Logikbasierte Systeme der Wissensverarbeitung

Sommersemester 2022

Prof. Dr. Manfred Schmidt-Schauß
und
PD Dr. David Sabel*

Institut für Informatik
Fachbereich Informatik und Mathematik
Goethe-Universität Frankfurt am Main
Postfach 11 19 32
D-60054 Frankfurt am Main
Email: schauss@em.uni-frankfurt.de

(*) Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München
Email: david.sabel@ifi.lmu.de

Stand: 28. Februar 2024

Inhaltsverzeichnis

1 Aussagenlogik

In diesem Kapitel werden die Aussagenlogik und dort angesiedelte Deduktionsverfahren erörtert. Wir zeigen, wie diese (relativ einfache) Logik verwendet werden kann, um intelligente Schlüsse zu ziehen und Entscheidungsverfahren zu implementieren. Bereits in Kapitel **??** haben wir in Abschnitt **??** die Wissenrepräsentationshypothese nach Brian Smith erörtert. Wir wiederholen sie an dieser Stelle, da sie als Paradigma insbesondere dieses Kapitel motiviert:

> „Die Verarbeitung von Wissen lässt sich trennen in: Repräsentation von Wissen, wobei dieses Wissen eine Entsprechung in der realen Welt hat; und in einen Inferenzmechanismus, der Schlüsse daraus zieht."

Diese Hypothese ist die Basis für solche Programme, die Fakten, Wissen, Beziehungen modellieren und entsprechende Operationen (und Simulationen) darauf aufbauen. Ein Repräsentations- und Inferenz-System besteht, abstrakt gesehen dabei aus den Komponenten: *Formale Sprache*, die die syntaktisch gültigen Formeln und Anfragen festlegt, der *Semantik* die den Sätzen der formalen Sprache eine Bedeutung zuweist, und der *Inferenzprozedur* (operationale Semantik), die (algorithmisch) angibt, wie man Schlüsse aus der gegebenen Wissensbasis ziehen kann. Die Inferenzen müssen korrekt bezüglich der Semantik sein. Die Korrektheit eines solchen Systems kann geprüft werden, indem man nachweist, dass die operationale Semantik die Semantik erhält.

Die Implementierung des Repräsentations- und Inferenz-Systems besteht im Allgemeinen aus einem *Parser* für die Erkennung der Syntax (der formalen Sprache) und der Implementierung der Inferenzprozedur.

1.1 Syntax und Semantik der Aussagenlogik

Die Aussagenlogik ist in vielen anderen Logiken enthalten; sie hat einfache verständliche Inferenzmethoden; man kann einfache Beispiele modellhaft in der Aussagenlogik betrachten. Bei der Verallgemeinerung auf Prädikatenlogik und andere Logiken startet man meist auf der Basis der Aussagenlogik.

Wir werden in diesem Kapitel detailliert auf Aussagenlogik, Inferenzverfahren und Eigenschaften eingehen. Das Ziel dabei ist jeweils, vereinfachte Varianten von allgemeinen Verfahren zum Schlussfolgern zu verstehen, damit man einen Einblick in die Wirkungs-

weise von Inferenzverfahren für Prädikatenlogik und andere Logiken gewinnt. Ziel ist eher nicht, optimale und effiziente Verfahren für die Aussagenlogik vorzustellen.

Definition 1.1.1 (Syntax der Aussagenlogik)**.** *Sei X ein Nichtterminal für aussagenlogische* Variablen (*aus einer Menge von abzählbar unendlich vielen Variablen*)*. Die folgende Grammatik legt die* Syntax *aussagenlogischer Formeln fest:*

$$A ::= X \mid (A \wedge A) \mid (A \vee A) \mid (\neg\, A) \mid (A \Rightarrow A) \mid (A \Leftrightarrow A) \mid 0 \mid 1$$

Die Konstanten 0 und 1 entsprechen der falschen bzw. wahren Aussage. Üblicherweise werden bei der Notation von Aussagen Klammern weggelassen, wobei man die Klammerung aus den Prioritäten der Operatoren wieder rekonstruieren kann: Die Prioritätsreihenfolge ist: $\neg, \wedge, \vee, \Rightarrow, \iff$.

Die Zeichen $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ nennt man Junktoren*. Die Bezeichnungen der Junktoren sind:*

- $A \wedge B$: *Konjunktion* (*Verundung*).
- $A \vee B$: *Disjunktion* (*Veroderung*).
- $A \Rightarrow B$: *Implikation* .
- $A \Leftrightarrow B$: *Äquivalenz* (*Biimplikation*).
- $\neg A$: *negierte Formel* (*Negation*).

Aussagen der Form $0, 1$ oder X nennen wir Atome*. Ein* Literal *ist entweder ein Atom oder eine Aussage der Form $\neg A$, wobei A ein Atom ist.*

Oft wählt man aussagekräftige Namen (Bezeichnungen) für die Variablen, um die entsprechenden Aussagen auch umgangsprachlich zu verstehen. Z.B. kann man den Satz „Wenn es heute nicht regnet, gehe ich Fahrradfahren.“ als Aussagenlogische Formel mit den Variablen $esregnetheute$ und $fahrradfahren$ darstellen als $\neg esregnetheute \Rightarrow fahrradfahren$.

Wir lassen auch teilweise eine erweiterte Syntax zu, bei der auch noch andere Operatoren zulässig sind wie: NOR, XOR, NAND.

Dies erweitert nicht die Fähigkeit zum Hinschreiben von logischen Ausdrücken, denn man kann diese Operatoren simulieren. Auch könnte man $0, 1$ weglassen, wie wir noch sehen werden, denn man kann 1 als Abkürzung von $A \vee \neg A$, und 0 als Abkürzung von $A \wedge \neg A$ auffassen.

Definition 1.1.2 (Semantik der Junktoren)**.** *Zunächst definiert man für jeden Junktor $\neg, \wedge, \vee, \Rightarrow$ und $\Leftrightarrow$ Funktionen $\{0,1\} \to \{0,1\}$ bzw. $\{0,1\}^2 \to \{0,1\}$. Die Funktion $f_\neg$ ist definiert als $f_\neg(0) = 1, f_\neg(1) = 0$. Ebenso definiert man $f_0 := 0, f_1 := 1$. Die anderen Funktionen f_{op} sind definiert gemäß folgender Tabelle:*

		$\wedge$	$\vee$	$\Rightarrow$	$\Leftarrow$	NOR	$NAND$	$\Leftrightarrow$	XOR
1	1	1	1	1	1	0	0	1	0
1	0	0	1	0	1	0	1	0	1
0	1	0	1	1	0	0	1	0	1
0	0	0	0	1	1	1	1	1	0

Der Einfachheit halber werden oft die syntaktischen Symbole auch als Funktionen gedeutet.

Definition 1.1.3 (Interpretation)**.** *Eine* Interpretation (*Variablenbelegung*) I *ist eine Funktion:*

$$I : \{\textit{aussagenlogische Variablen}\} \to \{0, 1\}.$$

Eine Interpretation I definiert für jede Aussage einen Wahrheitswert, wenn man sie auf Aussagen wie folgt fortsetzt:

- $I(0) := 0, I(1) := 1$
- $I(\neg A) := f_\neg(I(A))$
- $I(A \; op \; B) := f_{op}(I(A), I(B))$, *wobei* $op \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow \ldots\}$

Eine Interpretation I ist genau dann ein Modell *für die Aussage F, wenn $I(F) = 1$ gilt. Ist I ein Modell für F so schreiben wir $I \models F$. Als weitere Sprechweisen verwenden wir in diesem Fall „F gilt in I“ und „I macht F wahr“.*

Aufbauend auf dem Begriff der Interpretation, werden in der Aussagenlogik verschiedene Eigenschaften von Formeln definiert.

Definition 1.1.4. *Sei A ein Aussage.*

- *A ist genau dann eine* Tautologie (Satz, allgemeingültig)*, wenn für alle Interpretationen I gilt: $I \models A$.*
- *A ist genau dann ein* Widerspruch (widersprüchlich, unerfüllbar)*, wenn für alle Interpretationen I gilt: $I(A) = 0$.*
- *A ist genau dann* erfüllbar (konsistent)*, wenn es eine Interpretation I gibt mit: $I \models A$ (d.h. wenn es ein Modell für A gibt).*
- *A ist genau dann* falsifizierbar*, wenn es eine Interpretation I gibt mit $I(A) = 0$.*

Man kann jede Aussage in den n Variablen $X_1, \ldots, X_n$ auch als eine Funktion mit den n Argumenten $X_1, \ldots, X_n$ auffassen. Dies entspricht dann *Booleschen Funktionen*.

Beispiel 1.1.5. *Wir betrachten einige Beispiele:*

- *$X \vee \neg X$ ist eine Tautologie, denn für jede Interpretation I gilt: $I(X \vee \neg X) = f_\vee(f_\neg(I(X)), I(X)) = 1$, da $f_\neg(I(X))$ oder $I(X)$ gleich zu 1 sein muss.*
- *$X \wedge \neg X$ ist ein Widerspruch, denn für jede Interpretation I gilt: $I(X \wedge \neg X) = f_\wedge(f_\wedge(I(X)), I(X)) = 0$, da $f_\neg(I(X))$ oder $I(X)$ gleich zu 0 sein muss.*
- *$(X \Rightarrow Y) \Rightarrow ((Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z))$ ist eine Tautologie.*

- *$X \vee Y$ ist erfüllbar. Z.B. ist I mit $I(X) = 1, I(Y) = 1$ ein Modell für $X \vee Y$: $I(X \vee Y) = f_\vee(I(X), I(Y)) = f_\vee(1, 1) = 1$.*
- *I mit $I(X) = 1, I(Y) = 0$ ist ein Modell für $X \wedge \neg Y$*

Beispiel 1.1.6. *Bauernregeln:*

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

  Die tautologischen Aussagen sind in Bezug auf die Wirklichkeit ohne Inhalt. Erst wenn man sie verwendet zum Finden von Folgerungen ergibt sich etwas neues.

Theorem 1.1.7.

- *Es ist entscheidbar, ob eine Aussage eine Tautologie (Widerspruch, erfüllbar) ist.*
- *Die Frage „Ist A erfüllbar?" ist $\mathcal{NP}$-vollständig.*
- *Die Frage „Ist A falsifizierbar?" ist ebenso $\mathcal{NP}$-vollständig.*
- *Die Frage „Ist A Tautologie?" ist $Co\mathcal{NP}$-vollständig.*
- *Die Frage „Ist A ein Widerspruch?" ist $Co\mathcal{NP}$-vollständig.*

Das einfachste und bekannteste Verfahren zur Entscheidbarkeit ist das der Wahrheitstafeln. Es werden einfache alle Interpretation ausprobiert.

Zur Erläuterung: $\mathcal{NP}$-vollständig bedeutet, dass jedes Problem der Problemklasse mit einem *nicht-deterministischen* Verfahren in *polynomieller Zeit* gelöst werden kann, und dass es keine bessere obere Schranke gibt (unter der allgemein vermuteten Hypothese $\mathcal{P} \neq \mathcal{NP}$).

Eine Problemklasse ist $Co\mathcal{NP}$-vollständig, wenn die Problemklasse, die aus dem Komplement gebildet wird, $\mathcal{NP}$-vollständig ist. [1]

[1] Üblicherweise werden Problemklassen als Wortproblem über einer formalen Sprache aufgefasst. Die Frage nach der Erfüllbarkeit wäre in dieser Darstellung die Sprache SAT definiert als:

$$\text{SAT} := \{F \mid F \text{ ist erfüllbare aussagenlogische Formel }\}.$$

Das Wortproblem ist die Frage, ob eine gegebene Formel in der Sprache SAT liegt oder nicht.
Die Komplexitätsklasse $\mathcal{NP}$ umfasst alle Sprachen, deren Wortproblem auf einer nichtdeterministischen Turingmaschine in polynomieller Zeit lösbar ist. Die Komplexitätsklasse $Co\mathcal{NP}$ ist definiert als:

$$Co\mathcal{NP} := \{L \mid L^C \in \mathcal{NP}\}, \text{ wobei } L^C \text{ das Komplement der Sprache } L \text{ ist.}$$

Sequentielle Algorithmen zur Lösung haben für beide Problemklassen nur Exponential-Zeit Algorithmen. Genaugenommen ist es ein offenes Problem der theoretischen Informatik, ob es nicht bessere sequentielle Algorithmen gibt (d.h. ob $\mathcal{P} = \mathcal{NP}$ bzw. $\mathcal{P} = Co\mathcal{NP}$ gilt).

Die Klasse der $\mathcal{NP}$-vollständigen Probleme ist vom praktischen Standpunkt aus leichter als $Co\mathcal{NP}$-vollständig: Für $\mathcal{NP}$-vollständige Probleme kann man mit Glück (d.h. Raten) oft schnell eine Lösung finden. Z.B. eine Interpretation einer Formel. Für $Co\mathcal{NP}$-vollständige Probleme muß man i.a. viele Möglichkeiten testen: Z.B. muß man i.A. alle Interpretationen einer Formel ausprobieren.

1.2 Folgerungsbegriffe

Man muß zwei verschiedene Begriffe der Folgerungen für Logiken unterscheiden:

- semantische Folgerung
- syntaktische Folgerung (Herleitung, Ableitung) mittels einer prozeduralen Vorschrift. Dies ist meist ein nicht-deterministischer Algorithmus (Kalkül), der auf Formeln oder auf erweiterten Datenstrukturen operiert. Das Ziel ist i.A. die Erkennung von Tautologien (oder Folgerungsbeziehungen).

In der Aussagenlogik fallen viele Begriffe zusammen, die in anderen Logiken verschieden sind. Trotzdem wollen wir die Begriffe hier unterscheiden.

Definition 1.2.1. *Sei $\mathcal{F}$ eine Menge von (aussagenlogischen) Formeln und G eine weitere Formel.*

Wir sagen G folgt semantisch *aus $\mathcal{F}$*
gdw.
Für alle Interpretationen I gilt: $\big($wenn für alle Formeln $F \in \mathcal{F}$ die Auswertung $I(F) = 1$ ergibt $\big)$, dann auch $I(G) = 1$.

Anders ausgedrückt: Jedes Modell für alle Formeln aus $\mathcal{F}$ ist auch ein Modell für G.

Die semantische Folgerung notieren wir auch als $\mathcal{F} \models G$

Wir schreiben auch $F_1, \ldots, F_n \models G$ statt $\{F_1, \ldots, F_n\} \models G$.
Es gilt

Die Sprache UNSAT $:= \{F \mid F$ ist Widerspruch$\}$ ist eine der Sprachen in $Co\mathcal{NP}$, da UNSAT^C = Alle Formeln \UNSAT = SAT, und SAT $\in \mathcal{NP}$.

Ein Sprache $L \in \mathcal{NP}$ ist $\mathcal{NP}$-vollständig, wenn es für jede Sprache $L' \in \mathcal{NP}$ eine Polynomialzeit-Kodierung R gibt, die jedes Wort $x \in L'$ in die Sprache L kodiert (d.h. $R(x) \in L$) (dies nennt man auch Polynomialzeitreduktion). Die $\mathcal{NP}$-vollständigen Sprachen sind daher die „schwersten" Probleme in $\mathcal{NP}$.

Analog dazu ist die $Co\mathcal{NP}$-Vollständigkeit definiert: Eine Sprache $L \in Co\mathcal{NP}$ ist $Co\mathcal{NP}$-vollständig, wenn jede andere Sprache aus $Co\mathcal{NP}$ in Polynomialzeit auf L reduziert werden kann.

Man kann relativ leicht nachweisen, dass sich auch die Vollständigkeit mit den Komplementen überträgt, d.h. dass eine Sprache L genau dann $Co\mathcal{NP}$-vollständig ist, wenn ihr Komplement L^C eine $\mathcal{NP}$-vollständige Sprache ist.

Satz 1.2.2. *Wenn ein F_i ein Widerspruch ist, dann kann man alles folgern: Es gilt dann für jede Formel G: $F_1, \ldots, F_n \models G$.*

Wenn ein F_i eine Tautologie ist, dann kann man dies in den Voraussetzungen weglassen: Es gilt dann für alle Formeln G: $F_1, \ldots, F_n \models G$ ist dasselbe wie $F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n \models G$

In der Aussagenlogik gibt es eine starke Verbindung von semantischer Folgerung mit Tautologien:

Theorem 1.2.3 (Deduktionstheorem der Aussagenlogik)**.**
$\{F_1, \ldots, F_n\} \models G$ gdw. $F_1 \wedge \ldots \wedge F_n \Rightarrow G$ ist Tautologie.

Zwei Aussagen F, G nennen wir *äquivalent* ($F \sim G$), gdw. $F \iff G$ eine Tautologie ist. Beachte, dass F und G genau dann äquivalent sind, wenn für alle Interpretationen I gilt: $I \models F$ gdw. $I \models G$ gilt.

Es ist auch zu beachten, dass z.B. $X \wedge Y$ nicht zu $X' \wedge Y'$ äquivalent ist. D.h. bei diesen Beziehungen spielen die Variablennamen eine wichtige Rolle.

Definition 1.2.4. *Gegeben sei ein (nicht-deterministischer) Algorithmus $\mathcal{A}$ (ein Kalkül), der aus einer Menge von Formeln $\mathcal{H}$ eine neue Formel H berechnet, Man sagt auch, dass H* syntaktisch *aus $\mathcal{H}$* folgt *(bezeichnet mit $\mathcal{H} \vdash_A H$ oder auch $\mathcal{H} \to_A H$).*

- *Der Algorithmus $\mathcal{A}$ ist* korrekt (sound)*, gdw. aus $\mathcal{H} \vdash_A H$ stets $\mathcal{H} \models H$ folgt.*
- *Der Algorithmus $\mathcal{A}$ ist* vollständig (complete)*, gdw. $\mathcal{H} \models H$ impliziert, dass $\mathcal{H} \vdash_A H$*

Aus obigen Betrachtungen folgt, dass es in der Aussagenlogik für die Zwecke der Herleitung im Prinzip genügt, einen Algorithmus zu haben, der Aussagen auf Tautologieeigenschaft prüft. Gegeben $\{F_1, \ldots, F_n\}$, zähle alle Formeln F auf, prüfe, ob $F_1 \wedge \ldots \wedge F_n \Rightarrow F$ eine Tautologie ist, und gebe F aus, wenn die Antwort ja ist. Das funktioniert, ist aber nicht sehr effizient, wegen der Aufzählung aller Formeln. Außerdem kann man immer eine unendliche Menge von Aussagen folgern, da alle Tautologien immer dabei sind.

Es gibt verschiedene Methoden, aussagenlogische Tautologien (oder auch erfüllbare Aussagen) algorithmisch zu erkennen: Z.B. BDDs (binary decision diagrams) : eine Methode, Aussagen als Boolesche Funktionen anzusehen und möglichst kompakt zu repräsentieren; Genetische Algorithmen zur Erkennung erfüllbarer Formeln; Suchverfahren die mit statischer Suche und etwas Zufall und Bewertungsfunktionen operieren; DPLL-Verfahren: Fallunterscheidung mit Simplifikationen (siehe unten 1.7); Tableaukalkül, der Formeln syntaktisch analysiert (analytic tableau).

1.3 Tautologien und einige einfache Verfahren

Wir wollen im folgenden Variablen in Aussagen nicht nur durch die Wahrheitswerte $0, 1$ ersetzen, sondern auch durch Aussagen. Wir schreiben dann $A[B/x]$, wenn in der Aussage A die Aussagenvariable x durch die Aussage B ersetzt wird. In Erweiterung dieser

Notation schreiben wir auch: $A[B_1/x_1, \ldots, B_n/x_n]$, wenn mehrere Aussagevariablen ersetzt werden sollen. Diese Einsetzung passiert gleichzeitig, so dass dies kompatibel zur Eigenschaft der Komposition als Funktionen ist: Wenn A n-stellige Funktion in den Aussagevariablen ist, dann ist $A[B_1/x_1, \ldots, B_n/x_n]$ die gleiche Funktion wie $A \circ (B_1, \ldots, B_n)$.

Theorem 1.3.1. *Gilt $A_1 \sim A_2$ und $B_1, \ldots, B_n$ sind weitere Aussagen, dann gilt auch $A_1[B_1/x_1, \ldots, B_n/x_n] \sim A_2[B_1/x_1, \ldots, B_n/x_n]$.*

Beweis. Man muß zeigen, dass alle Zeilen der Wahrheitstafeln für die neuen Ausdrücke gleich sind. Seien $y_1, \ldots, y_m$ die Variablen. Den Wert einer Zeile kann man berechnen, indem man zunächst die Wahrheitswerte für B_i berechnet und dann in der Wahrheitstabelle von A_i nachschaut. Offenbar erhält man für beide Ausdrücke jeweils denselben Wahrheitswert. □

Theorem 1.3.2. *Sind A, B äquivalente Aussagen, und F eine weitere Aussage, dann sind $F[A]$ und $F[B]$ ebenfalls äquivalent.* [2]

Die Junktoren $\wedge$ und $\vee$ sind kommutativ, assoziativ, und idempotent, d.h. es gilt:

$$\begin{array}{ccc}
\mathcal{F} \wedge \mathcal{G} & \Longleftrightarrow & \mathcal{G} \wedge \mathcal{F} \\
\mathcal{F} \wedge \mathcal{F} & \Longleftrightarrow & \mathcal{F} \\
\mathcal{F} \wedge (\mathcal{G} \wedge \mathcal{H}) & \Longleftrightarrow & (\mathcal{F} \wedge \mathcal{G}) \wedge \mathcal{H} \\
\mathcal{F} \vee \mathcal{G} & \Longleftrightarrow & \mathcal{G} \vee \mathcal{F} \\
\mathcal{F} \vee (\mathcal{G} \vee \mathcal{H}) & \Longleftrightarrow & (\mathcal{F} \vee \mathcal{G}) \vee \mathcal{H} \\
\mathcal{F} \vee \mathcal{F} & \Longleftrightarrow & \mathcal{F}
\end{array}$$

Weiterhin gibt es für Aussagen noch Rechenregeln und Gesetze, die wir im folgenden auflisten wollen. Alle lassen sich mit Hilfe der Wahrheitstafeln beweisen, allerdings erweist sich das bei steigender Variablenanzahl als mühevoll, denn die Anzahl der Überprüfungen ist 2^n wenn n die Anzahl der Aussagenvariablen ist. Die Frage nach dem maximal nötigen Aufwand (in Abhängigkeit von der Größe der Aussagen) für die Bestimmung, ob eine Aussage erfüllbar ist, ist ein berühmtes offenes Problem der (theoretischen) Informatik, das $SAT \in \mathcal{P}$-Problem, dessen Lösung weitreichende Konsequenzen hätte, z.B. würde $\mathcal{P} = \mathcal{NP}$ daraus folgen. Im Moment geht man davon aus, dass dies nicht gilt.

[2] Hierbei ist $F[B]$ die Aussage, die dadurch entsteht, dass in F die Subaussage A durch B ersetzt wird.

Lemma 1.3.3 (Äquivalenzen)**.**

$\neg(\neg A))$	$\iff$	A	
$(A \Rightarrow B)$	$\iff$	$(\neg A \vee B)$	
$(A \iff B)$	$\iff$	$((A \Rightarrow B) \wedge (B \Rightarrow A))$	
$\neg(A \wedge B)$	$\iff$	$\neg A \vee \neg B$	(*DeMorgansche Gesetze*)
$\neg(A \vee B)$	$\iff$	$\neg A \wedge \neg B$	
$A \wedge (B \vee C)$	$\iff$	$(A \wedge B) \vee (A \wedge C)$	*Distributivität*
$A \vee (B \wedge C)$	$\iff$	$(A \vee B) \wedge (A \vee C)$	*Distributivität*
$(A \Rightarrow B)$	$\iff$	$(\neg B \Rightarrow \neg A)$	*Kontraposition*
$A \vee (A \wedge B)$	$\iff$	A	*Absorption*
$A \wedge (A \vee B)$	$\iff$	A	*Absorption*

Diese Regeln sind nach Satz 1.3.1 nicht nur verwendbar, wenn A, B, C Aussagevariablen sind, sondern auch, wenn A, B, C für Aussagen stehen.

1.4 Normalformen

Für Aussagen der Aussagenlogik gibt es verschiedene Normalformen. U.a. die *disjunktive Normalform* (*DNF*) und die *konjunktive Normalform* (*CNF*): Die letzte nennt man auch Klauselnormalform.

- *disjunktive Normalform* (*DNF*). Die Aussage ist eine Disjunktion von Konjunktionen von Literalen. D.h. von der Form

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

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

- *konjunktive Normalform* (*CNF*). Die Aussage ist eine Konjunktion von Disjunktionen von Literalen. D.h. von der Form

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

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

Die Klauselnormalform wird oft als Menge von Mengen notiert und auch behandelt. Dies ist gerechtfertigt, da sowohl $\wedge$ als auch $\vee$ assoziativ, kommutativ und idempotent sind, so dass Vertauschungen und ein Weglassen der Klammern erlaubt ist. Wichtig ist die Idempotenz, die z.B. erlaubt, eine Klausel $\{A, B, A, C\}$ als unmittelbar äquivalent zur Klausel $\{A, B, C\}$ zu betrachten. Eine Klausel mit einem Literal bezeichnet man auch als *1-Klausel* (Unit-Klausel). Eine Klausel (in Mengenschreibweise) ohne Literale wird als *leere Klausel* bezeichnet. Diese ist äquivalent zu 0, dem Widerspruch.

Lemma 1.4.1. *Eine Klausel C ist genau dann eine Tautologie, wenn es eine Variable A gibt, so dass sowohl A als auch $\neg A$ in der Klausel vorkommen*

Beweis. Übungsaufgabe.

Es gilt:

Lemma 1.4.2. *Zu jeder Aussage kann man eine äquivalente CNF finden und ebenso eine äquivalente DNF. Allerdings nicht in eindeutiger Weise.*

Lemma 1.4.3.

- *Eine Aussage in CNF kann man in Zeit $O(n * log(n))$ auf Tautologie-Eigenschaft testen.*
- *Eine Aussage in DNF kann man in Zeit $O(n * log(n))$ auf Unerfüllbarkeit testen.*

Beweis. Eine CNF $C_1 \wedge \ldots \wedge C_n$ ist eine Tautologie, gdw. für alle Interpretationen I diese Formel stets wahr ist. D.h. alle C_i müssen ebenfalls Tautologien sein. Das geht nur, wenn jedes C_i ein Paar von Literalen der Form $A, \neg A$ enthält.

Das gleiche gilt in dualer Weise für eine DNF, wenn man dualisiert, d.h. wenn man ersetzt: $\wedge \leftrightarrow \vee$, $0 \leftrightarrow 1$, CNF $\leftrightarrow$ DNF, Tautologie $\leftrightarrow$ Widerspruch. □

Der duale Test: CNF auf Unerfüllbarkeit bzw. DNF auf Allgemeingültigkeit ist die eigentliche harte Nuß.

Dualitätsprinzip Man stellt fest, dass in der Aussagenlogik (auch in der Prädikatenlogik) Definitionen, Lemmas, Methoden, Kalküle, Algorithmen stets eine duale Variante haben. Die Dualität ist wie im Beweis oben durch Vertauschung zu sehen: $\wedge \leftrightarrow \vee$, $0 \leftrightarrow 1$, CNF $\leftrightarrow$ DNF, Tautologie $\leftrightarrow$ Widerspruch, Test auf Allgemeingültigkeit $\leftrightarrow$ Test auf Unerfüllbarkeit. D.h. auch, dass die Wahl zwischen Beweissystemen, die auf Allgemeingültigkeit testen und solchen, die auf Unerfüllbarkeit testen, eine Geschmacksfrage ist und keine prinzipielle Frage.

Lemma 1.4.4. *Sei F eine Formel. Dann ist F allgemeingültig gdw. $\neg F$ ein Widerspruch ist*

Bemerkung 1.4.5. *Aus Lemma 1.4.3 kann man Schlussfolgerungen ziehen über die zu erwartende Komplexität eines Algorithmus, der eine Formel (unter Äquivalenzerhaltung) in eine DNF (CNF) transformiert. Wenn dieser Algorithmus polynomiell wäre, dann könnte man einen polynomiellen Tautologietest durchführen:*

> *Zuerst überführe eine Formel in CNF und dann prüfe, ob diese CNF eine Tautologie ist.*

Dies wäre ein polynomieller Algorithmus für ein $CoNP$-vollständiges Problem.

Allerdings sehen wir später, dass die DNF (CNF-)Transformation selbst nicht der Engpass ist, wenn man nur verlangt, dass die Allgemeingültigkeit (Unerfüllbarkeit) in eine Richtung erhalten bleibt.

Definition 1.4.6 (Transformation in Klauselnormalform)**.** *Folgende Prozedur wandelt jede aussagenlogische Formel in konjunktive Normalform (CNF, Klauselnormalform) um:*

1. *Elimination von $\Leftrightarrow$ und $\Rightarrow$:*

$$\begin{array}{lcl} F \Leftrightarrow G & \rightarrow & (F \Rightarrow G) \wedge (G \Rightarrow F) \\ F \Rightarrow G & \rightarrow & \neg F \vee G \end{array}$$

2. *Negation ganz nach innen schieben:*

$$\begin{array}{lcl} \neg\neg F & \rightarrow & F \\ \neg(F \wedge G) & \rightarrow & \neg F \vee \neg G \\ \neg(F \vee G) & \rightarrow & \neg F \wedge \neg G \end{array}$$

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

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

$$(L_{1,1} \vee \ldots \vee L_{1,n_1}) \wedge (L_{2,1} \vee \ldots \vee L_{2,n_2}) \wedge \ldots \wedge (L_{k,1} \vee \ldots \vee L_{k,n_k})$$

oder in (Multi-)Mengenschreibweise:

$$\{\{L_{1,1}, \ldots, L_{1,n_1}\}, \{L_{2,1}, \ldots, L_{2,n_2}\}, \ldots \{L_{k,1}, \ldots, L_{k,n_k}\}\}$$

Die so hergestellte CNF ist eine Formel, die äquivalent zur eingegebenen Formel ist. Dieser CNF-Algorithmus ist im schlechtesten Fall exponentiell, d.h. die Anzahl der Literale in der Klauselform wächst exponentiell mit der Größe der Ausgangsformel.

Es gibt zwei Schritte, die zum exponentiellem Anwachsen der Formel führen können,

- die Elimination von $\Leftrightarrow$: Betrachte die Formel $(A_1 \Leftrightarrow A_2) \Leftrightarrow (A_3 \Leftrightarrow A_4)$ und die Verallgemeinerung.

- Ausmultiplikation mittels Distributivgesetz:

  $$(A_1 \wedge \ldots \wedge A_n) \vee B_2 \vee \ldots \vee B_m \rightarrow ((A_1 \vee B_2) \wedge \ldots \wedge (A_n \vee B_2)) \vee B_3 \ldots \vee B_n$$

  Dies verdoppelt B_2 und führt zum Iterieren des Verdoppelns, wenn B_i selbst wieder zusammengesetzte Aussagen sind.

Beispiel 1.4.7.

$$
\begin{aligned}
& ((A \wedge B) \Rightarrow C) \Rightarrow C \\
\rightarrow \quad & \neg(\neg(A \wedge B) \vee C) \vee C \\
\rightarrow \quad & (A \wedge B) \wedge \neg C) \vee C \\
\rightarrow \quad & (A \vee C) \wedge (B \vee C) \wedge (\neg C \vee C)
\end{aligned}
$$

1.5 Lineare CNF

Wir geben einen Algorithmus an, der eine aussagenlogische Formel F in polynomieller (sogar fast-linearer) Zeit in eine CNF F_{CNF} umwandelt, wobei die Formel F erfüllbar ist gdw. F_{CNF} erfüllbar ist. Es ist hierbei nicht gefordert, dass F und F_{CNF} äquivalent als Formeln sind! Beachte, dass bei diesem Verfahren die Anzahl der Variablen erhöht wird.

Der Trick ist, komplexe Subformeln iterativ durch neue Variablen abzukürzen. Sei $F[G]$ eine Formel mit der Subformel G. Dann erzeuge daraus $(A \iff G) \wedge F[A]$, wobei A eine neue aussagenlogische Variable ist.

Lemma 1.5.1. *$F[G]$ ist erfüllbar gdw. $(G \iff X) \wedge F[X]$ erfüllbar ist. Hierbei muß X eine Variable sein, die nicht in $F[G]$ vorkommt.*

Beweis. "$\Rightarrow$" Sei $F[G]$ erfüllbar und sei I eine beliebige Interpretation mit $I(F[G]) = 1$. Erweitere I zu I', so dass $I'(X) := I(G)$. Werte die Formel $(G \iff X) \wedge F[X]$ unter I' aus: Es gilt $I'(G \iff X) = 1$ und $I'(F[G]) = I'(F[X]) = 1$.

"$\Leftarrow$" Sei $I(G \iff X) \wedge F[X]) = 1$ für eine Interpretation I. Dann ist $I(G) = I(A)$ und $I(F[X]) = 1$. Damit muss auch $I(F[G]) = 1$ sein. □

Analog dazu erhält diese Transformation auch die Widersprüchlichkeit.

Zunächst benötigen wir den Begriff Tiefe einer Aussage und Subformel in Tiefe n. Beachte hierbei aber, dass es jetzt auf die genaue syntaktische Form ankommt: Es muß voll geklammert sein. Man kann es auch für eine flachgeklopfte Form definieren, allerdings braucht man dann eine andere Syntaxdefinition usw. Wir betrachten die Tiefe der Formeln, die in einer Formel-Menge emthalten sind.

Definition 1.5.2. *Die Tiefe einer Subformel in Tiefe n definiert man entsprechend dem Aufbau der Syntax: Zunächst ist jede Formel F eine Subformel der Tiefe 0 von sich selbst. Sei H eine Subformel von F der Tiefe n. Dann definieren wir Subformeln von F wie folgt:*

- *Wenn $H \equiv \neg G$, dann ist G eine Subformel der Tiefe $n+1$*
- *Wenn $H \equiv (G_1 \; op \; G_2)$, dann sind G_1, G_2 Subformeln der Tiefe $n+1$, wobei op einer der Junktoren $\vee, \wedge, \Rightarrow, \iff$ sein kann.*

Die Tiefe einer Formel sei die maximale Tiefe einer Subformel.

Algorithmus **Schnelle CNF-Berechnung**

Eingabe: Aussagenlogische Formel F

Verfahren:
Sei F von der Form $H_1 \wedge \ldots \wedge H_n$, wobei H_i keine Konjunktionen sind.
while ein H_i Tiefe ≥ 4 besitzt **do**
  for $i \in \{1, \ldots, n\}$ **do**
    if H_i hat Tiefe ≥ 4 **then**
      Seien $G_1, \ldots, G_m$ alle Subformeln von H_i in Tiefe 3, die selbst Tiefe ≥ 1 haben
      Ersetze H_i wie folgt:
      $H_i[G_1, \ldots, G_m] \rightarrow (G_1 \iff X_1) \wedge \ldots \wedge (G_m \iff X_m) \wedge H_i[X_1, \ldots, X_m]$
    end if
  end for
  Iteriere mit der entstandenen Formeln als neues F
end while
Wende die (langsame) CNF-Erstellung auf die entstandene Formel an

Beachte, dass nach Ablauf der while-Schleife, alle Subformeln H_i eine Tiefe ≤ 3 besitzen und die langsame CNF-Erstellung nur die H_i Formeln einzeln transformieren muss, da die Konjunktionen zwischen den H_i bereits in der richtigen Form sind. Jede dieser Formeln hat Tiefe ≤ 3, daher kann man die Laufzeit für die Transformation einer solchen kleinen Formel als konstant ansehen.

Beachte dass die entstande Teilformel $H_i[X_1, \ldots, X_m]$ Tiefe 3 hat und daher nicht nochmal bearbeitet wird. Die Formeln $(G_j \iff X_j)$ müssen evtl. weiter bearbeitet werden, allerdings nur innerhalb von G_j. Insbesondere gilt dabei: Wenn H_i vorher Tiefe $n > 3$ hat, dann hat $H_i[X_1, \ldots, X_m]$ Tiefe 3 und jede der Formeln G_j hat selbst Tiefe maximal $n - 3$. Jede neuen Formeln $(G_j \iff X_j)$ hat der Tiefe maximal $n - 2$. D.h. insgesamt wird im schlechsten Fall ein H_i der Tiefe n durch mehrere Formeln G_j der Tiefe $n - 2$ ersetzt (wobei $n > 3$). Dies zeigt schon die Terminierung des Verfahrens. Da jede Subformel der ursprünglichen Formel höchstens einmal durch eine neue Abkürzung abgekürzt wird, können höchstens linear viele solcher Ersetzungen passieren. Wenn man das Verfahren geschickt implementiert (durch Merken der nicht zu betrachtenden Subformeln), kann daher die CNF in linearer Zeit in der Größe der Eingabeformel erstellt werden.

Satz 1.5.3. *Zu jeder aussagenlogischen Formel kann unter Erhaltung der Erfüllbarkeit eine CNF in Zeit $O(n)$ berechnet werden, wobei n die Größe der aussagenlogischen Formel ist.*

Beachte, dass die schnelle CNF-Berechnung (sofern mindestens eine Abkürzung eingeführt wird), die Tautologieeigenschaft *nicht* erhält.

Bemerkung 1.5.4. *Sei $F[G]$ eine Tautologie. Dann ist die Formel $F[X] \wedge (X \iff G)$, wobei X eine neue aussagenlogische Variable, keine Tautologie: Seie I eine Interpretation, die $F[G]$*

wahr macht (*d.h.* $I(F[G]) = 1$).

$$\textit{Setze } I'(Y) := \begin{cases} I(Y) & \textit{falls } Y \neq X \\ f_\neg(I(G)) & \textit{falls } Y = X \end{cases}$$

Dann falsifiziert I' *die Formel* $F[X] \wedge (X \iff G)$, *denn* $I'(F[X] \wedge (X \iff G)) = f_\wedge(I'(F[X]), f_\iff(I'(X), I'(G))) = f_\wedge(I'(F[X]), f_\iff(f_\neg I(G), I(G))) = f_\wedge(I'(F[X]), 0) = 0$.

Beachte auch, dass es gar keinen polynomiellen Algorithmus zur CNF-Berechnung geben kann (solange $\mathcal{P} \neq Co\mathcal{NP}$), der eine äquivalente CNF berechnet, da man dann einen polynomiellen Algorithmus zum Test auf Tautologieeigenschaft hätte.

Dual zur schnellen Herstellung der CNF ist wieder die schnelle Herstellung einer DNF, die allerdings mit einer anderen Transformation durchgeführt wird: $F[G] \longrightarrow (G \iff X) \Rightarrow F[X]$. Diese Transformation erhält die Falsifizierbarkeit (und daher auch die Tautologieeigenschaft), aber die Erfüllbarkeit bleibt nicht erhalten. Genauer gilt $(G \iff X) \Rightarrow F[X]$ ist stets erfüllbar, unabhängig davon, ob $F[G]$ erfüllbar ist oder nicht (Beweis: Übungsaufgabe).

Beispiel 1.5.5. *Wandle eine DNF in eine CNF auf diese Art und Weise um unter Erhaltung der Erfüllbarkeit. Sei* $(D_{11} \wedge D_{12}) \vee \ldots \vee (D_{n1} \wedge D_{n2})$ *die DNF. Wenn man das Verfahren leicht abwandelt, damit man besser sieht, was passiert: ersetzt man die Formeln* $(D_{i1} \wedge D_{i2})$ *durch neue Variablen* A_i *und erhält am Ende:*
$(A_1 \vee \ldots \vee A_n) \wedge (\neg A_1 \vee D_{11}) \wedge (\neg A_1 \vee D_{12}) \wedge (\neg D_{11} \vee \neg D_{12} \vee A_1) \wedge \ldots$. *Diese Formel hat* $8n$ *Literale.*

Beispiel 1.5.6. *Wandle die Formel*

$$(((((X \iff Y) \iff Y) \iff Y) \iff Y) \iff X)$$

um. Das ergibt:

$$(A \iff ((X \iff Y) \iff Y)) \Rightarrow ((((A \iff Y) \iff Y) \iff Y) \iff X)$$

Danach kann man diese Formel dann auf übliche Weise in DNF umwandeln.

