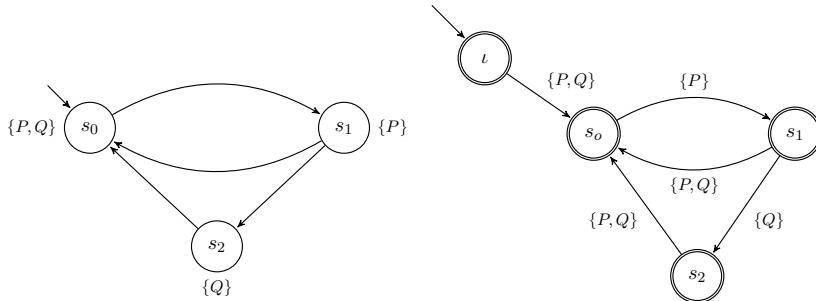


Abbildung 19.4: Kripke-Struktur und entsprechender Büchi-Automat

Beispiel 19.2. Abbildung 19.4 zeigt links eine Kripke-Struktur und daneben den entsprechenden Büchi-Automaten.

Wesentlich ist nun die Beobachtung, dass einer Formel φ , die in der Kripke-Struktur erfüllt ist, gerade die Läufe entsprechen, die vom korrespondierenden Büchi-Automat akzeptiert werden. Wenn man im Beispiel von Abbildung 19.4 die Formel $\square \diamond P$ betrachtet, so ist sie in der Kripke-Struktur wahr, denn auf allen Berechnungspfaden wird immer wieder der Zustand s_1 erreicht, in dem $P = T$ gilt. Wenn man nun andererseits einen beliebigen Lauf im korrespondierenden Büchi-Automat betrachtet, dann ist der Zustand s_1 ein akzeptierender Zustand, der in jedem möglichen Lauf unendlich oft durchlaufen wird, das bedeutet,

dass in einem Lauf immer wieder der Übergang vorkommt, der mit $\{P\}$ markiert ist.

19.4.2 Eigenschaften von Büchi-Automaten

Man unterscheidet zwischen *deterministischen* und *nichtdeterministischen* Büchi-Automaten. In der Definition 19.7 können in der Transitionsrelation Δ zwei Tupel (q, a, r) und (q, a, r') vorkommen, was bedeutet dass Nichtdeterminismus in der Definition erlaubt ist. Wenn es zu einem Buchstaben in jedem Zustand nur einen möglichen Übergang gibt, dann ist der Automat deterministisch.

Handelt es sich um endliche Worte und endliche Automaten, dann kann zu jedem nichtdeterministischen Automat ein deterministischer Automat konstruiert werden, der dieselbe Sprache erkennt. Bei Büchi-Automaten ist dies nicht so [HL11, Abschnitt 1.1.3 und 5.2].

Büchi-Automaten sind abgeschlossen bezüglich der Bildung des Durchschnitts. Das bedeutet, dass es einen Büchi-Automaten \mathcal{A} gibt, der exakt den Durchschnitt der Sprachen zweier Automaten $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ akzeptiert. Der Büchi-Automat \mathcal{A} wird konstruiert als sogenanntes *synchronisiertes Produkt* von \mathcal{A}_1 und \mathcal{A}_2 , siehe [HL11, Satz 5.14].

Hat man zwei Büchi-Automaten $\mathcal{A}_1 = (\Sigma, Q_1, \Delta_1, q_1, F_1)$ und $\mathcal{A}_2 = (\Sigma, Q_2, \Delta_2, q_2, F_2)$ gegeben, dann definiert man das synchronisierte Produkt $\mathcal{A}_1 \cap \mathcal{A}_2$ folgendermaßen:

- $\mathcal{A}_1 \cap \mathcal{A}_2 = (\Sigma, Q_1 \times Q_2 \times \{0, 1, 2\}, \Delta, \langle q_1, q_2, 0 \rangle, F_1 \times F_2 \times \{2\})$
- $(\langle q_i, q_j, z \rangle a \langle q_k, q_l, z' \rangle) \in \Delta \Leftrightarrow (q_i, a, q_k) \in \Delta_1 \text{ und } (q_j, a, q_l) \in \Delta_2$
und für z und z' gilt
 - $z = 0 \wedge q_k \in F_1 \Rightarrow z' = 1$
 - $z = 1 \wedge q_l \in F_2 \Rightarrow z' = 2$
 - $z = 2 \Rightarrow z' = 0$
 - sonst $z' = z$

Die Idee besteht darin, dass man die beiden Automaten gewissermaßen parallel durchläuft und durch die Verbindungen sicherstellt, dass akzeptierende Zustände von *beiden* Automaten unendlich oft durchlaufen werden müssen.

Diese Idee wird in Abbildung 19.5 illustriert. Die beiden Automaten erkennen Läufe mit unendlich vielen *as*, bzw. *bs*. Das synchronisierte Produkt ist ein Automat für unendlich viele *as* und *bs*. In der Abbildung von $\mathcal{A}_1 \cap \mathcal{A}_2$ sind die Zustände, die gar nicht erreicht werden können, nicht eingezzeichnet.