Bemerkung 1.5.7. *Es gibt zahlreiche Implementierungen von Algorithmen, die Formeln in Klauselmengen verwandelt. Z.B. siehe die www-Seite* `http://www.spass-prover.org/`.

1.6 Resolution für Aussagenlogik

Das Resolutionsverfahren dient zum Erkennen von Widersprüchen, wobei statt des Testes auf Allgemeingültigkeit einer Formel F die Formel $\neg F$ auf Unerfüllbarkeit getestet wird.

Eine Begründung wurde bereits gegeben. Eine erweiterte liefert das folgende Lemma zum Beweis durch Widerspruch:

Lemma 1.6.1. *Eine Formel $A_1 \wedge \ldots \wedge A_n \Rightarrow F$ ist allgemeingültig gdw. $A_1 \wedge \ldots \wedge A_n \wedge \neg F$ widersprüchlich ist.*

Beweis. Übungsaufgabe □

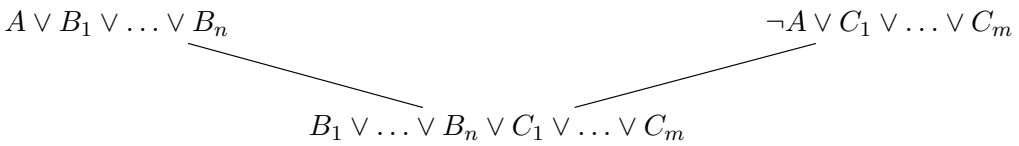
Die semantische Entsprechung ist:

Lemma 1.6.2. *$\{A_1, \ldots, A_n\} \models F$ gdw. es keine Interpretation I gibt, so dass $I \models \{A_1, \ldots, A_n, \neg F\}$.*

Die Resolution ist eine Regel mit der man aus zwei Klauseln einer Klauselmenge eine weitere herleiten und dann zur Klauselmenge hinzufügen kann.
Resolution:

$$\begin{array}{ll} A & \vee B_1 \vee \ldots \vee B_n \\ \neg A & \vee C_1 \vee \ldots \vee C_m \\ \hline & B_1 \vee \ldots \vee B_n \vee C_1 \vee \ldots \vee C_m \end{array}$$

Man nennt die ersten beiden Klauseln auch Elternklauseln und die neu hergeleitete Klausel *Resolvente*. Oft verwendet man auch die folgende Schreibweise als „Graph“:



Auf der Ebene der Klauselmengen sieht das Verfahren so aus:

$$C \quad \rightarrow \quad C \cup \{R\}$$

wobei R eine Resolvente ist, die aus zwei Klauseln von C berechnet worden ist. Man nimmt der Einfachheit halber an, dass Klauseln Mengen sind, d.h. es kommen keine Literale doppelt vor. Außerdem nimmt man auch noch an, dass die Konjunktion der Klauseln eine Menge ist, d.h. nur neue Klauseln, die nicht bereits vorhanden sind, können hinzugefügt werden. Wird die leere Klausel hergeleitet, ist das Verfahren beendet, denn ein Widerspruch wurde hergeleitet.

Eingesetzt wird die Resolution zur Erkennung unerfüllbarer Klauselmengen. Es werden solange Resolventen hergeleitet, bis entweder die leere Klausel hinzugefügt wurde oder keine neue Resolvente mehr herleitbar ist. Dies geschieht meist in der Form, dass man Axiome (als Konjunktion) eingibt und ebenso eine negierte Folgerung, so dass die Unerfüllbarkeit bedeutet, dass man einen Widerspruch hergeleitet hat.

Wenn man keine neuen Klauseln mehr herleiten kann, oder wenn besonders kurze (aussagekräftige) Klauseln hergeleitet werden, kann man diese als echte Folgerungen aus den eingegebenen Formeln ansehen, und evtl. ein Modell konstruieren. Allerdings ist das bei obiger Widerspruchsvorgehensweise nicht unbedingt ein Modell der Axiome, da die negiert eingegebene Folgerung dazu beigetragen haben kann. Ist die kleine Formel nur aus den Axiomen hergeleitet worden, dann hat man ein Lemma, dass in allen Modellen der Axiome gilt.

Lemma 1.6.3. *Wenn $C \to C'$ mit Resolution, dann ist C äquivalent zu C'.*

Beweis. Wir zeigen den nichttrivialen Teil:

Sei I eine Interpretation, die sowohl $A \vee B_1 \vee \ldots \vee B_n$ und $\neg A \vee C_1 \vee \ldots \vee C_m$ wahr macht. Wenn $I(A) = 1$, dann gibt es ein C_j, so dass $I(C_j) = 1$. Damit ist auch die Resolvente unter I wahr. Im Fall $I(A) = 0$ gilt $I(B_j) = 1$ für ein j und somit ist die Resolvente wahr unter der Interpretation. □

Lemma 1.6.4. *Die Resolution auf einer aussagenlogischen Klauselmenge terminiert, wenn man einen Resolutionsschritt nur ausführen darf, wenn sich die Klauselmenge vergrößert.*

Beweis. Es gibt nur endlich viele verschiedene Klauseln, da Resolution keine neuen Variablen einführt. □

Übungsaufgabe 1.6.5. *Gebe ein Beispiel an, so dass R sich aus C_1, C_2 mit Resolution herleiten läßt, aber $C_1 \wedge C_2 \iff R$ ist falsch*

Da Resolution die Äquivalenz der Klauselmenge als Formel erhält, kann man diese auch verwenden, um ein Modell zu erzeugen, bzw. ein Modell einzuschränken. Leider ist diese Methode nicht immer erfolgreich: Zum Beispiel betrachte man die Klauselmenge $\{\{A, B\}, \{\neg A, \neg B\}\}$. Resolution ergibt:

$$\{\{A, B\}, \{A, \neg A\}, \{B, \neg B\}, \{\neg A, \neg B\}\}$$

D.h. es wurden zwei tautologische Klauseln hinzugefügt. (Eine Klausel ist eine Tautologie, wenn $P, \neg P$ vorkommt für ein P.) Ein Modell läßt sich nicht direkt ablesen. Zum Generieren von Modellen ist z.B. die DPLL-Prozedur (siehe 1.7) geeigneter.

Was jetzt noch fehlt, ist ein Nachweis der naheliegenden Vermutung, dass die Resolution für alle unerfüllbaren Klauselmengen die leere Klausel auch findet.

Theorem 1.6.6. *Für eine unerfüllbare Klauselmenge findet Resolution nach endlich vielen Schritten die leere Klausel.*

Beweis. Wir verwenden das folgende Maß für Klauselmengen C:

$\#aou(C) =$ Anzahl der verschiedenen Atome in C, die nicht innerhalb von Literalen von 1-Klauseln in C vorkommen

Z.B. ist $\#aou(\{\{A, \neg B\}, \{B, C\}, \{\neg A\}, \{\neg B\}\}) = 1$, da von den drei Atomen A, B, C nur C nicht in einer 1-Klausel vorkommt.

Sei C eine unerfüllbare Klausel. Wir zeigen per Induktion über $\#aou(C)$, dass das Resolutionsverfahren stets die leere Klausel herleitet.

Für die Induktionsbasis nehmen wir an, dass $\#aou(C) = 0$ ist, d.h. sämtliche Atome kommen in 1-Klauseln vor. Wenn C bereits die leere Klausel enthält, oder zwei 1-Klauseln der Form $\{A\}$ und $\{\neg A\}$ für ein Atom A, dann sind wir fertig (im zweiten Fall muss die leere Klausel irgendwann durch Resolution der Klauseln $\{A\}$ und $\{\neg A\}$ hergeleitet werden). Anderenfalls gibt es 1-Klauseln und Klauseln mit mehr als einem Literal, wobei die 1-Klauseln untereinander nicht resolvierbar sind. O.B.d.A. sei $C = \{\{L_1\}, \ldots, \{L_n\}, K_1, \ldots, K_m\}$, wobei alle Klauseln K_i keine 1-Klauseln sind. Betrachte die Interpretation I mit $I(L_i) = 1$ für $i = 1, \ldots, n$. Da C unerfüllbar ist, muss C jedoch eine Klausel K enthalten mit $I(K) = 0$. K kann keines der Literale L_i enthalten, sonst würde $I(K) = 1$ gelten. Daher enthält K nur Literale $\overline{L_i}$, wobei $\overline{\neg A} = A$ und $\overline{A} = \neg A$. Offensichtlich kann man die passenden 1-Klauseln $\{L_i\}$ zusammen mit K verwenden, um mit mehrfacher Resolution die leere Klausel herzuleiten.

Nun betrachten wir den Induktionsschritt. Sei $\#aou(C) = n > 0$. Dann gibt es mindestens ein Atom, dass in keiner 1-Klausel vorkommt. O.B.d.A. sei A dieses Atom. Betrachte die Klauselmenge $C' := C \cup \{\{A\}\}$. Jede Interpretation die C falsch macht, macht offensichtlich auch C' falsch. Daher ist C' auch unerfüllbar. Außerdem gilt $\#aou(C') = n - 1$. Daher folgt aus der Induktionshypothese, dass es eine Resolutionsherleitung für C' gibt, welche die leere Klausel herleitet. Verwende diese Herleitung wie folgt: Streiche die Klausel $\{A\}$ aus der Herleitung heraus. Dies ergibt eine Herleitung für C, die entweder immer noch die leere Klausel herleitet (dann sind wir fertig), oder die Klausel $\{\neg A\}$ herleitet. Im letzten Fall betrachte die Klauselmenge $C'' := C \cup \{\{\neg A\}\}$. Auch C'' ist unerfüllbar (aufgrund der Unerfüllbarkeit von C). Ebenso gilt $\#aou(C'') = n - 1$, und daher (aufgrund der Induktionshypothese) existiert eine Herleitung der leeren Klauseln beginnend mit C''. Nun streichen wir aus dieser Herleitung die 1-Klausel $\{\neg A\}$. Erneut ergibt sich eine Resolutionsherleitung für C, die entweder die leere Klausel herleitet (dann sind wir fertig) oder, die die Klausel $\{A\}$ herleitet. Im letzten Fall verknüpfen wir die beiden Herleitungen von $\{\neg A\}$ und $\{A\}$ zu einer Herleitung (z.B. sequentiell). Aus der entstandenen Klauselmenge muss die Resolution die leere Klausel herleiten, da sie die beiden 1-Klauseln $\{\neg A\}$ und $\{A\}$ enthält. □

Theorem 1.6.7. *Resolution erkennt unerfüllbare Klauselmengen.*

Die Komplexität im schlimmsten Fall wurde von A. Haken (Haken, 1985) (siehe auch (Eder, 1992) nach unten abgeschätzt: Es gibt eine Folge von Formeln (die sogenannten Taubenschlag-formeln (pigeon hole formula, Schubfach-formeln), für die gilt, dass die kürzeste Herleitung der leeren Klausel mit Resolution eine exponentielle Länge (in der Größe der Formel) hat.

Betrachtet man das ganze Verfahren zur Prüfung der Allgemeingültigkeit einer aussagenlogischen Formel, so kann man eine CNF in linearer Zeit herstellen, aber ein Resolutionsbeweis muß mindestens exponentiell lange sein, d.h. auch exponentiell lange dauern.

Beachte, dass es formale Beweisverfahren gibt, die polynomiell lange Beweise für die Schubfachformeln haben. Es ist theoretisch offen, ob es ein Beweisverfahren gibt, das für alle Aussagen polynomiell lange Beweise hat: Dies ist äquivalent zum offenen Problem ob $\mathcal{NP} = Co\mathcal{NP}$.

1.6.1 Optimierungen des Resolutionskalküls

Es gibt verschiedene Optimierungen, die es erlauben, bestimmte Klauseln während der Resolution nicht weiter zu betrachten. Diese Klauseln können gelöscht, bzw. erst gar nicht erzeugt werden, ohne dass die Korrektheit und Vollständigkeit des Verfahrens verloren geht. In diesem Abschnitt werden wir drei solche Löschregeln behandeln und kurz deren Korrektheit begründen.

1.6.1.1 Tautologische Klauseln

Ein Literal ist positiv, wenn es Atom ist (z.B. A), und negativ, wenn es ein negiertes Atom (z.B. $\neg A$) ist.

Definition 1.6.8. *Eine Klausel ist* tautologisch, *wenn sie ein Literal sowohl positiv als auch negativ enthält.*

Aus einer Klauselmenge kann man tautologische Klauseln löschen, denn es gilt:

Satz 1.6.9. *Sei $C \cup \{K\}$ eine Klauselmenge, wobei K eine tautologische Klausel ist. Dann sind $C \cup \{K\}$ und C äquivalente Formeln.*

Beweis. Offensichtlich gilt: Jede Interpretation I macht die tautologische Klausel K wahr. Daher gilt $I(C \cup \{K\}) = f_\wedge(I(C), I(K)) = f_\wedge(I(C), 1) = I(C)$. □

D.h. während der Resolution kann man auf das Erzeugen von solchen Resolventen, die tautologische Klauseln sind, verzichten. In der initialen Klauselmenge kann man tautologische Klauseln bereits entfernen. Beachte insbesondere: Eine CNF ist eine Tautologie gdw. alle Klauseln tautologisch sind. D.h. für Tautologien (die komplette Klauselmenge) braucht man das Resolutionsverfahren gar nicht starten: Alle Klauseln werden durch das Löschen tautologischer Klauseln entfernt und man erhält die leere Klauselmenge.

1.6.1.2 Klauseln mit isolierten Literalen

Definition 1.6.10. *Ein Literal L ist* isoliert *in einer Klauselmenge C, wenn das Literal $\overline{L}$ in C nicht vorkommt. Dabei ist $\overline{L}$ das zu L negierte Literal, d.h.*

$$\overline{L} := \begin{cases} \neg X, & \text{wenn } L = X \\ X, & \text{wenn } L = \neg X \end{cases}$$

Beispiel 1.6.11. *In $\{\{A, \neg B\}, \{A, \neg B, C\}, \{\neg A, C\}\}$ ist das Literal C isoliert, da $\neg C$ nicht in der Klauselmenge vorkommt. Ebenso ist $\neg B$ isoliertes Literal, da B nicht in der Klauselmenge vorkommt.*

Man kann Klauseln, die isolierte Literale enthalten, aus einer Klauselmenge entfernen. Dabei wird die Erfüllbarkeit der Klauselmenge nicht verändert:

Satz 1.6.12. *Sei C eine Klauselmenge und C' die Klauselmenge, die aus C entsteht, indem alle Klauseln entfernt werden, die isolierte Literale enthalten. Dann ist C erfüllbar gdw. C' erfüllbar ist.*

Beweis. Sei C erfüllbar und I eine Interpretation mit $I(C) = 1$. Dann gilt offensichtlich $I(C') = 1$, da $C' \subseteq C$ (C' enthält weniger Klauseln als C).

Für die umgekehrte Richtung sei C' erfüllbar und I eine Interpretation mit $I(C') = 1$. Betrachte die Interpretation I' mit

$$I'(X) := \begin{cases} 1, & \text{wenn } X \text{ isoliertes Literal in } C \\ 0, & \text{wenn } \neg X \text{ isoliertes Literal in } C \\ I(X), & \text{sonst} \end{cases}$$

Es muss gelten $I'(C) = 1$, denn Klauseln mit isolierten Literalen werden offensichtlich wahr gemacht und alle anderen Klauseln werden bereits durch I wahr gemacht. D.h. C ist erfüllbar. □

Beachte, dass das Entfernen von Klauseln mit isolierten Literalen nicht die Äquivalenz der Klauselmengen erhält:

Bemerkung 1.6.13. *Betrachte die Klauselmenge $C = \{\{A, \neg A\}, \{B\}\}$. Das Literal B ist isoliert. Nach Löschen der Klausel entsteht $C' = \{\{A, \neg A\}\}$. Während C' eine Tautologie ist, ist C falsifizierbar (mit $I(B) = 0$).*

Für das Resolutionsverfahren reicht die Erfüllbarkeitsäquivenz jedoch aus, da das Verfahren auf Widersprüchlichkeit testet. Daher können Klauseln mit isolierten Literalen entfernt werden.

1.6.1.3 Subsumtion

Definition 1.6.14. *Eine Klausel K_1* subsumiert *eine Klausel K_2, wenn $K_1 \subseteq K_2$ gilt. Man sagt in diesem Fall auch: „K_2 wird von K_1 subsumiert“.*

Gibt es in einer Klauselmenge, Klauseln, die durch andere Klauseln subsumiert werden, so können die subsumierten Klauseln entfernt werden, denn es gilt:

Satz 1.6.15. *Sei $C \cup \{K_1\} \cup \{K_2\}$ eine Klauselmenge, wobei K_1 die Klausel K_2 subsumiert. Dann sind die Formeln $C \cup \{K_1\} \cup \{K_2\}$ und $C \cup \{K_1\}$ äquivalent.*

Beweis. Sei I eine Interpretation. Wir betrachten zwei Fälle:

- $I(C \cup \{K_1\}) = 1$. Dann muss gelten $I(K_1) = 1$. Da $K_1 \subseteq K_2$ muss ebenso gelten $I(K_2) = 1$ (beachte $K_1 \neq \emptyset$, da $I(K_1) = 1$) und daher macht I auch $C \cup \{K_1\} \cup \{K_2\}$ wahr.

- $I(C \cup \{K_1\}) = 0$, dann kann nur $I(C \cup \{K_1\} \cup \{K_2\}) = 0$ gelten.

□

Das zeigt, dass subsumierte Klauseln im Resolutionsverfahren entfernt werden können.

1.7 DPLL-Verfahren

Die Prozedur von Davis, Putnam, Logemann und Loveland (DPLL) dient zum Entscheiden der Erfüllbarkeit (und Unerfüllbarkeit) von aussagenlogische Klauselmengen. Sie ist eine Abwandlung eines vorher von Davis und Putnam vorgeschlagenen Verfahrens (und wird daher manchmal auch nur als Davis-Putnam-Prozedur bezeichnet).

Das Verfahren beruht auf Fallunterscheidung und Ausnutzen und Propagieren der Information. Die Korrektheit und Vollständigkeit des Verfahrens lässt sich aus der Resolution samt den Löschregeln leicht folgern.

Der Algorithmus hat eine Klauselmenge als Eingabe und liefert genau dann `true` als Ausgabe, wenn die Klauselmenge unerfüllbar ist. Wir nehmen an, dass tautologische Klauseln in der Eingabe bereits entfernt sind. Der genaue Algorithmus ist wie folgt:

Algorithmus **DPLL-Prozedur**

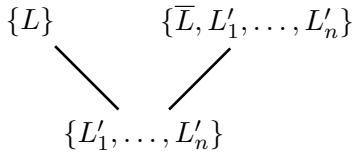
Eingabe: Klauselmenge C, die keine tautologischen Klauseln enthält

Funktion DPLL(C):
`if` $\emptyset \in C$ // *C enthält die leere Klausel*
`then return true;` // *unerfüllbar*
`endif`
`if` $C = \emptyset$ // *C ist die leere Menge*
`then return false;` // *erfüllbar*
`endif`
`if` C enthält 1-Klausel $\{L\}$ `then`
Sei C' die Klauselmenge, die aus C entsteht, indem
(1) alle Klauseln, die L enthalten, entfernt werden und
(2) in den verbleibenden Klausel alle Literale $\overline{L}$ entfernt werden
(wobei $\overline{L} = \neg X$, wenn $L = X$ und $\overline{L} = X$, wenn $L = \neg X$)
DPLL(C'); // *rekursiver Aufruf*
`endif`
`if` C enthält isoliertes Literal L `then`
Sei C' die Klauselmenge, die aus C entsteht, indem alle Klauseln, die L enthalten, entfernt werden.
DPLL(C'); // *rekursiver Aufruf*
`endif`
// *Nur wenn keiner der obigen Fälle zutraf:*
Wähle Atom A, dass in C vorkommt;
`return` DPLL($C \cup \{\{A\}\}$) $\wedge$ DPLL($C \cup \{\{\neg A\}\}$) // *Fallunterscheidung*

Dies ist ein vollständiges, korrektes Entscheidungsverfahren für die (Un-)Erfüllbarkeit von aussagenlogischen Klauselmengen.

Das Entfernen der Klauseln, die das Literal aus der 1-Klausel enthalten, ((1) im Algorithmus) entspricht der Subsumtion, da die 1-Klausel $\{L\}$ alle Klauseln subsumiert, die L enthalten. Das Entfernen der 1-Klausel selbst entspricht der Vorgehensweise, das entsprechende Literal L in der Interpretation wahr zu machen, da jedes Modell die 1-Klausel wahr machen muss.

Das Entfernen der Literale $\overline{L}$ in (2) entspricht der Resolution der jeweiligen Klausel mit der 1-Klausel $\{L\}$ und anschließender Subsumtion, denn die Resolvente subsumiert die ursprüngliche Klausel:



Das Entfernen der Klauseln, die isolierten Literale enthalten, entspricht genau der oben behandelten Löschregel für isolierte Literale.

Wenn all diese Regeln nicht mehr anwendbar sind (es gibt weder 1-Klauseln noch isolierte Literale), wird in der letzten Zeile eine Fallunterscheidung durchgeführt, die die beiden Fälle A ist wahr, oder A ist falsch probiert.

Diese DPLL-Prozedur ist im Allgemeinen (aus praktischer Sicht) sehr viel besser als eine vollständige Fallunterscheidung über alle möglichen Variablenbelegungen, d.h. besser als die Erstellung einer Wahrheitstafel. Die DPLL-Prozedur braucht im schlimmsten Fall exponentiell viel Zeit (in der Anzahl der unterschiedlichen Variablen), was nicht weiter verwundern kann, denn ein Teil des Problems ist gerade SAT, das Erfüllbarkeitsproblem für aussagenlogische Klauselmengen, und das ist bekanntlich $\mathcal{NP}$-vollständig.

Der DPLL-Algorithmus ist erstaunlich schnell, wenn man bei der Fallunterscheidung am Ende noch darauf achtet, dass man Literale auswählt, die in möglichst kurzen Klauseln vorkommen. Dies erhöht nämlich die Wahrscheinlichkeit, dass nach wenigen Schritten große Anteile der Klauselmenge gelöscht werden.

Der DPLL-Algorithmus kann leicht so erweitert werden, dass im Falle der Erfüllbarkeit der Klauselmenge auch ein Modell (eine erfüllende Interpretation) berechnet wird. Der Algorithmus arbeitet depth-first mit backtracking. Wenn die Antwort „erfüllbar“ ist, kann man durch Rückverfolgung der folgenden Annahmen ein Modell bestimmen:

1. Isolierte Literale werden als wahr angenommen.

2. Literale in 1-Klauseln werden ebenfalls als wahr angenommen.

3. Dadurch nicht belegt Variablen können für das vollständige Modell beliebig belegt werden

Beispiel 1.7.1. *Betrachte folgende Klauselmenge, wobei jede Zeile einer Klausel entspricht.*

$$
\begin{array}{lll}
P, & Q & \\
\neg P, & Q & R \\
P, & \neg Q, & R \\
\neg, P & \neg Q, & R \\
P, & Q, & \neg R \\
\neg P, & Q, & \neg R \\
P, & \neg Q, & \neg R \\
\neg P, & \neg Q, & \neg R
\end{array}
$$

Fall 1: Addiere die Klausel $\{P\}$. Das ergibt nach einigen Schritten:

$$\begin{array}{ll} Q, & R \\ \neg Q, & R \\ Q, & \neg R \\ \neg Q, & \neg R \end{array}$$

Fall 1.1: Addiere $\{Q\}$: ergibt die leere Klausel.
Fall 1.2: Addiere $\{\neg Q\}$: ergibt die leere Klausel.
Fall 2: Addiere die Klausel $\{\neg P\}$. Das ergibt nach einigen Schritten:

$$\begin{array}{ll} Q & \\ \neg Q & R \\ Q & \neg R \\ \neg Q & \neg R \end{array}$$

Weitere Schritte für Q ergeben

$$\begin{array}{l} R \\ \neg R \end{array}$$

Auch dies ergibt sofort die leere Klausel. Damit hat die DPLL-Prozedur die eingegebene Klauselmenge als unerfüllbar erkannt.

Beispiel 1.7.2. *Wir nehmen uns ein Rätsel von Raymond Smullyan vor: Die Frage nach dem Pfefferdieb. Es gibt drei Verdächtige: Den Hutmacher, den Schnapphasen und die (Hasel-)Maus. Folgendes ist bekannt:*

- *Genau einer von ihnen ist der Dieb.*
- *Unschuldige sagen immer die Wahrheit*
- *Schnapphase: der Hutmacher ist unschuldig.*
- *Hutmacher: die Hasel-Maus ist unschuldig*

Kodierung: H, S, M sind Variablen für Hutmacher, Schnapphase, Maus und bedeuten jeweils "ist schuldig". Man kodiert das in Aussagenlogik und fragt nach einem Modell.

- $H \vee S \vee M$
- $H \Rightarrow \neg(S \vee M)$
- $S \Rightarrow \neg(H \vee M)$
- $M \Rightarrow \neg(H \vee S)$
- $\neg S \Rightarrow \neg H$
- $\neg H \Rightarrow \neg M$

Dies ergibt eine Klauselmenge (doppelte sind schon eliminiert):

1. H, S, M
2. $\neg H, \neg S$
3. $\neg H, \neg M$
4. $\neg S, \neg M$
5. $S, \neg H$
6. $H, \neg M$

Wir können verschiedene Verfahren zur Lösung verwenden.

1. *Resolution ergibt:* $2+5 : \neg H$, $3+6 : \neg M$. *Diese beiden 1-Klauseln mit 1 resolviert ergibt:* S. *Nach Prüfung, ob* $\{\neg H, \neg M, S\}$ *ein Modell ergibt, können wir sagen, dass der Schnapphase schuldig ist.*
2. *DPLL: Wir verfolgen nur einen Pfad im Suchraum:*
   Fall 1: $S = 0$. *Die 5-te Klausel ergibt dann* $\neg H$. *Danach die 6te Klausel* $\neg M$. *Zusammen mit (den Resten von) Klausel 1 ergibt dies ein Widerspruch.*
   Fall 2: $S = 1$. *Dann bleibt von der vierten Klausel nur* $\neg M$ *übrig, und von der zweiten Klausel nur* $\neg H$. *Diese ergibt somit das gleiche Modell.*

Beispiel 1.7.3. *Eine Logelei aus der Zeit: Abianer sagen die Wahrheit, Bebianer Lügen. Es gibt folgende Aussagen:*

1. *Knasi: Knisi ist Abianer.*
2. *Knesi: Wenn Knösi Bebianer, dann ist auch Knusi ein Abianer.*
3. *Knisi: Wenn Knusi Abianer, dann ist Knesi Bebianer.*
4. *Knosi: Knesi und Knüsi sind beide Abianer.*
5. *Knusi: Wenn Knüsi Abianer ist, dann ist auch Knisi Abianer.*
6. *Knösi: Entweder ist Knasi oder Knisi Abianer.*
7. *Knüsi: Knosi ist Abianer.*

Eine offensichtliche Kodierung ergibt

```
A <=> I
E <=>  (-OE => U)
I <=> (U => -E)
O <=> (E /\ UE)
```

```
U <=> (UE => I)
OE <=> (A XOR I)
UE <=> O
```

Die Eingabe in den DPLL-Algorithmus[3] *ergibt:*

```
abianer1Expr = "((A <=> I) /\\ (E <=> (-OE => U)) /\\ (I <=> (U => -E))
               /\\ (O <=> (E /\\ UE)) /\\ (U <=> (UE => I))
                /\\ (OE <=> -(A <=> I)) /\\ (UE <=> O))"


 Resultat:
"Modell: -OE, -O, -UE, E, U, -I, -A"
```

Damit sind Knesi und Knusi Abianer, die anderen sind Bebianer.

Beispiel 1.7.4. *Ein weiteres Rätsel von Raymond Smullyan:*

Hier geht es um den Diebstahl von Salz. Die Verdächtigen sind: Lakai mit dem Froschgesicht, Lakai mit dem Fischgesicht, Herzbube.

Die Aussagen und die bekannten Tatsachen sind:

- *Frosch: der Fisch wars*
- *Fisch: ich wars nicht*
- *Herzbube: ich wars*
- *Genau einer ist der Dieb*
- *höchstens einer hat gelogen*

Man sieht, dass es nicht nur um die Lösung des Rätsels selbst geht, sondern auch um etwas Übung und Geschick, das Rätsel so zu formalisieren, dass es von einem Computer gelöst werden kann. Man muß sich auch davon überzeugen, dass die Formulierung dem gestellten Problem entspricht. Wir wollen Aussagenlogik verwenden. Wir verwenden Variablen mit folgenden Namen und Bedeutung:

FRW	*Frosch sagt die Wahrheit*
FIW	*Fisch sagt die Wahrheit*
HBW	*Herzbube sagt die Wahrheit*
FID	*der Fisch ist der Dieb*
FRD	*der Frosch ist der Dieb*
HBD	*der Herzbube ist der Dieb*

Die Formulierung ist:

[3] Eine Implementierung zum einfachen Ausprobieren ist über www.ki.informatik.uni-frankfurt.de/lehre/allgemein/DP/ zu finden.

höchstens einer sagt nicht die Wahrheit:

$$\begin{aligned}\neg FRW &\Rightarrow FIW \wedge HBW\\ \neg FIW &\Rightarrow FRW \wedge HBW\\ \neg HBW &\Rightarrow FRW \wedge FIW\end{aligned}$$

genau einer ist der Dieb:

$$FID \vee FRD \vee HBD$$

$$\begin{aligned}FID &\Rightarrow \neg FRD \wedge \neg HBD\\ FRD &\Rightarrow \neg FID \wedge \neg HBD\\ HBD &\Rightarrow \neg FID \wedge \neg FRD\end{aligned}$$

Die Aussagen:

$$\begin{aligned}FRW &\Rightarrow FID\\ FIW &\Rightarrow \neg FID\\ HBW &\Rightarrow HBD\end{aligned}$$

Eingabe in den DPLL-Algorithmus:

```
dp "((-FRW => FIW /\\ HBW) /\\ (-FIW =>  FRW /\\  HBW)
    /\\ (-HBW =>  FRW /\\ FIW)
    /\\  (FID => -FRD /\\ -HBD) /\\ (FRD => -FID /\\ -HBD)
    /\\ (HBD => -FID /\\ -FRD) /\\  (FRW =>  FID)
    /\\ (FIW =>  -FID) /\\ (HBW =>  HBD))"
```

Die berechnete Lösung ist: $-FRD, -FID, -FRW, FIW, HBW, HBD$ *D.h FRW ist falsch, d.h. der Lakai mit dem Froschgesicht hat gelogen und der Herzbube war der Dieb.*

Der Aufruf

```
dpalle fischfroschexpr
```

(siehe Programme zur Veranstaltung) ergibt, dass es genau ein Modell gibt, also gibt es nur die eine Lösung.

Beispiel 1.7.5. *Anwendung auf ein Suchproblem: Das n-Damen Problem. s sollen Königinnen auf einem quadratischen Schachbrett der Seitenlänge n so platziert werden, dass diese sich nicht schlagen können. Damit die Formulierung einfacher wird, erwarten wir, dass sich in jeder Zeile und Spalte eine Königin befindet. Man kann leicht ein Programm schreiben, dass die entsprechende Klauselmenge direkt erzeugt. Wir demonstrieren dies in Haskell, wobei wir Literale durch Zahlen darstellen: Negative Zahlen entsprechen dabei negativen Literalen.*

Zunächst muss man sich verdeutlichen, welche Bedingungen zur Lösung erforderlich sind: Pro Zeile und pro Spalte muss eine Dame stehen und es darf keine bedrohenden Paare von Damen geben. In Haskell programmiert ergibt das:

```
nDamen n =
    (proZeileEineDame n)
 ++ (proSpalteEineDame n)
 ++ (bedrohendePaare n)
```

Der Einfachheit halber kodieren wir die $n \times n$ Felder durch Zahlen von 1 bis n^2, verwenden dafür aber eine Funktion, die uns Koordinaten (im Bereich $(1,1)$ bis (n,n) direkt umrechnet:

```
koordinateZuZahl (x,y) n =  (x-1)*n+y
```

Die Bedingungen, dass eine Dame pro Zeile bzw. pro Spalte vorkommt, lassen sich leicht als Klauselmengen Programmieren: Z.B. wird dies für Zeile i zugesichert, indem alle Felder in Zeile i in einer Klausel vereint werden, damit ist eines dieser Felder sicher belegt. In Haskell kann man das leicht mit List Comprehensions programmieren:

```
proZeileEineDame  n =  [[koordinateZuZahl (i,j) n | j <- [1..n]] | i <- [1..n]]
proSpalteEineDame n =  [[koordinateZuZahl (i,j) n | i <- [1..n]] | j <- [1..n]]
```

Schließlich fügen wir Klauseln der Form $\{\neg X, \neg Y\}$ hinzu, die zusichern, dass niemals zwei Felder X und Y gleichzeitig belegt sind, sofern sich die Felder in der gleichen Zeile, Spalte, oder Diagonalen befinden:

```
bedrohendePaare n =  [[negate (koordinateZuZahl  (x1,y1) n),  negate (koordinateZuZahl  (x2,y2) n)]
| x1 <- [1..n],
y1 <- [1..n],
x2 <- [1..n],
y2 <- [1..n],
(x1,y1) < (x2,y2),
bedroht (x1,y1,x2,y2)]

bedroht (a,x,b,y)
 | a == b = True
 | x == y = True
 | abs (a-b) == abs (y-x) = True
 | otherwise = False
```

Die Bedingung `(x1,y1) < (x2,y2)` *ist dabei schon eine Optimierung die Symmetrien vermeidet. Iim Fall $n = 4$ erzeugt dies die Klauselmenge*

```
[[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16],[1,5,9,13],[2,6,10,14],[3,7,11,15],[4,8,12,16],
[-1,-5],[-1,-9],[-1,-13],[-1,-2],[-1,-6],[-1,-3],[-1,-11],[-1,-4],[-1,-16],[-5,-9],[-5,-13],
[-5,-2],[-5,-6],[-5,-10],[-5,-7],[-5,-15],[-5,-8],[-9,-13],[-9,-6],[-9,-10],[-9,-14],[-9,-3],[-9,-11],
[-9,-12],[-13,-10],[-13,-14],[-13,-7],[-13,-15],[-13,-4],[-13,-16],[-2,-6],[-2,-10],[-2,-14],[-2,-3],
[-2,-7],[-2,-4],[-2,-12],[-6,-10],[-6,-14],[-6,-3],[-6,-7],[-6,-11],[-6,-8],[-6,-16],[-10,-14],[-10,-7],
[-10,-11],[-10,-15],[-10,-4],[-10,-12],[-14,-11],[-14,-15],[-14,-8],[-14,-16],[-3,-7],[-3,-11],[-3,-15],
[-3,-4],[-3,-8],[-7,-11],[-7,-15],[-7,-4],[-7,-8],[-7,-12],[-11,-15],[-11,-8],[-11,-12],[-11,-16],
[-15,-12],[-15,-16],[-4,-8],[-4,-12],[-4,-16],[-8,-12],[-8,-16],[-12,-16]]
```

Das Ergebnis der DPLL-Prozedur sind zwei Interpretationen:

```
[[-4,-8,-15,5,-13,14,-6,-2,12,-9,-1,3,-16,-10,-7,-11],
 [-4,2,8,-6,-1,9,-12,-14,-13,-5,15,-3,-16,-10,-7,-11]]
```

Die entsprechen den zwei möglichen Platzierungen im Fall $n = 4$.

Der Aufruf `dpqueens 8` *ergibt nach kurzer Zeit:*

```
- - D - - - - -
- - - - - - - D -
- D - - - - - -
- - - - - - - - D
- - - - D - - -
D - - - - - - -
- - - D - - - -
- - - - - D - -
```

Beispiel 1.7.6. *Das n-Damen Problem gibt es noch in weiteren Varianten: Die Torus-Variante: In dem Fall setzt sich die Bedrohung über der Rand an der Seite und oben und unten fort. Die die Bedrohung entlang Diagonallinien des* 8×8 *Feldes ist so dass die Dame in der ersten Zeile im unteren Beispiel die Dame in der vierten Zeile bedroht, wenn man die Diagonale nach links unten verfolgt.*

```
- - D - - - - -
- - - - - - - D -
- D - - - - - -
- - - - - - - - D
- - - - D - - -
D - - - - - - -
- - - D - - - -
- - - - - D - -
```

Recherche im Internet ergibt (*z.B Polya hat sich schon damit beschäftigt*)*: es gibt offenbar Lösungen, wenn* n *teilerfremd zu* 6 *ist und* $n \geq 5$*. D.h. für* $n = 5, 7, 11, 13, 17, ..$ *gibt es Lösungen, für* $n = 4, 6, 8, 9, 10, 12, 14, 15, 16, \ldots$ *gibt es keine Lösungen. Eine Lösung* (*Aufruf:* `dpqueenscyl 5`) *ist:*

```
D - - - -
- - D - -
- - - - D
- D - - -
- - - D -
```

Für $n = 11$ *werden 4 Lösungen ermittelt und für* $n = 13$ *werden 174 Lösungen ermittelt* (*allerdings nach einer Symmetrieoptimierung*)*. Eine generische Lösung ist:*

```
D - - - - - - - - - - - -
- - D - - - - - - - - - -
- - - - D - - - - - - - -
- - - - - - D - - - - - -
- - - - - - - - D - - - -
- - - - - - - - - - D - -
- - - - - - - - - - - - D
- D - - - - - - - - - - -
- - - D - - - - - - - - -
- - - - - D - - - - - - -
- - - - - - - D - - - - -
- - - - - - - - - D - - -
- - - - - - - - - - - D -
```

Aus diesen Beispielen kann man sich auch eine Serie von erfüllbaren und unerfüllbaren Klauselmengen generieren, die man für Testzwecke verwenden kann.

1.8 Modellierung von Problemen als Aussagenlogische Erfüllbarkeitsprobleme

In diesem Abschnitt wollen wir kurz darauf eingehen, wie man aus einem gegebenen Problem eine passende Kodierung als Erfüllbarkeitsproblem der Aussagenlogik findet. Die Beispiele am Ende des letzten Abschnitts geben hier schon die Richtung vor. Man kann allerdings für häufige auftretende Kodierungsprobleme, allgemein Formeln angeben, die es erlauben, schnell Kodierungen zu finden. Außerdem soll dieser Abschnitt verdeutlichen, dass das *Modellieren* von Problemen als Aussagenlogische Erfüllbarkeitsprobleme oft systematisch möglich ist. Schließlich werden wir zum Abschluss des Kapitels die entwickelten Formeln verwenden, um als Anwendungsbeispiel das Lösen von Sudokus durch Verwendung der Aussagenlogik zu automatisieren.

Ein häufiges Problem bei der Modellierung sind Nebenbedingungen, die erfordern, dass eine gewisse Anzahl von Literalen im Modell wahr ist, oder nicht wahr ist. Im einfacheren Fall besteht die Aufgabe darin sicherzustellen, dass mindestens eine, höchstens eine oder genau ein Literal (oder auch Subformel) aus einer Menge von Literalen (bzw. Formeln) wahr ist.

Wir geben hier allgemein an, wie man diese Formeln erstellen kann. Sei S eine Menge von Formeln (oder im einfacheren Fall Literalen). Die Formel, die mindestens eine der Formeln aus S wahr macht, ist einfach die Disjunktion aller Formeln in S:

$$at\_least\_one(S) = (F_1 \vee \ldots \vee F_n)$$

Beachte, dass die Formel genau einer Klausel entspricht, wenn $F_1, \ldots, F_n$ Literale sind.

Die Formel, die höchstens eine der Formeln aus S wahr macht, kann man konstruieren, indem man sich überlegt, dass diese Anforderung identisch dazu ist, dass nie zwei Formeln aus S gleichzeitig wahr sein dürfen. Statt $\neg(F_i \wedge F_j)$ kann man gleich $(\neg F_i \vee \neg F_j)$ verwenden:

$$at\_most\_one(S) = \bigwedge\{(\neg F_i \vee \neg F_j) \mid i \in \{1,\ldots,n\}, j \in \{1,\ldots,n\}, i < j\}$$

Die Formel ist in CNF, wenn $F_1, \ldots, F_n$ Literale sind.

Die Kodierung der Bedingung, dass genau eine Formel wahr ist, ist nun die Konjunktion der at_least_one und at_most_one Formeln:

$$exactly\_one(S) = at\_least\_one(S) \wedge at\_most\_one(S)$$

Beispiel 1.8.1. *Wir betrachten das folgende Problem: s Studenten schreiben Klausuren. Der Dozent hat k verschiedene Klausurtypen entworfen und er weiß bereits, wie die Studenten im Klausurraum sitzen (er kennt die Paarungen aller benachbart sitzenden Studenten). Der Dozent möchte verhindern, dass benachbart sitzende Studenten den gleichen Klausurtyp erhalten. Das Problem besteht daher darin, welcher Student welchen Klausurtyp erhält.*

Wir verwenden die Aussagenlogischen Variablen:

$$s_i^j = \textit{Student } i \textit{ schreibt Klausurtyp } j$$

Das Problem hat im Grunde zwei Bedingungen: Jeder Student erhält genau eine Klausur und benachbarte Studenten erhalten nicht die gleiche Klausur. Dies lässt sich ausdrücken durch:

$$klausur\_formel(s,k,benachbart) =$$

Jeder Student erhält genau eine Klausur:
$$\underbrace{\bigwedge_{i=1}^{s}}_{\substack{\textit{alle Studenten}}} \underbrace{exactly\_one(\{s_i^j \mid j \in \{1..k\}\})}_{\textit{Student } i \textit{ genau eine Klausur}}$$

Benachbarte Studenten, nicht die gleiche Klausur:
$$\wedge \bigwedge_{(a,b)\in benachbart} \bigwedge_{j=1}^{k} (\neg s_a^j \vee \neg s_b^j)$$

Beachte, dass direkt eine CNF erzeugt wird, die z.B. als Eingabe für die DPLL-Prozedur verwendet werden kann. Ist die Formel widersprüchlich, dann gibt es zu wenige Klausurtypen. Ist die Formel erfüllbar, so lässt sich aus dem Modell ablesen, welche Klausur jeder Student erhält.

Die Verallgemeinerung der bisher definierten Formeln, ist es K viele mindestens, höchstens, oder genau als wahr zu fordern. Hierfür bietet es sich an, als Vorarbeit alle Teilmengen der Mächtigkeit K der Menge S von Formeln zu berechnen.

Sei $all\_subsets(S,K) = \{S' \subseteq S \mid |S'| = K\}$. Eine rekursive Berechnung lässt sich z.B. aus folgender Definition von $all\_subsets$ leicht herleiten:

$$\begin{array}{lcl} all\_subsets(S,0) & = & \{\{\}\} \\ all\_subsets(S,K) & = & \{S\}, \text{wenn } |S| = K \\ all\_subsets(\{s\} \cup S, K) & = & all\_subsets(S,K) \\ & & \cup \{\{s\} \cup S' \mid S' \in all\_subsets(S, K-1)\} \end{array}$$

Wir betrachten zunächst den Fall, dass höchstens K Formeln aus S erfüllt werden sollen. Offensichtlich reicht hierfür aus, zu fordern, dass in allen $K+1$-elementigen Teilmengen von S mindestens eine Formel falsch ist. Das ergibt:

$$at\_most(K,S) = \bigwedge \left\{ \underbrace{(\neg F_1 \vee \ldots \vee \neg F_{k+1})}_{\text{mind. 1 falsch}} \mid \{F_1,\ldots,F_{K+1}\} \in all\_subsets(S, K+1) \right\}$$

Der Vorteil ist erneut, dass die erzeugte Formel bereits in CNF ist, wenn S nur Literale enthält.

Zum Erzeugen einer Formel, die zusichert, dass mindestens K Formeln aus S wahr sind, kann man zunächst die Idee verwenden, dass in mindestens einer K-Elementigen Teilmenge von S alle Formeln erfüllt sein müssen. Dies ergibt:

$$at\_least(K,S) = \bigvee \left\{ \underbrace{\bigwedge S'}_{K \text{ wahr}} \mid S' \in all\_subsets(S,K) \right\}$$

Allerdings ist diese Formel keine CNF, daher bietet es sich an die folgende Formel zu verwenden, die die Idee verwendet: Zunächst muss eine Formel wahr sein, und zusätzlich müssen, wenn Formel F_i wahr ist, $K-1$ weitere Formeln wahr sein. Daraus lässt sich eine Formel in CNF (wenn S nur Literale enthält) herleiten:

$$\begin{aligned}
&at_least(K,S)\\
&= at_least_one(S)\\
&\wedge \bigwedge \{f \Rightarrow \bigvee S' \mid f \in S, S' \in all_subsets(S \setminus \{f\}, |S| - (K-1))\}\\
&= at_least_one(S)\\
&\wedge \bigwedge \{\neg f \vee \bigvee S' \mid f \in S, S' \in all_subsets(S \setminus \{f\}, |S| - (K-1))\}
\end{aligned}$$

Die Formel, die zusichert, dass genau K Formeln aus S wahr sind, ist die Konjunktion der at_least und at_most Formeln:

$$exactly(K,S) = at\_least(K,S) \wedge at\_most(K,S)$$

Beispiel 1.8.2. *Wir betrachten eine Logelei aus der Zeit:*
Tom, ein Biologiestudent, sitzt verzweifelt in der Klausur, denn er hat vergessen, das Kapitel über die Wolfswürmer zu lernen. Das Einzige, was er weiß, ist, dass es bei den Multiple-Choice-Aufgaben immer genau 3 korrekte *Antworten gibt. Und dies sind die angebotenen Antworten:*

a) *Wolfswürmer werden oft von Igelwürmern gefressen.*
b) *Wolfswürmer meiden die Gesellschaft von Eselswürmern.*
c) *Wolfswürmer ernähren sich von Lammwürmern.*
d) *Wolfswürmer leben in der banesischen Tundra.*
e) *Wolfswürmer gehören zur Gattung der Hundswürmer.*
f) *Wolfswürmer sind grau gestreift.*
g) *Genau eine der beiden Aussagen b*) *und e*) *ist richtig.*
h) *Genau eine der beiden Aussagen a*) *und d*) *ist richtig.*
i) *Genau eine der beiden Aussagen c*) *und h*) *ist richtig.*
j) *Genau eine der beiden Aussagen f*) *und i*) *ist richtig.*
k) *Genau eine der beiden Aussagen c*) *und d*) *ist richtig.*
l) *Genau eine der beiden Aussagen d*) *und h*) *ist richtig.*

Wir verwenden $A, B, \ldots, L$ *als Variablen, so dass die entsprechende Aussage wahr ist, gdw. die Variable wahr ist. Für die Aussagen a*) *bis f*) *müssen wir zunächst nichts kodieren. Für g*) *bis l*)*:*

- $G \iff B$ *XOR* E
- $H \iff A$ *XOR* D
- $I \iff C$ *XOR* H
- $J \iff F$ *XOR* I
- $K \iff C$ *XOR* D
- $L \iff D$ *XOR* H

Die Bedingung, dass genau 3 Antworten richtig sind, kann mit der $exactly$*-Formel erzeugt werden:*

$$exactly(3, \{A, B, C, D, E, F, G, H, I, J, K, L\})$$

Nach Konjunktion aller Aussagen und Eingabe in den DPLL-Algorithmus erhält man als Modell `[-K,-J,-I,-G,-F,-E,-B,C,-L,-A,H,D]`*, d.h. Tom muss*
c) *Wolfswürmer ernähren sich von Lammwürmern.*
d) *Wolfswürmer leben in der banesischen Tundra.*
h) *Genau eine der beiden Aussagen a*) *und d*) *ist richtig.*
ankreuzen.

Schließlich betrachten wir größeres Beispiel, das Lösen Sudokus. Dabei ist eine 9x9 Matrix gegeben, wobei einige Zelle schon Zahlen enthalten. Die Aufgabe ist es, alle Zellen mit Zahlen aus dem Bereich 1 bis 9 zu füllen, so dass in jede Zeile, in jeder Spalte, und in jeder der 9 disjunkten 3x3 Teilmatrizen, eine Permutation der Zahlen von 1 bis 9 steht.

Z.B. kann man das Sudoku

		6	4		1	5		
		3	6	5				
5	8			2		6		
4	6		8					3
	3	5					2	
2					3	9	8	
9		1			5			
				4	6			5
			1				7	

folgendermaßen vervollständigen (lösen):

7	9	6	4	8	1	5	3	2
1	2	3	6	5	9	8	4	7
5	8	4	3	2	7	6	9	1
4	6	9	8	1	2	7	4	3
8	3	5	7	9	4	1	2	6
2	1	7	5	6	3	9	8	4
9	4	1	2	7	5	3	6	8
3	7	8	9	4	6	2	1	5
6	5	2	1	3	8	4	7	9

Wir kodieren ein Sudoku indem wir direkt Zahlen als Variablen verwenden, negative Zahlen stehen dabei für die negierten Variablen. Wir verwenden dreistellige Zahlen XYZ, wobei

- X = Zeile
- Y = Spalte
- V = Zahl die in der Zelle stehen kann in $\{1, \ldots, 9\}$

D.h. es gibt pro Zelle 9 Variablen von denen genau eine Wahr sein muss.

Die Klauselmenge zur Lösung des Sudokus besteht aus mehreren Teilen:

$$\begin{aligned} \text{Klauselmenge} = \quad & \text{Startbelegung} \\ & \cup \text{ FelderEindeutigBelegt} \\ & \cup \text{ ZeilenBedingung} \\ & \cup \text{ SpaltenBedingung} \\ & \cup \text{ QuadratBedingung} \end{aligned}$$

Die Startbelegung gibt für manche Zellen vor, welche Zahl dort steht. In unserer Kodierung fügen wir hierfür die entsprechenden 1-Klauseln `[XYV]` hinzu, wenn in Zelle (X, Y) die Zahl V steht.

Die restlichen Klauseln sind statisch für alle 9×9 Sudokus. Zunächst müssen wir zusichern, dass in jedem Feld genau eine Zahl steht. Hierfür lässt sich die $exactly\_one$-Formel verwenden:

$$\text{FelderEindeutigBelegt} = \bigwedge_{X=1}^{9} \bigwedge_{Y=1}^{9} exactly\_one(\{XY1, \ldots, XY9\})$$

Die Zeilen- und Spaltenbedingungen lassen sich ebenfalls durch $exactly\_one$-Formeln ausdrücken, es reicht jedoch aus, die at_most_one-Formel zu verwenden, da der Rest bereits durch die eindeutige Belegung zugesichert wird.

$$\text{ZeilenBedingung} = \bigwedge_{X=1}^{9} \bigwedge_{V=1}^{9} at\_most\_one(\{X1V, \ldots, X9V\})$$

$$\text{SpaltenBedingung} = \bigwedge_{Y=1}^{9} \bigwedge_{V=1}^{9} at\_most\_one(\{1YV, \ldots, 9YV\})$$

Schließlich brauchen wir noch die Bedingungen für die 3×3 Quadrate

$$\text{QuadratBedingung} = \bigwedge_{V=1}^{9} \left(\begin{array}{l} at\_most\_one(\{11V, 12V, 13V, 21V, 22V, 23V, 31V, 32V, 33V\}) \wedge \\ at\_most\_one(\{14V, 15V, 16V, 24V, 25V, 26V, 34V, 35V, 36V\}) \wedge \\ at\_most\_one(\{17V, 18V, 19V, 27V, 28V, 29V, 37V, 38V, 39V\}) \wedge \\ at\_most\_one(\{41V, 42V, 43V, 51V, 52V, 53V, 61V, 62V, 63V\}) \wedge \\ at\_most\_one(\{44V, 45V, 46V, 54V, 55V, 56V, 64V, 65V, 66V\}) \wedge \\ at\_most\_one(\{47V, 48V, 49V, 57V, 58V, 59V, 67V, 68V, 69V\}) \wedge \\ at\_most\_one(\{71V, 72V, 73V, 81V, 82V, 83V, 91V, 92V, 93V\}) \wedge \\ at\_most\_one(\{74V, 75V, 76V, 84V, 85V, 86V, 94V, 95V, 96V\}) \wedge \\ at\_most\_one(\{77V, 78V, 79V, 87V, 88V, 89V, 97V, 98V, 99V\}) \end{array} \right)$$

Die erzeugten Klauselmengen lassen sich anschließend direkt in der DPLL-Prozedur verwenden. Die Programme zur Veranstaltung enthalten ein Programm, das vollautomatisch nach diesem Verfahren Sudokus löst.

2
Prädikatenlogik

2.1 Syntax und Semantik der Prädikatenlogik (PL_1)

Prädikatenlogik (PL) ist eine ausdrucksstarke Logik, die im Prinzip für sehr viele Anwendungen ausreicht.

Man unterscheidet verschiedene Stufen der Prädikatenlogik. Prädikatenlogik 0.Stufe (PL_0) ist die Aussagenlogik. Sie erlaubt keine Quantifikationen. Prädikatenlogik erster Stufe (PL_1) dagegen erlaubt schon Quantifikationen über Individuenvariablen. Prädikatenlogik 2.Stufe (PL_2) erlaubt darüberhinaus noch unabhängig Quantifikationen über die Funktionen und Relationen über diese Trägermenge. Man kann also z.B. in PL_2 hinschreiben „$\forall x : \exists f : \forall P : P(f(x, f))$“, was in PL_1 nicht geht. Prädikatenlogik noch höherer Stufe erlaubt Quantifizierungen über die Funktionen und Relationen über der Funktions- und Relationsmenge usw. (Ebbinghaus et al., 1986).

Aus praktischer Sicht gibt es viele Zusammenhänge, die sich in anderen Logiken wesentlich eleganter und einfacher formulieren lassen als in PL_n.

2.1.1 Syntax der Prädikatenlogik erster Stufe

PL_1 ist eine Erweiterung der Aussagenlogik. Wie für die meisten Logiken besteht die Syntaxbeschreibung aus den drei Komponenten:

- Signatur
- Bildungsregeln für Terme
- Bildungsregeln für Formeln

Die *Signatur* gibt das Alphabet an, aus dem die zusammengesetzten Objekte bestehen. Man unterscheidet *Funktions-* und *Prädikatensymbole*. Neben diesen Symbolen gibt es noch unendliche viele *Variablensymbole*, die nicht zur Signatur gerechnet werden.

Beispiel 2.1.1. *In einer Formel „$\forall x : \exists y : P(x, f(y))$“ sind x und y Variablensymbole, f ist ein einstelliges Funktionssymbol und P ein zweistelliges Prädikatensymbol.*

Aus den Variablen- und Funktionssymbolen lassen sich *Terme* aufbauen ($f(y)$ ist zum Beispiel ein Term) und damit und mit den Prädikatensymbolen *Atome*, *Literale* und *Formeln*. Terme bezeichnen Objekte einer Trägermenge und Formeln bezeichnen Wahrheits-

werte.
Die formale Definition ist:

Definition 2.1.2 (Syntax von PL_1)**.** *Die Syntax von PL_1 besteht zunächst aus einer* Signatur Σ*, darauf aufbauend werden* Terme *erzeugt, die schließlich innerhalb von* Formeln *verwendet werden:*

Signatur: *Eine Signatur Σ ist ein Paar $\Sigma = (\mathcal{F}, \mathcal{P})$, wobei*

- *$\mathcal{F}$ ist die Menge der Funktionssymbole*
- *$\mathcal{P}$ ist die Menge der Prädikatensymbole*

Diese Mengen sind disjunkt. Daneben braucht man noch die Menge V der Variablensymbole (abzählbar unendlich viele). Diese Menge ist ebenfalls disjunkt zu $\mathcal{F}$ und $\mathcal{P}$.

Jedem Funktions- und Prädikatensymbol $f \in \Sigma$ ist eindeutig eine Stelligkeit $arity(f) \geq 0$ *zugeordnet, die angibt wieviele Argumente das Funktions- oder Prädikatensymbole erhält. Funktionssymbole mit der Stelligkeit 0 bezeichnet man auch als* Konstantensymbole*, d.h. die Menge der Konstantensymbole ist gerade $\{f \in \mathcal{F} \mid arity(f) = 0\}$. Es muß mindestens ein Konstantensymbol in F vorhanden sein!*

Terme: *Die Menge der* Terme $T(\Sigma, V)$ *über der Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und den Variablen V wird induktiv als die kleinste Menge definiert, die folgendes erfüllt:*

- $V \subseteq T(\Sigma, V)$
- *falls $f \in \mathcal{F}, arity(f) = n, t_1, \ldots t_n \in T(\Sigma, V)$ dann $f(t_1, \ldots t_n) \in T(\Sigma, V)$. Hierbei ist $f(t_1, \ldots t_n)$ zu lesen als Zeichenkette bzw. als ein Baum.*

Formeln: *Die Menge der* Formeln $F(\Sigma, V)$ *über der Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und den Variablen V wird induktiv als die kleinste Menge definiert, die folgendes erfüllt:*

- *falls $P \in \mathcal{P}, arity(P) = n, t_1, \ldots, t_n \in T(\Sigma, V)$ dann $P(t_1, \ldots t_n) \in F(\Sigma, V)$* (Atom) *. Auch hier ist $P(t_1, \ldots t_n)$ als Zeichenkette zu lesen.*
- *falls $F, G \in F(\Sigma, V)$ und $x \in V$, dann auch: $(\neg F), (F \vee G), (F \wedge G), (F \Rightarrow G), (F \Leftrightarrow G), (\forall x : F)$ und $(\exists x : F) \in F(\Sigma, V)$.*

Wir machen bei den Schreibweisen einige Vereinfachungen: Geschachtelte gleiche Quantoren schreiben wir als Quantor über mehreren Variablen: $\forall x : \forall y : F$ wird als $\forall x, y : F$ geschrieben. In Formeln werden zum Teil Klammern weggelassen, wenn die Eindeutigkeit gewährleistet bleibt, mit den üblichen Prioritätsregeln. Weiterhin erlauben wir auch als Formelkonstanten die Formeln `false` und `true`.

Beispiel 2.1.3. *Signatur $\Sigma := (\{a, b, f, g\}, \{P, Q, R\})$ mit $arity(a) = arity(b) = arity(P) = 0$, $arity(f) = arity(Q) = 1$ $arity(g) = arity(R) = 2$. $V = \{x, y, z, \ldots\}$*

$T(\Sigma, V) =$	$F(\Sigma, V) =$
	$\{P, Q(a), Q(b), Q(x), \ldots, R(a,a), \ldots R(a,b), \ldots$
$f(a), f(b), f(x), \ldots$	$\neg P, \neg Q(a), \ldots$
$g(a,a), g(a,b), g(a, f(a)), \ldots$	$P \wedge Q(a), P \wedge \neg Q(a), \ldots$
$f(f(a)), f(f(b)), \ldots f(g(a,a)), \ldots$	$P \vee Q(a), P \vee \neg Q(a), \ldots$
$g(f(f(a)), a), \ldots$	$P \Rightarrow Q(a), P \Leftrightarrow \neg Q(a), \ldots$
$\ldots$	$\forall x : Q(x), \ldots$
	$\exists x : R(x,y) \Rightarrow Q(x), \ldots$

Mit der Forderung, dass mindestens ein Konstantensymbol vorhanden sein muss, ist implizit verbunden, dass die Trägermengen, über die in einer jeweiligen Interpretation quantifiziert wird, nicht leer sein kann – es muss mindestens ein Element vorhanden sein, auf das dieses Konstantensymbol abgebildet wird. Damit verhindert man, dass Quantifizierungen der Art „$\forall x : (P \wedge \neg P)$“ wahr gemacht werden können; indem die Menge, über die quantifiziert wird, leer ist. Einige der logischen Verknüpfungen, wie z.B. $\Rightarrow$ und $\Leftrightarrow$ sind redundant. Sie können durch die anderen dargestellt werden. Bei der Herstellung der Klauselnormalform (Abschnitt 2.1.4) werden sie dann auch konsequenterweise wieder eliminiert. Nichtsdestotrotz erleichtern sie die Lesbarkeit von Formeln beträchtlich und sind daher mit eingeführt worden.

Definition 2.1.4 (Konventionen)**.** *Wir verwenden die folgenden Konventionen:*

- *Da 0-stellige Funktionssymbole als Konstantensymbole dienen*[1]*, schreibt man im allgemeinen nicht „$a()$“ sondern einfach nur „a“*

- *Variablen sind genau einem Quantor zugeordnet Insbesondere gelten ähnlich wie in den meisten Programmiersprachen die schon gewohnten Bereichsregeln (lexical scoping) für Variable. D. h. Formeln der Art $\forall x : \exists x : P(x)$ haben nicht die vermutete Bedeutung, nämlich dass für alle x das gleiche x existiert, sondern x ist gebunden in $\exists x : P(x)$ und dass äußere x in alle $\forall x :$ kann das innere x nicht beeinflussen. D.h. die Formel $\forall x : \exists x : P(x)$ ist zu $\forall y : \exists x : P(x)$ und daher zu $\exists x : P(x)$ äquivalent. Da man unendlich viele Variablensymbole zur Verfügung hat, kann man in jedem Fall für jeden Quantor ein anderes Variablensymbol wählen.*

- *Meist haben die logischen Verknüpfungssymbole die Bindungsordnung $\Rightarrow, \Leftrightarrow, \wedge, \vee, \neg$, d.h. $\Rightarrow$ bindet am schwächsten und $\neg$ am stärksten. Danach gilt eine Formel $\neg A \wedge B \vee C \Rightarrow D \Leftrightarrow E \wedge F$ als folgendermaßen strukturiert: $((\neg A) \wedge (B \vee C)) \Rightarrow (D \Leftrightarrow (E \wedge F))$. Quantoren binden, soweit die quantifizierte Variable vorkommt. D.h. $\forall x : P(x) \wedge Q$ steht*

[1] Indem man Konstantensymbole nicht extra ausweist, spart man sich in vielen Fallunterscheidungen eben diesen speziellen Fall.

für $(\forall x : P(x)) \wedge Q$ *während* $\forall x : P(x) \wedge R(x)$ *für* $(\forall x : (P(x) \wedge R(x)))$ *steht. Um Zweifel auszuschließen, werden aber meist die Klammern explizit angegeben.*

- *Im Folgenden wird die allgemein übliche Konvention für die Benutzung des Alphabets verwendet: Buchstaben am Ende des Alphabets, d.h.* u, v, w, x, y, z *bezeichnen Variablensymbole. Buchstaben am Anfang des Alphabets, d.h.* a, b, c, d, e *bezeichnen Konstantensymbole. Die Buchstaben* f, g, h *werden für Funktionssymbole benutzt. Die großen Buchstaben* P, Q, R, T *werden für Prädikatensymbole benutzt.*

Die Syntax der Terme und Formeln wurde induktiv definiert: Aus den einfachen Objekten, Variable im Fall von Termen und Atome im Fall von Formeln wurden mit Hilfe von Konstruktionsvorschriften die komplexeren Objekte aufgebaut. Damit hat man eine Datenstruktur, die eine Syntaxbaum hat, der mit verschiedenen Konstruktoren aufgebaut wurde. Definitionen, Algorithmen, Argumentation und Beweise müssen nun jeweils rekursiv (induktiv) gemacht werden, wobei man stets Fallunterscheidung und Induktion *über die Struktur der Formeln und Terme* machen muss

Variablen, die nicht im Skopus eine Quantors stehen nennt man *freie Variablen*. Für eine Formel F bzw. Term t bezeichnen wir mit $FV(F)$ bzw. $FV(t)$ die Menge der freien Variablen.

Beispiel 2.1.5. *Seien* x, y, z *Variablensymbole, dann gilt:*

- $FV(x) = FV(f(x)) = FV(g(x, g(x, a))) = \{x\}$
- $FV(P(x) \wedge Q(y)) = \{x, y\}$
- $FV(\exists x : R(x, y)) = \{y\}$.

Definition 2.1.6 (Sprechweisen)**.** *Wir führen einige übliche Sprechweisen für spezielle Formeln und Terme ein:*

Atom:	*Eine Formel der Art* $P(t_1, \ldots t_n)$ *wobei* P *ein Prädikatensymbol und* $t_1, \ldots, t_n$ *Terme sind heißt Atom.*
Literal:	*Ein Atom oder ein negiertes Atom heißt Literal. (Beispiele:* $P(a)$ *und* $\neg P(a)$*)*
Grundterm:	*Ein Term t ohne Variablensymbole, d.h.* $FV(t) = \emptyset$*, heißt Grundterm (engl. ground term).*
Grundatom:	*Ein Atom* F *ohne Variablensymbole, d.h.* $FV(F) = \emptyset$*, heißt Grundatom.*
geschlossene Formel:	*Eine Formel* F *ohne freie Variablensymbole, d.h.* $FV(F) = \emptyset$ *heißt geschlossen.*
Klausel	*Formel mit einem Quantorpräfix nur aus Allquantoren besteht . d.h* $F = \forall^n.F'$ *und* F' *ist eine Disjunktionen von Literalen.*

Beispiel 2.1.7. *Für eine geschlossene Formel:* $\forall x : \exists y : \ P(x, y)$*. Nicht geschlossen ist:* $\exists y : P(x, y)$*, da* $FV(\exists y : \ P(x, y)) = \{y\})$.

2.1.2 Semantik

Die Semantik der Prädikatenlogik kann man analog zur Aussagenlogik definieren, man benötigt in diesem Fall noch eine Menge von Individuen (den Domain) und Interpretationsmöglichkeiten für Terme (Funktionen) und Prädikate.

Definition 2.1.8. Interpretation *Gegeben eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und Variablensymbole V. Eine* Interpretation $I = (S, I_V)$ *mit $S = (D_S, \mathcal{F}_S, \mathcal{P}_S)$ besteht aus einer nichtleeren Menge D_S (die Trägermenge), einer Interpretation der Funktionssymbole $\mathcal{F}_S$, die jedem Funktionssymbol $f \in \mathcal{F}$ eine totale Funktionen $f_S \in \mathcal{F}_S$ über D zuordnet und der Interpretation der Prädikatensymbole $\mathcal{P}_S$, die jedem Prädikat $P \in \mathcal{P}$ ein Prädikat $P_S \in \mathcal{P}_S$ (als Relation über D) zuordnet. Schließlich ist $I_V : V \to D_S$ eine Variablenbelegung, die jedem Variablensymbol einen Wert in der Trägermenge von D_S zuordnet.*

Diese Interpretation wird verträglich erweitert auf Terme, als $I(x) = I_V(x)$ und $I(f(t_1, \ldots, t_n)) = f_S(I(t_1), \ldots, I_(t_n))$.

Das Paar $(D_S, \mathcal{F}_S)$ nennt man auch Σ-Algebra, und das 3-Tupel $(D_S, \mathcal{F}_S), \mathcal{P}_S)$ Σ-Struktur. Eine ganz spezielle Σ-Algebra, ist die Termalgebra. Diese interpretiert die Funktionssymbole als syntaktische Terme:

Definition 2.1.9. (*Termalgebra*) *Für eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und eine Menge von Variablen V sei $(T(\Sigma, V)$ die Menge der Terme über der Signatur Σ und den Variablen V. Sei F_T die Menge der Funktionen $\{f_\Sigma \mid f \in \mathcal{F}, arity(f) = n, f_\Sigma(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)\})$. Dann nennt man $(T(\Sigma, V), F_T)$ die* Termalgebra *über der Signatur Σ.*

Für eine gegebene Interpretation I definieren wir eine abgeänderte Interpretation $I[a/x]$, mit $I[a/x](x) := a$ und $I[a/x](y) := I(y)$ falls $y \neq x$.

Nun können wir den Wahrheitswert von Formeln bzgl. einer Interpretation I definieren:

Definition 2.1.10 (Auswertung von Formeln)**.** *Sei I eine Interpretation. Die Basisfälle sind:*

Fall: $H = P(t_1, \ldots t_n)$	*falls $(I(t_1), \ldots, I(t_n)) \in P_S$*[2]*,*	*dann $I(H) := 1$.*
	falls $(I(t_1), \ldots, I(t_n)) \notin P_S$,	*dann $I(H) := 0$.*
Fall $H = P$		$I(P) := P_S$.

Rekursionsfälle:

[2] P_S ist die in $\mathcal{P}_S$ dem Symbol P zugeordnete Relation

Fall: $H = \texttt{false}$	*dann* $I(H) = 0$
Fall: $H = \texttt{true}$	*dann* $I(H) = 1$
Fall: $H = \neg F$	*dann* $I(H) = 1$ *falls* $I(F) = 0$
Fall: $H = F \vee G$	*dann* $I(H) = 1$ *falls* $I(F) = 1$ *oder* $I(G) = 1$
Fall: $H = F \wedge G$,	*dann* $I(H) = 1$ *falls* $I(F) = 1$ *und* $I(G) = 1$
Fall: $H = F \Rightarrow G$	*dann* $I(H) = 1$ *falls* $I(F) = 0$ *oder* $I(G) = 1$
Fall: $H = F \Leftrightarrow G$	*dann* $I(H) = 1$ *falls* $I(F) = 1$ *gdw.* $I(G) = 1$
Fall: $H = \forall x : F$	*dann* $I(H) = 1$ *falls für alle* $a \in D_S : I[a/x](F) = 1$
Fall: $H = \exists x : F$	*dann* $I(H) = 1$ *falls für ein* $a \in D_S : I[a/x](F) = 1$

Diese Definition erlaubt es, für eine Formel und eine Interpretation I zu bestimmen, ob die Formel in dieser Interpretation wahr oder falsch ist. Im nächsten Schritt lassen sich dann die Formeln danach klassifizieren, ob sie in allen Interpretationen wahr werden (Tautologien), in irgendeiner Interpretation wahr werden (erfüllbare Formeln), in irgendeiner Interpretation falsch werden (falsifizierbare Formeln) oder in allen Interpretationen falsch werden (unerfüllbare oder widersprüchliche Formeln).

Definition 2.1.11 (Modelle, Tautologien etc.)**.** *Eine Interpretation I, die eine Formel F wahr macht (erfüllt) heißt* Modell *von F. Man sagt auch: F gilt in I (F ist wahr in I, I erfüllt F). Bezeichnung: $I \models F$.*

Eine Formel F heißt:

allgemeingültig *wenn sie von allen Interpretationen erfüllt wird*

erfüllbar *wenn sie von einer Interpretation erfüllt wird, d.h. wenn es ein Modell gibt*

unerfüllbar (widersprüchlich) *wenn sie von keiner Interpretation erfüllt wird.*

falsifizierbar *wenn sie in einer Interpretation falsch wird.*

Eine allgemeingültige geschlossene Formel nennt man auch Tautologie oder Satz der Prädikatenlogik.

Es gibt dabei folgende Zusammenhänge:

- Eine Formel F ist allgemeingültig gdw. $\neg F$ unerfüllbar
- Falls F nicht allgemeingültig ist: F ist erfüllbar gdw. $\neg F$ erfüllbar

Die Menge der unerfüllbaren und die Menge der allgemeingültigen Formeln sind disjunkt. Formeln die weder unerfüllbar noch allgemeingültig sind sind gleichzeitig erfüllbar und falsifizierbar.

Beispiel 2.1.12. *Einige Beispiele für allgemeingültige, unerfüllbare, erfüllbare und falsifizierbare Formeln sind:*

allgemeingültig:	$P \vee \neg P$
unerfüllbar	$P \wedge \neg P$
erfüllbar, falsifizierbar:	$\forall x.P(x)$

Für den letzten Fall geben wir Interpretation an, die die Formel erfüllt und eine die sie falsifiziert:

1. *Als Menge D_S wählen wir $\{0,1\}$, als Interpretation für P ebenfalls die Menge $P_S = \{0,1\}$. Dann ergibt eine Interpretation I: $I(\forall x.P(x))$ gdw. $0 \in P_S$ und $1 \in P_S$. D.h. $I(\forall x.P(x)) = 1$.*

2. *Als Menge D_S wählen wir $\{0,1\}$, als Interpretation für P die Menge $P_S = \{0\}$. Dann ergibt eine Interpretation I: $I(\forall x.P(x))$ gdw. $0 \in P_S$ und $1 \in P_S$. D.h. $I(\forall x.P(x)) = 0$.*

Als weiteren, einfachen Testfall untersuchen wir Klauseln.

Beispiel 2.1.13. *Wann ist die Klausel $\{P(s), \neg P(t)\}$ allgemeingültig? Zunächst ist einfach zu sehen, dass $\{P(s), \neg P(s)\}$ eine Tautologie ist: Für jede Interpretation I gilt, dass $I(P(s))$ gerade der negierte Wahrheitswert von $I(\neg P(s))$ ist.*

Angenommen, $s \neq t$. Vermutung: dann ist sie nicht allgemeingültig. Um dies nachzuweisen, muss man eine Interpretation finden, die diese Klausel falsch macht.

Zuerst definieren wir eine Trägermenge D_S und eine Σ-Algebra. Man startet mit einer Menge A_0, die mindestens soviele Elemente enthält, wie die Terme s, t Konstanten und Variablen enthalten. Also $A_0 := \{a_1, \ldots, a_n, c_{x_1}, \ldots c_{x_m}\}$, wobei a_i die Konstanten in s, t sind und x_i die Variablen.

Danach definiert man alle Funktionen so, die in s, t vorkommen, so dass keinerlei Beziehungen gelten. D.h. man kann genau alle Terme nehmen, die sich aus den A_0 als Konstanten und den Funktionssymbolen aufbauen lassen. D.h. es ist die Termmenge einer Termalgebra über einer erweiterten Signatur Σ'.

Danach wählt man als Interpretation $I(a_i) := a_i$, $I(x_i) = c_{x_i}$, $f_S = f$. Beachte, dass der Quantorpräfix nur aus Allquantoren besteht.

Damit gilt nun: $I(S) \neq I(t)$. Da man die einstellige Relation P_S , die P zugeordnet wird, frei wählen kann, kann man dies so machen, dass $I(s) \notin P_S$ und $I(t) \in P_S$. Damit wird aber die Klausel falsch unter dieser Interpretation. D.h. sie kann nicht allgemeingültig sein.

Analog kann man diese Argumentation für Klauseln mit mehrstelligen Prädikaten verwenden.

Jede Klausel, die mehr als ein Literal enthält, ist erfüllbar. Wie nehmen an, dass die Formeln `true`, `false` nicht in Klauseln verwendet werden.

Mit der Einführung des Begriffs eines Modell (und einer Interpretation) sind alle Voraussetzungen gegeben, um – in Verallgemeinerung der Begriffe für Aussagenlogik – eine semantische Folgerungsbeziehung $\models$ zwischen zwei Formeln zu definieren:

Definition 2.1.14 (Semantische Folgerung)**.**
$F \models G$ gdw. G gilt (ist wahr) in allen Modellen von F.

Diese Definition ist zwar sehr intuitiv, da es aber i.A. unendlich viele Modelle für eine Formel gibt, ist sie jedoch in keiner Weise geeignet, um für zwei konkrete Formeln F und G zu testen, ob G aus F semantisch folgt. Die semantische Folgerungsbeziehung dient aber als Referenz; jedes konkrete, d.h. programmierbare Testverfahren muss sich daran messen, ob und wie genau es diese Beziehung zwischen zwei Formeln realisiert.

Bemerkung 2.1.15. *Das Beispiel der natürlichen Zahlen und der darin geltenden Sätze ist nicht vollständig mit PL_1 zu erfassen. Der Grund ist, dass man nur von einer einzigen festen Σ-Struktur ausgeht (die natürlichen Zahlen) und dann nach der Gültigkeit von Sätzen fragt.*

Versucht man die natürlichen Zahlen in PL_1 zu erfassen, so stellt sich heraus, dass man mit endlich vielen Axiomen nicht auskommt. Es ist sogar so, dass die Menge der Axiome nicht rekursiv aufzählbar ist.

Beispiel 2.1.16. *Ein Beispiel für eine in PL_1 modellierbare Theorie sind die Gruppen. Man benötigt nur endlich viele Axiome. Und man kann dann danach fragen, welche Sätze in allen Gruppen gelten.*

Ein erster Schritt zur Mechanisierung der Folgerungsbeziehung liefert das sogenannte *Deduktionstheorem*, welches die semantische Folgerungsbeziehung in Beziehung setzt mit dem syntaktischen Implikationszeichen. Dieses Theorem erlaubt die Rückführung der semantischen Folgerung auf einen Tautologietest.

Theorem 2.1.17 (Deduktionstheorem)**.** *Für alle Formeln F und G gilt: $F \models G$ gdw. $F \Rightarrow G$ ist allgemeingültig.*

Bemerkung 2.1.18. *Will man wissen, ob eine Formel F aus einer Menge von Axiomen $A_1, \ldots, A_n$ folgt, so kann man dies zunächst auf die äquivalente Frage zurückführen, ob F aus der Konjunktion der Axiome $A_1 \wedge \ldots \wedge A_n$ folgt und dann auf die äquivalente Frage, ob die Implikation $(A_1 \wedge \ldots \wedge A_n) \Rightarrow F$ eine Tautologie ist.*

Das Deduktionstheorem gilt in anderen Logiken i.A. nicht mehr (z.B. Modallogik). Das kann daran liegen, dass die semantische Folgerungsbeziehung dort anders definiert ist oder auch, dass die Implikation selbst anders definiert ist. Die Implikation, so wie sie in PL_1 definiert ist, hat nämlich den paradoxen Effekt, dass aus etwas Falschem alles folgt, d.h. die Formel `false` $\Rightarrow F$ ist eine Tautologie. Versucht man, diesen Effekt durch eine geänderte Definition für $\Rightarrow$ zu vermeiden, dann muss das Deduktionstheorem nicht mehr unbedingt gelten. Da F eine Tautologie ist genau dann wenn $\neg F$ unerfüllbar ist, folgt unmittelbar:

$$
\begin{array}{c}
F \models G \\
\text{gdw.} \\
\neg(F \Rightarrow G) \text{ ist unerfüllbar (widersprüchlich)} \\
\text{gdw.} \\
F \wedge \neg G \text{ ist unerfüllbar.}
\end{array}
$$

Das bedeutet, dass man in PL_1 den Test der semantischen Folgerungsbeziehung weiter zurückführen kann auf einen Unerfüllbarkeitstest. Genau dieses Verfahren ist die häufig verwendete Methode des **Beweis durch Widerspruch**: Um zu zeigen, dass aus Axiomen ein Theorem folgt, zeigt man, dass die Axiome zusammen mit dem negierten Theorem einen Widerspruch ergeben.

2.1.3 Berechenbarkeitseigenschaften der Prädikatenlogik

Es gilt die Unentscheidbarkeit der Prädikatenlogik:

Theorem 2.1.19. *Es ist unentscheidbar, ob eine geschlossene Formel ein Satz der Prädikatenlogik ist.*

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

Analog dazu gilt jedoch, die Semi-Entscheidbarkeit, d.h.