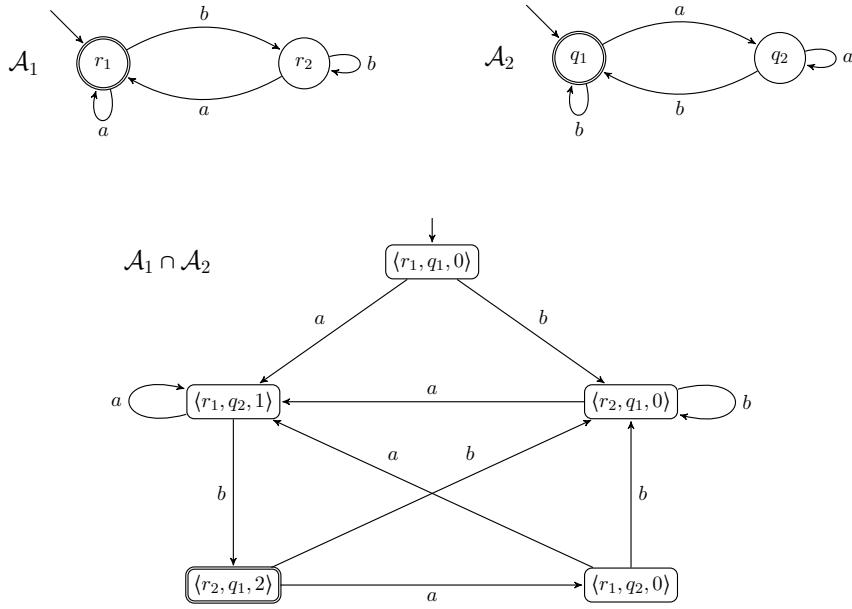


Abbildung 19.5: Beispiel für das synchronisierte Produkt

Nicht-deterministische Büchi-Automaten sind auch abgeschlossen unter der Komplementbildung. Das Komplement zum Automaten \mathcal{A} ist ein Büchi-Automat, der die Sprache $\overline{\mathcal{L}(\mathcal{A})} = \Sigma^\omega - \mathcal{L}(\mathcal{A})$ akzeptiert. In [HL11, Abschnitt 6.2] wird zu einem Büchi-Automat \mathcal{A} ein Automat \mathcal{A}' konstruiert mit $\mathcal{L}(\mathcal{A}') = \overline{\mathcal{L}(\mathcal{A})}$.

19.4.3 Das Leerheitsproblem für Büchi-Automaten

Die Frage, ob die von einem Automaten erkannte Sprache leer ist, also ob $\mathcal{L}(\mathcal{A}) = \emptyset$ gilt, nennt man das *Leerheitsproblem*.

Ein gerichteter Graph heißt *stark zusammenhängend*, wenn es zwischen allen Knoten einen gerichteten Weg gibt. Anders ausgedrückt, jedes beliebige Paar von Knoten im Graphen ist in der reflexiv-transitiven Hülle des Graphen enthalten. Eine *starke Zusammenhangskomponente* ist ein Teilgraph, der stark zusammenhängend ist.

Die von einem Büchi-Automaten erkannte Sprache ist nicht leer, wenn es einen starken Zusammenhangskomponenten gibt, die erreichbar ist und einen akzeptierenden Zustand enthält. Das bedeutet dasselbe wie: es gibt einen erreichbaren akzeptierenden Zustand, der Teil eines Zyklus ist.

Üblicherweise werden die starken Zusammenhangskomponenten eines

Graphen mit dem Algorithmus von Tarjan⁴ ermittelt. In [CGP99, Abschnitt 9.3] wird ein Algorithmus beschrieben, der in einem Büchi-Automaten Zyklen mit einem akzeptierenden Zustand findet.

19.4.4 Verallgemeinerte Büchi-Automaten

Oft ist es praktisch mit verallgemeinerten Büchi-Automaten zu arbeiten.

Definition 19.8. Ein *verallgemeinerter Büchi-Automat* ist ein Tupel $(A) = (\Sigma, Q, \Delta, I, F_1, F_2, \dots, F_k)$ wie bereits definiert, jedoch mit einer Menge $I \subseteq Q$ von Initialzuständen und mehreren Mengen $F_1, F_2, \dots, F_k \subseteq Q$ akzeptierender Zustände.

Ein Lauf eines verallgemeinerten Büchi-Automaten auf einem unendlichen Wort kann im Unterschied zu den bisher betrachteten Automaten in einem beliebigen Zustand aus der Menge I beginnen. Ein Lauf q_0, q_1, \dots heißt akzeptierend, wenn es für alle $i = 1, 2, \dots, k$ ein $q \in F_i$ gibt, so dass für unendlich viele j gilt: $q = q_j$.

Ein verallgemeinerter Büchi-Automat hat also mehrere Initialzustände und mehrere Mengen akzeptierender Zustände. Die Akzeptanzbedingung besteht darin, dass jede dieser Mengen unendlich oft besucht werden muss.

Man kann einen verallgemeinerten Büchi-Automaten leicht zu einem Büchi-Automaten mit einem Initialzustand und einer einzigen Menge von akzeptierenden Zuständen machen.

Gegeben sei $\mathcal{A} = (\Sigma, Q, \Delta, I, F_1, \dots, F_k)$. Man konstruiert daraus $\mathcal{A}' = (\Sigma, Q \times \{0, 1, \dots, k\}, \Delta', I \times \{0\}, Q \times \{k\})$ mit folgender Transitionsrelation Δ' :

$((q, z), a, (q', z')) \in \Delta'$, wenn $(q, a, q') \in \Delta$ ist und für z und z' gilt

- Ist $q' \in F_i$ und $z = i - 1$, dann ist $z' = i$.
- Ist $z = k$, dann ist $z' = 0$.
- In allen anderen Fällen ist $z' = z$.

Die Idee besteht darin, aus den bisherigen Zuständen Paare mit einem zusätzlichen Zähler z zu machen. Dieser Zähler gibt an, aus welcher der ursprünglichen Menge akzeptierender Zustände als nächstes ein akzeptierender Zustand durchlaufen werden soll.

⁴Robert Tarjan, amerikanischer Informatiker.

Um zu einem einzigen Initialzustand zu kommen, erweitert man den Automaten durch einen neuen Initialzustand, der zu jedem der bisherigen Initialzuständen führt.

19.5 Erfüllbarkeit in der LTL

Wie wir gleich sehen werden, ist es möglich zu einer Formel der linearen temporalen Logik einen Büchi-Automaten zu konstruieren, der die Eigenschaft hat, dass die von diesem Automaten erkannte Sprache genau den Berechnungspfaden entspricht, die die Formel erfüllen.

Um die Erfüllbarkeit einer Formel der LTL zu entscheiden, konstruiert man also zu dieser Formel den korrespondierenden Büchi-Automat. Wenn dieser Automat nicht leer ist, ist die Formel erfüllbar.

Um eine Kripke-Struktur zu finden, in der die Formel wahr wird, nimmt man einen Pfad im Büchi-Automaten mit einem Zyklus, der eine akzeptierenden Zustand enthält. Man kann dann den Büchi-Automaten als Kripke-Struktur interpretieren, indem man aus den Aussagen, die die Übergänge des Automaten markieren, die Aussagen ermittelt, die im Zielzustand zutreffen müssen.

19.5.1 Negationsnormalform (NNF) in der LTL

Für die Konstruktion eines Büchi-Automaten zu einer Formel ist es nötig, in der Formel folgende Junktoren zu erlauben: $\neg, \wedge, \vee, \circ, \mathcal{U}, \mathcal{R}$ sowie die beiden Konstanten \perp und \top .

Definition 19.9. Eine Formel φ der LTL ist in *Negationsnormalform*, wenn der Junktor \neg nur in Literalen vorkommt.