Theorem 2.1.20. *Die Menge der Sätze der Prädikatenlogik ist rekursiv aufzählbar.*

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

Über das theoretische Verhalten eines automatischen Deduktionssystems kann man daher folgendes sagen: Es kann terminieren und antworten: ist oder ist kein Satz. Wenn das System sehr lange läuft, kann das zwei Ursachen haben: Der Satz ist zu schwer zu zeigen (zu erkennen) oder die eingegebene Formel ist kein Satz und das System kann auch dies nicht erkennen.

Die Einordnung der sogenannten quantifizierten Booleschen Formeln – Quantoren über null-stellige Prädikate sind erlaubt, aber keine Funktionssymbole, und keine mehrstelligen Prädikate – ist in PL_1 nicht möglich. Diese QBF sind ein „Fragment“ von PL_2. Es ist bekannt, dass die Eigenschaft „Tautologie“ für QBF entscheidbar ist (PSPACE-vollständig).

2.1.4 Normalformen von PL_1-Formeln

Ziel diese Abschnitts ist es, einen Algorithmus zu entwickeln, der beliebige Formeln in eine Klauselnormalform (conjunctive normal form, CNF), d.h. eine Konjunktion ($\wedge$) von Disjunktionen ($\vee$) von Literalen, transformiert. Diese Klauselnormalform ist nur „ganz außen“ allquantifiziert, es gibt keine inneren Quantoren. Folgendes Lemma erlaubt die

Transformation von Formeln, wobei die Regeln in Tautologien entsprechen, aber als Transformationen benutzt werden. Diese sind Erweiterungen der Tautologien der Aussagenlogik.

Lemma 2.1.21. Elementare Rechenregeln

$$
\begin{array}{lcll}
\neg\forall x : F & \Leftrightarrow & \exists x : \neg F & \\
\neg\exists x : F & \Leftrightarrow & \forall x : \neg F & \\
(\forall x : F) \wedge G & \Leftrightarrow & \forall x : (F \wedge G) & \textit{falls } x \textit{ nicht frei in } G \\
(\forall x : F) \vee G & \Leftrightarrow & \forall x : (F \vee G) & \textit{falls } x \textit{ nicht frei in } G \\
(\exists x : F) \wedge G & \Leftrightarrow & \exists x : (F \wedge G) & \textit{falls } x \textit{ nicht frei in } G \\
(\exists x : F) \vee G & \Leftrightarrow & \exists x : (F \vee G) & \textit{falls } x \textit{ nicht frei in } G \\
\forall x : F \wedge \forall x : G & \Leftrightarrow & \forall x : (F \wedge G) & \\
\exists x : F \vee \exists x : G & \Leftrightarrow & \exists x : (F \vee G) &
\end{array}
$$

Bemerkung 2.1.22. *Mithilfe der obigen Tautologien kann man die sogenannte Pränexform und auch die Negationsnormalform einer Formel herstellen. Die* Pränex-form *ist dadurch gekennzeichnet, dass in der Formel zuerst alle Quantoren kommen (Quantorpräfix), und dann eine quantorenfreie Formel. Die* Negations-normalform *ist dadurch gekennzeichnet, dass alle Negationszeichen nur vor Atomen vorkommen und dass die Junktoren $\Rightarrow, \Leftrightarrow$ eliminiert sind.*

Zur Umwandlung einer Formel in Pränexform braucht man nur die Äquivalenzen zu verwenden, die Quantoren nach außen schieben. Hierzu müssen alle gebundenen Variablen verschiedene Namen haben. Die Äquivalenzen, die es erlauben, Subformeln unter Quantoren $\forall x.$ zu schieben, falls diese die Subformel die Variable x nicht enthält, spielen eine wichtige Rolle.

Die Negationsnormalform wird erreicht, indem man zunächst $\Rightarrow, \Leftrightarrow$ eliminiert und dann alle Äquivalenzen nutzt, um Negationszeichen nach innen zu schieben.

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

Idee: Ersetze in $\exists x : P(x)$ das x, „das existiert“ durch ein Konstantensymbol a, d.h. $\exists x : P(x) \to P(a)$ Ersetze in $\forall x_1 \ldots x_n : \exists y : P(x_1, \ldots, x_n, y)$ das y durch eine Funktion von $x_1, \ldots, x_n$, d.h. $\forall x_1 \ldots x_n : \exists y : P(x_1, \ldots, x_n, y) \to \forall x_1 \ldots x_n : P(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$.

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

Theorem 2.1.23. Skolemisierung

Eine Formel $F = \forall x_1 \ldots x_n : \exists y : G[x_1, \ldots, x_n, y]$ ist (un-)erfüllbar gdw. $F' = \forall x_1 \ldots x_n : G[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]$ (un-)erfüllbar ist, wobei f ein n-stelliges Funktionssymbol ist mit $n \geq 0$, das nicht in G vorkommt.

Beispiel 2.1.24. Skolemisierung

$$\begin{aligned} \exists x : P(x) &\rightarrow P(a) \\ \forall x : \exists y : Q(f(y,y),x,y) &\rightarrow \forall x : Q(f(g(x),g(x)),x,g(x)) \\ \forall x,y : \exists z : x + z = y &\rightarrow \forall x,y : x + h(x,y) = y. \end{aligned}$$

Beispiel 2.1.25. *Skolemisierung erhält i.a. nicht die Allgemeingültigkeit (Falsifizierbarkeit):*

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

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

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

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

Beachte, dass es dual dazu auch eine Form der Skolemisierung gibt, bei der die all-quantifizierten Variablen skolemisiert werden. Dies wird verwendet, wenn man statt auf Widersprüchlichkeit die Formeln auf Allgemeingültigkeit testet. Im Beweis ersetzt man dann die Begriff erfüllbar durch falsifizierbar und unerfüllbar durch allgemeingültig.

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

Theorem 2.1.26 (Allgemeinere Skolemisierung)**.** *Sei F eine geschlossene Formel, G eine existentiell quantifizierte Unterformel in F an einer Position p, Weiterhin sei G nur unter All-quantoren, Konjunktionen, und Disjunktionen. Die All-quantoren über G binden die Variablen $x_1, \ldots, x_n$ mit $n \geq 0$. D.h. F ist von der Form $F[\exists y : G'[x_1, \ldots, x_n, y]]$.*

Dann ist $F[G]$ (un-)erfüllbar gdw. $F[G'[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]]$ (un-)erfüllbar ist, wobei f ein n-stelliges Funktionssymbol ist, das nicht in G vorkommt.

Definition 2.1.27 (Transformation in Klauselnormalform)**.** *Folgende Prozedur wandelt jede prädikatenlogische Formel in Klauselform (CNF) unter Erhaltung der Erfüllbarkeit um:*

1. *Elimination von $\Leftrightarrow$ und $\Rightarrow$: $F \Leftrightarrow G \rightarrow F \Rightarrow G \wedge G \Rightarrow F$ (Lemma 2.1.21) und*
   $F \Rightarrow G \rightarrow \neg F \vee G$
2. *Negation ganz nach innen schieben:*

$$\begin{aligned} \neg\neg F &\rightarrow F \\ \neg(F \wedge G) &\rightarrow \neg F \vee \neg G \\ \neg(F \vee G) &\rightarrow \neg F \wedge \neg G \\ \neg\forall x : F &\rightarrow \exists x : \neg F \\ \neg\exists x : F &\rightarrow \forall x : \neg F \end{aligned}$$

3. *Skopus von Quantoren minimieren, d.h. Quantoren so weit wie möglich nach innen schieben (kann auch nur teilweise geschehen)*

$$\begin{array}{lcll}
\forall x : (F \wedge G) & \rightarrow & (\forall x : F) \wedge G & \text{falls } x \text{ nicht frei in } G \\
\forall x : (F \vee G) & \rightarrow & (\forall x : F) \vee G & \text{falls } x \text{ nicht frei in } G \\
\exists x : (F \wedge G) & \rightarrow & (\exists x : F) \wedge G & \text{falls } x \text{ nicht frei in } G \\
\exists x : (F \vee G) & \rightarrow & (\exists x : F) \vee G & \text{falls } x \text{ nicht frei in } G \\
\forall x : (F \wedge G) & \rightarrow & \forall x : F \wedge \forall x : G & \\
\exists x : (F \vee G) & \rightarrow & \exists x : F \vee \exists x : G &
\end{array}$$

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

5. *Existenzquantoren werden durch Skolemisierung eliminiert*

6. *Allquantoren nach außen schieben*

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

8. *Allquantoren vor die Klauseln schieben, anschließend umbenennen.*

9. *Quantoren kann man in der Darstellung dann weglassen unter der Annahme, dass alle Variablen (implizit) als allquantifiziert angenommen werden.*

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

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

oder in Mengenschreibweise:

$$\begin{array}{l}
\{\{L_{1,1}, \ldots, L_{1,n_1}\}, \\
\{L_{2,1}, \ldots, L_{2,n_2}\}, \\
\ldots \\
\{L_{k,1}, \ldots, L_{1,n_k}\}\}
\end{array}$$

Beispiel 2.1.28. *Analog zur Aussagenlogik kann das CNF-Verfahren so optimiert werden, dass es statt exponentiell großen Formeln nur linear große Klauselmengen generiert.*

Beispiel 2.1.29.

$A1$: $Dieb(Anton) \vee Dieb(Ede) \vee Dieb(Karl)$

$A2$: $Dieb(Anton) \Rightarrow (Dieb(Ede) \vee Dieb(Karl))$

$A3$: $Dieb(Karl) \Rightarrow (Dieb(Ede) \vee Dieb(Anton))$

$A4$: $Dieb(Ede) \Rightarrow (\neg\, Dieb(Anton) \wedge \neg\, Dieb(Karl))$

$A5$: $\neg\, Dieb(Anton) \vee \neg\, Dieb(Karl)$

Klauselform:

$A1$: $Dieb(Anton), Dieb(Ede), Dieb(Karl)$

$A2$: $\neg\, Dieb(Anton), Dieb(Ede), Dieb(Karl)$

$A3$: $\neg\, Dieb(Karl), Dieb(Ede), Dieb(Anton)$

$A4a$: $\neg\, Dieb(Ede), \neg\, Dieb(Anton)$

$A4b$: $\neg\, Dieb(Ede), \neg\, Dieb(Karl)$

$A5$: $\neg\, Dieb(Anton), \neg\, Dieb(Karl)$

Beispiel 2.1.30. *verschiedene Typen von Normalformen:*

Original Formel: $\forall\varepsilon : (\varepsilon > 0 \Rightarrow \exists\delta : (\delta > 0 \wedge \forall x, y : (|x - y| < \delta \Rightarrow |g(x) - g(y)| < \varepsilon)))$

Negations Normalform : (*Alle Negationen innen;* $\Rightarrow, \Leftrightarrow$ *eliminiert*)

$\forall\varepsilon : (\neg\varepsilon > 0 \vee \exists\delta : (\delta > 0 \wedge \forall x, y : (\neg|x - y| < \delta \vee |g(x) - g(y)| < \varepsilon)))$

Pränex Form : (*Alle Quantoren außen*) $\forall\varepsilon : \exists\delta : \forall x, y : \varepsilon > 0 \Rightarrow \delta > 0 \wedge (|x - y| < \delta \Rightarrow |g(x) - g(y)| < \varepsilon)$

Skolemisierte Pränex Form : $\varepsilon > 0 \Rightarrow f_\delta(\varepsilon) > 0 \wedge (|x - y| < f_\delta(\varepsilon) \Rightarrow |g(x) - g(y)| < \varepsilon)$

Disjunktive Normalform : $(\neg\varepsilon > 0) \vee (fd(\varepsilon) > 0 \wedge \neg|x - y| < f_\delta(\varepsilon)) \vee (f_\delta(\varepsilon) > 0 \wedge |g(x) - g(y)| < \varepsilon)$

Konjunktive Normalform : $(\neg\varepsilon > 0 \vee f_\delta(\varepsilon) > 0) \wedge (\neg\varepsilon > 0 \vee \neg|x - y| < fd(\varepsilon) \vee |g(x) - g(y)| < \varepsilon)$

Klauselform : $\{\{\neg\varepsilon > 0, f_\delta(\varepsilon) > 0\}, \{\neg\varepsilon > 0, \neg|x - y| < f_\delta(\varepsilon), |g(x) - g(y)| < \varepsilon\}\}$.

2.2 Resolution

Ein Kalkül soll die semantische Folgerungsbeziehung durch syntaktische Manipulation nachbilden, d.h. genau dann wenn $F \models G$, soll es möglich sein, entweder G aus F durch syntaktische Manipulation abzuleiten ($F \vdash G$, positiver Beweis) oder $F \wedge \neg G$ durch syntaktische Manipulation zu widerlegen ($F \wedge \neg G \vdash false$, Widerlegungsbeweis). Für jeden Kalkül muss die Korrektheit gezeigt werden, d.h. wann immer $F \vdash G$, dann $F \models G$. Die Vollständigkeit, d.h. wann immer $F \models G$, dann $F \vdash G$ ist nicht notwendig. Was möglichst gelten sollte (aber bei manchen Logiken nicht möglich ist), ist die Widerlegungsvollständigkeit, d.h. wann immer $F \models G$ dann $F \wedge \neg G \vdash false$. Für PL_1 gibt es eine ganze Reihe unterschiedlicher Kalküle. Ein wichtiger und gut automatisierbarer ist der 1963 von John

Alan Robinson entwickelte Resolutionskalkül (Robinson, 1965). Er arbeitet in erster Linie auf Klauseln.

2.2.1 Grundresolution: Resolution ohne Unifikation

Im folgenden schreiben wir Klauseln teilweise als Folge von Literalen: $L_1, \ldots L_n$ und behandeln diese als wären es Multimengen. (teilweise auch als Mengen).

Grundresolution behandelt Grund-Klauselmengen, d.h. Klauselmengen die nur Grundterme enthalten und somit variablenfrei sind.

Definition 2.2.1. *Resolution im Fall direkt komplementärer Resolutionsliterale.*

$$\begin{array}{ll} \textit{Elternklausel 1:} & L, K_1, \ldots, K_m \\ \textit{Elternklausel 2:} & \neg L, N_1, \ldots, N_n \\ \hline \textit{Resolvente:} & K_1, \ldots, K_m, N_1, \ldots, N_n \end{array}$$

Hierbei kann es passieren, dass die Resolvente keine Literale mehr enthält. Diese Klausel nennt man die *leere Klausel*; Bezeichnung: $\square$. Sie wird als „falsch“ interpretiert und stellt i.a. den gesuchten Widerspruch dar.

Beispiel 3.2.14 weiter fortgesetzt:

Beispiel 2.2.2. *siehe Beispiel 2.1.29*

A1: $Dieb(Anton), Dieb(Ede), Dieb(Karl)$
A2: $\neg\, Dieb(Anton), Dieb(Ede), Dieb(Karl)$
A3: $\neg\, Dieb(Karl), Dieb(Ede), Dieb(Anton)$
A4a: $\neg\, Dieb(Ede), \neg\, Dieb(Anton)$
A4b: $\neg\, Dieb(Ede), \neg\, Dieb(Karl)$
A5: $\neg\, Dieb(Anton), \neg\, Dieb(Karl)$

Resolutionsableitung:

A2,2 & A4a,1	$\vdash$	*R1:* $\neg\, Dieb(Anton), Dieb(Karl)$
R1,2 & A5,2	$\vdash$	*R2:* $\neg\, Dieb(Anton)$
R2 & A3,3	$\vdash$	*R3:* $\neg\, Dieb(Karl), Dieb(Ede)$
R3,2 & A4b,1	$\vdash$	*R4:* $\neg\, Dieb(Karl)$
R4 & A1,3	$\vdash$	*R5:* $Dieb(Anton), Dieb(Ede)$
R5,1 & R2	$\vdash$	*R6:* $Dieb(Ede)$

Also, Ede wars.

Satz 2.2.3. *Die Grund-Resolution ist korrekt:*

$$\begin{array}{lll} C_1 & := & L, K_1, \ldots, K_m \\ C2 & :=: & \neg L, N_1, \ldots, N_n \\ \hline R & = & K_1, \ldots, K_m, N_1, \ldots, N_n \end{array}$$

Dann gilt $C_1 \wedge C_2 \models R$.

Beweis. Wir müssen zeigen: Jede Interpretation, die die beiden Elternklauseln wahr macht, macht auch die Resolvente wahr. Das geht durch einfache Fallunterscheidung:

Falls L wahr ist, muss $\neg L$ falsch sein. Da C_2 wahr ist, muss ein N_i wahr sein. Da dieses Literal in R vorkommt, ist auch R (als Disjunktion betrachtet) wahr. Falls L falsch ist, muss ein K_j wahr sein. Da das ebenfalls in der Resolvente vorkommt, ist R auch in diesem Fall wahr. □

Bemerkung 2.2.4. *Im Sinne der Herleitbarkeit ist Resolution unvollständig: Nicht jede Formel, die semantisch folgt, läßt sich durch Anwenden der Resolution ableiten: Denn $P \models P \vee Q$. aber auf P alleine kann man keine Resolution anwenden.*

Für den eingeschränkten Fall, dass die Klauselmenge keine Variablen enthält (Grundfall) können wir schon die Widerlegungsvollständigkeit der Resolution beweisen.

Theorem 2.2.5 (Widerlegungsvollständigkeit der Grundresolution)**.** *Jede endliche unerfüllbare Grundklauselmenge läßt sich durch Resolution widerlegen.*

2.2.2 Resolution im allgemeinen Fall

Für den allgemeinen Fall, wenn in den Literalen auch Variablen vorkommen, benötigt man eine zusätzliche Operation, um potentielle Resolutionspartner, d.h. Literale mit gleichem Prädikat und verschiedenem Vorzeichen durch Einsetzung von Termen für Variablen komplementär gleich zu machen. Dazu führen wir zunächst das Konzept der Substitution ein.

Definition 2.2.6 (Substitution)**.** *Eine Substitution σ ist eine Abbildung endlich vieler Variablen auf Terme. Die Erweiterung von σ auf Terme ist definiert durch:*

$$\begin{array}{lcl} \sigma(x) & = & x, \text{ wenn } \sigma \text{ die Variable } x \text{ nicht abbildet} \\ \sigma(f(t_1, \ldots, t_n)) & = & f(\sigma(t_1), \ldots, \sigma(t_n)) \end{array}$$

Entsprechend kann man die Anwendung von Substitutionen auf Literale und Klauseln definieren. D.h. die Substitution σ kann man sehen als gleichzeitige Ersetzung der Variablen x durch den Term $\sigma(x)$.

Substitutionen werden meist geschrieben wie eine Menge von Variable – Term Paaren:

$$\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_1\}$$

Beispiel 2.2.7.

$$\begin{array}{ll}
\sigma = \{x \mapsto a\} & \sigma(x) = a, \sigma(f(x,x)) = f(a,a) \\
\sigma = \{x \mapsto g(x)\} & \sigma(x) = g(x), \sigma(f(x,x)) = f(g(x),g(x)), \\
 & \sigma(\sigma(x)) = g(g(x)) \\
\sigma = \{x \mapsto y, y \mapsto a\} & \sigma(x) = y, \sigma(\sigma(x)) = a, \\
 & \sigma(f(x,y)) = f(y,a) \\
\sigma = \{x \mapsto y, y \mapsto x\} & \sigma(x) = y, \sigma(f(x,y)) = f(y,x)
\end{array}$$

Definition 2.2.8. *Für eine Substitution* $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *heißt*

$$\begin{array}{lcll}
dom(\sigma) & := & \{x_1, \ldots, x_n\} & \textit{die Domain} \\
cod(\sigma) & := & \{t_1, \ldots, t_n\} & \textit{die Codomain}
\end{array}$$

Gleichheit (modulo einer Menge von Variablen W*) zwischen Substitutionen* σ *und* τ *wird folgendermaßen definiert:* $\sigma = \tau[W]$ *gdw.* $\sigma x = \tau x$ *für alle* $x \in W$*.*

Eine Instanziierungrelation $\leq [W]$ *zwischen Substitutionen* σ *und* τ *(für eine Menge von Variablen* W*) ist folgendermaßen definiert:*

$\sigma \leq \tau[W]$ *gdw. es existiert eine Substitution* λ *mit* $\lambda\sigma = \tau[W]$*. Zwei Substitutionen* σ *und* τ *sind* äquivalent*, d.h.* $\sigma \approx t[W]$ *gdw.* $\sigma \leq \tau[W]$ *und* $\tau \leq \sigma[W]$ *gilt.Ist* W *die Menge aller Variablen, so kann man* W *weglassen.*

Beispiel 2.2.9.

Komposition:

- $\{x \mapsto a\}\{y \mapsto b\} = \{x \mapsto a, y \mapsto b\}$
- $\{y \mapsto b\}\{x \mapsto f(y)\} = \{x \mapsto f(b), y \mapsto b\}$
- $\{x \mapsto b\}\{x \mapsto a\} = \{x \mapsto a\}$

Instanziierungsrelation:

- $\{x \mapsto y\} \leq \{x \mapsto a\}[x]$
- $\{x \mapsto y\} \leq \{x \mapsto a\}[x, y]$ *gilt nicht !*
- $\{x \mapsto y\} \leq \{x \mapsto a, y \mapsto a\}$
- $\{x \mapsto f(x)\} \leq \{x \mapsto f(a)\}$

Äquivalenz: $\{x \mapsto y, y \mapsto x\} \approx Id$ *(*Id *ist die identische Substitution)*

Definition 2.2.10. *(Resolution mit Unifikation)*

$$\begin{array}{lll}
\textit{Elternklausel 1:} & L, K_1, \ldots, K_m & \sigma \textit{ ist eine Substitution (Unifikator)} \\
\textit{Elternklausel 2:} & \neg L', N_1, \ldots, N_n & \sigma(L) = \sigma(L') \\
\textit{Resolvente:} & \sigma(K_1, \ldots, K_m, N_1, \ldots, N_n) &
\end{array}$$

Die Operation auf einer Klauselmenge, die eine Klausel C auswählt, auf diese eine Substitution σ anwendet und $\sigma(C)$ zur Klauselmenge hinzufügt, ist korrekt. Damit ist auch die allgemeine Resolution als Folge von Variableneinsetzung und Resolution korrekt.

Beispiel 2.2.11. *Dieses Beispiel (Eine Variante der Russelschen Antinomie) zeigt, dass noch eine Erweiterung der Resolution, die Faktorisierung, notwendig ist. Die Aussage ist: Der Friseur rasiert alle, die sich nicht selbst rasieren:*

$$\forall x : \neg(rasiert(x, x)) \Leftrightarrow rasiert(Friseur, x)$$

$$\frac{\begin{array}{ll} rasiert(x,x), rasiert(Friseur,x) & \sigma = \{x \mapsto Friseur, y \mapsto Friseur\} \\ \neg rasiert(Friseur,y), \neg rasiert(y,y) & \end{array}}{rasiert(Friseur, Friseur), \neg rasiert(Friseur, Friseur)}$$

Die Klauseln sind widersprüchlich, was aber ohne eine Verschmelzung der Literale mittels Resolution nicht ableitbar ist. In der folgenden Herleitung werden die Variablen mit „Friseur“ instanziiert, und dann die Literale verschmolzen.

$$\begin{array}{rcl} rasiert(x,x), rasiert(Friseur,x) & \vdash & rasiert(Friseur, Friseur) \\ \neg rasiert(Friseur,y), \neg rasiert(y,y) & \vdash & \neg rasiert(Friseur, Friseur) \end{array}$$

Danach ist es möglich, diese beiden Literale durch Resolution zur leeren Klausel abzuleiten.

Definition 2.2.12. (Faktorisierung)

$$\begin{array}{ll} \textit{Elternklausel:} & \dfrac{L, L', K_1, \ldots, K_m}{\sigma(L, K_1, \ldots, K_m)} \quad \sigma(L) = \sigma(L') \\ \textit{Faktor:} & \end{array}$$

Damit besteht der Resolutionskalkül jetzt aus Resolution und Faktorisierung.

Definition 2.2.13. *Der Resolutionskalkül transformiert Klauselmengen S wie folgt:*

1. *$S \to S \cup \{R\}$, wobei R eine Resolvente von zwei (nicht notwendig verschiedenen) Klauseln aus S ist.*

2. *$S \to S \cup \{F\}$, wobei F ein Faktor einer Klausel aus S ist.*

Der Resolutionskalkül terminiert mit Erfolg, wenn die leere Klausel abgeleitet wurde, d.h. wenn $\square \in S$.

Bei Klauselmengen nehmen wir wie üblich an, dass die Klauseln variablendisjunkt sind.

Beispiel 2.2.14. *Wir wollen die Transitivität der Teilmengenrelation mit Resolution beweisen. Wir starten mit der Definition von $\subseteq$ unter Benutzung von $\in$:*

$$\forall x, y : x \subseteq y \Leftrightarrow \forall w : w \in x \Rightarrow w \in y$$

Das zu beweisende Theorem ist:

$$\forall x, y, z : x \subseteq y \wedge y \subseteq z \Rightarrow x \subseteq z$$

Umwandlung in Klauselform ergibt:

H1:	$\neg x \subseteq y, \neg w \in x, w \in y$	($\Rightarrow$ *Teil der Definition*)
H2:	$x \subseteq y, f(x,y) \in x$	(*zwei* $\Leftarrow$ *Teile der Definition,*
H3:	$x \subseteq y, \neg f(x,y) \in y$	f *ist die Skolem Funktion für* w)
C1:	$a \subseteq b$	(*drei Teile der negierten Behauptung,*
C2:	$b \subseteq c$	a, b, c *sind Skolem Konstanten für* x, y, z)
C3:	$\neg a \subseteq c$	

Resolutionswiderlegung:

H1,1 & C1,	$\{x \mapsto a, y \mapsto b\}$	$\vdash$	*R1:* $\neg w \in a, w \in b$
H1,1 & C2,	$\{x \mapsto b, y \mapsto c\}$	$\vdash$	*R2:* $\neg w \in b, w \in c$
H2,2 & R1,1,	$\{x \mapsto a, w \mapsto f(a,y)\}$	$\vdash$	*R3:* $a \subseteq y, f(a,y) \in b$
H3,2 & R2,2,	$\{y \mapsto c, w \mapsto f(x,c)\}$	$\vdash$	*R4:* $x \subseteq c, \neg f(x,c) \in b$
R3,2 & R4,2,	$\{x \mapsto a, y \mapsto c\}$	$\vdash$	*R5:* $a \subseteq c, a \subseteq c$
R5 & (*Faktorisierung*)		$\vdash$	*R6:* $a \subseteq c$
R6 & C3		$\vdash$	*R7:* $\square$

2.2.3 Unifikation

Die Resolutions- und Faktorisierungsregel verwenden Substitutionen (Unifikatoren), die zwei Atome syntaktisch gleich machen. Meist will man jedoch nicht irgendeinen Unifikator, sondern einen möglichst allgemeinen. Was das bedeutet, zeigen die folgenden Beispiele:

Beispiel 2.2.15. *Unifikatoren und allgemeinste Unifikatoren.*

$P(x), Q(x)$	
$\neg P(y), R(y)$	$\sigma = \{x \mapsto a, y \mapsto a\}$
$Q(a), R(a)$	σ *ist ein Unifikator*
$P(x), Q(x)$	
$\neg P(y), R(y)$	$\sigma = \{x \mapsto y\}$
$Q(y), R(y)$	σ *ist ein allgemeinster Unifikator*

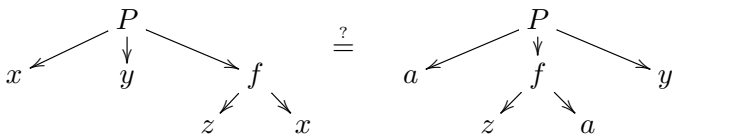
Fragen:

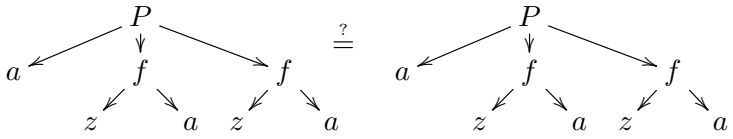
- Was heißt „allgemeinster“ Unifikator?
- Wieviele gibt es davon? ($\sigma' = \{y \mapsto x\}$ im obigen Beispiel ist offensichtlich auch einer.)
- Wie berechnet man sie?

Ein allgemeinster Unifikator (von zwei Atomen) kann man intuitiv dadurch erklären, dass es eine Substitution ist, die zwei Terme oder Atome gleich macht, und möglichst wenig instanziiert. Optimal ist es dann, wenn alle Unifikatoren durch weitere Einsetzung in den allgemeinsten Unifikator erzeugt werden können.

Beispiel 2.2.16. *Ein Beispiel zur Unifikation.*

$$P(x, y, f(z, x)) = P(a, f(z, a), y)$$





$$\begin{aligned} x &= a \\ y &= f(z, a) \end{aligned}$$

Ein Anwendung der Unifikation ist der polymorphe Typcheck, der als eine Basisoperation genau die Unifikation von Typen verwendet, wobei Typvariablen instanziiert werden. Dieser wird in Haskell, ML, und mittlerweile auch in einer Variante auch Java benutzt.

Definition 2.2.17. *Unifikatoren und allgemeinste Unifikatoren.*

Seien s und t die zu unifizierenden Terme (Atome) und $W := FV(s,t)$. Eine Substitution σ heißt Unifikator *(von s,t), wenn $\sigma(s) = \sigma(t)$. Die Menge aller Unifikatoren bezeichnet man auch mit $U(s,t)$,*

Eine Substitution σ heißt allgemeinster Unifikator *für zwei Terme s und t wenn*

σ ein Unifikator ist	*(Korrektheit)*
für alle Unifikatoren τ gilt $\sigma \leq \tau[W]$	*(Vollständigkeit)*

Beachte, dass es bis auf Variablenumbenennung immer einen allgemeinsten Unifikator gibt. Wir geben den Unifikationsalgorithmus in Form einer Regelmenge an, die auf Mengen von (zu lösenden) Gleichungen operiert.

Definition 2.2.18. *Unifikationsalgorithmus $U1$:*

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

Ausgabe: *„nicht unifizierbar“ oder einen allgemeinsten Unifikator:*

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

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

Unifikationsregeln:

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

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

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

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

Abbruchbedingungen:

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

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

Steuerung:

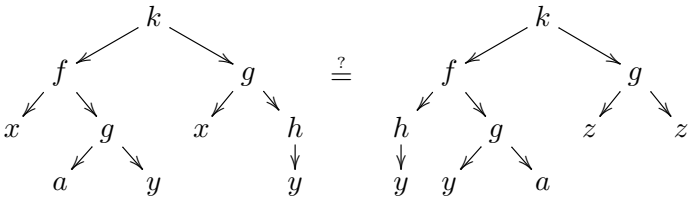
Starte mit $\Gamma = \Gamma_0$, und transformiere Γ solange durch (nichtdeterministische, aber nicht verzweigende) Anwendung der Regeln, bis entweder eine Abbruchbedingung erfüllt ist oder keine Regel mehr anwendbar ist. Falls eine Abbruchbedingung erfüllt ist, terminiere mit „nicht unifizierbar". Falls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

Beispiel 2.2.19. *Unifikation von zwei Termen durch Anwendung der obigen Regeln:*

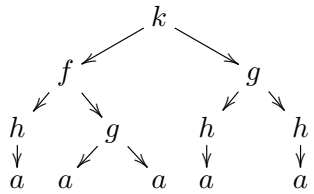
$$\begin{array}{lll}
 & \{k(f(x,g(a,y)),g(x,h(y)) \stackrel{?}{=} k(f(h(y),g(y,a)),g(z,z))\} & \\
\to & \{f(x,g(a,y)) \stackrel{?}{=} f(h(y),g(y,a)),g(x,h(y)) \stackrel{?}{=} g(z,z)\} & (\textit{Dekomposition}) \\
\to & x \stackrel{?}{=} h(y),g(a,y) \stackrel{?}{=} g(y,a),g(x,h(y)) = g(z,z) & (\textit{Dekomposition}) \\
\to & x \stackrel{?}{=} h(y),a \stackrel{?}{=} y,y \stackrel{?}{=} a,g(x,h(y)) \stackrel{?}{=} g(z,z) & (\textit{Dekomposition}) \\
\to & x \stackrel{?}{=} h(y),y \stackrel{?}{=} a,g(x,h(y)) \stackrel{?}{=} g(z,z) & (\textit{Orientierung}) \\
\to & x \stackrel{?}{=} h(a),y \stackrel{?}{=} a,g(x,h(a)) \stackrel{?}{=} g(z,z) & (\textit{Anwendung, y}) \\
\to & x \stackrel{?}{=} h(a),y \stackrel{?}{=} a,x \stackrel{?}{=} z,h(a) \stackrel{?}{=} z & (\textit{Dekomposition}) \\
\to & x \stackrel{?}{=} h(a),y \stackrel{?}{=} a,x \stackrel{?}{=} z,z \stackrel{?}{=} h(a) & (\textit{Orientierung}) \\
\to & x \stackrel{?}{=} h(a),y \stackrel{?}{=} a,z \stackrel{?}{=} h(a) & (\textit{Anwendung,z})
\end{array}$$

Unifizierte Terme: $k(f(h(a), g(a, a)), g(h(a), h(a)))$.

In der Baumdarstellung sieht es so aus:



Unifizierter Term: $k(f(h(a), g(a, a)), g(h(a), h(a)))$.



Theorem 2.2.20. *Der Unifikationsalgorithmus terminiert, ist korrekt und vollständig Er liefert, falls er nicht abbricht, genau einen allgemeinsten Unifikator.*

Was man hieraus folgern kann, ist dass der Unifikationsalgorithmus alleine ausreicht, um die Unerfüllbarkeit einer Menge von Unit-Klauseln [3] zu entscheiden.

2.3 Vollständigkeit

Theorem 2.3.1. (*Gödel-Herbrand-Skolem Theorem*) *Zu jeder unerfüllbaren Menge C von Klauseln gibt es eine endliche unerfüllbare Menge von Grundinstanzen* (*Grundklauseln*) *von C.*

Zusammen mit dem Satz, dass jede unerfüllbare Menge von Grundklauseln sich mit Resolution widerlegen lässt, kommt man dem Beweis der Vollständigkeit der allgemeinen Resolution näher. Man braucht als Verbindung noch ein Lifting-Lemma, das die Grundresolutionsschritte in Resolutionsschritte überträgt (liftet). Bei dieser Übertragung erkennt man, dass man auf der allgemeinen Ebene der Klauseln mit Variablen noch die Faktorisierung benötigt, sonst lassen sich nicht alle Grundresolutionsschritte als Resolutionsschritte mit anschließender Instanzbildung der Resolvente verstehen.

2.4 Löschregeln: Subsumtion, Tautologie und Isoliertheit

Wenn man einen automatischen Beweiser, der nur mit Resolution und Faktorisierung arbeitet, beobachtet, dann wird man sehr schnell zwei Arten von Redundanzen feststellen: Der Beweiser wird Tautologien ableiten, d.h. Klauseln, die ein Literal positiv und negativ

[3] Das sind Klauseln mit nur einem Literal

enthalten, z.B. $\{P, \neg P, \ldots\}$. Diese Klauseln sind in allen Interpretationen wahr und können daher zur Suche des Widerspruchs (leere Klausel) nicht beitragen. Man sollte entweder ihre Entstehung verhindern oder sie wenigstens sofort löschen. Weiterhin wird der Beweiser Klauseln ableiten, die Spezialisierungen von schon vorhandenen Klauseln sind. Z.B. wenn schon $\{P(x), Q\}$ vorhanden ist, dann ist $\{P(a), Q\}$ aber auch $\{P(y), Q, R\}$, eine Spezialisierung. Alles was man mit diesen (subsumierten) Klauseln machen kann, kann man genausogut oder besser mit der allgemeineren Klausel machen. Daher sollte man subsumierte Klauseln sofort löschen. Es kann auch passieren, dass eine neue abgeleitete Klausel allgemeiner ist als schon vorhandene Klauseln (die neue subsumiert die alte).

Pragmatisch gesehen, muss man auch verhindern, dass bereits hergeleitete und hinzugefügte Resolventen (Faktoren) noch einmal hinzugefügt werden. D.h. eine Buchführung kann sinnvoll sein.

Im folgenden wollen wir uns die drei wichtigsten Löschregeln und deren Wirkungsweise anschauen und deren Vollständigkeit im Zusammenhang mit der Resolution zeigen.

Definition 2.4.1. Isoliertes Literal
Sei $\mathcal{C}$ eine Klauselmenge, D eine Klausel in $\mathcal{C}$ und L ein Literal in D. L heißt isoliert, *wenn es keine Klausel $D' \neq D$ mit einem Literal L' in $\mathcal{C}$ gibt, so dass L und L' verschiedenes Vorzeichen haben und L und L' unifizierbar sind.*

Die entsprechende Reduktionsregel ist:

Definition 2.4.2. ISOL: Löschregel für isolierte Literale
Wenn D eine Klausel aus $\mathcal{C}$ ist mit einem isolierten Literal, dann lösche die Klausel D aus $\mathcal{C}$.

Der Test, ob ein Literal in einer Klauselmenge isoliert ist, kann in Zeit $O(n^3 \log(n))$ durchgeführt werden.

Dass diese Regel korrekt ist, kann man mit folgender prozeduralen Argumentation einsehen: Eine Klausel D, die ein isoliertes Literal enthält, kann niemals in einer Resolutionsableitung der leeren Klausel vorkommen, denn dieses Literal kann mittels Resolution nicht mehr entfernt werden und ist deshalb in allen nachfolgenden Resolventen enthalten. Dies gilt auch für eine eventuelle spätere Resolution mit einer Kopie von D. Also kann ein Resolutionsbeweis in der restlichen Klauselmenge gefunden werden. Somit gilt:

Theorem 2.4.3. *Die Löschregel für isolierte Literale kann zum Resolutionskalkül hinzugenommen werden, ohne die Widerlegungsvollständigkeit zu verlieren.*

Die Löschregel für isolierte Literale gehört gewissermaßen zur Grundausstattung von Deduktionssystemen auf der Basis von Resolution. Sie kann mögliche Eingabefehler finden und kann Resolventen mit isolierten Literalen wieder entfernen. Der Suchraum wird im allgemeinen jedoch nicht kleiner, denn die Eingabeformeln enthalten normalerweise keine isolierten Literale. Das Löschen von Resolventen mit isolierten Literalen ist noch nicht ausreichend. Denn ein nachfolgender Resolutionsschritt könnte genau denselben

Resolutionsschritt noch einmal machen. Das Klauselgraphverfahren z.B.hat diese Schwächen nicht.

Beispiel 2.4.4. *Betrachte die Klauselmenge*

$$\begin{aligned} C1: &\quad P(a) \\ C2: &\quad P(b) \\ C3: &\quad \neg Q(b) \\ C4: &\quad \neg P(x), Q(x) \end{aligned}$$

Diese Klauselmenge ist unerfüllbar. Am Anfang gibt es keine isolierten Literale. Resolviert man $C1 + C4$ so erhält man die Resolvente $\{Q(a)\}$. Das einzige Literal ist isoliert. Somit kann diese Resolvente gleich wieder gelöscht werden. D.h. dieser Versuch war eine Sackgasse in der Suche.

Eine weitere mögliche Redundanz ist die Subsumtion

Definition 2.4.5. *Seien D und E Klauseln. Wir sagen dass D die Klausel E* subsumiert, *wenn es eine Substitution σ gibt, so dass $\sigma(D) \subseteq E$*

D subsumiert E wenn D eine Teilmenge von E ist oder allgemeiner als eine Teilmenge von E ist. Zum Beispiel $\{P(x)\}$ subsumiert $\{P(a), P(b), Q(y)\}$. Dass eine Klausel E, die von D subsumiert wird, redundant ist, kann man sich folgendermaßen klarmachen:

Wenn eine Resolutionsableitung der leeren Klausel irgendwann E benutzt, dann müssen in nachfolgenden Resolutionsschritten die „überflüssigen“ Literale wieder wegresolviert werden. Hätte man statt dessen D benutzt, wären diese extra Schritte überflüssig.

Die entsprechende Reduktionsregel ist:

Definition 2.4.6. SUBS: Löschregel für subsumierte Klauseln
Wenn D und E Klauseln aus $\mathcal{C}$ sind, D subsumiert E und E hat nicht weniger Literale als D, dann lösche die Klausel E aus $\mathcal{C}$.

Beispiel 2.4.7.

- *P subsumiert $\{P, S\}$.*
- *$\{Q(x), R(x)\}$ subsumiert $\{R(a), S, Q(a)\}$*
- *$\{E(a, x), E(x, a)\}$ subsumiert $\{E(a, a)\}$ D.h eine Klausel subsumiert einen ihren Faktoren. In diesem Fall wird nicht gelöscht.*
- *$\{\neg P(x), P(f(x))\}$ impliziert $\{\neg P(x), P(f(f(x))\}$ aber subsumiert nicht.*

Die Subsumtionslöschregel unterscheidet man manchmal noch nach Vorwärts- und Rückwärtsanwendung. *Vorwärtsanwendung* bedeutet, dass man gerade neu erzeugte Klauseln, die subsumiert werden, löscht. *Rückwärtsanwendung* bedeutet, dass man alte Klauseln löscht, die von gerade erzeugten Klausel subsumiert werden. Die Bedingung, dass D nicht weniger Literale als C haben muss, verhindert, dass Elternklauseln ihre Faktoren

subsumieren. Die Einschränkung auf das syntaktische Kriterium $\theta(C) \subseteq D$ für Subsumtion ist zunächst mal pragmatischer Natur. Es ist nämlich so, dass man die allgemeine Implikation, $C \Rightarrow D$, nicht immer entscheiden kann. Selbst wenn man es entscheiden könnte, wäre es nicht immer geschickt, solche Klauseln zu löschen. Z.B. folgt die Klausel $\{\neg P(x), P(f(f(f(f(f(x))))))\}$ aus der Klausel $\{\neg P(x), Pf(x))\}$. Um eine Widerlegung mit den beiden unären Klauseln $P(a)$ und $\neg P(f(f(f(f(f(a))))))$ zu finden, benötigt man mit der ersten Klausel gerade zwei Resolutionsschritte, während man mit der zweiten Klausel 6 Resolutionsschritte benötigt. Würde man die implizierte Klausel löschen, würde der Beweis also viel länger werden.

Ein praktisches Problem bei der Löschung subsumierter Klauseln ist, dass der Test, ob eine Klausel C eine andere subsumiert, $\mathcal{NP}$-vollständig ist. Die Komplexität steckt in den Permutationen der Literale beim Subsumtionstest. In der Praxis macht das keine Schwierigkeiten, da man entweder die Länge der Klauseln für die Subsumtion testet, beschränken kann, oder den Subsumtionstest unvollständig ausführt, indem man nicht alle möglichen Permutationen von Literalen mit gleichem Prädikat ausprobiert.

Um zu zeigen, dass man gefahrlos subsumierte Klauseln löschen kann, d.h. dass man Subsumtion zu einem Resolutionsbeweiser hinzufügen kann ohne dass die Widerlegungsvollständigkeit verlorengeht, zeigen wir zunächst ein Lemma.

Lemma 2.4.8. *Seien D, E Klauseln in der Klauselmenge $\mathcal{C}$, so dass D die Klausel E subsumiert. Dann wird jede Resolvente und jeder Faktor von E von einer Klausel subsumiert, die ableitbar ist, ohne E zu verwenden, wobei statt E die Klausel D oder Faktoren von D verwendet werden.*

Beweis. Faktoren von E werden offensichtlich von D subsumiert. Seien $E = \{K\} \cup E_R$ und $F = \{M\} \cup F_R$, wobei K und M komplementäre Literale sind und sei $R := \tau E_R \cup \tau F_R$ die Resolvente.

Sei $\sigma(D) \subseteq E$.

1. $\sigma(D) \subseteq E_R$.
   Dann ist $\tau\sigma(D) \subseteq F_R$, d.h. D subsumiert die Resolvente R.

2. $D = \{L\} \cup D_R$, $\sigma D_R \subseteq E_R$ und $\sigma(L) = K$
   Die Resolvente von D mit F auf den Literalen L, M und dem allgemeinsten Unifikator μ ist dann $\mu D_R \cup \mu F_R$. Sei τ' die Substitution, die auf E wie $\tau\sigma$ wirkt und auf F wie τ. Diese Definition ist möglich durch Abänderung auf Variablen, da wir stets annehmen, dass verschiedene Klauseln variablendisjunkt sind. Da μ allgemeinst ist, gibt es eine Substitution λ mit $\lambda\mu = \tau'$. Dann ist $\lambda\mu D_R \cup \lambda\mu F_R = \tau\sigma D_R \cup \tau F_R \subseteq \tau E_R \cup \tau F_R$. Also wird die Resolvente Rvon E und F subsumiert von einer Resolvente von D und F.

3. $\sigma D_R \subseteq E$ aber nicht $\sigma D_R \subseteq E_R$.
   Dann ist $D_R = \{L_1, \ldots, L_m\} \cup D'_R$ und $\sigma L_i = \sigma L = K$. Mit einer Argumentation ähnlich zu der in Fall 1 sieht man, dass es einen Faktor D' von D gibt, der L und

alle L_i verschmilzt und immer noch E subsumiert. Auf diesen kann dann Fall 1 angewendet werden.

□

Theorem 2.4.9. *Der Resolutionskalkül zusammen mit der Löschung subsumierter Klauseln ist widerlegungsvollständig.*

Beweis. Induktion nach der Länge einer Resolutionsableitung mit Lemma 2.4.8 als Induktionsschritt und der Tatsache, dass die einzige Klausel, die die leere Klausel subsumiert, selbst nur die leere Klausel sein kann, liefert die Behauptung. □

Definition 2.4.10. *Sei D eine Klausel. Wir sagen dass D eine* Tautologie *ist, wenn D in allen Interpretationen wahr ist.*

Beispiele für Tautologien sind $\{Pa, \neg Pa\}$, $\{Qa, P(f(x)), \neg P(f(x), Qb\}$ oder $\{Px, \neg Px\}$. Keine Tautologien sind $\{Px, \neg Pf(y)\}$ und $\{\neg P(x, y), P(y, x)\}$.
Ein syntaktisches Kriterium zur Erkennung von tautologischen Klauseln ist der Test, ob zwei komplementäre Literale L, L' enthalten sind mit gleichen Atomen. (siehe Beispiel 2.1.13). Dieser Test ist für die ganze Klauselmenge in Zeit $O(n^3)$ durchführbar.
Die entsprechende Reduktionsregel ist:

Definition 2.4.11. TAUT: Löschregel für tautologische Klauseln
Wenn D eine tautologische Klausel aus der Klauselmenge $\mathcal{C}$ ist, dann lösche die Klausel D aus $\mathcal{C}$.

Da tautologische Klauseln in allen Interpretationen wahr sind, ist die Löschung von Tautologien unerheblich für die Unerfüllbarkeit.

Theorem 2.4.12. *Die Löschregel für tautologische Klauseln ist widerlegungsvollständig.*

Beweis. Hierzu zeigt man, dass ein Beweis, der eine tautologische Klausel benutzt, verkürzt werden kann:
Sei $C = C_1 \vee L \vee \neg L$ eine tautologische Klausel, die im Beweis benutzt wird. Sei $D = D_1 \vee L'$ die Klausel, mit der als nächstes resolviert wird. Wir können annehmen, dass L' und $\neg L$ komplementär sind mit allgemeinstem Unifikator σ. Das Resultat ist $\sigma(C_1 \vee L \vee D_1)$. Diese Resolvente wird von D subsumiert. Mit Lemma 2.4.8 können wir den Resolutionsbeweis verkürzen, indem D statt dieser Resolvente genommen wird. Mit Induktion können so alle Tautologien aus einer Resolutionsherleitung der leeren Klausel eliminiert werden. □
Es gilt:

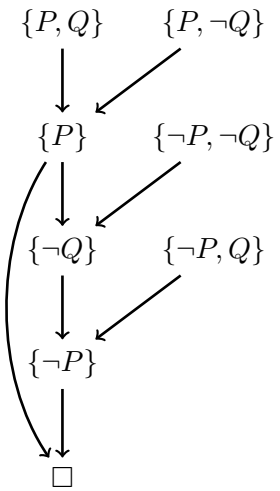
Theorem 2.4.13. *Der Resolutionskalkül zusammen mit Löschung subsumierter Klauseln, Löschung von Klauseln mit isolierten Literalen und Löschung von Tautologien ist widerlegungsvollständig.*

Die Löschung von subsumierten Klauseln kann sehr zur Verkleinerung von Suchräumen beitragen. Umgekehrt kann das Abschalten der Subsumtionsregel einen Beweis dadurch praktisch unmöglich machen, dass mehr als 99% aller abgeleiteten Resolventen subsumierte Klauseln sind. Es gibt noch verschiedene destruktive Operationen auf der Menge der Klauseln, die als Zusammensetzung von Resolution, Faktorisierung und Subsumtion verstanden werden können. Zum Beispiel gibt es den Fall, dass ein Faktor eine Elternklausel subsumiert, wie in $\{P(x,x), P(x,y)\}$. Faktorisierung zu $\{P(x,x)\}$ und anschließende Subsumtionslöschung kann man dann sehen als Ersetzen der ursprünglichen Klausel durch den Faktor. Diese Operation wird auch *Subsumtionsresolution* genannt. Die Prozedur von Davis und Putnam (siehe Abschnitt zu Aussagenlogik) zum Entscheiden der Unerfüllbarkeit von aussagenlogische Klauselmengen kann man jetzt leicht aus Resolution, Subsumtionsregel, Isolationsregel und Fallunterscheidung zusammenbauen.

2.5 Lineare Resolution

Wir betrachten als (eingeschränkte) Variante der Resolution die sogenannte „lineare Resolution“. Die lineare Resolution beginnt mit einer *Zentralklausel*. Am Anfang muss diese als eine der Elternklausel verwendet werden, im Anschluss muss stets die erhaltene Resolvente als Elternklausel verwendet werden. Als zweite Elternklausel kann dabei sowohl eine Eingabeklausel oder auch eine vorher berechnete Resolvente verwendet werden.

Zum Beispiel kann man die Klauselmenge $\{\{P,Q\},\{P,\neg Q\},\{\neg P,Q\},\{\neg P,\neg Q\}\}$ durch lineare Resolution mit der Zentralklausel $\{P,Q\}$ wie folgt widerlegen:



Wenn man Faktorisierung auch erlaubt, so lässt sich nachweisen, dass lineare Resolution widerlegungsvollständig ist. Im Falle von Hornklauseln ist die Faktorisierung nicht notwendig.

2.5.1 Hornklauseln und SLD-Resolution

Hornklauseln sind syntaktisch eingeschränkte Klauseln, d.h. nicht alle Klauseln sind Hornklauseln, und daher werden nicht alle Prädikatenlogischen Formeln von Hornklauselmengen erfasst.

Definition 2.5.1 (Hornklausel)**.** *Eine Hornklausel ist eine Klausel, die* höchstens *ein positives Literal enthält. Eine Klauselmenge, die nur aus Hornklauseln besteht, nennt man* Hornklauselmenge.

Beispiel 2.5.2. $\{\neq R(x), P(a), Q(f(y))\}$ *ist keine Hornklausel, da sie zwei positive Literale enthält. Hingegen sind die Klauseln* $\{\neg R(f(x)), \neg P(g(x,a),)Q(y)\}$ *und* $\{\neg R(g(y)), \neg P(h(b))\}$ *beide Hornklauseln, da die erste Klausel genau ein positives Literal (das Literal* $Q(y)$*) und die zweite Klausel gar keine positiven Literale enthält.*

Hornklauseln finden Verwendung in logischen Programmiersprache, wie z.B. Prolog, das im Skript noch etwas genauer beschrieben wird.
Hornklauseln lassen sich weiter unterteilen:

- *Definite Klauseln* sind Klauseln mit genau einem positiven Literal.
- Definite Einsklauseln(mit positivem Literal) werden auch als *Fakt* bezeichnet.
- Klauseln, die nur negative Literale enthalten, nennt man auch ein *definites Ziel*.

Eine Menge von definiten Klauseln nennt man auch *definites Programm*.

Wie bereits erwähnt sind die lineare Resolution ohne Verwendung der Faktorisierung und auch die Unit-Resolution vollständige Strategien für Hornklauseln.

Die sogenannte *SLD-Resolution* ist nur für Hornklauselmengen definiert. Das D steht dabei für Definite Klauseln. Beginnend mit einer Zentralklausel (die ein definites Ziel ist, also keine positiven Literale enthält), wird Lineare Resolution durchgeführt, wobei eine deterministische Selektionsfunktion bestimmt, welches Literal aus der aktuellen Zentralklausel wegresolviert wird.

Tatsächlich lässt sich nachweisen, dass die Auswahl dieses Literals nicht entscheidend für die Widerlegungsvollständigkeit ist, d.h. unabhängig davon, welches Literal gewählt wird, findet die Resolution die leere Klausel; oder umgekehrt: wenn man Literal L_i zu erst wegresolviert und man findet die leere Klausel nicht, dann findet man sie auch nicht, wenn man zunächst Literal L_j wegresolviert. D.h. die Wahl des Literals ist sogenannter don't care-Nichtdeterminismus der durch die Selektionsfunktion entfernt wird. Man kann hier verschiedene Heuristiken verwenden. Üblicherweise werden zunächst durch die Resolution neu hinzugefügte Literal wegresolviert. Als weitere Selektionsfunktion kann man die Aufschreibereihenfolge verwenden, oder beispielsweise zuerst die am meisten instanziierten Literale (oder die am wenigsten instanziierten Literale) verwenden.

Als weitere Besonderheit der SLD-Resolution ist zu beachten, dass bei einem definiten Ziel als Zentralklausel und definitem Programm als Eingabeklauseln, jede lineare Resolution stets die Resolvente und *eine Seitenklausel* verwendet, d.h. es werden nie zwei Resolventen miteinander resolviert (beachte: bei allgemeiner linearer Resolution ist dieser Fall möglich). Der Grund liegt darin, dass alle Resolventen stets nur aus negativen Literalen bestehen, zur Resolution muss daher eine definite Klausel (mit positivem Literal) als zweite Elternklausel gewählt werden.

Die so definierte SLD-Resolution ist für Hornklauselmengen korrekt und vollständig. Beachte, dass die Resolutionsreihenfolge allerdings nicht-deterministisch ist: Zwar legt die Selektionsfunktion das wegzuresolvierende Literal fest, jedoch kann die zweite Elternklausel beliebig gewählt werden. Ein vollständige Strategie wäre es eine Breitensuche zu verwenden, die alle diese Möglichkeiten quasi gleichzeitig versucht.

Die SLD-Resolution zusammen mit der Breitensuche hat noch eine weitere Vollständigkeitseigenschaft:

SLD-Resolution berechnet für Hornklauselmengen Antworten auf Anfragen: Anfragen sind die negativen Klauseln. Eine Antwort ist eine Grund-Substitution σ und eine Anfrage A, so dass die definiten Klauseln und $\sigma(A)$ unerfüllbar sind.

Die SLD-Resolution ist **vollständig in Bezug auf alle Antworten:**
Es wird bei fairer Vorgehensweise zu jeder möglichen Antwort eine (evtl. allgemeinere, d.h mit Variablen-) Antwort berechnet.

3

Logisches Programmieren

Nicht Teil der Vorlesung im Sommer-Semester 2016

In diesem Kapitel werden wir erläutern, wie logische Programmierung – insbesondere mit Prolog – funktioniert und welche Konzepte dahinter stecken. Wir werden aufgrund des Umfangs nicht auf alle Feinheiten und Besonderheiten von Prolog eingehen. Hierfür sei auf die entsprechenden Bücher verwiesen.

3.1 Von der Resolution zum Logischen Programmieren

Zunächst werden wir erörtern wie man das Herleiten der leeren Klausel mithilfe der Prädikatenlogischen Resolution als *Ausführung eines Programms* auffassen kann.

Wo ist das Programm? Nicht nur in funktionalen Programmiersprachen, sondern in fast allen Programmiersprachen steht das Konzept / Konstrukt der Funktion zu Verfügung, also die Definition einer Abbildung von Eingaben auf Ausgaben. Erinnern wir uns an die Semantik der Prädikatenlogischen Prädikate: Diese stellen *Relationen* dar. Daher kann man Relationen auch als Abbildungen auffassen. Betrachte zum Beispiel die Funktion, die eine Zahl um Eins inkrementiert:

$$f(x) = x + 1.$$

Relational kann man dies als Relation auffassen, die die Eingabe x zur Ausgabe $x + 1$ in Beziehung setzt. Wenn wir daher in der Prädikatenlogik schreiben $\forall x.P(x, x + 1)$ (wobei $+$ als zweistelliges Funktionssymbol aufzufassen ist), so kann man das Prädikat P gerade als Abbildung jedes x auf seinen Nachfolger $x + 1$ interpretieren. Allerdings kann man in diesem Fall für x auch einen beliebigen Term einsetzen und eigentlich ist $x + 1$ rein syntaktisch erstmal nur eine Folge von Symbolen. Tatsächlich erfordert die Behandlung von Zahlen etwas mehr Aufwand und muss durch spezielle Operationen in logische Programmiersprachen eingebaut werden. Außerdem fehlt noch die möglich rekursiv zu programmieren und das Definieren von Bedingungen (z.B. möchte man nicht durch 0 dividieren). Betrachten wir daher ein etwas einfacheres Beispiel:

Die Klauseln:

$$\{vater(peter, maria)\},$$
$$\{mutter(susanne, maria)\},$$
$$\{vater(peter, monika)\},$$
$$\{mutter(susanne, monika)\},$$
$$\{vater(karl, peter)\},$$
$$\{mutter(elisabeth, peter)\},$$
$$\{vater(karl, pia)\},$$
$$\{mutter(elisabeth, pia)\},$$
$$\{vater(karl, paul)\},$$
$$\{mutter(elisabeth, paul)\}$$

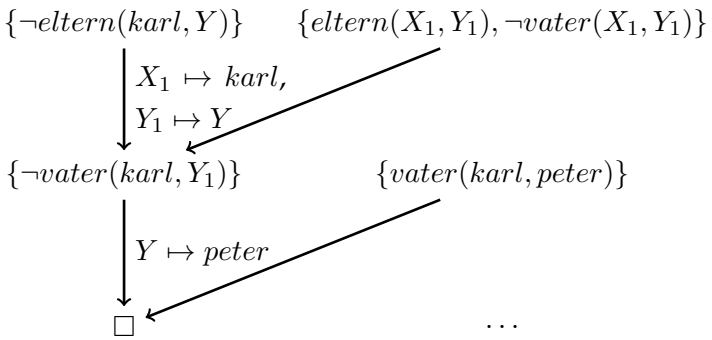
wobei $vater$ und $mutter$ zweistellige Prädikate und alle Vornamen Konstanten sind, kann man als Wissenbasis oder auch als Definition / Fakten auffassen.

Die PL_1-Formel

$$\forall X, Y : vater(X, Y) \implies eltern(X, Y) \wedge \forall X, Y : mutter(X, Y) \implies eltern(X, Y)$$

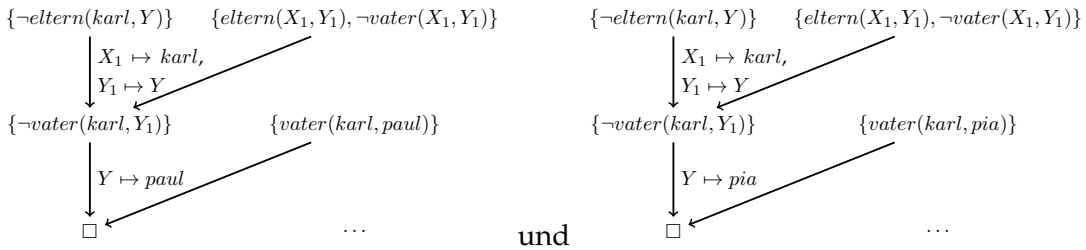
kann man als Definition der Relation $eltern$ auffassen. Beachte, dass wir sowohl X als Eingabe und Y als Ausgabe auffassen können, aber auch umgekehrt. In CNF ergibt dies gerade die beiden Klauseln $\{eltern(X, Y), \neg vater(X, Y)\}$ und $\{eltern(X, Y), \neg mutter(X, Y)\}$.

Ein *Aufruf* des Programms könnte nun z.B. die Anfrage $\exists Y.eltern(karl, Y)$ sein, um zu fragen, ob Karl Kinder hat. D.h. wir möchten aus obiger Klauselmenge (die Wissenbasis / Programm) die Formel $\exists Y.eltern(karl, Y)$ *folgern*. Die Eingabe für den Resolutionskalkül muss daher die negierte Anfrage $\forall Y.\neg eltern(karl, Y)$ sein, oder als Klausel $\{\neg eltern(karl, Y)\}$. Resolution kann hier einen Widerspruch herleiten



Wir würden aber trotzdem gerne wissen, wer nun ein Kind von Karl ist. Die Resolution liefert hierfür zunächst kein echtes Ergebnis (außer dem Beweis der Unerfüllbarkeit der Eingabeformel). Es fehlt daher die Erzeugung einer *Antwort* bzw. Ausgabe des Programms. Hierfür kann man die errechneten Substitutionen verwenden. Beachte: Wir haben oben schon lineare Resolution verwendet, und die Zentralklausel war gerade die An-

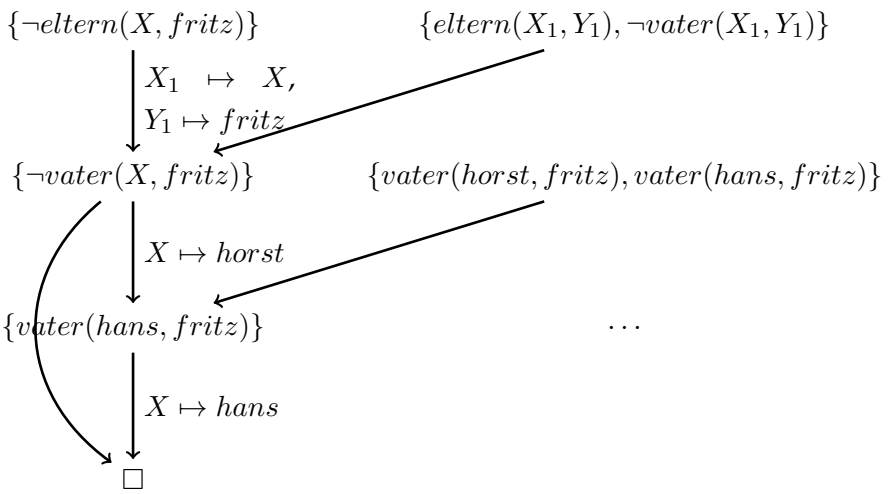
frage $\{\neg eltern(karl, Y)\}$. Wendet man alle in der Herleitung entstanden Substitutionen nacheinander auf diese Klausel an, so erhält man gerade $\{\neg eltern(karl, peter)\}$. Weil wir die leere Klausel hergeleitet haben, können wir daher nicht nur schließen, dass Karl ein Kind hat, sondern dass Peter ein Kind von Karl ist. Es gibt noch zwei weitere Kinder von Karl (nämlich Pia und Paul), diese können durch Verwendung anderer Eingabeklauseln bei der Resolution ebenfalls hergeleitet werden:



Dies sind gerade *alle Möglichkeiten* durch lineare Resolution die leere Klausel herzuleiten.

Zunächst stellt sich die Frage: Funktioniert das Erzeugen einer Antwort mit dieser Methode stets?

Wir erweitern das Beispiel, um die Aussage: „Der Vater von Fritz ist Horst oder Hans“, d.h. wir fügen die Klausel $\{vater(horst, fritz), vater(hans, fritz)\}$ hinzu. Zusätzlich fügen wir noch die Mutter von Fritz hinzu: $\{mutter(anna, fritz)\}$. Die Anfrage, ob Fritz Eltern hat (und welche), entspricht der Formel $\exists X : eltern(X, fritz)$, d.h. wir starten die Resolution mit der Zielklausel $\{\neg eltern(X, fritz)\}$:



D.h. wir können zwar die leere Klausel herleiten, aber die Substitution legt kein eindeutiges Ergebnis fest: X wird einmal auf hans und einmal auf horst abgebildet. Die Antwort wäre daher eine Oder-Verknüpfung. Diese Situation kann immer dann auftreten, wenn eine Klausel mehr als ein positives Literal enthält, was im Beispiel gerade die Klausel $\{vater(horst, fritz), vater(hans, fritz)\}$ war.

Eine andere Möglichkeit keine eindeutige Antwort zu erhalten, besteht darin, wenn die Anfrage selbst eine Disjunktion ist, z.B. $\exists X_1, X_2 : eltern(X_1, maria) \vee eltern(X_2, monika)$. Nach Negation zerfällt diese Anfrage in zwei Klauseln $\{\neg eltern(X_1, maria)\}$ und $\{\neg eltern(X_2, monika)\}$. Da es dann keine eindeutige Zentralklausel gibt, gibt es unter Umständen auch keine eindeutige Antwort.

Die Konsequenz aus diesen Uneindeutigkeiten ist es, dass in Prolog nur *Hornklauseln* erlaubt sind (diese enthalten nie mehr als ein positives Literal), und nur eine Zielklausel erlaubt ist. Neben diesem Vorteil in der Verwendung von Hornklauseln, gibt es weitere Vorteile: Zum Einen sind viele logische Zusammenhänge in Form von Hornklauseln modellierbar, und zum anderen ist die SLD-Resolution widerlegungsvollständig und entspricht gerade dem sequentiellen Ablauf eines Programms.

Obwohl wir Hornklauseln schon im letzten Kapitel definiert haben, werden wir dies nun erneut machen, und gleich die entsprechende Syntax in Prolog einführen.

Definition 3.1.1.

- *Eine* Hornklausel *ist eine Klausel mit maximal einem positiven Literal.*
- *Eine* definite Klausel *ist eine Klausel mit genau einem positiven Literal. D.h. die Klausel ist von der Form:* $A \vee \neg B_1 \vee \ldots \vee \neg B_m$*, was gerade der Implikation* $B_1 \wedge \ldots \wedge B_m \implies A$ *entspricht. In der logischen Programmierung verwendet man die umgekehrte Notation:*

  $$A \Leftarrow B_1, \ldots, B_m.$$

  Beachte: Dabei werden Kommata als Konjunktion interpretiert, der Punkt legt das Ende der Klausel fest.

  In Prolog *wird diese Schreibweise verwendet, allerdings wird anstelle von* $\Leftarrow$ *das Konstrukt :- verwendet, d.h. eine definite Klausel wird in Prolog als*

  $$A \text{ :- } B_1, \ldots, B_m.$$

  geschrieben.

  Man nennt A *den* Kopf *und* $B_1, \ldots, B_m$ *den* Rumpf *der Klausel.*
- *Eine Klausel ohne positive Literale nennt man* definites Ziel (Anfrage,Query,goal)*. Die Notation in der logischen Programmierung ist*

  $$\Leftarrow B_1, \ldots, B_n.$$

  Die B_i *werden* Unterziele (Subgoals) *genannt.*

  Im Prolog-Interpreter gibt man einfach $B_1, \ldots, B_n$*. ein, der Prompt selbst repräsentiert das* $\Leftarrow$ *(der üblicherweise als ?- dargestellt wird).*

- *Eine* Unit-Klausel (*oder* Fakt) *ist eine definite Klausel mit leerem Rumpf. Notation in der logischen Programmierung ist* $A \Leftarrow$. *In Prolog wird der* $\Leftarrow$ *weggelassen, d.h. man schreibt einfach* A.
- *Ein* definites Programm *ist eine Menge von definiten Klauseln.*
- *Die Menge aller Klauseln, deren Kopfliteral das Prädikat* Q *hat, nennen wir* Definition von Q.

Ein definites Programm entspricht einer Und-Verknüpfung aller definiten Klauseln. Beachte, dass jede Hornklausel entweder eine definite Klausel oder ein definites Ziel ist.

Wir geben die Elternbeziehungen vom Anfang als Prolog-Programm an:

```
vater(peter,maria).
vater(peter,monika).
vater(karl, peter).
vater(karl, pia).
vater(karl, paul).
mutter(susanne,monika).
mutter(susanne,maria).
mutter(elisabeth, peter).
mutter(elisabeth, pia).
mutter(elisabeth, paul).

eltern(X,Y) :-  vater(X,Y).
eltern(X,Y) :-  mutter(X,Y).
```

Wenn man dieses Prolog-Programm in den Interpreter lädt[1], kann man Anfragen durchführen. Beachte: Nachdem Ausdrucken der ersten möglichen Antwort, kann man ein Semikolon eingeben, um nach weiteren Antworten zu suchen. Wir führen dies mit einigen Beispielen vor:

```
?- eltern(karl,X).
X = peter ;
X = pia ;
X = paul ;
false.

?- eltern(karl,x).
false.

?- eltern(peter,X).
X = maria ;
X = monika ;
false.
```

[1] Wir verwenden SWI-Prolog (http://www.swi-prolog.org/), das Laden wird über `consult('filename').` ausgeführt werden, wenn `filename.pl` das Programm enthält.

```
?- eltern(X,peter).
X = karl ;
X = elisabeth.

?- eltern(X,Y).
X = peter,
Y = maria ;
X = peter,
Y = monika ;
X = karl,
Y = peter ;
X = karl,
Y = pia ;
X = karl,
Y = paul ;
X = susanne,
Y = monika ;
...
```

Beachte: Variablen werden in Prolog beginnend mit Großbuchstaben oder _ geschrieben. Prädikate und Funktionssymbole beginnen mit einem Kleinbuchstaben. Daher liefert die Anfrage `?- eltern(karl,x).` kein Ergebnis (`x` wird als Konstante interpretiert!).

3.2 Semantik von Hornklauselprogrammen

Wir werden zunächst allgemein die Semantik von Hornklauselprogrammen erörtern. Prolog-Programme und -Interpreter basieren im Grunde auf dieser Semantik, nehmen jedoch praktische Anpassungen vor, die auch die Semantik verändern. Daher werden wir erst im Anschluss auf die Prolog-Semantik eingehen und die Unterschiede herausstellen.

Wir betrachten zunächst die SLD-Resolution für Hornklauselprogramme erneut, und bauen den Antwortmechanismus mit ein. Bei der Resolution sind die Elternklauseln stets mit unterschiedlichen Variablennamen frisch umzubenennen (da die Variablen einer Klausel impliziert allquantifiziert sind). Verwendet man SLD-Resolution für Hornklauselprogramme mit einem definiten Ziel als Zentralklausel, so haben wir bereits gesehen, dass jede Resolution die letzte Resolvente (bzw. am Anfang die Zentralklausel) und eine Seitenklausel als Elternklauseln verwendet. Zur richtigen Umbenennung vor jeder Resolution genügt es, nur die Variablen einer der beiden Elternklauseln umzubenennen. Hierfür können wir stets die Seitenklausel wählen. Wir bezeichnen dieses Vorgehen als *standardisierte SLD-Resolution*.

Wir definieren einen solchen Resolutionsschritt formal:

Definition 3.2.1. *Sei $G = \Leftarrow A_1, \ldots, A_m, \ldots, A_k$ ein definites Ziel und $C = A \Leftarrow B_1, \ldots, B_q$ eine definite Klausel, wobei C frisch umbenannt ist. Dann kann man aus G und C ein neues Ziel G' wie folgt mit Resolution* herleiten*:*

1. *A_m ist das durch die Selektionsfunktion selektierte Atom des definiten Ziels G.*
2. *θ ist ein allgemeinster Unifikator von A_m und A, dem Kopf von C.*
3. *G' ist das neue Ziel: $\theta(A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)$.*

Man sieht: G' ist eine Resolvente von G und C. Die Ableitungsrelation sei bezeichnet durch $G \to_{\theta,C,m} G'$, wobei man die genaue Kennzeichnung C, m auch weglassen kann, wenn diese aus dem Kontext hervorgeht.

Die Semantik eines Hornklauselprogramms besteht dann aus einer Folge dieser Schritte, die auch als *SLD-Ableitung* bezeichnet wird.

Definition 3.2.2. *Sei P ein definites Programm und G ein definites Ziel. Eine* SLD-Ableitung *von $P \cup \{G\}$ ist eine Folge*

$$G \to_{\theta_1,C_1,m_1} G_1 \to_{\theta_2,C_2,m_2} G_2 \ldots$$

von SLD-Schritten, wobei C_i jeweils eine Variante einer Klausel aus P ist mit neuen Variablen.

Die SLD-Ableitung ist eine SLD-Widerlegung, *wenn sie mit einer leeren Klausel endet.*

Im Sinne der linearen Resolution nennt man die Klauseln C_i die Eingabeklauseln.

Weitere Sprechweisen:

erfolgreiche SLD-Ableitung: Wenn es eine SLD-Widerlegung ist.

fehlgeschlagene SLD-Ableitung: Wenn diese nicht fortsetzbar ist.

unendliche SLD-Ableitung

Definition 3.2.3. *Sei P ein definites Programm und G ein definites Ziel.*

- *Eine* korrekte Antwort *ist eine Substitution θ, so dass $P \models \theta(\neg G)$ gilt.*
- *Eine* berechnete Antwort *θ für $P \cup \{G\}$ ist eine Substitution, die man durch eine erfolgreiche SLD-Ableitung $G \to_{\theta_1,C_1,m_1} G_1 \to_{\theta_2,C_2,m_2} G_2 \ldots \to_{\theta_n,C_n,m_n} \square$ erhält: θ ist die Komposition $\theta_n \circ \ldots \circ \theta_1$, wobei man diese auf die Variablen von G einschränkt.*

Theorem 3.2.4 (Soundness der SLD-Resolution)**.** *Sei P ein definites Programm und G ein definites Ziel. Dann ist jede berechnete Antwort θ auch korrekt. D.h. $P \models \theta(\neg G)$*

3.2.1 Vollständigkeit der SLD-Resolution

Wir werden verschiedene Vollständigkeitsaussagen für die SLD-Resolution formulieren.

Theorem 3.2.5. (*Widerlegungsvollständigkeit*)
Sei P ein definites Programm und G ein definites Ziel. Wenn $P \cup \{G\}$ unerfüllbar ist, dann gibt es eine SLD-Widerlegung von $P \cup \{G\}$.

Es gilt der folgende stärkere Satz über die Vollständigkeit der berechneten Antworten, nämlich, dass es zu jeder Antwortsubstitution eine berechnete Antwort gibt, die allgemeiner ist.

Theorem 3.2.6 (Vollständigkeit der SLD-Resolution)**.** *Sei P ein definites Programm und G ein definites Ziel. Zu jeder korrekten Antwort θ gibt es eine berechnete Antwort σ für $P \cup \{G\}$ und eine Substitution γ, so dass für alle Variablen $x \in FV(G)$: $\gamma\sigma(x) = \theta(x)$.*

3.2.2 Strategien zur Berechnung von Antworten

Definition 3.2.7. *Der folgende Algorithmus berechnet alle Antworten mit einer Breitensuche:*

1. *Gegeben ein Ziel $\Leftarrow A_1, \ldots, A_n$:*
   a) *Probiere alle Möglichkeiten aus, ein Unterziel A aus $A_1, \ldots, A_n$ auszuwählen*
   b) *Probiere alle Möglichkeiten aus, eine Resolution von A mit einem Kopf einer Programmklausel durchzuführen.*
2. *Erzeuge neues Ziel B: Löschen von A, Instanziieren des restlichen Ziels, Hinzufügen des Rumpfs der Klausel.*
3. *Wenn $B = \square$, dann gebe die Antwort aus.*

   Sonst: mache weiter mit 1 mit dem Ziel B.

Dieser Algorithmus hat zwei Verzweigungspunkte pro Resolutionsschritt:

- Die Auswahl eines Atoms
- Die Auswahl einer Klausel

Es stellt sich heraus, dass bei der Auswahl des Atoms die restlichen Alternativen nicht betrachtet werden müssen.

Satz 3.2.8. *Vertauscht man in einer SLD-Widerlegung die Abarbeitung zweier Atome in einem Ziel, so sind die zugehörigen Substitutionen bis auf Variablenumbenennung gleich und die Widerlegung hat die gleiche Länge.*

Weiterhin gilt: Die Suchstrategie, die irgendein Atom auswählt, dann alle Klauseln durchprobiert, usw. ist vollständig bzgl. der Antworten.

Beachte: Dies betrifft nicht die *Reihenfolge, in der Seitenklauseln ausgewählt werden*, diese ist nach wie vor nichtdeterministisch.

Aber man kann daher leicht zu lösende Atome zuerst auswählen und schwerer zu lösende zurückstellen.

Es kann aber sein, dass bei günstiger Auswahl nur endlich viele Alternativen ausprobiert werden müssen, aber bei ungünstiger Auswahl evtl. eine unendliche Ableitung ohne Lösungen mitbetrachtet werden muss.

Wir werden dies zeigen, indem wir zunächst *SLD-Bäume* definieren, diese repräsentieren den ganzen Suchraum für ein Ziel, gegeben ein definites Programm und eine Selektionsfunktion.

Definition 3.2.9. *Gegeben sei ein definites Programm P und ein definites Ziel G: Ein* SLD-Baum *für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:*

1. *Jeder Knoten ist markiert mit einem definiten Ziel.*
2. *Die Wurzel ist markiert mit G.*
3. *In jedem Knoten mit nichtleerem Ziel wird ein Atom A des Ziels mit der Selektionsfunktion ausgewählt. Die Kinder dieses Knotens sind dann die möglichen Ziele nach genau einem SLD-Resolutionsschritt mit einer definiten Klausel in P.*
4. *Knoten, die mit der leeren Klausel markiert sind, sind Blätter.*
5. *Ein Blatt ist entweder mit dem leeren Ziel markiert, oder es gibt von dem Blatt aus keine SLD-Resolution.*

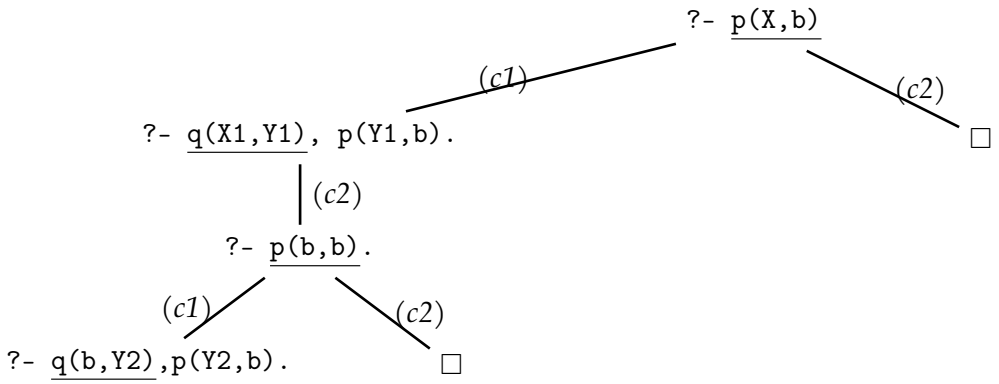
Die Äste (Kanten) entsprechen dabei SLD-Ableitungen. Erfolgreiche Widerlegungen sind Erfolgspfade, unendliche SLD-Ableitungen entsprechen unendlichen Pfaden. Fehlschläge entsprechen Pfaden zu Blättern, die mit einem nichtleeren Ziel markiert sind.