Um die Negationsnormalform von φ zu erreichen, müssen wir zunächst die Implikation beseitigen. Dazu eignet sich die Funktion `IMPL_FREE` aus Abschnitt 6.1.

Ist die Formel ohne Implikationen, erhält man die Negationsnormalform mit folgender Funktion:

```
function NNFLTL( $\varphi$ ) {
    // pre:  $\varphi$  hat keine Implikationen
    // post: äquivalente Umformung von  $\varphi$  in NNF
    case {
         $\varphi$  ist Literal oder eine Konstante:
            return  $\varphi$ ;
         $\varphi$  hat die Form  $\neg\neg\varphi_1$ :
```

```

        return NNFLTL(φ1);
    φ hat die Form φ1 ∧ φ2:
        return NNFLTL(φ1) ∧ NNFLTL(φ2);
    φ hat die Form φ1 ∨ φ2:
        return NNFLTL(φ1) ∨ NNFLTL(φ2);
    φ hat die Form ¬(φ1 ∧ φ2):
        return NNFLTL(¬φ1) ∨ NNFLTL(¬φ2);
    φ hat die Form ¬(φ1 ∨ φ2):
        return NNFLTL(¬φ1) ∧ NNFLTL(¬φ2);
    φ hat die Form ¬○φ1:
        return ○NNFLTL(¬φ1);
    φ hat die Form ¬(φ1 U φ2):
        return NNFLTL(¬φ1) R NNFLTL(¬φ2);
    φ hat die Form ¬(φ1 R φ2):
        return NNFLTL(¬φ1) U NNFLTL(¬φ2);
    }
}

```

19.5.2 Von LTL zum Büchi-Automat

In diesem Abschnitt wird der Algorithmus von Gerth, Peled, Vardi und Wolper [GPVW95] vorgestellt, der eine Formel der LTL in Negationsnormalform in einen Büchi-Automaten transformiert.

Die Grundidee der Konstruktion des Automaten aus dem Formel besteht darin, dass die Formel von ihrer Struktur her so zerlegt wird, dass sich ein Teil der Aussage auf den aktuellen Zustand und ein anderer Teil der Aussage auf den nächsten Zustand bezieht. Dieses Vorgehen erklärt auch, weshalb in der Negationsnormalform gerade die binären Junktoren U und R gewählt wurden. Denn für diese beiden Operatoren hat man die folgenden Expansionsstrategien, auch *Abwicklung* genannt:

$$\begin{aligned}\varphi_1 U \varphi_2 &\leftrightarrow \varphi_2 \vee (\varphi_1 \wedge ○(\varphi_1 U \varphi_2)) \\ \varphi_1 R \varphi_2 &\leftrightarrow \varphi_2 \wedge (\varphi_1 \vee ○(\varphi_1 R \varphi_2))\end{aligned}$$

Für den Algorithmus wird die Datenstruktur $node = [id, incoming, old, new, next]$ verwendet, wobei

- id ist eine eindeutige Id des Knotens.
- $incoming$ ist eine Liste von Ids von Knoten, die eine Kante zu diesem Knoten haben.

- old enthält die Teilformeln von φ , die bereits vom Algorithmus berechnet wurden.
- new enthält die Teilformeln von φ , die im aktuellen Schritt berechnet werden.
- $next$ enthält die Teilformeln, die im nächsten Schritt berechnet werden müssen.

Als Hilfsfunktion wird die Funktion `NEW_ID()` verwendet, die bei jedem Aufruf eine frische Id generiert.

Der Algorithmus wird initialisiert mit einem Knoten $node_0$, der eine Kante von einem speziellen Zustand namens *init* hat, welcher später der Initialzustand des Automaten werden wird:

```
 $node_0 := [id := NEW_ID(),$ 
 $incoming := \{init\},$ 
 $old := \emptyset,$ 
 $new := \{\varphi\},$ 
 $next := \emptyset]$ 
```

Die rekursive Funktion $nodes' = \text{EXPAND}(q, nodes)$ bearbeitet den Knoten q und die bisherige Liste $nodes$ von bereits konstruierten Knoten. Sie gibt eine Liste von Knoten zurück. D.h. der Graph wird erzeugt durch $\text{expand}(node_0, \emptyset)$.

In der Funktion $\text{expand}(q, nodes)$ sind folgende Möglichkeiten zu unterscheiden:

- Das Feld new von q ist leer.
 - Es gibt einen Knoten $r \in nodes$ mit $r.old = q.old$ und $r.next = q.next$
dann tritt r an die Stelle von q , d.h.
 $r.incoming = r.incoming \cup q.incoming$
 $\text{return } nodes$.
 - andernfalls muss man einen neuen Knoten einfügen:

```
 $node_{new} := [id := NEW_ID(),$ 
 $incoming := \{q.id\},$ 
 $old := \emptyset,$ 
 $new := q.next,$ 
 $next := \emptyset]$ 
```

Die Rekursion geht weiter mit $\text{EXPAND}(node_{new}, nodes \cup \{q\})$.

- $q.new$ ist nicht leer.

Dann nehmen wir eine Formel $\psi \in q.new$ und setzen

$$q.new := q.new - \psi.$$

- Es kann sein, dass $\psi \in q.old$ gilt
Die Rekursion geht weiter mit $\text{EXPAND}(q, nodes)$
- Es kann sein, dass $\neg\psi \in q.old$ ist oder $\psi = \perp$
Wir sind auf einen Widerspruch gestoßen, d.h. der aktuelle Knoten muss verworfen werden:
`return nodes.`
- andernfalls werden Knoten q' bzw. q_1 und q_2 erzeugt, je nach Struktur der Formel ψ und die Rekursion geht dann weiter mit
 $\text{EXPAND}(q', nodes)$ bzw. $\text{EXPAND}(q_2, \text{EXPAND}(q_1, nodes))$

Nun müssen wir noch sehen, wie im letzteren Fall der Knoten q' gebildet wird oder unser Knoten gespalten wird in q_1 und q_2 . Dazu müssen wir uns die Struktur der Formel ansehen:

- ψ ist ein Literal oder $\psi = \top$:

$$\begin{aligned} q' := & [id := \text{NEW_ID}(), \\ & incoming := q.incoming, \\ & old := q.old \cup \{\psi\}, \\ & new := q.new, \\ & next := q.next] \end{aligned}$$

- ψ hat die Form $\psi = \psi_1 \vee \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ incoming := q.incoming, & incoming := q.incoming, \\ old := q.old \cup \{\psi\}, & old := q.old \cup \{\psi\}, \\ new := q.new \cup \{\psi_1\}, & new := q.new \cup \{\psi_2\}, \\ next := q.next] & next := q.next] \end{array}$$

- ψ hat die Form $\psi = \psi_1 \wedge \psi_2$:

$$\begin{aligned} q' := & [id := \text{NEW_ID}(), \\ & incoming := q.incoming, \\ & old := q.old \cup \{\psi\}, \\ & new := q.new \cup \{\psi_1, \psi_2\}, \\ & next := q.next] \end{aligned}$$

- ψ hat die Form $\psi = \psi_1 \mathcal{U} \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ \quad incoming := q.incoming, & \quad incoming := q.incoming, \\ \quad old := q.old \cup \{\psi\}, & \quad old := q.old \cup \{\psi\}, \\ \quad new := q.new \cup \{\psi_1\}, & \quad new := q.new \cup \{\psi_2\}, \\ \quad next := q.next \cup \{\psi_1 \mathcal{U} \psi_2\}] & \quad next := q.next] \end{array}$$

- ψ hat die Form $\psi = \psi_1 \mathcal{R} \psi_2$:

$$\begin{array}{ll} q_1 := [id := \text{NEW_ID}(), & q_2 := [id := \text{NEW_ID}() \\ \quad incoming := q.incoming, & \quad incoming := q.incoming, \\ \quad old := q.old \cup \{\psi\}, & \quad old := q.old \cup \{\psi\}, \\ \quad new := q.new \cup \{\psi_1, \psi_2\}, & \quad new := q.new \cup \{\psi_2\}, \\ \quad next := q.next] & \quad next := q.next \cup \{\psi_1 \mathcal{R} \psi_2\}] \end{array}$$

- ψ hat die Form $\psi = \circ \psi_1$:

$$\begin{array}{l} q' := [id := \text{NEW_ID}(), \\ \quad incoming := q.incoming, \\ \quad old := q.old \cup \{\psi\}, \\ \quad new := q.new, \\ \quad next := q.next \cup \{\psi_1\}] \end{array}$$

Beispiel 19.3. Als Beispiel wollen wir den Algorithmus mit der Formel $\varphi = (P \mathcal{U} (Q \wedge R))$ durchspielen:

Der Algorithmus beginnt mit dem initialen Knoten n_0 , der die Formel selbst repräsentiert: $[0, \{init\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}, \{\}]$.

$\begin{array}{ccccccc} id & incoming & old & new & & & next \\ 0 & & & & & & \end{array}$

Der Ergebnisgraph ist leer: $nodes = \{\}$.

Schritt 1: Die zu bearbeitende Formel hat \mathcal{U} als Haupt-Junktor, d.h. es entstehen zwei neue Knoten:

$$n_1 = [1, \{init\}, \{P \mathcal{U} (Q \wedge R)\}, \{P\}, \{P \mathcal{U} (Q \wedge R)\}].$$

$$n_2 = [2, \{init\}, \{P \mathcal{U} (Q \wedge R)\}, \{Q \wedge R\}, \{\}].$$

$$n_2 = [2, \{init\}, \{P \mathcal{U} (Q \wedge R)\}, \{Q \wedge R\}, \{\}].$$

Nun muss man erst n_1 rekursiv expandieren und dann mit dem dadurch entstehenden Ergebnisgraph mit der Expansion von n_2 in Schritt 13 fortfahren.

Schritt 2: Die zu bearbeitende Formel kommt nicht in $n_1.old$ vor. Sie ist ein Literal, somit muss ein neuer Knoten erzeugt werden:

$$n_3 = [3, \{init\}, \{P \mathcal{U} (Q \wedge R), P\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}].$$

$$n_3 = [3, \{init\}, \{P \mathcal{U} (Q \wedge R), P\}, \{\}, \{P \mathcal{U} (Q \wedge R)\}].$$

Schritt 3: Nun muss n_3 verarbeitet werden. $n_3.new$ ist leer und $nodes$ ist bisher auch leer, also muss man einen neuen Knoten einfügen:

$$n_4 = [4, \{n_3\}, \{\}, \{P \cup (Q \wedge R)\}, \{\}] \text{ und } nodes = \{init, n_3\}.$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 4: Dieser neue Knoten muss jetzt wieder für die Abwicklung des \cup geteilt werden: $n_5 = [5, \{n_3\}, \{P \cup (Q \wedge R)\}, \{P\}, \{P \cup (Q \wedge R)\}]$.

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

$$n_6 = [6, \{n_3\}, \{P \cup (Q \wedge R)\}, \{Q \wedge R\}, \{\}].$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Wieder muss man zunächst n_5 verarbeiten und mit n_6 in Schritt 7 fortfahren.

Schritt 5: Die zu bearbeitende Formel kommt nicht in $n_5.old$ vor. Sie ist ein Literal, somit muss ein neuer Knoten erzeugt werden:

$$n_7 = [7, \{n_3\}, \{P \cup (Q \wedge R), P\}, \{\}, \{P \cup (Q \wedge R)\}].$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 6: $n_7.new$ ist leer. Jetzt gibt es aber einen Knoten in $nodes$, dessen Felder *old* und *new* mit denen von n_7 übereinstimmen, nämlich n_3 . n_3 bekommt also eine Schleife:

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

$$n_3 = [3, \{init, n_3\}, \{P \cup (Q \wedge R), P\}, \{\}, \{P \cup (Q \wedge R)\}] \text{ und die Rekursion bricht in diesem Zweig mit } nodes = \{init, n_3\} \text{ ab.}$$

Schritt 7: Es geht mit n_6 aus Schritt 4 weiter. Die zu bearbeitende Formel hat als Haupt-Junktor das \wedge , also muss eine neuer Knoten erzeugt werden:

$$n_8 = [8, \{n_3\}, \{P \cup (Q \wedge R), Q \wedge R\}, \{Q, R\}, \{\}].$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 8: In der Menge n_8 beginnen wir mit der ersten Formel, der Primformal Q :

$$n_9 = [9, \{n_3\}, \{P \cup (Q \wedge R), Q \wedge R, Q\}, \{R\}, \{\}].$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 9: Nun muss man R verarbeiten:

$$n_{10} = [10, \{n_3\}, \{P \cup (Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}].$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 10: $n_{10}.new$ ist jetzt leer und es muss ein neuer Knoten erzeugt werden:

$$n_{11} = [11, \{n_{10}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}\}.$$