Beispiel 3.2.10. *Programm:*

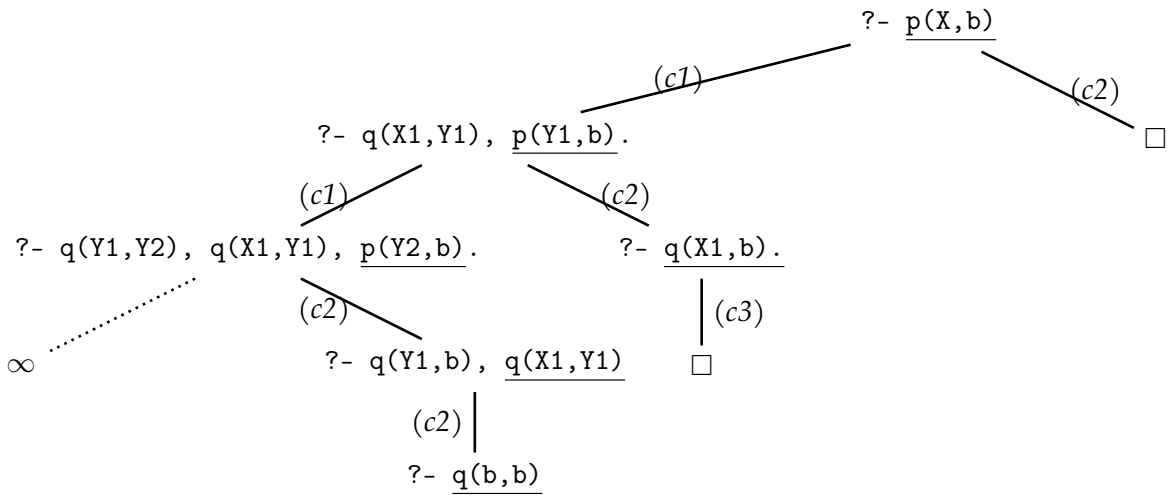
```
(c1)        p(X,Z) :-  q(X,Y), p(Y,Z).
(c2)        p(X,X).
(c3)        q(a,b).
```

Anfrage: `?- p(X,b)` *(Zur Erinnerung: Das entspricht der Klausel $\{\neg p(X, b)\}$)*

Wir schreiben zusätzlich die entsprechende Seitenklausel an die Kanten. Die Selektionsfunktion nehme stets Atome mit q vor den Atomen mit p. Wir unterstreichen das selektierte Atom zur Verdeutlichung. Dann ergibt sich der SLD-Baum:



Nimmt man die Selektionsfunktion: Erst Atome mit p dann Atome mit q, dann kann der Baum unendlich werden:



An diesen SLD-Bäumen kann man (im Prinzip) den ganzen Suchraum ablesen.

Breitensuche in diesen Bäumen findet jedoch alle Lösungen (jede Lösung nach endlich vielen Schritten). Beachte, dass dies nicht für die Tiefensuche gilt, da sie in den unendlichen Pfad laufen kann.

3.3 Implementierung logischer Programmiersprachen: Prolog

In Implementierungen – insbesondere in Prolog – wird i.A. die Suche durch verschiedene Vereinfachungen und Festlegungen deterministisch gemacht:

- Die Anfrage wird als *Stack von Literalen* implementiert. Es wird immer das erste Unterziel zuerst bearbeitet.
- Die Programmklauseln werden in der Reihenfolge abgesucht, in der sie im Programm stehen.
- Bei Ausführung eines SLD-Resolutionsschritts wird der Rumpf an die Stelle des Unterziels gesetzt, und zwar in der Reihenfolge der Literale in der Programmklausel.

D.h. Prolog benutzt die Tiefensuche, wobei die Rechts-Links-Ordnung durch die Reihenfolge der Klauseln im Programm und die Reihenfolge der Literale in den Rümpfen definiert wird.

Beachte, dass der Stack von Literalen die *Selektionsfunktion* festlegt, und daher nicht die Vollständigkeit des Verfahrens ändert. Die *Reihenfolge der Programmklauseln* zu verwenden entspricht jedoch gerade der Tiefensuche, die – wie wir bereits gesehen haben – unvollständig ist. Der Grund hierfür ist praktischer Natur: Zum einen kann die Tiefensuche in

vielen Fällen schneller sein, als die Breitensuche, und zum anderen verbraucht sie bekanntlich viel weniger Platz.

Aus Sicht der Prologgemeinde ist daher der Programmierer dafür verantwortlich die Programmklauseln entsprechend in einer Reihenfolge anzuordnen, so dass die Nichtterminierung nicht eintritt. Im Grunde widerspricht dies dem großen Ziel der *deklarativen Programmierung*, welches fordert, dass im Programm nur zu spezifizieren ist, was berechnet werden soll, aber nicht genau wie. Um Nichtterminierung zu vermeiden, muss der Prolog-Programmierer jedoch die Hintergründe kennen und durch Anordnung der Programmklauseln auch zum Teil das Wie spezifizieren.

3.3.1 Syntaxkonventionen von Prolog

Namen können aus Großbuchstaben, Kleinbuchstaben, Ziffern und _ (Unterstrich) gebildet werden, oder nur aus Sonderzeichen.

Konstanten sind Namen, die mit Kleinbuchstaben beginnen;
Variablen sind Namen, die mit Großbuchstaben oder mit einem Unterstrich _ beginnen. Der Unterstrich alleine _ ist der Name einer anonymen Variablen (wildcard, joker, Leerstelle).

Zusammengesetzte Terme (komplexe Terme) haben die Form

$$Functor(component_1, \ldots, component_n)$$

Die Stelligkeit des Funktors ist nicht variabel. Verwendet man den gleichen Namen mit verschiedener Stelligkeit, so werden diese Vorkommen als verschiedene Objekte interpretiert.

Funktoren, die auf oberster Ebene stehen, nennt man auch Prädikate. Z.B. `besitzt(peter, buch(lloyd, prolog)).`

3.3.2 Beispiele zu Prologs Tiefensuche

Wir betrachten zunächst ein einfaches Beispiel, dass auf dem schon eingeführten Programm basiert.

Beispiel 3.3.1. *Nehmen wir an, wir wollen das Prädikat $vorfahr(X, Y)$ implementieren, das wahr ist, wenn X ein Vorfahre von Y ist (über beliebig viele Generationen hinweg). Mit dem bestehenden Prädikat $eltern$ kann man dies in Prolog implementieren als:*

```
vorfahr1(X,Z) :- vorfahr1(X,Y),vorfahr1(Y,Z).
vorfahr1(X,Y) :- eltern(X,Y).
```

Jeder Aufruf von `vorfahr1` *(z.B. mit konkreten Werten* `vorfahr1(karl,maria)`*) führt direkt zur nicht Terminierung, da mit der ersten Klausel*

`vorfahr1(X,Z) :- vorfahr1(X,Y),vorfahr1(Y,Z).` *resolviert wird, und im Anschluss wieder usw.*

Dreht man die Klauseln um, so erhält man:

```
vorfahr2(X,Y) :- eltern(X,Y).
vorfahr2(X,Z) :- vorfahr2(X,Y),vorfahr2(Y,Z).
```

Dann erhält man Antworten, aber die Suche nach weiteren Antworten führt in den nichtterminierenden Zweig (nachdem alle Instanziierungen für `eltern(X,Y)` *probiert wurden, wird die zweite Klausel verwendet, die dann zur Nichtterminierung führt).*

Eine bessere Variante ist:

```
vorfahr(X,Y) :- eltern(X,Y).
vorfahr(X,Z) :- eltern(X,Y),vorfahr(Y,Z).
```

Diese terminiert stets, da sie stets die Relation `eltern(X,Y)` *instanzieren muss: Da es nur endlich viele solcher Instanziierungen gibt, terminiert die Suche.*

Ein weiteres Beispiel zeigt, dass man die Klauseln nicht immer so anordnen kann, dass die Suche terminiert:

Beispiel 3.3.2. *Sei p ein zweistelliges Prädikat, das symmetrisch und transitiv ist. Wir geben einige Fakten an und die beiden Eigenschaften:*

```
p(a,b).
p(c,b).
p(X,Y) :- p(Y,X).
p(X,Z) :- p(X,Y),p(Y,Z).
```

Für die Anfrage `p(a,c)` *(die offensichtlich* `true` *geben sollte) findet Prolog keine Anwort (sondern terminiert nicht), unabhängig davon, wie man die 4 Klauseln anordnet, es gibt aber eine erfolgreiche SLD-Ableitung, diese muss jedoch einmal die Symmetrie-Klausel und einmal die Transitivität-Klausel anwenden. Die Prolog-Ausführung wendet aber je nach Reihenfolge immer die gleiche Klausel an.*

Eine mögliche Abhilfe ist das Einführen eines zusätzlichen Parameters, der als Tiefenschranke verwendet wird (beachte den Abschnitt 3.3.3 zu Zahlen und arithmetischen Operationen).

```
p(I,a,b).
p(I,c,b).
p(I,X,Y) :-  I > 0, J is I-1,  p(J,Y,X).
p(I,X,Z) :-  I > 0, J is I-1,  p(J,X,Y) , p(J,Y,Z).
```

Ruft man `p(2,a,c)` *auf so erhält man true. I stellt hier gerade die Tiefenschranke dar.*

Eine weitere Veränderung gegenüber der Logik ist, dass in Prologimplementierungen der Occurs Check während der Unifikation aus Effizienzgründen nicht durchgeführt wird. Da es in diesem Fall quasi eine Lösung als unendlichen Term gibt (der natürlich kein PL_1-Term ist), wird einfach weitergerechnet. Das ist aus Sicht der Prädikatenlogik falsch. Aus Prolog-Sicht muss sich der Programmierer darüber bewusst sein.

Gibt man z.B.

```
k(X,X)=k(Y,h(Y)).
```

im Prolog-Interpreter ein (beachte = steht für Gleichheit bzgl. der Terme), so erhält man eine Lösung:

```
X = Y, Y = h(Y).
```

Die Unifikation mit Occurs Check verbietet diese Lösung, da $Y = h(Y)$ bedeutet, dass Y ein unendlicher Term ist.

Weitere Veränderungen in Prolog gegenüber der theoretischen Fundierung sind:

- Es gibt extra Operatoren zum Beeinflussen des Backtracking (*Cut*), der u.a. bewirken kann, dass für ein Unterziel statt vielen Lösungen maximal eine berechnet wird. Dies werden wir unten noch genauer behandeln.
- Assert/Retract: Damit können Programmklauseln zum Programm hinzugefügt bzw. gelöscht werden. I.a, betrifft dies nur Fakten.
- Negation: Negation wird definiert als Fehlschlagen der Suche.
- Es wird z.T. Typinformation zu Programmen hinzugefügt.
- Argumente von Prädikaten kann man mit dem Zusatz I bzw. O versehen. Diese Angaben bedeuten, dass das jeweilige Argument nur als Eingabe bzw. nur als Ausgabe verwendet wird.
- Zusätzlich kann man ein Prädikat als Funktion definieren, wenn die ersten Argumente die Eingabe sind, das letzte Argument die Ausgabe, wobei die Funktion noch zusätzlich als deterministisch (d.h. als mathematische Funktion) deklariert werden muss.

Bevor wir einige dieser Einschränkungen und Erweiterungen betrachten, führen wir zunächst eine wichtige Konstrukte und Prinzipien zur Prolog-Programmierung ein.

3.3.3 Arithmetische Operationen

Die Zeichen $+, -, *, /$ sind erlaubt.

Ausdrücke der Form $a * b + c$ sind zunächst lesbare Abkürzungen für den Term $+(*(a, b), c)$. Hier kann zur Darstellung abhängig von der Prolog-Implementierung Infix, Postfix, Präfix, Assoziativität, Priorität benutzt werden.

Zahlen sind erlaubt und als Standard in Prolog implementiert. Der Zahlbereich ist abhängig von der Implementierung.

Das Spezialprädikat $=$ (Gleichheit) kann definiert werden durch

```
X = X.
```

Z.B.

```
?- besitzt(peter, X) = besitzt(peter,buch(lloyd,prolog))
     yes; X = buch(lloyd,prolog)
```

Diese Operation der Berechnung der Instanzen von Variablen wird mittels Unifikation durchgeführt.

Beachte, dass diese Gleichheit jedoch nicht wahr wird für `1+1 = 2`, da `1+1` als syntaktischer Term aufgefasst wird, der anders aussieht, als der Term (die Konstante) `2`.

Prolog stellt neben diesem Gleichheitstest auch weitere Gleichheitstests und arithmetische Vergleiche zur Verfügung, die jedoch anders wirken:

`=`	Syntaktische Gleichheit (Unfikation erlaubt)
`=:=`	Wertgleichheit, die Argumente werden zu Zahlen ausgewertet und dürfen keine Variablen enthalten
`=/=`	Wertungleichheit
`<`	kleiner als Wertgleichheit
`>`	größer als Wertgleichheit
`=<`	kleiner gleich als Wertgleichheit
`>=`	größer gleich als Wertgleichheit

Außer bei der syntaktischen Gleichheit mit Unifikation =, muss für alle anderen Operationen gelten: Wenn exp1 op exp2 ausgewertet wird, müssen exp1 und exp2 zu einer Zahl auswertbar sein. Beachte:

$3 + 4 = 7$	ergibt: no
$3 + 4 =:= 7$	ergibt yes

Wir testen weitere Beispiele im Prolog-Interpreter:

```
?- 3+4 = X.
X = 3+4.

?- 3+4 =:= 7.
true.

?- 3+4 =:= X.
ERROR: =:=/2: Arguments are not sufficiently instantiated
?- X < 6.
ERROR: </2: Arguments are not sufficiently instantiated
?- 5 < 6.
true.

?- (2*2) < 6.
true.
```

Mit diesen Operatoren kann man daher Variablen keine Werte zuweisen. Deshalb gibt es hierfür das Spezialprädikat `is`, welches in der Form

Variable `is` arithmetischer Ausdruck

verwendet wird und bewirkt, dass der arithmetische Ausdruck ausgewertet wird, und dessen Wert an die Variable gebunden wird. Der arithmetische Ausdruck darf daher zum Zeitpunkt der Auswertung von `is` keine Variablen mehr enthalten.

Wir demonstrieren die Unterschiede zwischen = und `is` anhand einiger Beispielaufrufe im Prolog-Interpreter:

```
?- X is 5.
X = 5.

?- X is (2+2*4).
X = 10.

?-  (2+2*4) is Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?- Y is Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?- X is Y.
ERROR: is/2: Arguments are not sufficiently instantiated
?- X=2, Y is X.
X = 2,
Y = 2.
?- Y=Y.
true.

?- (2+2*4) = Y.
Y = 2+2*4.
```

Beachte, dass alle Prädikate, die Werte auswerten, nicht mehr direkt zur PL_1-Semantik passen.

3.3.4 Listen und Listenprogrammierung

Listen können in logischen Programmiersprachen direkt dargestellt werden, da sie direkt als Terme mit zwei verschiedenen Funktionsymbolen konstruiert werden können:

- Einer Konstante `[]` (gesprochen „Nil“) für die leere Liste
- Einem zweistelligen Funktionssymbol, dass das Listenelement und die Restliste enthält. Dieses wird üblicherweise als „Cons“ bezeichnet. In Prolog ist dieses Funktionssymbol als Punkt `.` dargestellt.

Man kann daher die Liste `[1,2,3]` mit dieser Syntax als `.(1,.(2,.(3,[])))` darstellen. Prolog erlaubt aber auch die Schreibweise `[1,2,3]` als *syntaktischen Zucker* (die intern in den Term `.(1,.(2,.(3,[])))` überführt wird). Das kann man auch im Interpreter testen:

```
?- [1,2,3]   = .(1,.(2,.(3,[]))).
true.
?- [1,2,3]   = .(1,.(2,.(3,.(4,[])))).
false.
?- [1,2,3,4] = .(1,.(2,.(3,.(4,[])))).
true.
```

Als Listenelemente können beliebig Elemente (nicht nur Zahlen) verwendet werden, die auch verschieden sein dürfen, d.h. die Listen in Prolog sind (im Unterschied zu Haskell) heterogen. Einige Beispiele für Listen (gleich mit der internen Darstellung) sind:

```
 ?- [[1], [2,3], [4,5]] = .(.(1,[]), .(.(2,.(3,[])),.(.(4,.(5,[])),[]))).
true.
?- [1,[1],[[1]],[[[1]]]] = .(1,.(.(1,[]),.(.(.(1,[]),[]),.( .(.(.(1,[]),[]),[]),[])))).
true.
?- [f(g(a)),h(b,x),f(f(f(f(c))))] = .(X,.(Y,.(Z,[]))).
X = f(g(a)),
Y = h(b, x),
Z = f(f(f(f(c)))).
```

Prädikate, die der Berechnung des Kopfelementes (head) und der Restliste ohne Kopfelement (tail) entsprechen, kann man damit schon definieren als:

```
head(.(X,_),X).
tail(.(_,XS),XS).
```

Wir testet das gleich:

```
?- head([1,2,3,4],X).
X = 1.

?- tail([1,2,3,4],X).
X = [2, 3, 4].
```

Prolog bietet jedoch eine komfortablere Schreibweise, um Pattern für Listen zu definieren (und damit Listen zusammenzusetzen und zu zerlegen). `.(x,xs)` kann als `[X|XS]` geschrieben werden. Genauer, kann man `|` zum Zerlegen einer Liste an einer beliebigen Position der Liste verwenden, d.h. auch `[X,Y,Z|YS]` ist erlaubt. Wir definieren zunächst `head` und `tail` erneut mit dieser Syntax:

```
head([X|_],X).
tail([_|XS],XS).
```

Einige Aufrufe:

```
?- head([1,2,3],X).
X = 1.

?- tail([1,2,3],X).
X = [2, 3].

?- [X,Y,Z|ZS] = [1,2,3,4,5].
X = 1,
Y = 2,
Z = 3,
ZS = [4, 5].
```

Man kann in Prolog auch für beide Argumente eine Variable eingeben, man erhält dann für den nichtinstanziierten Teil (zumindest in SWI-Prolog) eine interne Variable:

```
?- head(X,Y).
X = [Y|_G304].

?- tail(X,Y).
X = [_G303|Y].
```

Ausgestattet mit diesem Handwerkzeugs, können wir nun verschiedene Listenoperationen in Prolog definieren. Wir beginnen mit dem Prädikat `member`, welches ein Element und eine Liste erwartet, und genau dann wahr sein soll, wenn das Element in der Liste vorkommt. Die Idee dabei ist: Man arbeitet die Liste rekursiv ab:

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Einige Beispielaufrufe:

```
?- member(2,[1,2,3]).
true ;
false.

?- member(2,[1,2,3,2]).
true ;
true ;
false.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
false.

?- member(X,Y).
Y = [X|_G304] ;
Y = [_G303, X|_G307] ;
Y = [_G303, _G306, X|_G310] ;
Y = [_G303, _G306, _G309, X|_G313] ;
Y = [_G303, _G306, _G309, _G312, X|_G316] ;
Y = [_G303, _G306, _G309, _G312, _G315, X|_G319] ;
Y = [_G303, _G306, _G309, _G312, _G315, _G318, X|_G322] ;
Y = [_G303, _G306, _G309, _G312, _G315, _G318, _G321, X|_G325] ;
Y = [_G303, _G306, _G309, _G312, _G315, _G318, _G321, _G324, X|...] ;
Y = [_G303, _G306, _G309, _G312, _G315, _G318, _G321, _G324, _G327|...]
```

Man kann sich fragen, ob `member` in allen möglichen Situationen terminiert.

- `member(s,t)`: wenn t keine Variablen enthält. dann wird t' beim nächsten mal kleiner.
- `member(Y,Y)`: Die erste Klausel unifiziert nicht, wenn der occurs-check eingeschaltet ist, (aber unifiziert, wenn occurs-check aus ist). Die zweite Klausel ergibt:

  ```
  member(X_1,[_|Y_1]) :-  member(X_1,Y_1).
  X_1 = Y = [_|Y_1]
  ```

  Die neue Anfrage ist:

  ```
  member([_|Y_1] ,Y_1)
  ```

  Die nächste Unifikation ebenfalls mit zweiter Klausel

  ```
  member(X_2,[_|Y_2]) :-  member(X_2,Y_2)
  X_2 = [_|Y_1] , Y_1 = [_|Y_2]
  ```

usw. Man sieht: das terminiert nicht.

Das kann man in Prolog auch testen (da eine Tiefensuche verwendet wird). Wir verwenden zwei Varianten: `member` wie oben angegeben und `member2`, für die die beiden Programmklauseln vertauscht sind:

```
?- member2(X,[1,2,3]).
X = 3 ;
X = 2 ;
X = 1.

?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3.

?- member(Y,Y).
Y = [**|_G460]

?- member2(Y,Y).
^C... terminiert nicht
```

Ein weiteres Prädikat ist `islist`, das versucht die Liste komplett auszuwerten. Ein Möglichkeit wäre die Definition:

```
islist([_|X]) :- islist(X).
islist([]).
```

Diese Definition terminiert es für Listen ohne Variablen, aber nicht für `islist(X)`.

Stellt man die Klauseln um, erhält man:

```
islist([]).
islist([_|X]) :- islist(X).
```

Dieses Programm terminiert für die Anfrage `islist(X)`.

Als nächsten Beispiel programmieren wir die Berechnung der Länge einer Liste, d.h. die Anzahl der Elemente die diese enthält. Beachte: Wir verwenden `is`, um die rekursive erhaltene Länge auszuwerten (vgl. Abschnitt zu arithmetischen Operationen).

```
laenge([],0).
laenge([_|X],N) :- laenge(X,N_1), N is N_1 + 1.
```

Damit kann man folgende Beispiele rechnen:

```
?- laenge([1,2,3],N).
N = 3.

?- laenge(XS,2).
XS = [_G880, _G883] ;
^C
?- laenge(XS,N).
XS = [],
N = 0 ;
XS = [_G892],
N = 1 ;
XS = [_G892, _G895],
N = 2 ;
XS = [_G892, _G895, _G898],
N = 3 ;
XS = [_G892, _G895, _G898, _G901],
N = 4 ;
XS = [_G892, _G895, _G898, _G901, _G904],
...
```

Beachte: Die Anfrage

```
?- laenge(XS,2).
```

terminiert in manchen Implementierungen nicht, aber in SWI-Prolog. Allerdings: Wenn man nach weiteren Antworten fragt, terminiert es nicht mehr.

Beispiel 3.3.3. *Sortieren von Listen von Zahlen:*

```
% Praedikat, das testet ob eine Liste sortiert ist

sortiert([]).
sortiert([X]).
sortiert([X|[Y|Z]]) :- X =< Y, sortiert([Y|Z]).

% sortiert_einfuegen(A,BS,CS): fuegt A sortiert in BS ein, Ergebnis: CS

sortiert_einfuegen(X,[],[X]).
sortiert_einfuegen(X,[Y|Z], [X|[Y|Z]]) :-  X =< Y.
sortiert_einfuegen(X,[Y|Z], [Y|U]) :-  Y < X, sortiert_einfuegen(X,Z,U).

% ins_sortiere sortiert die Liste

ins_sortiere(X,X) :-  sortiert(X).
ins_sortiere([X|Y], Z) :- ins_sortiere(Y,U), sortiert_einfuegen(X,U,Z).
```

Einige Testaufrufe:

```
?- sortiert([1,2,3]).
true.
?- sortiert([2,3,1]).
false.
?- sortiert([1,2,X]).
ERROR: =</2: Arguments are not sufficiently instantiated

?- sortiert(X).
X = [] ;
X = [_G312] ;
ERROR: =</2: Arguments are not sufficiently instantiated


?- ins_sortiere([5,1,4,2,3],X).
X = [1, 2, 3, 4, 5] ;
X = [1, 2, 3, 4, 5] ;
X = [1, 2, 3, 4, 5] ;
false.
```

Beispiel 3.3.4. *Programmierung von* `append` *zum Zusammenhängen von Listen:*

```
% append(A,B,C) haengt Listen A und B zusammen

append([],X,X).
append([X|Y],U,[X|Z]) :- append(Y,U,Z).
```

Beispiele:

```
?- append([1,2],[3,4],Z).
Z = [1, 2, 3, 4].

?- append([1,2],Y,Z).
Z = [1, 2|Y].

?- append(X,Y,Z).
X = [],
Y = Z ;
X = [_G663],
Z = [_G663|Y] ;
X = [_G663, _G669],
Z = [_G663, _G669|Y]
...
```

Beispiel 3.3.5. *Mischen von zwei sortierten Listen:*

```
merge([],YS,YS).
merge(XS,[],XS).
merge([X|XS], [Y|YS], [X|ZS]) :- merge(XS,[Y|YS],ZS), X =< Y.
merge([X|XS], [Y|YS], [Y|ZS]) :- merge([X|XS],YS,ZS), X > Y.
```

Beispiele:

```
?- merge([1,10,100],[0,5,50,500],Z).
Z = [0, 1, 5, 10, 50, 100, 500].

?- merge(X,Y,[1,2,3]).
X = [],
Y = [1, 2, 3] ;
X = [1, 2, 3],
Y = [] ;
X = [1],
Y = [2, 3] ;
X = [1, 2],
Y = [3] ;
X = [1, 3],
Y = [2] ;
X = [2, 3],
Y = [1] ;
X = [2],
Y = [1, 3] ;
X = [3],
Y = [1, 2] ;
false.
```

Den letzten Aufruf kann man gerade so auffassen: Berechne alle Listen, die nach dem Mischen `[1,2,3]` *ergeben.*

Beispiel 3.3.6. *Einige weitere Beispiele für Listenfunktionen*

```
listtoset([],[]).
listtoset([X|R],[X|S]) :- remove(X,R,S1),listtoset(S1,S).

remove(E,[],[]).
remove(E,[E|R],S):- remove(E,R,S).
remove(E,[X|R],[X|S]):- not(E == X), remove(E,R,S).

union(X,Y,Z) :- append(X,Y,XY), listtoset(XY,Z).

intersect([],X,[]).
intersect([X|R],S,[X|T]) :- member(X,S),intersect(R,S,T1),listtoset(T1,T).
intersect([X|R],S,T) :- not(member(X,S)),intersect(R,S,T1),listtoset(T1,T).

reverse([],[]).
reverse([X|R],Y) :- reverse(R,RR), append(RR,[X],Y).

reversea(X,Y) :- reverseah(X,[],Y).
reverseah([X|Xs],Acc,Y):- reverseah(Xs,[X|Acc],Y).
reverseah([],Y,Y).
```

3.3.5 Differenzlisten

Wir führen an dieser Stelle eine weitere Darstellung von Listen ein, die sogenannten *Differenzlisten*. Diese werden später im Abschnitt 3.4 zum Parsen und Definite Clause Grammars eine Rolle spielen und verwendet. Diese Darstellung besitzt den Vorteil, dass das append-Prädikat wesentlich effizienter implementiert werden kann.

Betrachten wir zunächst die bisherige Implementierung von append erneut:

```
append([],X,X).
append([X|Y],U,[X|Z]) :- append(Y,U,Z).
```

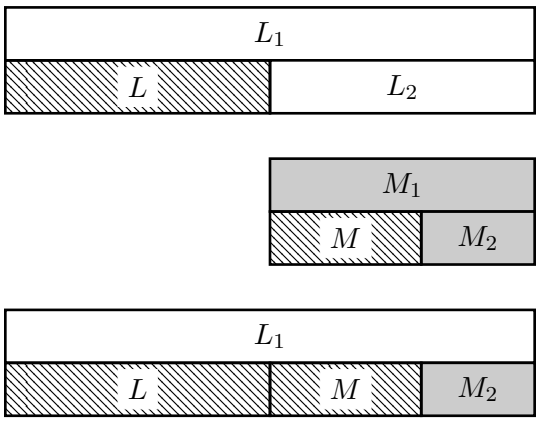
Der Nachteil dieser Implementierung liegt darin, dass die Auswertung der Ergebnisliste die gesamte erste Liste abarbeiten muss, d.h. die Laufzeit ist linear in der Länge der ersten Liste. Dies kann man sich klar machen, wenn man die Abarbeitung beispielhaft durchgeht:

append($[a_1,\ldots,a_n]$,$[b_1,\ldots,b_m]$,L) $X_1 = a_1, Y_1=[a_2,\ldots,a_n], U_1 = [b_1,\ldots,b_m], L = [a_1|Z_1]$
append($[a_2,\ldots,a_n]$,$[b_1,\ldots,b_m]$,Z_1) $X_2 = a_2, Y_2=[a_3,\ldots,a_n], U_2 = [b_1,\ldots,b_m], Z_1 = [a_2|Z_2]$
append($[a_3,\ldots,a_n]$,$[b_1,\ldots,b_m]$,Z_2) $X_3 = a_3, Y_3=[a_4,\ldots,a_n], U_3 = [b_1,\ldots,b_m], Z_2 = [a_3|Z_3]$
.....
append($[]$,$[b_1,\ldots,b_m]$,Z_n) $Z_n = [b_1,\ldots,b_m]$

Da Listen rekursiv dargestellt werden, gibt es zunächst keine bessere Möglichkeit das append-Prädikat zu implementieren. Ein Ausweg stellen die sogenannten Differenzlisten

dar: Anstelle der Liste selbst, wird dabei eine Liste durch *zwei* Listen dargestellt. D.h. eine Liste wird durch ein Paar (L_1, L_2) dargestellt (oder, da in Prolog das Minuszeichen nur einen Term konstruiert auch gleich als Differenz $L_1 - L_2$). Die eigentliche Liste ist dabei die Differenz von L_1 und L_2, d.h. L_2 muss ein *Suffix* von L_1 sein. Z.B. kann die Liste `[1,2,3]` als Differenzliste `([1,2,3,4,5] - [4,5])` dargestellt werden. Offensichtlich ist diese Darstellung nicht eindeutig, vielmehr kann $[1, 2, 3]$ durch jede Differenzliste der Form `([1,2,3|Y] - Y)` dargestellt werden. Zum Programmieren eignet sich die letzte Darstellung (mit `Y` als nicht instanziierte Variable), denn diese schafft gerade den Platz, um etwas an die Liste in konstanter Zeit anzuhängen. Genauer lässt sich dann `append` mit konstanter Laufzeit implementieren. Wir beschreiben zunächst die Idee und geben dann die Implementierung an.

Seien die Listen L und M als Differenzlisten $(L_1 - L_2)$ und $(M_1 - M_2)$ gegeben. Wenn M_1 gerade gleich zu L_2 ist, dann entspricht das Anhängen von M an L gerade der Differenzliste $(L_1 - M_2)$. Das folgende Bild verdeutlich dies



Man sieht, L_2 muss gleich zu M_1 sein. Anderenfalls funktioniert das Aneinanderhängen nicht. Die Implementierung in Prolog ist daher gerade:

```
appendD(L1 - L2,M1 - M2,L1 - M2) :- L2 = M1.
```

Diese kann noch vereinfacht werden, sodass ein einzelner Fakt ausreicht:

```
append(L1 - L2,L2 - M2,L1 - M2).
```

Die Auswertung einer Anfrage `append(Liste1 - Liste1Rest, Liste2 - Liste2, Ergebnis)` ist offensichtlich in konstanter Zeit möglich, wenn `Liste1Rest` eine Variable ist: In diesem Fall muss unifiziert werden: `Liste1Rest = Liste2`, was in konstanter Zeit möglich ist, wenn `Liste1Rest` eine Variable ist.

Ein Beispielaufruf zeigt dies:

```
?- appendD([1,2,3|Y]-Y,[4,5,6|Z]-Z,R).
Y = [4, 5, 6|Z],
R = [1, 2, 3, 4, 5, 6|Z]-Z
```

3.3.6 Der Cut-Operator: Steuerung der Suche

Prolog verfügt über den sogenannten Cut-Operator (der durch das Ausrufezeichen `!` dargestellt wird). Die Wirkung des Operators lässt sich grob dadurch beschreiben, dass die Suche an diesem Operator abgeschnitten wird (daher der Name).

Formaler: Betrachte die Programmklausel $B \Leftarrow A_1, \ldots, A_m, !, A_{m+1}, \ldots, A_n$. Die Wirkung von `!` ist die folgende: Wenn die SLD-Resolutionen, die $A_1, \ldots, A_m$ wegresolviert haben erfolgreich waren, dann wird bei Fehlschlagen der weiteren Suche in $A_{m+1}, \ldots, A_n$ kein Backtracking in $A_1, \ldots, A_m$ durchgeführt. Im SLD-Baum werden daher gerade die anderen Möglichkeiten, $A_1, \ldots, A_m$ wegzuresolvieren, nicht mehr betrachtet.

Wir betrachten ein Beispiel, um die Wirkung und die Nützlichkeit des Cut-Operators zu erläutern.

Betrachte das folgende Beispiel: Ein Mitarbeiter des Wetteramts muss je nach Windstärke X eine der drei Mitteilungen $Y =$ „normal“, „windig“ oder „stürmisch“ weitergeben. Die genauen Regeln dafür sind:

1. Wenn $X < 4$, dann $Y = normal$.
2. Wenn $4 \leq 8$ und $X < 40$ dann $Y = windig$.
3. Wenn $8 \leq X$ dann $Y = stuermisch$.

Da sich der Mitarbeiter die Regeln nicht merken kann, hat er sich ein Prolog-Programm geschrieben:

```
mitteilung(X,normal)     :- X < 4.
mitteilung(X,windig)     :- 4 =< X, X < 8.
mitteilung(X,stuermisch) :- 8 =< X.
```

Angenommen die aktuelle Messung hat 3 ergeben, und der Mitarbeiter nimmt an, er muss „windig“ mitteilen, überprüft dies aber in Prolog:

```
?- mitteilung(3,Y), Y=windig.
false.
```

Wir betrachten den Ablauf der SLD-Resolution: Das erste Ziel ergibt zunächst die Instanziierung $Y \mapsto normal$. Anschließend führt die Unifikation von $windig = normal$ zum Fail. Prolog wird nun Backtracken und die beiden anderen Programmklauseln für mitteilung ausprobieren, obwohl dies eigentlich unnötig ist. Der trace in SWI-Prolog zeigt dies:

```
[trace] 3 ?- mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 4=<3 ? creep
   Fail: (8) 4=<3 ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 8=<3 ? creep
   Fail: (8) 8=<3 ? creep
   Fail: (7) mitteilung(3, _G2797) ? creep
false.
```

Mithilfe des Cut-Operators können wir diese weitere Suche abschneiden, und genauso auch für den Fall, dass die zweite Programmklausel schon instanziiert wurde (dann muss die dritte nicht mehr versucht werden). Als effizientere Version des Programms erhalten wir daher:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- 4 =< X, X < 8,!.
mitteilung(X,stuermisch) :- 8 =< X.
```

Schauen wir uns den trace erneut an, so sehen wir, dass die Suche früher abgebrochen wird:

```
[trace] 4 ?-
|    mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2803) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
false.
```

Die Verwendung des Cut hat hier also nur die Ausführung verändert (optimiert), aber die logische Semantik ist die selbe geblieben. Insbesondere gilt: Das Programm mit Cut verhält sich logisch genauso, wie wenn man die Cuts weglässt.

Eine weitere Optimierung im Beispiel ist, die Abfragen `4 =< X` in der zweiten Programmklauseln und `8 =< X` in der dritten Programmklausel wegzulassen, da die Suche dort erst hinkommt, wenn die entsprechenden Literale sowieso wahr sind. Das ergibt als dritte Variante:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- X < 8,!.
mitteilung(X,stuermisch).
```

Beachte: Diese Implementierung sollte man nicht mit direkten Werten für die Ausgabe aufrufen. Da z.B `?- mitteilung(3,windig)` wahr wird. Im Gegensatz dazu ergibt `?- mitteilung(3,Y), Y=windig.` falsch.

Es stimmt nun auch nicht mehr: Das Programm verhält sich mit Cuts genauso wie ohne Cuts. Denn: Wenn wir die Cuts weglassen, erhalten wir

```
mitteilung(X,normal) :- X < 4.
mitteilung(X,windig) :- X < 8.
mitteilung(X,stuermisch).
```

Nun ergibt die Anfrage `?- mitteilung(3,Y), Y=windig.` als Ergebnis `true`!.

Das Beispiel demonstriert, dass das Hantieren mit Cuts zu semantischen Problemen führt, d.h. die Bedeutung der Programme kann sich ändern. Eine gute Programmierregel ist es, Cuts nur dann einzusetzen, wenn das Weglassen der Cuts die logische Bedeutung nicht ändert, da ansonsten die Verständlichkeit und Wartbarkeit der Programme schwieriger wird.

Wir betrachten einige Beispiele zur Programmierung mit Cut:

Beispiel 3.3.7. *Die Berechnung des Maximums zweier Zahlen kann wiefolgt in Prolog implementiert werden:*

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y < X.
```

Mit Cut kann man dies umschreiben zu:

```
max(X,Y,X) :- X >= Y,!.
max(X,Y,Y).
```

Allerdings sollte man dieses Prädikat nicht direkt mit dem Ergebniswert aufrufen, denn z.B. `?- max(3,1,1).` *ergibt wahr. Man sollte es daher in der Form* `?- max(3,1,Z), Z=3.` *verwenden.*

Beispiel 3.3.8. *Wir haben bereits den Enthaltentest eines Elements in einer Liste programmiert:*

```
member(X,[X|_]).
member(X,[_|Y]) :-  member(X,Y).
```

Mit Cut können wir dies effizienter programmieren als:

```
member(X,[X|_]) :-!.
member(X,[_|Y]) :-  member(X,Y).
```

Wenn die erste Klausel erfolgreich war, muss die zweite nicht mehr betrachtet werden.

Beachte: Die alte Version liefert bei Anfrage `?- member(X,[1,2,3]).` *drei verschiedene Lösungen* `X = 1 ; X = 2 ; X = 3.` *Die Version mit Cut liefert nur eine* `X = 1.`

Beispiel 3.3.9. *Ein if-then-else ist eigentlich nicht darstellbar in purem Prolog. Mit dem Cut-Operator kann dies jedoch programmiert werden als*

```
ifthenelse(B,P,Q) :- B,!,P.
ifthenelse(B,P,Q) :- Q.
```

Beispiel 3.3.10. *Das folgende Beispiel soll nochmal verdeutlichen, dass der Cut-Operator semantikverändernd ist. Betrachte das Programm*

```
p :- a,b.
p :-c.
```

Die logische Bedeutung ist $(a \wedge b \implies p) \wedge (c \implies p)$. *Drehen wir die Reihenfolge der Programmklauseln um, so ändert sich an der Bedeutung nichts. Betrachten wir nun die Variante mit cut:*

```
p :- a,!,b.
p :-c.
```

Die logische Bedeutung ist nun $(a \wedge b \implies p) \wedge (\neg a \wedge c \implies p)$, *da die zweite Klausel nur bei* $\neg a$ *betrachtet wird.*

Umdrehen der Klauseln ergibt:

```
p :-c.
p :- a,!,b.
```

Die logische Bedeutung ist nun anders: $(c \implies p) \wedge (a \wedge b \implies p)$.

3.3.7 Negation: Die Closed World-Annahme

In Prolog kann man in Programmklauseln keine Negation verwenden, da die Programmklausel $A \Leftarrow \neg B$ der Klausel $\{A, B\}$ entspräche, die keine definite Klausel mehr ist, da sie zwei positive Literale enthält. Trotzallem gibt es in Prolog einen `not`-Operator. Dessen Semantik ist jedoch nicht die logische Negation, sondern wird als Fehlschlagen der Suche interpretiert. Wir betrachten als Beispiel die Aussage: Maria mag alle Tiere außer Schlangen. Den ersten Teil der Aussage können wir durch

```
mag(maria,Y) :- tier(Y).
```

ausdrücken. Wir müssen aber noch die Schlangen ausschließen. Es gibt in Prolog das vordefinierte Literal `fail`, dass zum sofortigen Fehlschlagen der Suche führt.

Mit diesem `fail` und dem Cut-Operator können wir obige Aussage ausdrücken:

```
mag(maria,Y) :- Y=schlange,!,fail.
mag(maria,Y) :- tier(Y).
```

Zunächst wird geprüft, ob es sich um eine Schlange handelt. Ist dies der Fall, so wird kein Backtracking durchgeführt und die Suche mit fail beendet. Im Prolog-Interpreter erhält man das richtige Ergebnis (nach Hinzufügen einiger Fakten):

```
?- mag(maria,schlange).
false.