<i>id</i>	<i>incoming</i>	<i>old</i>	<i>new</i>	<i>next</i>
-----------	-----------------	------------	------------	-------------

Schritt 11: Es geht mit der Expansion von n_{11} weiter. $n_{11}.new$ ist leer und es muss ein neuer Knoten erzeugt werden:

$$n_{12} = [12, \{n_{11}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}, n_{11}\}.$$

id incoming old new next

Schritt 12: Bei der Verarbeitung von n_{12} tritt der Fall ein, dass wir einen Knoten in $nodes$ haben, dessen Felder *old* und *next* mit denen von n_{12} übereinstimmen, nämlich mit n_{11} , also

$$n_{11} = [11, \{n_{10}, n_{11}\}, \{\}, \{\}, \{\}] \text{ und } nodes = \{init, n_3, n_{10}, n_{11}\}.$$

id incoming old new next

Schritt 13: Mit Schritt 12 ist dieser Zweig beendet und mit dem bisher entstandenen Graph $nodes = \{init, n_3, n_{10}\}$ geht es mit der Expansion von n_2 aus Schritt 1 weiter:

$$n_2 = [2, \{init\}, \{PU(Q \wedge R)\}, \{Q \wedge R\}, \{\}].$$

id incoming old new next

Man muss also $Q \wedge R$ verarbeiten. Jetzt wiederholt sich das Vorgehen:

$$n_{13} = [13, \{init\}, \{PU(Q \wedge R), Q \wedge R\}, \{Q, R\}, \{\}].$$

id incoming old new next

Schritt 14:

$$n_{14} = [14, \{init\}, \{PU(Q \wedge R), Q \wedge R, Q\}, \{R\}, \{\}].$$

id incoming old new next

Schritt 15:

$$n_{15} = [15, \{init\}, \{PU(Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}].$$

id incoming old new next

Schritt 16: Jetzt ist $n_{15}.new$ leer und wir müssen prüfen, ob wir bereits einen Knoten haben, dessen Felder *old* und *next* mit denen von n_{15} übereinstimmen. Dies ist tatsächlich der Fall, nämlich n_{10} . Also

$$n_{10} = [10, \{n_3, init\}, \{PU(Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}]$$

id incoming old new next

und $nodes = \{init, n_3, n_{10}, n_{11}\}$ wird zurückgegeben.

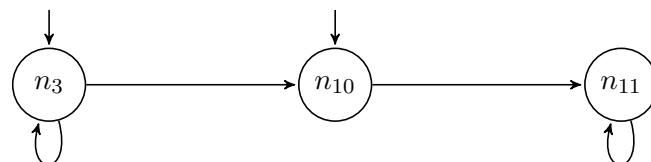


Abbildung 19.6: Graph, konstruiert zu $PU(Q \wedge R)$

$$n_3 = [3, \{init, n_3\}, \{PU(Q \wedge R), P\}, \{\}, \{PU(Q \wedge R)\}]$$

id incoming old new next

$$n_{10} = [10, \{n_3, init\}, \{PU(Q \wedge R), Q \wedge R, Q, R\}, \{\}, \{\}]$$

id incoming old new next

$$n_{11} = [11, \{n_{10}, n_{11}\}, \{\}, \{\}, \{\}]$$

id incoming old new next

Der konstruierte Graph wird in Abbildung 19.6 dargestellt.

Aus dem Graphen, den wir in der Liste der Knoten konstruiert haben, wird durch folgende Festlegungen ein verallgemeinerter Büchi-Automat:

- Das Alphabet Σ besteht aus Mengen von Aussagensymbolen, die in φ vorkommen.
- Die Menge Q der Zustände sind die Knoten in $nodes$ zusammen mit dem speziellen Zustand $init$.
- Ein Tupel $(q, a, q') \in \Delta$ genau dann wenn $q \in q'.incoming$ und a erfüllt alle Literale in $q'.old$.
- Der Initialzustand ist $init$.
- Für jede Teilformel der Form $\psi_1 \mathcal{U} \psi_2$ gibt es eine Akzeptanzmenge F_i mit $F_i = \{q \mid \psi_2 \in q.old \text{ oder } \psi_1 \mathcal{U} \psi_2 \notin r.old\}$.

Beispiel 19.4. In unserem Beispiel für die Formel $P \mathcal{U} (Q \wedge R)$ ergibt sich somit:

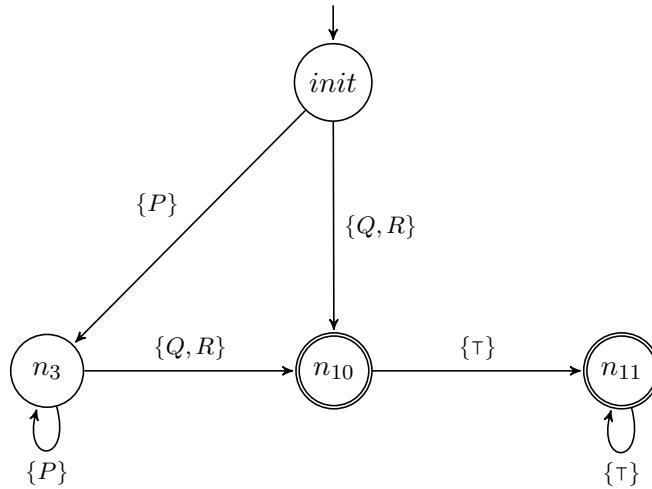
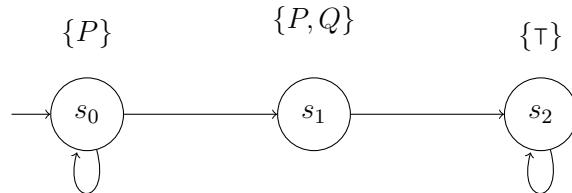
- $\Sigma = \{P, Q, R\}$,
- $Q = \{init, n_3, n_{10}, n_{11}\}$,
- $\Delta = \{[init, \{P\}, n_3], [n_3, \{P\}, n_3], [init, \{Q, R\}, n_{10}], [n_{10}, \{Q, R\}, n_{10}], [n_{10}, \{\top\}, n_{11}], [n_{11}, \{\top\}, n_{11}]\}$,
- $init$ ist der Initialzustand und
- $F_1 = \{n_{10}, n_{11}\}$ ist die Akzeptanzmenge.

Abbildung 19.7 zeigt den Büchi-Automaten zur Formel $P \mathcal{U} (Q \wedge R)$.

An dem Büchi-Automaten zu einer Formel der LTL kann man unmittelbar eine Kripke-Struktur ablesen, in der die Formel wahr ist. Man nimmt einen Pfad zu einer Akzeptanzmenge, die einen Zyklus enthält. Beginnend bei $init$ interpretiert man die Markierung der Kanten des Büchi-Graphen als Zuweisung der Menge der Literale zum nächsten Knoten.

Beispiel 19.5. In unserem Beispiel der Formel $P \mathcal{U} (Q \wedge R)$ kann man so verschiedene Kripke-Strukturen ablesen, etwa die in Abbildung 19.8 dargestellte.

Bemerkung Mit dem dargestellten Verfahren die Erfüllbarkeit einer Formel der LTL zu entscheiden, hat man natürlich auch ein Entscheidungsverfahren für die Erfüllbarkeit der Aussagenlogik. Es handelt sich die Methode der *semantischen Tableaux*, im Detail beschrieben z.B. in [BA12, Abschnitt 2.6].

Abbildung 19.7: Büchi-Automat zu $P \cup (Q \wedge R)$ Abbildung 19.8: Kripke-Struktur, in der $P \cup (Q \wedge R)$ gilt

*

In der Logic Workbench (lwb) wird für die Transformation einer Formel der LTL in einen Büchi-Automaten die Java-Bibliothek `ltl2buchi` eingesetzt. Die Bibliothek wurde implementiert von Dimitra Giannakopoulou und Flavio Lerda am **NASA Ames research Center** als Komponente des Model Checkers **Java Pathfinder**. Giannakopoulou und Lerda verwenden einen Algorithmus, der auf den oben erläuterten Algorithmus aufbaut, jedoch deutlich kleinere Büchi-Automaten generiert [GL02].

Die Funktion in der Logic Workbench ergeben in unserem Beispiel die in Abbildung 19.9 dargestellten Ergebnisse:

```
(def lex '(until P (and Q R)))

(lwb.ltl.buechi/ba lex)
;=> {:nodes [{:id 1, :accepting true} {:id 0, :init true}],
;      :edges [{:from 1, :to 1, :guard #{}}, {:from 0, :to 0, :guard #{P}},
;              {:from 0, :to 1, :guard #{R Q}}]}
```

```
(sat lex)
;=> {:atoms #{{R Q P}},
;      :nodes {:{s_1 #{R Q}}},
;      :initial :s_1,
;      :edges #{{{:{s_1 :s_1}}}}
```

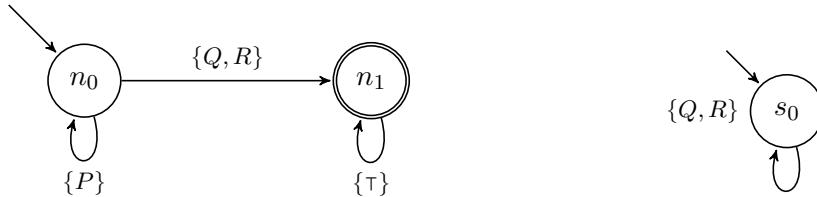


Abbildung 19.9: Mit lwb generiert: Büchi-Automat und Kripke-Struktur

19.6 Auswertung von Formeln in Kripke-Strukturen

Die Auswertung einer Formel φ der LTL in einer gegebenen Kripke-Struktur \mathcal{K} wird als *Model Checking* bezeichnet. Das Vorgehen zur Beantwortung der Frage $\mathcal{K} \models \varphi?$ besteht aus folgenden Schritten (siehe auch Abbildung 19.10):

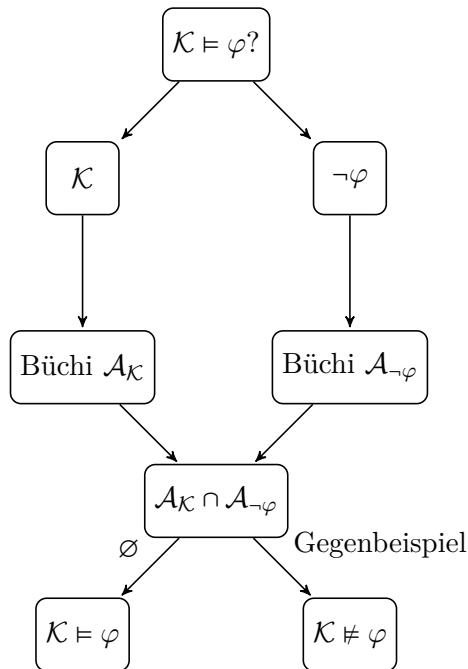


Abbildung 19.10: Evaluation von Formeln der LTL in Kripke-Strukturen

1. Zu der gegebenen Kripke-Struktur, dem Modell, konstruiert man den korrespondierenden Büchi-Automaten, siehe 19.4.1.
2. Die Negation der Formel, also zu $\neg\varphi$ wird in einen Büchi-Automaten übersetzt.
3. Nun bildet man den Durchschnitt der beiden Automaten. Wenn es einen erfolgreichen Lauf in diesem Automaten gibt, dann entspricht dies Berechnungspfaden in \mathcal{K} , die $\neg\varphi$ erfüllen.

Also: ist der Durchschnitt der beiden Automaten leer, dann ist φ in \mathcal{K} wahr. Andernfalls kann man dem Durchschnitt einen Berechnungspfad ablesen, der ein Gegenbeispiel darstellt.

Beispiel 19.6 (Wechselseitiger Ausschluss). Gegeben seien zwei Prozesse, die auf eine gemeinsame Resource zu greifen möchten. Ihre Programmschleife besteht darin, dass sie nicht-kritische Aktionen ausführen, dann versuchen den kritischen Bereich zu betreten und ihn schließlich wieder verlassen.