?- mag(maria,hund).
true.

?- mag(maria,katze).
true.
```

Im Grunde ist das Ergebnis `false` aber kein logisches Falsch, sondern sagt nur aus: Der Interpreter hat keine erfolgreiche SLD-Widerlegung gefunden. Genau dies liegt der sogenannten *Closed-World-Assumption* (CWA) zu Grunde: Alles was nicht aus dem Programm folgt, wird als falsch angesehen.

Das `not`-Prädikat ist entsprechend definiert als:

```
not(P) :- P,!,fail.
not(P) :- true.
```

D.h. `not(P)` liefert true, wenn Prolog keine erfolgreiche SLD-Widerlegung für `P` finden kann. Das entspricht natürlich nicht der logischen Negation. Der Programmierer muss sich deshalb darüber bewusst sein, was er tut, wenn er `not` verwendet.

Unser Beispiel kann daher auch programmiert werden als:

```
mag(maria,Y) :- tier(Y), not(Y = schlange).
```

Die Closed-World-Assumption wird in Prolog verwendet. Dies sieht man z.B. auch an folgendem Beispiel

```
rund(ball).
```

Die Anfrage `?- rund(ball)` liefert true. Hingegen liefert die Anfrage `?- rund(erde)` als Ergebnis `false`, obwohl wir eigentlich gar nicht wissen, ob die Erde rund ist, da keinerlei Aussage darüber aus dem Programm folgt. Die Closed-World-Assumption erzwingt das `false`, da angenommen wird, dass alles falsch ist, was nicht herleitbar ist. Ebenso ergibt die Anfrage `?- not(rund(erde))` das Ergebnis `true`, obwohl dies eigentlich nicht aus dem Programm folgt, sondern nur aufgrund der speziellen Semantik von `not`.

Abschließend betrachten wir einige Beispiele

Beispiel 3.3.11. *Unser Programm zu Verwandtschaftsbeziehungen, etwas erweitert:*

```
frau(maria).
frau(elisabeth).
frau(susanne).
frau(monika).
frau(pia).
mann(peter).
mann(karl).
mann(paul).
vater(peter,maria).
vater(peter,monika).
vater(karl, peter).
vater(karl, pia).
vater(karl, paul).
mutter(susanne,monika).
mutter(susanne,maria).
mutter(elisabeth, peter).
mutter(elisabeth, pia).
mutter(elisabeth, paul).
```

Mit `not` *können wir einfach Prädikate, für Geschwister-Beziehungen definieren:*

```
bruder(X,Y) :- mann(X),vater(Z,X),vater(Z,Y),not(X == Y), mutter(M,X),mutter(M,Y).
schwester(X,Y) :- frau(X),vater(Z,X),vater(Z,Y),not(X == Y),mutter(M,X),mutter(M,Y).
geschwister(X,Y) :-  vater(Z,X),vater(Z,Y),not(X == Y),mutter(M,X),mutter(M,Y).
```

Ein anderes, aber im Prinzip ähnliches Beispiel ist ein ungerichteter Graph:

Beispiel 3.3.12. *Ein ungerichteter Graph kann als Kantenrelation dargestellt werden:*

```
kante(a,b).
kante(a,c).
kante(c,d).
kante(d,b).
kante(b,e).
kante(f,g).
```

Das Prädikat `verbunden` *ist wahr, wenn zwei Knoten (über mehrere Kanten) verbunden sind:*

```
verbunden(X,Y) :- kante(X,Y).
verbunden(X,Y) :- kante(Y,X).
verbunden(X,Z) :- kante(X,Y),verbunden(Y,Z).
verbunden(X,Z) :- kante(Y,X),verbunden(Y,Z).
```

So terminiert es allerdings nicht immer. Eine terminierende Variante ist:

```
verbunden2(X,Y) :- verb(X,Y,[]).
ungkante(X,Y) :-  kante(X,Y).
ungkante(X,Y) :-  kante(Y,X).
verb(X,Y,_) :- ungkante(X,Y).
verb(X,Z,L) :- ungkante(X,Y), not(member(Y,L)), verb(Y,Z,[Y|L]).
```

3.3.8 Vergleich: Theoretische Eigenschaften und reale Implementierungen

Folgende Tabelle vergleicht die theoretischen Eigenschaften mit denen implementierter logischer Programmiersprachen.

Pures Prolog Definite Programme	nichtpures Prolog Cut, Negation, Klauselreihenfolge fest	nichtpur, ohne occurs-check
SLD: ist korrekt	SLD: korrekt	SLD: i.a. nicht korrekt
vollständig	unvollständig	unvollständig

3.4 Sprachverarbeitung und Parsen in Prolog

Eine häufige Anwendung der speziellen Abarbeitungsstrategie von Prolog ist die Verwendung als Implementierungssprache für Parser zu gegebenen Grammatiken, insbesondere für natürliche Sprachen (Bratko, 1990; Gal et al., 1991; Pereira & Shieber, 1987). Dies ist auch historisch gesehen die erste Anwendung von Prolog gewesen. Prolog verwendet als Suchstrategie Tiefensuche mit Backtracking. Dies kann direkt für einen rekursiv absteigenden Parser verwendet werden. Tatsächlich stehen nicht nur kontextfreie Grammatiken zur Verfügung, sondern auch erweiterte Grammatiken, z.B. für (schriftliche) Eingabe in natürlicher Sprache.

Wir betrachten zunächst kontextfreie Grammatiken in Prolog, und anschließend die erweiterten Grammatiken und die Sprachverarbeitung. Eine Kontextfreie Grammatik (CFG) besteht aus Nichtterminalen, Terminalen und Produktionen, wobei eine Produktion auf der linken Seite genau ein Nichtterminal hat und auf der rechten Seite eine Folge von Nichtterminalen und Terminalen. Z.B. erzeugt die folgende CFG mit Startsymbol S alle Worte der Form $a^n b^n$:

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow aSb \end{aligned}$$

In Prolog kann diese Grammatik mehr oder weniger direkt eingegeben werden als

```
s --> [a,b].
s --> [a], s, [b].
```

D.h. anstelle von $\rightarrow$ wird `-->` benutzt, Terminale werden in Listenklammern geschrieben und Folgen werden durch Kommata getrennt. Tatsächlich wird hier nur syntaktischer Zucker verwendet, d.h. Prolog übersetzt diese sogenannte Definite Clause Grammar (DCG) – die in diesem Fall auch eine CFG ist – in Hornklauseln. Dabei werden die Nichtterminale in Prädikate übersetzt, die jedoch zwei weitere Argumente zur Verarbeitung der Eingabe erhalten. D.h. im Beispiel wird durch die DCG das zweistellige Prädikat `s` definiert. Die Argumente stellen dabei gerade die Eingabeliste als Differenzliste dar (wobei ein Paar (L_1, L_2) verwendet wird, und nicht die Darstellung $L_1 - L_2$). Wir betrachten zunächst einige Beispielaufrufe für unser Programm.

```
?- s([a,a,b,b],[]).
true.
?- s([a,b,b,a],[]).
false.
?- s([a,a,a,b,b,b],[]).
true.
?- s([a,a,a,b,b,b|X],X).
true .
?- s([a,a,a,b,b,b,c,d],[c,d]).
true.
```

Die genaue Übersetzung ist: Sei die Produktion `n --> n1,...,nm`, wobei alle `ni` zunächst Nichtterminale seien. Dann wird für jede solche Produktion die Klausel

```
n(Eingabe,Rest) :- n1(Eingabe,Rest1), n2(Rest1,Rest2),..., nm(Restm,Rest)
```

erzeugt, d.h. die Differenzliste wird quasi sequentiell aufgeteilt: Jedes der Nichtterminale `ni` nimmt sich zur Verarbeitung gerade soviele Elemente aus der Liste wie es zur Verarbeitung benötigt.

Die Übersetzung der Terminale erfolgt dementsprechend: Man kann z.B. annehmen, dass ein Prädikat `terminal` zur Verarbeitung aller Terminale hinzugefügt wird, mit der Definition

```
terminal(Terminale,Eingabe,Rest) :- append(Terminale,Rest,Eingabe).
```

Wenn ni gerade die Liste `[t1,...,tk]` von Terminalen ist, dann wird ni ersetzt durch `terminal([t1,...,tk],Rest`$I-1$`,Rest`I`)` Man kann auch auf die Verwendung des Prädikats `terminal` verzichten und die Bedingungen direkt kodieren.

Kehren wir zurück zur Beispielgrammatik und übersetzen diese nach obiger Anleitung in Prolog-Hornklauseln:

```
s(Ein,Rest) :- terminal([a,b],Ein,Rest).
s(Ein,Rest) :- terminal([a],Ein,Rest1), s(Rest1,Rest2), terminal([b],Rest2,Rest).
terminal(Terminale,Eingabe,Rest) :- append(Terminale,Rest,Eingabe).
```

Wenn man auf `terminal` verzichtet, und die entsprechenden Bedingungen direkt reinkodiert, erhält man

```
s([a,b|Ein],Ein).
s([a|Ein],Rest) :-  s(Ein,[b|Rest]).
```

DCGs können noch erweitert verwendet werden, indem man zusätzliche Argumente als Parameter an den Nichtterminalsymbolen einfügt. Diese werden dann mitunifiziert und es können dadurch weitere Bedingungen in die Grammatik hineinkodiert werden. Damit gehen DCGs über die Klasse der kontextfreien Sprachen hinaus, denn es kann z.B. eine DCG für die Sprache aller Wörter $a^n b^n c^n d^n$ angegeben ist, die bekanntlich nicht kontextfrei ist.

Eine DCG-Grammatik für die nicht-kontextfreie formale Sprache $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$ ist die folgende:

```
s            --> aa(X),bb(X),cc(X),dd(X).
aa(0)        --> [a].
aa(succ(X))  --> [a], aa(X).
bb(0)        --> [b].
bb(succ(X))  --> [b], bb(X).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Die zusätzlichen Parameter an den Nichtterminalen sind gerade Peanozahlen (die aus `succ` und 0 aufgebaut sind). Die erste Produktion erwingt gerade, dass die Anzahl der erzeugten a's, b's, c's und d's genau gleich sein muss.

Wir testen dies:

```
?- s([a,a,b,b,c,c,d,d],[]).
true .

?- s([a,a,a,b,b,c,c,d,d],[]).
false.

?- s(Eingabe,[]).
Eingabe = [a, b, c, d] ;
Eingabe = [a, a, b, b, c, c, d, d] ;
Eingabe = [a, a, a, b, b, b, c, c, c|...]
...
```

Die Übersetzung in reines Prolog ist gerade

```
s(Ein,Rest) :- aa(X,Ein,Rest1),bb(X,Rest1,Rest2),cc(X,Rest2,Rest3),dd(X,Rest3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,Rest1),  aa(X,Rest1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,Rest1),  bb(X,Rest1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X),Ein,Rest) :- terminal([c],Ein,Rest1),  cc(X,Rest1,Rest).
dd(0,Ein,Rest) :- terminal([d],Ein,Rest).
dd(succ(X),Ein,Rest) :- terminal([d],Ein,Rest1),  dd(X,Rest1,Rest).
```

oder ohne Verwendung von `terminal`:

```
s(Ein,Rest) :- aa(X,Ein,Rest1),bb(X,Rest1,Rest2),cc(X,Rest2,Rest3),dd(X,Rest3,Rest).
aa(0,[a|Ein],Ein).
aa(succ(X),[a|Ein],Rest) :- aa(X,Ein,Rest).
bb(0,[b|Ein],Ein).
bb(succ(X),[b|Ein],Rest) :- bb(X,Ein,Rest).
cc(0,[c|Ein],Ein).
cc(succ(X),[c|Ein],Rest) :- cc(X,Ein,Rest).
dd(0,[d|Ein],Ein).
dd(succ(X),[d|Ein],Rest) :- dd(X,Ein,Rest).
```

Die zusätzlich erlaubten Parameter innerhalb der DCGs sind insbesondere für die Verarbeitung natürlicher Sprache nützlich, da sie bestimmte Attribute der Terminale testen können, die bei bestimmten Satzgliedern übereinstimmen müssen, z.B. Geschlecht, Fall, Zeit, usw. und die Akzeptanz einer Phrase steuern können.

Deutsch (Latein auch) z.B. bauen auf Wortendungen zum Anzeigen der Fälle, Singular/Plural Geschlecht an und erlauben somit auch eine freiere Wahl der Satzstellung. Als Beispielsprache ist Englisch einfacher, da es fast keine Wortendungen gibt, und die Reihenfolge der Worte eines Satzes fast nicht umstellbar ist.

3.4.1 Kontextfreie Grammatiken für Englisch

Eine Tabelle der *syntaktischen Kategorien* ist:

Det	Determiner (Artikel)	(the, a, some)
N	Noun (Nomen)	(table, computer, John)
V	verb	(writes, eats, having)
ADJ	adjectives	(big, fast)
ADV	adverbs	(very, slowly, yesterday)
AUX	auxiliaries (Hilfsverben)	(is, do, has will)
CON	conjunctions	(and, or)
PREP	prepositions	(to, on, with)
PRON	Pronouns (Pronomen)	(he, who, which)

Weitere Bezeichnungen (Subkategorisierung und Komponenten):

S	Sentence	(Satz)
PN	proper noun	("a ...." ist verboten)
IV	intransitive verb	hat kein Objekt
TV	transitives verb	hat Objekt.

Komponenten eines Satzes (Phrasen, Sätze):

S	(Sentence) Satz	
NP	Nounphrase Nominalphrase	das dicke Buch
VP	Verbphrase	schreibe ein Buch
PP	Präpositionalphrase	mit einem Fernglas
ADJP	adjective phrase	größer als man erwartet
ADVP	adverbial phrase	(yesterday evening, gestern abend)
OptRel	relative clause, optional	

Die (erste Annäherung an eine) Beschreibung der Grammatik einer natürlichen Sprache erfolgt mit *Phrasenstrukturregeln*. Diese können mit CFGs zur Konstruktion sowie zur Analyse von Sätzen bzw. Phrasen verwendet werden. Diese Grammatik ist natürlich abhängig von der beschriebenen Sprache (Deutsch, Englisch, Französisch, Russisch, usw.).

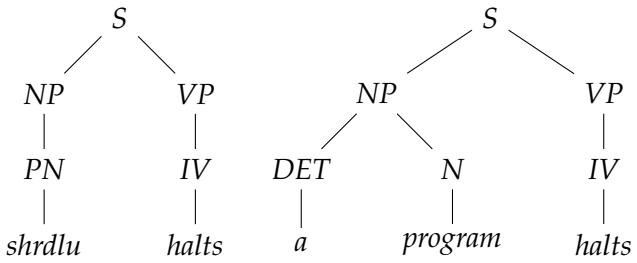
Eine kontextfreie Grammatik (CFG) zum Erzeugen einfacher englischer Sätze kann man wie folgt schreiben:

S	→	NP VP
NP	→	Det N
NP	→	Det N OptRel
NP	→	PN
Optrel	→	that VP
VP	→	TV NP
VP	→	IV

Lexikoneinträge:

PN	→	terry
PN	→	shrdlu
Det	→	a
N	→	program
IV	→	halts
TV	→	writes

Beispiel 3.4.1. *Herleitungsbäume für die Sätze „shrdlu halts“ und „a program halts“ sind:*



In Prolog kann man nun die Kontextfreie Grammatik als DCG eingeben, wobei wir gleich noch das Prädikat `istsatz` zur komfortableren Eingabe definieren:

```
istsatz(Ein) :-  s(Ein,[]).

s      --> np, vp.
np     --> pn.
np     --> det, n.
np     --> det, n, optrel.
optrel --> [that], vp.
vp     --> tv, np.
vp     --> iv.
% Lexikon:
pn     --> [terry].
pn     --> [shrdlu].
det    --> [a].
n      --> [program].
iv     --> [halts].
tv     --> [writes].
```

Einige Beispielaufrufe dazu sind:

```
?- istsatz([a,program,halts]).
true .
?- istsatz([shrdlu,halts]).
true .
?- istsatz([terry,writes,a,program,that,halts]).
true .
?- istsatz([terry,halts,a,program]).
false.
?- istsatz([terry,writes,a,shrdlu]).
false.
?- istsatz([terry,writes,a,program]).
true .
?- istsatz([terry,writes,shrdlu]).
true .
?- istsatz(X).
X = [terry, writes, terry] ;
X = [terry, writes, shrdlu] ;
X = [terry, writes, a, program] ;
X = [terry, writes, a, program, that, writes, terry] ;
X = [terry, writes, a, program, that, writes, shrdlu] ;
X = [terry, writes, a, program, that, writes, a, program] ;
...
```

Übersetzt man die DCG in reine Hornklauseln, so erhält man:

```
istsatz(Ein) :-  s(Ein,[]).

s(Ein,Rest)             :- np(Ein,Rest1), vp(Rest1,Rest).
np(Ein,Rest)            :- pn(Ein,Rest).
np(Ein,Rest)            :- det(Ein,Rest1), n(Rest1,Rest).
np(Ein,Rest)            :- det(Ein,Rest1),n(Rest1,Rest2), optrel(Rest2,Rest).
optrel([that|Ein],Rest) :- vp(Ein,Rest).
vp(Ein,Rest)            :- tv(Ein,Rest1), np(Rest1,Rest).
vp(Ein,Rest)            :- iv(Ein,Rest).
% Lexikon:
pn([terry|Ein],Ein).
pn([shrdlu|Ein],Ein).
det([a|Ein],Ein).
n([program|Ein],Ein).
iv([halts|Ein],Ein).
tv([writes|Ein],Ein).
```

Dieser Parser antwortet nur mit ja/nein bzw. terminiert nicht, insbesondere erhält man keinen Parsebaum.

Man teilt die Grammatik normalerweise ein in die eigentliche Grammatik und das *Lexikon*, das die Terminale beschreibt. Teilweise zählt man auch die *Präterminale* wie z.B. N (Nomen) zum Lexikon.

3.4.2 Verwendung von Definite Clause Grammars

Wir verwenden nun Definite Clause Grammars (DCG's), d.h. wir Verwenden die Erweiterung der kontextfreien Grammatiken um Argumente (Attribute) von Nichtterminalen. Z.B. sollte der Numerus von NP und VP übereinstimmen, d.h. beides sollte entweder im Singular oder Plural sein: "sie gehen. er geht", aber nicht "er gehen".

Dies ergibt z.B. folgendes Grammatikfragment, in dem das Lexikon der Einfachheit halber in der DCG enthalten ist.

In der Syntax der DCGs sieht das folgendermaßen aus, wobei Attribute als Argumente geschrieben werden. Diese Syntax ist in den meisten Prologimplementierungen zulässig.

```
s             -->  np(Number), vp(Number).
np(Number)    -->  pn(Number).
vp(Number)    -->  tv(Number), np(_).
vp(Number)    -->  iv(Number).
pn(singular)  -->  [shrdlu].
pn(plural)    -->  [they].
iv(singular)  -->  [halts].
iv(plural)    -->  [halt].
```

Die Übersetzung in Prolog ergibt sich, durch Hinzufügen der Differenzlisten:

```
s(P0,P)            :- np(Number,P0,P1), vp(Number,P1,P).
np(Number,P0, P)   :- pn(Number,P0,P).
vp(Number,P0, P)   :- tv(Number, P0, P1), np(_, P1, P).
vp(Number,P0, P)   :- iv(Number, P0, P).
pn(singular,P0,P)  :- terminal([shrdlu], P0, P).
pn(plural, P0, P)  :- terminal([they],P0, P).
iv(singular, P0, P):- terminal([halts], P0, P).
iv(plural, P0, P)  :- terminal([halt], P0, P).
```

Die Anfrage `s([shrdlu, halts],[])` hat folgende Verarbeitung:

```
s([shrdlu, halts],[])
   np(N,[shrdlu, halts],P1), vp(N,P1,[])
   pn(N,[shrdlu, halts],P1), iv(N,P1,[])
      terminal(shrdlu,[shrdlu, halts],P1),  iv(singular,P1,[])
            %%%   (N = singular)
terminal(shrdlu,[shrdlu, halts],P1),  terminal([halts],P1,[])
            %%%  (Unifikation mit terminal ergibt P1 = [halts])
terminal([halts],[halts],[]).
Ergibt ‘‘yes‘‘.
```

Es gibt noch weitere dieser Attribute[2] z.B. Person (erste, zweite, dritte) muss übereinstimmen (ich bin; du bist; er, sie, es ist); bei NP und VP muss das Geschlecht übereinstimmen: das Haus (nicht „die Haus“), ebenso gibt es eine Übereinstimmung bei Det und N.

Im Deutschen kann man aus der Endung den Casus (teilweise) ablesen mittels morphologischer Verarbeitung. Die sogenannte morphologische Verarbeitung dient dazu, Worte zu analysieren: auf Endungen, Zerlegung zusammengesetzter Worte, ... In dieser Vorlesungen werden wir darauf nicht weiter eingehen.

Bestimmte Satzkonstruktionen erfordern Dativ, bzw. Akkusativ, ...die ebenfalls als erforderliche Attribute entweder direkt in die Grammatikregeln eingefügt werden oder als entsprechende Verbmarkierungen im Lexikon.
Auch die Zeit oder (Aktiv/Passiv) kann man hinzufügen.

Um weitere Prolog-Tests in eine DCG-Klausel einzufügen, muss man die Prolog-Literale mit geschweiften Klammern einklammern:

Ein Beispiel für eine kleine Micro-DCG des Deutschen ist, wobei folgende Eingabe alle gültigen Sätze dieser Grammatik ausgibt: `s(X,[])`. Z.B. `[ich,fahre, ein, auto]`.

[2] Attribute werden in der Computerlinguistik auch features genannt; Die zugehörigen Grammatiken Unifikationsgrammatiken

```
s     -->  np(Number,Sex,Person), vp(Number,Sex,Person).
np(Number,Sex,Person)    -->  pn(Number,Sex,Person,nom).
vp(Number,Sex,Person)    -->  tv(Number,Sex,Person,Semtv),
      npakk(akk,Semnp),
        {intersect(Semtv,Semnp,Semschnitt), not (Semschnitt  = [])}.
vp(Number,Sex,Person)    -->  iv(Number,Sex,Person).
pn(singular,no,1,nom)  -->  [ich].
pn(singular,no,2,nom)  -->  [du].
pn(singular,male,3,nom)  -->  [er].
pn(singular,female,3,nom)  -->  [sie].
pn(singular,neutrum,3,nom)  -->  [es].
pn(plural,no,1,nom)    -->  [wir].
pn(plural,no,2,nom)    -->  [ihr].
pn(plural,mfn,3,nom)    -->  [sie].
npakk(Fall,Sem)  --> det(Number,Sex,3,Fall),nom(Number,Sex,3,Fall,Sem).
iv(singular,male,3)  -->  [geht].
iv(singular,female,3)  -->  [geht].
iv(singular,neutrum,3)  -->  [geht].
iv(singular,no,1)  -->  [gehe].
iv(singular,no,2)  -->  [gehst].
iv(plural,mfn,3)    -->  [gehen].
iv(plural,no,1)    -->  [gehen].
iv(plural,no,2)    -->  [geht].
tv(singular,male,3,[transport])  -->  [f\"ahrt].
tv(singular,female,3,[transport])  -->  [f\"ahrt].
tv(singular,neutrum,3,[transport])  -->  [f\"ahrt].
tv(singular,no,1,[transport])  -->  [fahre].
tv(singular,no,2,[transport])  -->  [f\"ahrst].
tv(plural,mfn,3,[transport])    -->  [fahren].
tv(plural,no,1,[transport])    -->  [fahren].
tv(plural,no,2,[transport])    -->  [fahrt].
tv(singular,male,3,[information])  -->  [liest].
tv(singular,female,3,[information])  -->  [liest].
tv(singular,neutrum,3,[information])  -->  [liest].
tv(singular,no,1,[information])  -->  [lese].
tv(singular,no,2,[information])  -->  [liest].
tv(plural,mfn,3,[information])    -->  [lesen].
tv(plural,no,1,[information])    -->  [lesen].
tv(plural,no,2,[information])    -->  [lest].

det(singular,neutrum,3,akk) --> [ein].
det(singular,neutrum,3,akk) --> [kein].
det(plural,neutrum,3,akk) --> [zwei].
det(singular,male,3,akk) --> [einen].
det(singular,male,3,akk) --> [keinen].
det(plural,male,3,akk) --> [zwei].
det(singular,female,3,akk) --> [eine].
det(singular,female,3,akk) --> [keine].
det(plural,female,3,akk) --> [zwei].
nom(singular,neutrum,3,akk,[transport]) --> [auto].
nom(plural,neutrum,3,akk,[transport]) --> [autos].
nom(singular,male,3,akk,[transport]) --> [bus].
nom(plural,male,3,akk,[transport]) --> [busse].
nom(singular,female,3,akk,[information]) --> [zeitung].
nom(plural,female,3,akk,[information]) --> [zeitungen].
```

3.4.2.1 Lexikon

Aus heutiger Sicht ist das Lexikon nicht nur eine Sammlung von Worten, sondern die *zentrale Struktur* der linguistischen Verarbeitung. Man kann sehr viel bereits im Lexikon kodieren und dann mit einfachen weiteren Regeln auskommen. D.h. dass z.B.viele Eigenschaften eines Wortes im Lexikoneintrag stecken, auch solche Angaben, die besagen in welchen Satzkonstruktionen bestimmte Verben erlaubt sind. Z.B. Passiv-verbot: („ich werde gelaufen"). z.B. dass ein Verb transitiv bzw intransitiv ist oder in beiden Versionen benutzt werden kann. Die Kontextinformation, die ein bestimmtes Wort benötigt, kann im Lexikon kodiert sein. Auch inhaltliche Informationen (Semantik) können dort enthalten sein.

Beispiel 3.4.2.

```
take: verb
 verbtype = transitive
 subject: role  = agent, semfeat= human
 object:  role  = instrument, semfeat=vehicle
 prep-obj: prep  = to, role=goal
 prep-obj: prep  = from, role=source, ...
```

3.4.2.2 Berechnung von Parse-Bäumen

Man kann den Nichtterminalen und Terminalen ein Argument mitgeben, das den Parsebaum mitberechnet. (Dies geht auch in einer attributierten Grammatik.)

```
s         --->  np vp
np        --->  pn
optrel    --->  that vp
vp        --->  iv
```

Die Implementierung kann dafür z.B. jedes Prädikat um ein Argument erweitern, das den (berechneten) Syntaxbaum aufnimmt.

```
s(sent(TR_np, TR_vp), Ein, Rest) :- np(TR_np,...), vp(TR_vp,...).
np(nphrase(TR_pn),...) :- PN(TR_pn,...).
optrel(relativ_satz(TR_iv),...) :- iv(TR_iv,...)
.....
```

Hier sollte z.B. der Satz "Shrdlu halts" als Ausgabe den Syntaxbaum
`sent(nphrase(propernounn(shrdlu)), verbphrase(intransverb(halts)))`
erzeugen.

3.4.2.3 DCG's erweitert um Prologaufrufe

Man kann in der Implementierung von DCGs Prolog-aufrufe einbetten, die bei der Übersetzung in Prolog mitübernommen werden. Syntaktisch macht man das durch Klammerung mit geschweiften Klammern:

```
s --> np,vp,{teste(X)}.
```

`s,np,vp` werden um die Ein- und Ausgabelisten ergänzt, das Literal in der Klammer wird übernommen.

3.4.2.4 DCG's als formales System

DCG's (ohne eingebettete Literale) sind als formales System aufzufassen, das analog zu CFGs eine formale Sprache definiert. Zusätzlich zu CFGs sind an den Nichtterminalen Argumente erlaubt, die Variablen und Konstanten sein können und auf Gleichheit getestet werden dürfen.

Die formale Sprache die zu einer DCG gehört, entspricht genau den erfolgreichen Parses nach der Übersetzung in definite Klauseln, wenn man SLD-Resolution als operationale Semantik nimmt.

Allerdings ist das nicht dasselbe wie die Gleichsetzung mit der Prologimplementierung derselben, da nach der Übersetzung das Parsen als rekursiv absteigend festgelegt ist.

DCGs kann man zu attributierten Grammatiken dadurch abgrenzen, dass die letzteren kontextfreie Grammatiken sind, die in den Regeln Berechnungsalgorithmen für eine festgelegte Menge von Attributen enthalten, aber aufgrund der Berechnungen keine Eingabe ablehnen können.

Eine theoretische Klassifizierung zwischen DCGs und CFG bzgl der erkannten formalen Sprachen ist: Offenbar sind DCGs eine echte Erweiterung der CFGs.

4

Qualitatives zeitliches Schließen

Um Repräsentation von Zeit und zeitlichen Zusammenhängen kommt man nicht herum. Es gibt viele Logiken hierfür, beispielsweise können Varianten der Modallogik Zeitpunkte als Zustände repräsentieren und erlauben Schlussfolgerungen aus gegebenen Aussagen. Wir betrachten in diesem Kapitel die Zeitlogik von James F. Allen (Allen, 1983), die auch *Allensche Intervall-Logik* genannt wird. In dieser Logik werden nicht konkrete Zeitpunkte oder Dauern repräsentiert, sondern es werden Aussagen über die *relative* Lage von Zeitintervallen getroffen. Dabei sind die Intervalle als Variablen (Namen) gegeben und es gibt Operatoren, um die relative Lage der Intervalle auszudrücken. Mithilfe üblicher aussgagenlogischer Junktoren können solche atomaren Aussagen zu komplexen Formeln zusammengesetzt werden. Nicht repräsentiert sind die genaue Dauer eines Intervalls oder dessen absolute zeitliche Lage. Sinnvoll ist der Einsatz der Allenschen Logik, wenn man nur die Information hat, wie verschiedene Aktionen, die eine gewisse Zeit dauern, zeitlich zueinander liegen können, und man daraus Schlussfolgerungen herleiten will. Sind die Intervalle genau bekannt, dann braucht man diese Logik nicht.

4.1 Allens Zeitintervall-Logik

Wir betrachten zunächst ein Beispiel, welches in Allenscher Intervall-Logik darstellbar ist.

Beispiel 4.1.1. *Wir nehmen ein Teil eines Rezeptes zum Apfelkuchenbacken.*

A1: *Hefeteig zubereiten*
A2: *Hefeteig gehen lassen*
A3: *Äpfel schälen und in Scheiben schneiden*
A4: *Blech einfetten*
A5: *Teig auf das Blech*
A6: *Äpfel auf den Kuchen setzen.*
…
A10: *Backofen heizt*
A11: *Kuchen im Backofen backen*

Da wir annehmen, dass nur eine Person in der Küche ist, können wir folgende Beziehungen angeben:

A2 folgt direkt auf A1
A3 ist später als A1
A3 und A4 sind nicht gleichzeitig
A1 und A4 sind nicht gleichzeitig
A11 ist während A10,
A11 und A10 enden gleichzeitig
A11 ist nach A3
A11 ist nach A4

Hier kann man z.B. mit Transitivität schließen, dass A11 nach A1 stattfindet. Man kann auch ohne Widerspruch hinzufügen, dass A3 nach A4 stattfindet (d.h. man kann sequentialisieren).

Das hat Ähnlichkeiten zu Job Shop Scheduling Problemen (JSSP), bei denen aber feste Zeitdauern pro Aktion und Ausschlusskriterien mittels benutzter Ressourcen festgelegt werden. Auch Abhängigkeiten von Aktionen kann man in JSSPs formulieren.

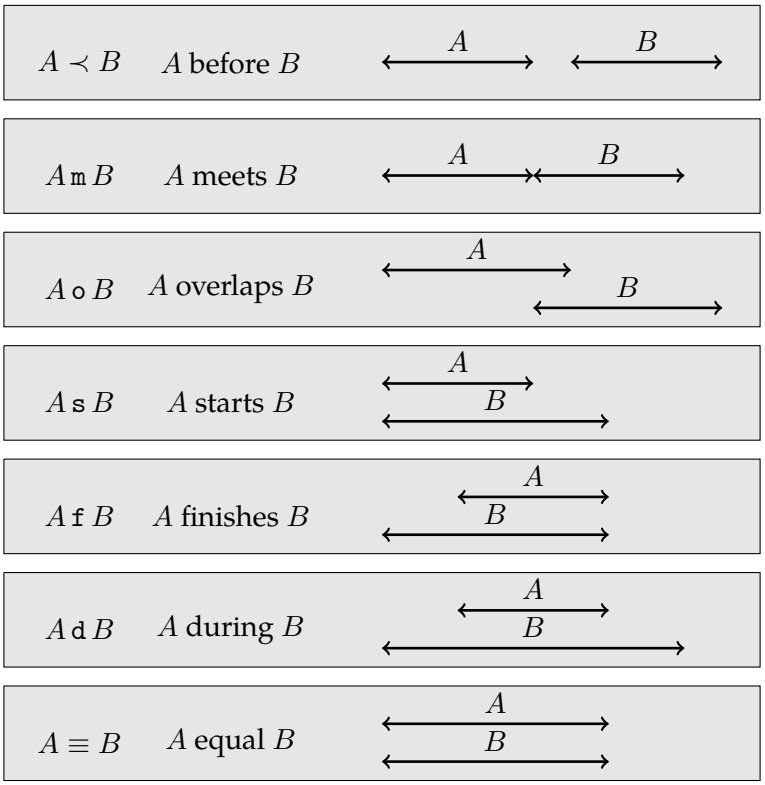
Der Allensche Intervall-Kalkül baut auf Zeitintervallen und binären Relationen zwischen den Intervallen auf, die deren zeitliche Lage beschreiben.

Die Interpretation der Intervalle kann man sich in den reellen Zahlen $\mathbb{R}$ vorstellen, aber in der Repräsentation der Allen-Logik kann man keine exakten Zeiten angeben und auch keine Zeitdauern formulieren.

Die Basis des Kalküls ist die Untersuchung der möglichen relativen Lagen zweier Intervalle $A = [A_a, A_e]$, $B = [B_a, B_e]$, wobei die Anfangs- und Enpunkte A_a, A_e, B_a, B_e reelle Zahlen sind und die Intervalle positive Länge haben (d.h. $A_a > B_e$ und $B_e > B_a$). Eine vollständige und disjunkte Liste der Möglichkeiten ist in folgender Tabelle enthalten, wobei man noch die Fälle hinzunehmen muss, in denen A, B vertauscht sind.

Bedingung	*Abkürzung*	*Bezeichnung*
$A_e < B_a$	$\prec$	A before B
$A_e = B_a$	`m`	A meets B
$A_a < B_a < A_e < B_e$	`o`	A overlaps B
$A_a = B_a < A_e < B_e$	`s`	A starts B
$B_a < A_a < A_e = B_e$	`f`	A finishes B
$B_a < A_a < A_e < B_e$	`d`	A during B
$B_a = A_a, A_e = B_e$	$\equiv$	A equal B

Als Bild kann man die Allenschen Relationen folgendermaßen illustrieren:



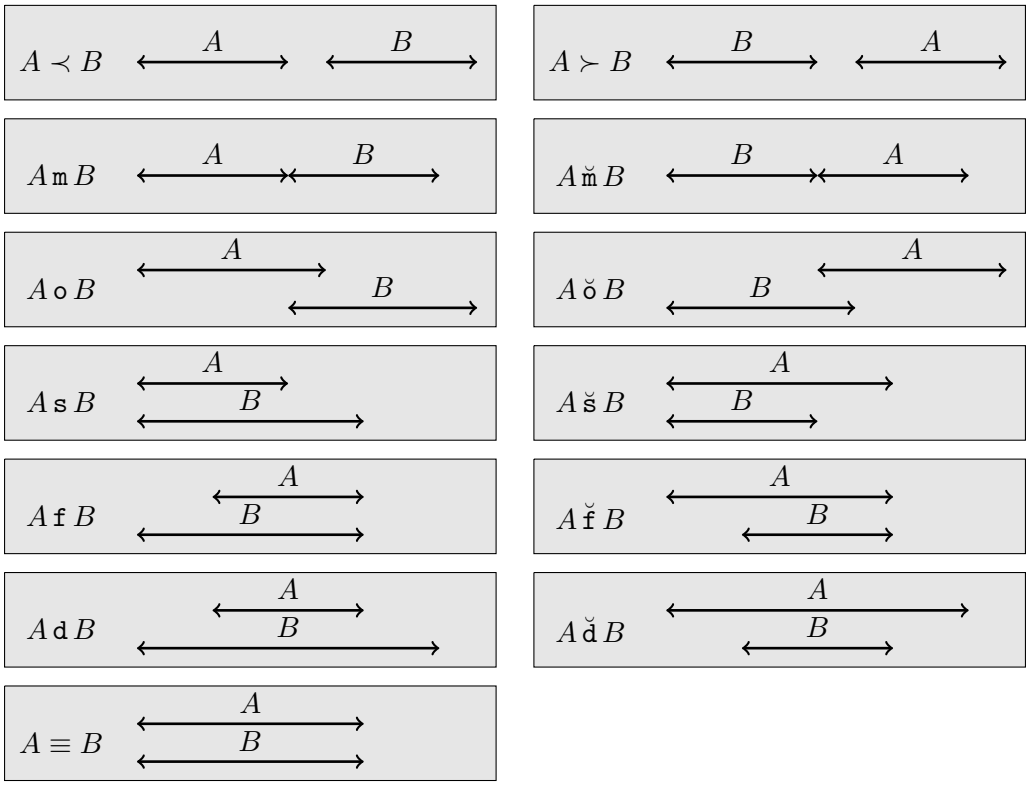
Nimmt man die (relationalen) Inversen dazu, dann hat man die 13 Allenschen Relationen zwischen Zeitintervallen. Die Inversen bezeichnet man mit $\breve{\mathtt{m}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}, \breve{\mathtt{f}}, \breve{\mathtt{d}}$, im Fall von $\prec$ mit $\succ$ und die relationale Inverse von $\equiv$ ist $\equiv$ selbst, da diese Relation symmetrisch ist.

Definition 4.1.2 (Allensche Basisrelationen)**.** *Die 13 Allenschen Basis-Relationen sind:*

$$\{\equiv, \prec, \mathtt{m}, \mathtt{o}, \mathtt{s}, \mathtt{d}, \mathtt{f}, \succ, \breve{\mathtt{m}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}, \breve{\mathtt{d}}, \breve{\mathtt{f}}\}.$$

Wir bezeichnen diese Menge mit $\mathcal{R}$.

Wir stellen zur Verdeutlichung nochmal alle 13 Relationen als Bild dar:



Durch Vergleichen aller Paare von Basis-Relationen sieht man leicht, dass die Relationen alle verschieden sind:

Satz 4.1.3. *Die Allenschen Basis-Relationen sind paarweise disjunkt, d.h.*

$$A\, r_1\, B \wedge A\, r_2\, B \implies r_1 = r_2.$$

Formeln der Allenschen Zeitlogik werden durch Variablen für die Intervalle, verknüpft durch die Basisrelationen und durch aussagenlogische Junktoren gebildet. Wir definieren die Menge der Formeln formal:

Definition 4.1.4 (Syntax der Allen-Formeln)**.** *Die Allen-Zeitlogik kann folgende* Allen-Formeln *bilden:*

- atomare Aussagen $A\ r\ B$, *wobei* A, B *Intervallnamen sind und* r *eine der Allenschen Basis-Relationen ist* (*d.h.* $r \in \mathcal{R}$)
- *aussagenlogische Kombinationen von atomaren Aussagen.*

Die Semantik dazu bildet Intervallnamen auf nichtleere, reelle Intervalle ab. Formeln sind wahr, wenn sie unabhängig von der exakten Zuordnung immer wahr sind. D.h. wenn für alle Abbildungen von Intervallnamen auf nichtleere, reelle Intervalle die Formel stets wahr ist.