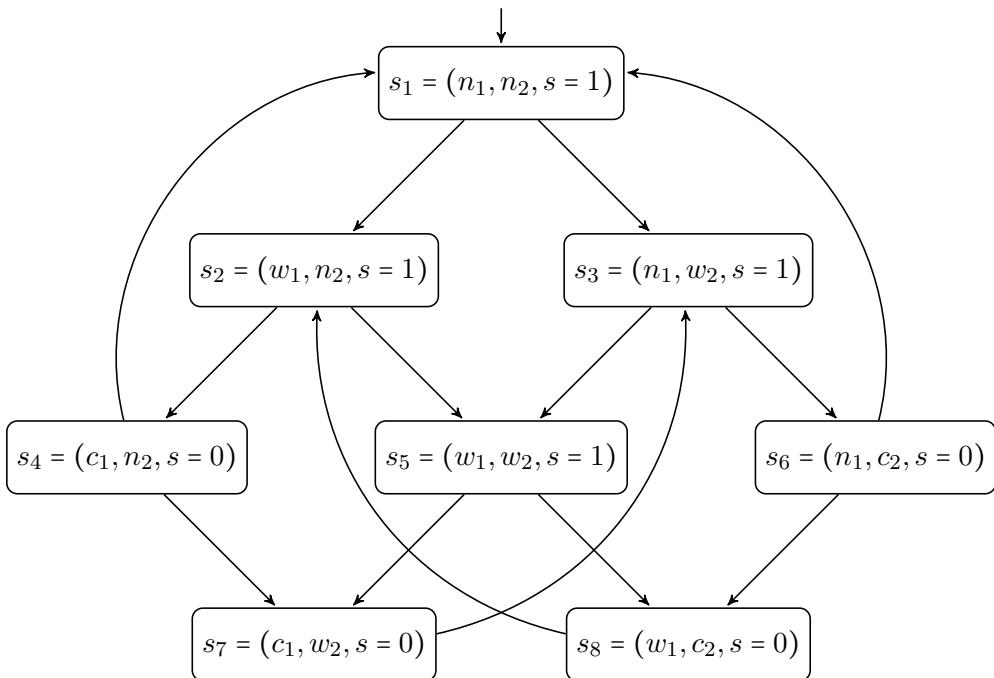


Abbildung 19.11: Kripke-Struktur zu einem Protokoll wechselseitigen Ausschlusses

Die beiden Prozesse verwenden eine gemeinsame binäre Semaphore s . Wenn $s = 1$ gilt, dann ist die Semaphore frei, wenn $s = 0$ ist, dann ist die Semaphore gerade durch einen der beiden Prozesse belegt.

In Abbildung 19.11 wird das Zusammenspiel der beiden Prozesse in einem Transitionssystem dargestellt. Dabei bezeichnet n_i , dass Prozess i nicht-kritische Aktionen ausführt, w_i , dass er auf den Zugang zu seinem kritischen Abschnitt wartet und c_i , dass er in seinem kritischen Abschnitt ist [BK08, Abschnitt 2.2].

In diesem Beispiel wollen wir nun folgende Formeln überprüfen:

- ① $\square (\neg c_1 \vee \neg c_2)$
- ② $\square \Diamond c_1 \vee \square \Diamond c_2$
- ③ $\square \Diamond c_1 \wedge \square \Diamond c_2$

Formel ① ist eine Sicherheitseigenschaft, die ausdrückt, dass niemals beide Prozesse gleichzeitig den kritischen Abschnitt verwenden dürfen. Die zweite Formel besagt, dass mindestens einer der Prozesse immer wieder den kritischen Sektor betreten kann. Und Formel ③ drückt Fairness aus: beide Prozesse können immer wieder den kritischen Abschnitt betreten.

Die Situation ist so einfach, dass man die Formeln durch genaues Hinsehen evaluieren kann. Wir wollen jedoch die Logic Workbench verwenden.

Zuerst wird die dem Transitionssystem entsprechende Kripke-Struktur in lwb definiert:

```
; Examples of algorithm for mutual exclusion based on a binary semaphore
; n = non-critical actions
; w = wait
; c = critical section
; s = semaphore s=1 true, s=0 false

(def ksx {:atoms '#{n1 n2 w1 w2 c1 c2 s}
          :nodes  #{:s1 '#{n1 n2 s}
                    :s2 '#{w1 n2 s}
                    :s3 '#{n1 w2 s}
                    :s4 '#{c1 n2}
                    :s5 '#{w1 w2 s}
                    :s6 '#{n1 c2}
                    :s7 '#{c1 w2}
                    :s8 '#{w1 c2}}
          :initial :s1
          :edges   #{{[:s1 :s2]
                     [:s1 :s3]
                     [:s2 :s4]
                     [:s2 :s5]
                     [:s3 :s5]
                     [:s3 :s6]
                     [:s4 :s1]
                     [:s4 :s7]}}}
```

```
[:s5 :s7]
[:s5 :s8]
[:s6 :s1]
[:s6 :s8]
[:s7 :s3]
[:s8 :s2}])
```

Dazu werden die verwendeten Atome definiert, die Zustände mit den Atomen, die in ihnen T sind, die anderen sind F. Ferner wird der Initialzustand festgelegt und die Zustandsübergänge.

Nun können wir Formeln in dieser Kripke-Struktur auswerten:

```
(def phi1 '(always (or (not c1) (not c2))))  
  
(eval-phi phi1 ksx)  
;=> true  
  
(def phi2 '(or (always (finally c1)) (always (finally c2))))  
  
(eval-phi phi2 ksx)  
;=> true  
  
(def phi3 '(and (always (finally c1)) (always (finally c2))))  
  
(eval-phi phi3 ksx)  
;=> false  
(eval-phi phi3 ksx :counterexample)  
;=> [:s1 :s3 :s5 :s7 :s3 :s5 :s8 :s2 :s5]
```

Die Formeln ① und ② werten zu T aus, wohingegen Formel ③ nicht zutrifft, also Fairness nicht gegeben ist. Man kann sich dann ein Gegenbeispiel anzeigen lassen. Es zeigt, dass es möglich ist, dass immer wieder der Zyklus s_5, s_8, s_2, s_5 durchlaufen werden kann und in diesem Fall wird Prozess 1 niemals mehr den kritischen Abschnitt betreten kann, er „verhungert“ in Zustand s_5 .

Kapitel 20

Natürliches Schließen in der LTL

siehe Vorlesung

Kapitel 21

Anwendungen der LTL in der Softwaretechnik

siehe Vorlesung

21.1 Model Checking

21.1.1 Konzept des Model Checkings

21.1.2 Der Model Checker **SPIN**

21.1.3 Beispiele für Model Checking

21.2 Zielemodell in der Anforderungsanalyse

Literaturverzeichnis

- [ABKS13] SVEN APEL, DON BATORY, CHRISTIAN KÄSTNER, ET AL. *Feature-Oriented Software Product Lines*. Berlin Heidelberg: Springer, 2013.
- [BA12] MORDECHAI BEN-ARI. *Mathematical Logic for Computer Science: Third Edition*. London: Springer, 2012.
- [Bat05] DON S. BATORY. Feature Models, Grammars, and Propositional Formulas. In: *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings, Lecture Notes in Computer Science*, Band 3714, S. 7–20. Springer, 2005.
- [BC04] YVES BERTOT, PIERRE CASTÉRAN. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Berlin: Springer, 2004.
- [BK08] CHRISTEL BAIER, JOOST-PIETER KATOEN. *Principles of Model Checking*. Cambridge, MA: MIT Press, 2008.
- [Bor05] RICHARD BORNAT. *Proof and Disproof in Formal Logic: An introduction for programmers*. Oxford: Oxford University Press, 2005.
- [BR18] DANIEL LE BERRE, PASCAL RAPICAULT. Boolean-Based Dependency Management for the Eclipse Ecosystem. *International Journal on Artificial Intelligence Tools*, 27(1):1–23, 2018.
- [CGP99] EDMUND M. CLARKE, JR., ORNA GRUMBERG, DORON A. PELED. *Model Checking*. Cambridge, MA: MIT Press, 1999.
- [DLL62] MARTIN DAVIS, GEORGE LOGEMANN, DONALD LOVELAND. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
- [dMB10] LEONARDO MENDONÇA DE MOURA, NIKOLAJ BJØRNER. Applications and Challenges in Satisfiability Modulo Theories. In: *Second International Workshop on Invariant Generation, WING 2009, York, UK, March 29, 2009 and Third*

- International Workshop on Invariant Generation, WING 2010, Edinburgh, UK, July 21, 2010, EPiC Series in Computing, Band 1, S. 1–11. 2010.*
- [DP60] MARTIN DAVIS, HILARY PUTNAM. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.
- [GBE⁺14] JÜRGEN GIESL, MARC BROCKSCHMIDT, FABIAN EMMES, ET AL. Proving Termination of Programs Automatically with AProVE. In: *Automated Reasoning - 7th International Joint Conference, IJCAR 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 19–22, 2014. Proceedings, Lecture Notes in Computer Science*, Band 8562, S. 184–191. Springer, 2014.
- [Gen35] GERHARD GENTZEN. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift*, 39:176–210, 1935. URL <http://gdz.sub.uni-goettingen.de/dms/resolveppn/?PPN=GDZPPN002375508>.
- [GL02] DIMITRA GIANNAKOPOULOU, FLAVIO LERDA. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In: DORON A. PELED, MOSHE Y. VARDI, Hg., *Formal Techniques for Networked and Distributed Systems - FORTE 2002, 22nd IFIP WG 6.1 International Conference Houston, Texas, USA, November 11–14, 2002, Proceedings, Lecture Notes in Computer Science*, Band 2529, S. 308–326. Springer, 2002.
- [GPVW95] ROB GERTH, DORON A. PELED, MOSHE Y. VARDI, ET AL. Simple on-the-fly automatic verification of linear temporal logic. In: PIOTR DEMBINSKI, MAREK SREDNIAWA, Hg., *Protocol Specification, Testing and Verification XV, Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland, June 1995, IFIP Conference Proceedings*, Band 38, S. 3–18. Chapman & Hall, 1995.
- [Gö29] KURT GÖDEL. *Über die Vollständigkeit des Logikkalküls*. Dissertation, 1929.
- [Hed04] SHAWN HEDMAN. *A First Course in Logic: An Introduction to Model Theory, Proof Theory, Computability, and Complexity*. Oxford: Oxford University Press, 2004.
- [HL11] MARTIN HOFMANN, MARTIN LANGE. *Automatentheorie und Logik*. Heidelberg: Springer, 2011.

- [Hof11] DIRK W. HOFFMANN. *Grenzen der Mathematik: Eine Reise durch die Kerngebiete der mathematischen Logik*. Heidelberg: Spektrum Akademischer Verlag, 2011.
- [HR04] MICHAEL HUTH, MARK RYAN. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge, UK: Cambridge University Press, 2. Auflage, 2004.
- [Jac06] MICHAEL JACKSON. The role of structure: a software engineering perspective. In: DENIS BESNARD, CRISTINA GACEK, CLIFFORD B. JONES, Hg., *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Kapitel 2, S. 16–45. Springer, 2006.
- [Jac12] DANIEL JACKSON. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA: MIT Press, revised Auflage, 2012.
- [Kay07] RICHARD KAYE. *The Mathematics of Logic: A guide to completeness theorems and their applications*. Cambridge, UK: Cambridge University Press, 2007.
- [KCH⁺90] KYO C. KANG, SHOLOM G. COHEN, JAMES A. HESS, ET AL. *Technical Report: Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Pittsburgh, PA: Software Engineering Institute (SEI), 1990.
- [KGN⁺09] ROOPE KAIVOLA, RAJNISH GHUGHAL, NAREN NARASIMHAN, ET AL. Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings, Lecture Notes in Computer Science*, Band 5643, S. 414–429. Springer, 2009.
- [Knu15] DONALD E. KNUTH. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Boston, MA: Addison-Wesley, 2015.
- [KS06] DANIEL KROENING, OFER STRICHMAN. *Decision Procedures: An Algorithmic Point of View*. Springer, 2006.
- [MH07] ANDREAS METZGER, PATRICK HEYMANS. *Technischer Bericht: Comparing Feature Diagram Examples Found in the Research Literature*. Software Systems Engineering, Universität Duisburg-Essen, 2007.
- [MTS⁺17] JENS MEINICKE, THOMAS THÜM, REIMAR SCHRÖTER, ET AL. *Mastering Software Variability with FeatureIDE*. Cham: Springer, 2017.

- [Rau08] WOLFGANG RAUTENBERG. *Einführung in die Mathematische Logik, 3., überarbeitete Auflage*. Wiesbaden: Vieweg + Teubner, 2008.
- [Sip13] MICHAEL SIPSER. *Introduction to the Theory of Computation, Third Edition*. Boston, MA: Cengage Learning, 2013.
- [SKK03] CARSTEN SINZ, ANDREAS KAISER, WOLFGANG KÜCHLIN. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.
- [Tse83] G. S. TSEITIN. On the Complexity of Derivation in Propositional Calculus. In: J. SIEKMANN, G. WRIGHTSON, Hg., *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, S. 466–483. Berlin, Heidelberg: Springer, 1983.
- [vD13] DIRK VAN DALEN. *Logic and Structure*. Berlin: Springer, 5. Auflage, 2013.
- [Zav12] PAMELA ZAVE. Using lightweight modeling to understand chord. *ACM SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, 2012.