Definition 4.1.5 (Semantik der Allen-Formeln)**.** *Eine* Interpretation *I bildet Intervallnamen auf nicht leere Intervalle $[a, b]$ ab, wobei $a, b \in \mathbb{R}$ und $a < b$. Die Erweiterung der Interpretation auf Allensche Formeln berechnet einen Wahrheitswert (0 oder 1) und ist wie folgt definiert:*

- *Wenn $A\ r\ B$ ein atomare Aussage ist und sei $I(A) = [A_a, A_e]$ und $I(B) = [B_a, B_e]$. Dann gilt:*
  - $r = \prec$: $I(A \prec B) = 1$, *gdw.* $A_e < B_a$
  - $r = \texttt{m}$: $I(A\ \texttt{m}\ B) = 1$, *gdw.* $A_e = B_a$
  - $r = \texttt{o}$: $I(A\ \texttt{o}\ B) = 1$, *gdw.* $A_a < B_a$, $B_a < A_e$ *und* $A_e < B_e$
  - $r = \texttt{s}$: $I(A\ \texttt{s}\ B) = 1$, *gdw.* $A_a = B_a$ *und* $A_e < B_e$
  - $r = \texttt{f}$: $I(A\ \texttt{f}\ B) = 1$, *gdw.* $A_a > B_a$ *und* $A_e = B_e$
  - $r = \texttt{d}$: $I(A\ \texttt{d}\ B) = 1$, *gdw.* $A_a > B_a$ *und* $A_e < B_e$
  - $r = \equiv$: $I(A \equiv B) = 1$, *gdw.* $A_a = B_a$ *und* $A_e = B_e$
  - $r = \breve{r_0}$: $I(A\ \breve{r_0}\ B) = 1$, *gdw.* $I(B\ r_0\ A) = 1$
- *Für komplexe Formeln gilt wie üblich:*
  - $I(F \wedge G) = 1$ *gdw.* $I(F) = 1$ *und* $I(G) = 1$
  - $I(F \vee G) = 1$ *gdw.* $I(F) = 1$ *oder* $I(G) = 1$.
  - $I(\neg F) = 1$ *gdw.* $I(F) = 0$
  - $I(F \iff G) = 1$ *gdw.* $I(F) = I(G)$
  - $I(F \Rightarrow G) = 1$ *gdw.* $I(F) = 0$ *oder* $I(G) = 1$

Eine Interpretation ist ein Modell *für eine Allen-Formel F, gdw. $I(F) = 1$ gilt. Eine Allensche Formel F ist:*

- *eine* Tautologie, *wenn jede Interpretation ein Modell für F ist.*
- *ein* Widerspruch (*inkonsistent*), *wenn es kein Modell für F gibt.*
- erfüllbar, *wenn es mindestens ein Modell für F gibt.*

Eine Allensche Formel F folgt semantisch *aus der Allen-Formel G, geschrieben als $G \models F$, genau denn wenn alle Modelle für G auch Modelle für F sind (d.h. falls $I(G) = 1$, dann gilt auch $I(F) = 1$). Zwei Allensche Formeln F, G sind* äquivalent, *gdw. $F \models G$ und $G \models F$ gilt.*

4.2 Darstellung Allenscher Formeln als Allensche Constraints

4.2.1 Abkürzende Schreibweise

Will man mehrere mögliche Lagebeziehungen (also vages Wissen) zwischen 2 Intervallen ausdrücken, so kann man dies disjunktiv machen. z.B.

$$A \prec B \vee A \,\mathtt{s}\, B \vee A \,\mathtt{f}\, B$$

Dies bedeutet: Aktion A ist früher als B, oder A, B starten gleichzeitig oder enden gleichzeitig, wobei in den letzten beiden Fällen A kürzer als B ist. Diese Disjunktion stellen wir verkürzt als $A \{\prec, \mathtt{s}, \mathtt{f}\} B$ dar .

Wir verwenden diese Schreibweise ab jetzt in der folgenden Form:

$$A \; S \; B, \text{ wobei } S \subseteq \mathcal{R}$$

und nennen eine solche Formel ein *atomares Allen-Constraint*. Damit kann man alle Disjunktionen von Einzel-Relationen zwischen zwei Intervallen A, B darstellen.

Dies ergibt eine Anzahl von 2^{13} verschiedenen (unpräzisen) Relationsbeziehungen zwischen Zeitintervallen, wobei man diese Relationen einfach durch eine höchstens 13-elementige Disjunktion bzw. deren Abkürzung darstellen kann. Hier ist auch $A \emptyset B$ dabei, die unmögliche Relation (definiert als $I(A \emptyset B) = 0$ für jede Interpretation I), und $A \mathcal{R} B$, die Relation, die alles erlaubt.

4.2.2 Allensche Constraints

Wir zeigen in diesem Abschnitt, dass man Allen-Formeln viel einfacher darstellen kann: Ein *Allensches Constraint* ist eine Konjunktion von atomaren Allen-Constraints, d.h. eine Formel der Form $A_1 S_1 A_2 \wedge \ldots \wedge A_{n-1} S_n A_n$ wobei $S_i \subseteq \mathcal{R}$ und die A_i nicht notwendigerweise verschieden sind. Jede Allensche Formel kann durch eine Disjunktion von Allenschen Constraints dargestellt werden. Hierfür genügt es zu zeigen, dass einige Vereinfachungen möglich sind, die diese „Normalform“ herstellen können. Im Wesentlichen reichen hierfür aussagenlogische Umformungen aus, jedoch müssen alle Negationen eliminiert werden.

Definition 4.2.1 (Vereinfachungsregeln für Allensche Formeln)**.** *Die Vereinfachungsregeln für Allensche Formeln sind:*

- *Ein atomare Aussage der Form $A \; r \; A$ kann man immer vereinfachen zu $0, 1$:*
  - $A \; r \; A \; \rightarrow \; 0$, *wenn* $r \neq \equiv$ *und*
  - $A \; \equiv \; A \; \rightarrow \; 1$.
- *Negationszeichen kann man nach innen schieben.*
- *Eine Formel $\neg(A \; R \; B)$ kann man zu $A \; (\mathcal{R} \setminus R) \; B$ umformen.*
- *Unterformeln der Form $A \; R_1 \; B \wedge A \; R_2 \; B$ kann man durch $A \; (R_1 \cap R_2) \; B$ ersetzen.*
- *Unterformeln der Form $A \; R_1 \; B \vee A \; R_2 \; B$ kann man durch $A \; (R_1 \cup R_2) \; B$ ersetzen.*
- *atomare Allen-Constraints der Form $A \emptyset B$ kann man durch 0 ersetzen.*

- *atomare Allen-Constraints der Form $A\,\mathcal{R}B$ kann man durch 1 ersetzen.*
- *Alle aussagenlogischen Umformungen sind erlaubt.*

Durch Anwenden dieser Vereinfachungen kann man jede Allensche Formel solange transformieren, bis man eine Disjunktion von Konjunktionen von atomaren Constraints hat. Z.B. $A\ \{\prec, \mathtt{s}, \mathtt{o}\}\ B \wedge A\ \{\succ, \mathtt{s}, \mathtt{o}\}\ B$ entspricht $A\ \{\mathtt{s}, \mathtt{o}\}\ B$. Das Ergebnis ist eine i.A. kleinere Formel, die eine Disjunktion von Konjunktionen von atomaren Allen-Constraints ist. Eine Konjunktion von atomaren Allen-Constraints nennt man *Allen-Constraint*, und die Disjunktion von Allen-Constraints ein *disjunktives Allen-Constraint*.

Es gilt:

Theorem 4.2.2. *Jede Vereinfachungsregel für Allensche Formeln erhält die Äquivalenz, d.h. wenn $F \to F'$, dann sind F und F' äquivalente Formeln.*

Beweis. Dies lässt sich durch Verwendung der Semantik verifizieren. Für Vereinfachungen wie $\neg(A\ R\ B) \to A\ (\mathcal{R} \setminus R)\ B$ muss man im Wesentlichen zeigen, dass $\mathcal{R}$ alle möglichen Lagen von zwei Intervallen auf der reellen Achse abdeckt, und dass die Allenschen Basisrelationen alle disjunkt sind. □

4.3 Der Allensche Kalkül

Der Allensche Kalkül definiert Regeln, um aus gegebenen Allenschen Constraints weitere Beziehungen zwischen Intervallen zu folgern. Die wesentlichen Regeln des Allenschen Kalküls betreffen dabei Beziehungen zwischen drei Intervallen. Man kann beispielsweise aus $A \prec B \wedge B \prec C$ mittels Transitivität von $\prec$ die neue Beziehung $A \prec C$ schließen. Aus $A \prec B \wedge C \prec B$ kann man dagegen nichts neues über die relative Lage von A zu C schließen: Alles (im Sinne der Allenschen Relationen) ist noch möglich.

Da die Menge der Kombinationen endlich ist, kann man sich alle möglichen neuen Relationen als Verknüpfungen $r_1 \circ r_2$ bereits erstellen und damit weitere Relationen zwischen Intervallen herleiten. D.h. wir definieren $r_1 \circ r_2 \subseteq \mathcal{R}$ gerade als minimale Menge von Basisrelationen mit: $A\,r_1\,B \wedge B\,r_2\,C \models A\,(r_1 \circ r_2)\,C$. Wir verallgemeinern dies auch für disjunktive Mengen von Basirelationen: Seien $R_1, R_2 \subseteq \mathcal{R}$, dann ist $R_1 \circ R_2 \subseteq \mathcal{R}$ gerade die minimale Menge von Basisrelationen für die gilt: $A\ R_1\ B \wedge B\ R_2\ C \models A\ (R_1 \circ R_2)\ C$.

Beachte, dass für Basisrelationen r_1, r_2 die Komposition $r_1 \circ r_2$ nicht notwendigerweise eine Basisrelation ist, z.B. ist $\prec \circ \succ = \mathcal{R}$, da man aus $A \prec B \wedge B \succ C$ für die Beziehung zwischen A und C alles folgen kann (also $A\,\mathcal{R}\,C$).

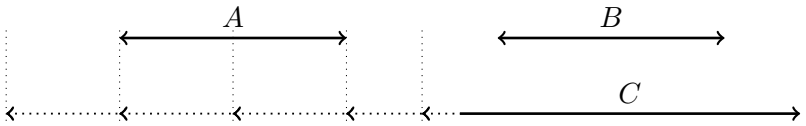
Die Werte für $r_1 \circ r_2$ kann man vor dem Start des Kalküls per Hand ausrechnen und in einer 13×13-Matrix abspeichern. Wir geben die Matrix als 12×12 (ohne die Einträge zu $\equiv$, denn es gilt stets $r \circ \equiv = \equiv \circ r = r$) in Abbildung 4.1 an. Dabei bedeuten die Einträge, die mit $\mathcal{R} \setminus$ beginnen, das Komplement der nachfolgenden Relationen.

Beispiele für Einträge in der 13×13-Matrix sind:

	$\prec$	$\succ$	d	$\breve{\text{d}}$	o	$\breve{\text{o}}$	m	$\breve{\text{m}}$	s	$\breve{\text{s}}$	f	$\breve{\text{f}}$
$\prec$	$\prec$	$\mathcal{R}$	$\prec$ o m d s	$\prec$	$\prec$	$\prec$ o m d s	$\prec$	$\prec$ o m d s	$\prec$	$\prec$	$\prec$ o m d s	$\prec$
$\succ$	$\mathcal{R}$	$\succ$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	$\succ$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	$\succ$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	$\succ$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	$\succ$	$\succ$	$\succ$
d	$\prec$	$\succ$	d	$\mathcal{R}$	$\prec$ o m d s	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	$\prec$	$\succ$	d	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f	d	$\prec$ o m d s
$\breve{\text{d}}$	$\prec$ o m $\breve{\text{d}}$ f	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	$\mathcal{R} \setminus$ $\prec$ $\succ$ m $\breve{\text{m}}$	$\breve{\text{d}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\breve{\text{d}}$	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	$\breve{\text{d}}$
o	$\prec$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o d s	$\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$	$\prec$ o m	$\mathcal{R} \setminus$ $\prec$ $\succ$ m $\breve{\text{m}}$	$\prec$	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o	$\breve{\text{d}}$ $\breve{\text{f}}$ o	d s o	$\prec$ o m
$\breve{\text{o}}$	$\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$	$\succ$	$\breve{\text{o}}$ d f	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	$\mathcal{R} \setminus$ $\prec$ $\succ$ m $\breve{\text{m}}$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\succ$	$\breve{\text{o}}$ d f	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$	$\breve{\text{o}}$	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$
m	$\prec$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o d s	$\prec$	$\prec$	o d s	$\prec$	$\equiv$ f $\breve{\text{f}}$	m	m	d s o	$\prec$
$\breve{\text{m}}$	$\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$	$\succ$	$\breve{\text{o}}$ d f	$\succ$	$\breve{\text{o}}$ d f	$\succ$	$\equiv$ s $\breve{\text{s}}$	$\succ$	d f $\breve{\text{o}}$	$\succ$	$\breve{\text{m}}$	$\breve{\text{m}}$
s	$\prec$	$\succ$	d	$\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$	$\prec$ o m	$\breve{\text{o}}$ d f	$\prec$	$\breve{\text{m}}$	s	$\equiv$ s $\breve{\text{s}}$	d	$\prec$ o m
$\breve{\text{s}}$	$\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$	$\succ$	$\breve{\text{o}}$ d f	$\breve{\text{d}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\breve{\text{o}}$	o $\breve{\text{d}}$ $\breve{\text{f}}$	$\breve{\text{m}}$	$\equiv$ s $\breve{\text{s}}$	$\breve{\text{s}}$	$\breve{\text{o}}$	$\breve{\text{d}}$
f	$\prec$	$\succ$	d	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o d s	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$	m	$\succ$	d	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$	f	$\equiv$ f $\breve{\text{f}}$
$\breve{\text{f}}$	$\prec$	$\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o d s	$\breve{\text{d}}$	o	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	m	$\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$	o	$\breve{\text{d}}$	$\equiv$ f $\breve{\text{f}}$	$\breve{\text{f}}$

Abbildung 4.1: Die Kompositionsmatrix der Allenschen Relationen

Beispiel 4.3.1. *Wenn $A \prec B \wedge B \; \mathtt{d} \; C$, dann kann man sich alle Möglichkeiten für $A \; (\prec \circ \mathtt{d}) \; C$ an folgendem Bild verdeutlichen, indem man alle möglichen linken Anfänge von C betrachtet:*



Man sieht, dass es genau die Möglichkeiten $\prec, \mathtt{o}, \mathtt{m}, \mathtt{s}, \mathtt{d}$ zwischen A und C gibt, d.h. $A \; \{\prec, \mathtt{o}, \mathtt{m}, \mathtt{s}, \mathtt{d}\} \; C$.

Hat man mehrere Elementarrelationen, dann kann man die Kombination der Elementarrelationen bilden, das entsprechende Kompositions-Resultat in einer Tabelle ablesen, und dann die Vereinigung bilden. Z.B. $A \; \{\mathtt{m}, \mathtt{d}\} \; B \wedge B \; \{\mathtt{f}, \mathtt{d}\} \; C$ erlaubt auf folgende Relation zu schließen: $A \; (\mathtt{m} \circ \mathtt{f} \cup \mathtt{m} \circ \mathtt{d} \cup \mathtt{d} \circ \mathtt{f} \cup \mathtt{d} \circ \mathtt{d}) \; C$.

Hier ergibt sich $A\ \{\mathtt{d},\mathtt{s},\mathtt{o}\} \cup \{\mathtt{d},\mathtt{s},\mathtt{o}\} \cup \{\mathtt{d}\} \cup \{\mathtt{d}\}\ C$. Das bedeutet: $A\ \{\mathtt{d},\mathtt{s},\mathtt{o}\}\ C$.

Allgemein gilt:

Satz 4.3.2. *Seien $r_1, \ldots, r_k, r'_1, \ldots, r'_k$ Allensche Basisrelationen. Dann gilt*

$$\{r_1, \ldots, r_k\} \circ \{r'_1, \ldots, r'_k\} = \bigcup \{r_i \circ r'_j \mid i = 1, \ldots, k, j = 1, \ldots, k'\}$$

D.h. man kann die Komposition von Teilmengen von Basisrelationen wieder auf die „punktweise“ Komposition der Basisrelationen zurückführen, die man aus obiger Tabelle ablesen kann.

Eine weitere Vereinfachung ist für die Inversion der Relationen möglich. Zunächst definieren wir:

Definition 4.3.3 (Inversion für Mengen von Basisrelationen)**.** *Sei $S = \{r_1, \ldots, r_k\} \subseteq \mathcal{R}$. Dann sei*

$$\breve{S} = \{\breve{r_1}, \ldots, \breve{r_k}\}.$$

Desweiteren definieren wir für eine Basisrelation r: $\breve{\breve{r}} = r$

Damit gilt:

Satz 4.3.4. *Für $S \subseteq \mathcal{R}$ gilt: $A\ S\ B$ und $B\ \breve{S}\ A$ sind äquivalente Allensche Formeln.*

Beweis. Für die Basisrelationen ist das klar. Sei $S = \{r_1, \ldots, r_k\}$. Dann ist $A\ S\ B$ gerade $A\, r_1\, B \vee \ldots \vee A\, r_k\, B$, und $A\, r_1\, B \vee \ldots \vee A\, r_k\, B$ ist äquivalent zu $B\, \breve{r_1}\, A \vee \ldots \vee B\, \breve{r_k}\, A$, was wiederum per Definition gerade $B\ \breve{S}\ A$ ist. □

Ebenso gilt:

Satz 4.3.5. $\breve{}(r_1 \circ r_2) = \breve{r_2} \circ \breve{r_1}$.

Man kann sich auf die Konjunktion von Relationsbeziehungen beschränken, d.h. auf (konjunktive) Allen-Constraints, da man die Disjunktionen unabhängig bearbeiten kann.

4.3.1 Berechnung des Allenschen Abschlusses eines Constraints

Die folgenden Regeln stellen den Allenschen Kalkül dar. Sie dienen dazu den sogenannten Allenschen Abschluss eines Allenschen Constraints zu berechnen.

Definition 4.3.6 (Regeln des Allenschen Kalküls)**.** *Die folgenden Regeln werden auf (Subformeln) von Allenschen Constraints angewendet:*

- *Aussagenlogische Umformungen dürfen verwendet werden.*
- $A\ R_1\ B \wedge A\ R_2\ B \rightarrow A\ (R_1 \cap R_2)\ B$
- $A\ \emptyset\ B \rightarrow 0$
- $A\ \mathcal{R}\ B \rightarrow 1$

- $A\ R\ A \to 0$, *wenn* $\equiv \notin R$.
- $A\ R\ A \to 1$, *wenn* $\equiv \in R$.
- $A\ R\ B \to A\ R\ B\ \wedge\ B\ \breve{R}\ A$
- $A\ R_1\ B \wedge B\ R_2\ C \to A\ R_1\ B \wedge B\ R_2\ C \wedge A\ (R_1 \circ R_2)\ C$.

Den Allenschen Abschluss eines (u.U. auch disjunktiven) Allenschen Constraints berechnet man, indem man obige Regel solange wie möglich anwendet:

Definition 4.3.7 (Allenscher Abschluss)**.** *Für konjunktive Formeln (d.h. Allensche Constraints) werden die Regeln des Allenschen Kalküls solange angewendet, bis sich keine neuen Beziehungen mehr herleiten lassen.*

Hat man einen disjunktiven Allenschen Constraint, so wendet man die Fixpunktiteration auf jede Komponente einzeln an, anschließend kann man u.U. vereinfachen: Erhält man für eine Komponente 1, so ist der disjunktive Allen-Constraint äquivalent zu 1. 0-en kann man wie üblich streichen. Sind alle Disjunktionen falsch, dann hat mann eine Inkonsistenz entdeckt (die Eingabe ist ein widersprüchliches Allen-Constraint).

Beispiel 4.3.8. *Wir betrachten nochmal das Beispiel des Kuchenbackens:*

A1: Hefeteig zubereiten
A2: Hefeteig gehen lassen
A3: Äpfel schälen und in Scheiben schneiden
A4: Blech einfetten
A5: Teig auf das Blech
A6: Äpfel auf den Kuchen setzen.

Als Eingabe kann man die folgenden Relationen zwischen $A1, \ldots, A6$ *nehmen, wenn mehrere Personen eine parallelisierte Verarbeitung ermöglichen:*

$A1\ \mathtt{m}\ A2$
$A2\ \{\mathtt{m}, \prec\}\ A5$
$A4\ \{\mathtt{m}, \prec\}\ A5$
$A3\ \{\mathtt{m}, \prec\}\ A6$
$A5\ \{\mathtt{m}, \prec\}\ A6$

Das ergibt das Allensche Constraint

$$A1\ \mathtt{m}\ A2 \wedge A2\ \{\mathtt{m}, \prec\}\ A5 \wedge A4\ \{\mathtt{m}, \prec\}\ A5 \wedge A3\ \{\mathtt{m}, \prec\}\ A6 \wedge A5\ \{\mathtt{m}, \prec\}\ A6.$$

Berechnet man den Allenschen Abschluss, muss man im Wesentlichen nur prüfen, ob man mit der Transitivitätsregel neue Beziehungen findet.

- *Aus* $A1\ \mathtt{m}\ A2 \wedge A2\ \{\mathtt{m}, \prec\}\ A5$ *erhält man* $A1 \prec A5$

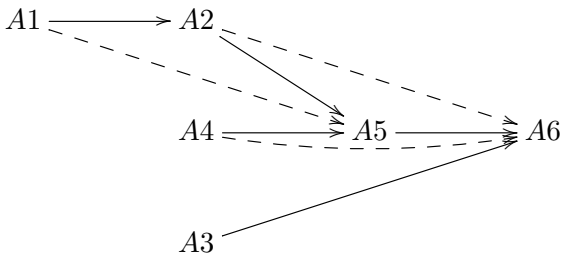
- *Aus* $A1 \prec A5 \wedge A5 \ \{\mathtt{m}, \prec\} \ A6$ *erhält man* $A1 \prec A6$
- *Aus* $A2 \ \{\mathtt{m}, \prec\} \ A5 \wedge A5 \ \{\mathtt{m}, \prec\} \ A6$ *erhält man* $A2 \prec A6$
- *Aus* $A4 \ \{\mathtt{m}, \prec\} \ A5 \wedge A5 \ \{\mathtt{m}, \prec\} \ A6$ *erhält man* $A4 \prec A6$

Mehr kann man nicht herleiten, d.h. der Allensche Abschluss ist gerade das Constraint

$A1 \ \mathtt{m} \ A2 \wedge A1 \prec A5 \wedge A1 \prec A6 \wedge A2 \ \{\mathtt{m}, \prec\} \ A5 \wedge A2 \prec A6 \wedge A4 \ \{\mathtt{m}, \prec\} \ A5 \wedge A4 \prec A6 \wedge A3 \ \{\mathtt{m}, \prec\} \ A6 \wedge A5 \ \{\mathtt{m}, \prec\} \ A6$.

Beachte: Hinzufügen von Beziehungen $Ai \ \mathcal{R} \ Aj$ *macht keinen Sinn, da diese direkt wieder gelöscht werden.*

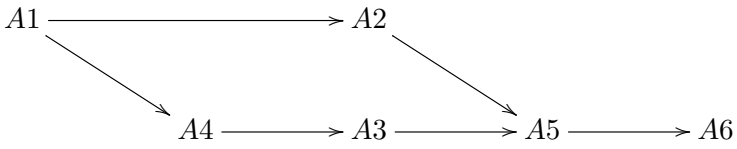
Ein Bild zu den Relationen ist das folgende, wobei Pfeile $A1 \rightarrow A2$ *bedeuten:* $A1$ *findet vor* $A2$ *statt (also* m *oder* $\prec$*). Die gestrichelten Pfeile, deuten die Beziehungen an, die der Allensche Kalkül herleitet.*



Beispiel 4.3.9. *Wir betrachten weiterhin die sechs Aktionen zum Kuchenbacken und nehmen an, dass nur eine Person alleine backt, d.h. es gibt keine parallelen Aktionen. Dann wären die entsprechenden Relationen:*

$A1 \ \mathtt{m} \ A2$
$A2 \ \{\mathtt{m}, \prec\} \ A5$
$A4 \ \{\mathtt{m}, \prec\} \ A5$
$A3 \ \{\mathtt{m}, \prec\} \ A6$
$A5 \ \{\mathtt{m}, \prec\} \ A6$
$A1 \ \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \ A3$
$A1 \ \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \ A4$
$A1 \ \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \ A5$
$A3 \ \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \ A4$
$A3 \ \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \ A4$

Eine Möglichkeit, die man durch Hinzufügen von Sequentialisierung bekommt, ist im Bild dargestellt:



4.4 Untersuchungen zum Kalkül

Wir sagen, der Kalkül ist *korrekt*, wenn bei Transformation von F nach F' gilt, dass F und F' äquivalente Formeln sind.

Wir sagen, der Kalkül ist *herleitungs-vollständig*, wenn er für jedes konjunktive Constraint alle semantisch folgerbaren Einzel-Relationen herleiten kann.

Wir sagen, der Kalkül ist widerspruchs-*vollständig*, wenn er für jedes konjunktive Constraint herausfinden kann, ob es widersprüchlich ist, indem der Kalkül irgendwann die atomare Formel 0 herleitet.

Es stellen sich die folgenden Fragen:

- Wie aufwändig ist die Berechnung des Abschlusses der Allenschen Relationen?
- Ist die Berechnung herleitungs- bzw- widerspruchs-vollständig? D.h. gibt es nicht doch noch nicht erkannte Relationsbeziehungen, die aus der Semantik der linearen, reellen Zeitachse aus einer vorgegebenen Formel folgen?
- Was ist die Komplexität der Logik und der Herleitungsbeziehung, evtl. für eingeschränkte Eingabeformeln?
- Wie kann man den Allenschen Kalkül für aussagenlogische Kombinationen von Intervallformeln verwenden?

4.4.1 Komplexität der Berechnung des Allenschen Abschlusses

Ein Vervollständigungsalgorithmus kann analog zur Berechnung des transitiven Abschlusses einer Relation implementiert werden. Methoden des dynamischen Programmierens kann man dazu verwenden. Das ergibt einen polynomiellen Aufwand zur Vervollständigung.

Wir geben zwei Algorithmen an. Beide Algorithmen nehmen ein konjunktives Allensches Constraint als Eingabe, wobei dieses bereits in Form eines zweidimensionalen Arrays R der Größe $n \times n$ vorliegt, dass über die n Intervallnamen indiziert ist, d.h. die Constraints sind alle von der Form $A_i \; R_{i,j} \; A_j$. Nicht bekannte Relationen werden dementsprechend auf $\mathcal{R}$ gesetzt (für $A_i \; R_{i,i} \; A_i$ kann $R_{i,i}$ auch initial auf $\equiv$ gesetzt werden). Sobald in einem Eintrag die leere Menge hergeleitet wurde, kann der Algorithmus gestoppt werden, da der Constraint unerfüllbar ist. Wir erwähnen diesen möglichen Abbruch nicht explizit in den folgenden Algorithmen.

Der erste Algorithmus in Abbildung 4.2 wendet die Transitivitätsregel solange an, bis sich nichts mehr ändert (d.h. ein Fixpunkt erreicht ist). Der Algorithmus ist ähnlich zum Floyd-Warshall-Algorithmus zur Berechnung der transitiven Hülle einer Relation, hat jedoch (den notwendigen) zusätzlichen Fixpunkttest:

Offensichtlich ist der Algorithmus korrekt, da er solange rechnet, bis sich nichts mehr ändert (also der Fixpunkt erreicht ist). Die Laufzeit des Algorithmus ist im Worst-Case

Algorithmus **Allenscher Abschluss, Variante 1**

Eingabe: $(n \times n)$-Array R, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$

Algorithmus:

```
repeat
  change := False;
  for i := 1 to n do
    for j := 1 to n do
      for k := 1 to n do
        R' := R_{i,j} ∩ (R_{i,k} ∘ R_{k,j});
        if R_{i,j} ≠ R' then
          R_{i,j} := R;
          change := True;
        endif
      endfor
    endfor
  endfor
until change=False
```

Abbildung 4.2: Algorithmus 1 zur Berechnung des Allenschen Abschlusses

$O(n^5)$: Im schlimmsten Fall wird in den drei for-Schleifen nur ein einziger Eintrag $R_{i,j}$ geändert. Jeder Eintrag $R_{i,j}$ kann höchstens 13 Mal geändert werden (Verkleinerung der Relation $R_{i,j}$ für den Constraint $A_i\ R_{i,j}\ A_j$ jedesmal um ein Element). Das ergibt $O(n^2)$ Durchläufe der repeat-Schleife. Ein Durchlauf der repeat-Schleife verbraucht aufgrund der drei for-Schleifen $O(n^3)$ Zeit, da alle anderen Operationen als konstant angenommen werden können.

Ein besserer Algorithmus ist in Abbildung 4.3 dargestellt. Die Korrekheit dieses Algorithmus ergibt sich daraus, dass bei einer Änderung des Werts $R_{i,j}$ wieder alle Nachbarn, deren Wert evtl. neu berechnet werden muss, in die Queue eingefügt werden.

Die Laufzeit des Algorithmus ist im Worst-Case $O(n^3)$: Am Anfang enthält die Menge queue $O(n^3)$ Tripel. Jedes $R_{i,j}$ kann maximal 13 mal verändert werden. Bei einer Änderung an $R_{i,j}$ werden $2 * n$ Tripel in queue eingefügt. Da es nur n^2 viele $R_{i,j}$ gibt, werden insgesamt maximal $13 * 2 * n * n^2 = O(n^3)$ Tripel zu queue hinzugefügt. Also gibt es nicht mehr als $O(n^3)$ Durchläufe der while-Schleife, von denen maximal $O(n^2)$ Durchläufe $O(n)$ Laufzeit verbrauchen und die restlichen $O(n)$ in konstanter Laufzeit laufen. Das ergibt in der Summe eine Laufzeit von $O(n^3)$.

4.4.2 Korrektheit

Es gilt:

Satz 4.4.1. *Der Allensche Kalkül ist korrekt.*

Algorithmus **Allenscher Abschluss, Variante 2**

Eingabe: $(n \times n)$-Array R, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$

Algorithmus:
queue := $\{(i,k,j) \mid 1 \leq i \leq n, 1 \leq k \leq n, 1 \leq j \leq n\}$;

while queue $\neq \emptyset$ **do**

  Wähle und entferne Tripel (i,k,j) aus queue;

  $R' := R_{i,j} \cap (R_{i,k} \circ R_{k,j})$;

  if $R_{i,j} \neq R'$ **then**

    $R_{i,j} := R'$;

    queue := queue ++ $\{(i,j,m) \mid 1 \leq m \leq n\}$ ++ $\{(m,i,j) \mid 1 \leq m \leq n\}$

  endif

endwhile

Abbildung 4.3: Algorithmus 2 zur Berechnung des Allenschen Abschlusses

Beweis. Die Korrekheit muss man mittels der Semantik nachweisen. Für die aussagenlogischen Umformungen ist dies offensichtlich.

- $A\ R_1\ B \wedge A\ R_2\ B$ ist äquivalent zu $A\ (R_1 \cap R_2)\ B$:
  Sei $R_1 = \{r_1, \ldots, r_k\}$, $R_2 = \{r'_1, \ldots, r'_{k'}\}$. Dann ist $A\ R_1\ B \wedge A\ R_2\ B$ äquivalent zu $\bigvee\{(A\ r_i\ B) \wedge (A\ r'_{i'} B) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Wenn man nun die Eigenschaft hinzunimmt, dass alle Basisrelationen echt disjunkt sind, so sieht man, dass die Formel äquivalent zu $\bigvee\{(A\ r_i\ B) \wedge (A\ r'_{i'} B) \mid 1 \leq i \leq k, 1 \leq i' \leq k', r_i = r'_{i'}\}$ ist, was gerade genau $A\ (R_1 \cap R_2)\ B$ entspricht.
- $A\ \emptyset\ B$ und 0 sind äquivalent, da für jede Interpretation I per Definition $I(A \emptyset B) = 0$ gilt.
- $A\ \mathcal{R}\ B$ ist äquivalent zu 1, da jede Interpretation I, die Intervalle $I(A)$ und $I(B)$ interpretiert, und $\mathcal{R}$ alle möglichen Lagen abdeckt.
- $A\ R\ A$ ist äquivalent zu 0, wenn $\equiv \notin R$: Jede Interpretation bildet $I(A)$ eindeutig auf ein Intervall ab.
- $A\ R\ A$ ist äquivalent zu 1, wenn $\equiv \in R$: Jede Interpretation bildet $I(A)$ eindeutig auf ein Intervall ab.
- $A\ R\ B$ ist äquivalent zu $B\ \breve{R}\ A$: Das haben wir bereits gezeigt (Satz 4.3.4).
- Für die Transitivitätsregel, die $A\ R_1\ B \wedge B\ R_2\ C$ durch $A\ R_1\ B \wedge B\ R_2\ C \wedge A\ R_1 \circ R_2\ C$ ersetzt, kann man zunächst die Fälle untersuchen, in denen R_1 und R_2 genau eine Basisrelation enthalten, d.h. man muss die Korrekheit der Einträge in

der 13×13-Matrix überprüfen (anhand der durch die Interpretation gegebenen Lagen der Intervalle auf der reellen Achse). Für mehrelementige Mengen: Sei $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C$ gegeben. Dies ist eine Abkürzung für $(A\ r_1\ B \vee \ldots \vee A\ r_k\ B) \wedge (B\ r'_1\ C \vee \ldots \vee B\ r'_{k'}\ C)$. Ausmultiplizieren ergibt gerade $\bigvee\{(A\ r_i\ B \wedge B\ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Nun können wir für jede Konjunktion $A\ r_i\ B \wedge B\ r'_{i'}\ C$ die bereits als korrekt gezeigte Ersetzung durchführen. Das ergibt $\bigvee\{(A\ r_i\ B \wedge B\ r'_{i'}\ C \wedge A\ r_i \circ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Nach Umklammern und dem Rückgängigmachen des Ausmultiplizierens erhalten wir: $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C \wedge \bigvee\{(A\ r_i \circ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Die letzte Veroderung entspricht genau der Definition von $\circ$ auf Mengen, d.h. wir dürfen umformen zu $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C \wedge A\ \{r_1, \ldots, r_k\} \circ \{r'_1, \ldots, r'_{k'}\}\ C$.

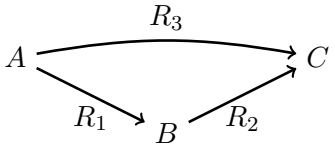
□

4.4.3 Partielle Vollständigkeit

Der Allensche Kalkül ist vollständig in eingeschränktem Sinn:

Satz 4.4.2. *Sind zwei Relationen $A_1\ R_1\ B_1\ \wedge\ A_2\ R_2\ B_2$ gegeben, dann ermittelt der Allensche Kalkül alle Relationsbeziehungen, die daraus folgen.*

Man kann einen Allenschen Constraint als sogenanntes Constraint-Netzwerk darstellen: Dabei werden die Intervallnamen als Knoten verwendet (pro Intervallname gibt es genau einen Knoten mit dem Intervallnamen als Markierung). Die Beziehungen zwischen den Intervallen werden als gerichtete Kanten mit Markierung der entsprechenden Basisrelationen dargestellt. Satz 4.4.2 sagt dann gerade aus, dass das Constraint-Netzwerk zum Allenschen Abschluss eines Constraints *Pfad-konsistent* ist, was gerade bedeutet, dass für je drei Knoten A, B, C mit Kanten $A\ R_1\ B$, $B\ R_2\ C$ und $A\ R_3\ C$ des Netzwerks stets gilt: Wenn es eine Interpretation I gibt, die $A\ R_3\ C$ erfüllt, dann können wir stets auch die anderen Kanten $A\ R_1\ B$ und $B\ R_2\ C$ erfüllen, ohne die Interpretation für A und C zu verändern, oder formaler: Wenn $I(A) = [A_a, A_e], I(C) = [CA, CZ]$, so dass $I(A\ R_3\ C) = 1$, dann gibt es eine Interpretation I' mit $I'(A) = I(A), I'(C) = I(C)$ und $I'(A\ R_1\ B) = 1, I'(B\ R_2\ C) = 1$. Als Teilausschnitt des Constraint-Netzwerks dargestellt:



Das der Allensche Abschluss die Pfadkonsistenz erfüllt ist klar, denn die Berechnung sichert gerade $R_3 \subseteq R_1 \circ R_2$ zu.

4.5 Unvollständigkeiten des Allenschen Kalküls.

Leider gilt:

Theorem 4.5.1. *Der Allensche Kalkül ist nicht herleitungs-vollständig.*

Beweis. Es genügt ein Gegenbeispiel anzugeben. Man benötigt vier Intervallkonstanten A, B, C, D. Die Beziehungen sind:

$$D \{\mathtt{o}\} B, D \{\mathtt{s},\mathtt{m}\} C, D \{\mathtt{s},\mathtt{m}\} A, A \{\mathtt{d},\breve{\mathtt{d}}\} B, C \{\mathtt{d},\breve{\mathtt{d}}\} B$$

Mit der kompositionellen Vervollständigung (nach Umdrehen) kann man aus $C \{\breve{\mathtt{s}},\breve{\mathtt{m}}\} D, D \{\mathtt{s},\mathtt{m}\} A$ die Relation $C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$ schließen. Aus $A \{\mathtt{d},\breve{\mathtt{d}}\} B, B \{\mathtt{d},\breve{\mathtt{d}}\} C$ kann man nur schließen, dass alles möglich ist. Der Schnitt ergibt somit $C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$. Oder anders ausgedrückt: Für das Allensche Constraint:

$$D \{\mathtt{o}\} B \wedge D \{\mathtt{s},\mathtt{m}\} C \wedge D \{\mathtt{s},\mathtt{m}\} A \wedge A \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{d},\breve{\mathtt{d}}\} B$$

ist der Allensche Abschluss:

$$D \{\mathtt{o}\} B \wedge D \{\mathtt{s},\mathtt{m}\} C \wedge D \{\mathtt{s},\mathtt{m}\} A \wedge A \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$$

Betrachtet man aber genauer die Möglichkeiten auf der reellen Achse, dann sieht man, dass in diesem speziellen Fall die Relation $C \{\mathtt{f},\breve{\mathtt{f}},\mathtt{o},\breve{\mathtt{o}}\} A$ nicht möglich ist. Die Lage von B zu D ist eindeutig. Wir betrachten daher die Möglichkeiten wie A zu D und C zu D liegen können. Da ergibt vier Fälle: (1) $D\ \mathtt{s}\ C$ und $D\ \mathtt{s}\ A$; (2) $D\ \mathtt{m}\ C$ und $D\ \mathtt{s}\ A$; (3) $D\ \mathtt{s}\ C$ und $D\ \mathtt{m}\ A$; (4) $D\ \mathtt{m}\ C$ und $D\ \mathtt{m}\ A$. Dabei muss man noch die Bedingungen zwischen A zu B und C zu B beachten (diese werden jedesmal eindeutig).

Die vier Fälle sind in Abbildung 4.4 als Bilder dargestellt sind:

Die Bilder zeigen: In Fall (1) ist $C \{\mathtt{s},\breve{\mathtt{s}}\ \equiv\} A$ möglich, in Fall (2) nur $C\ \mathtt{d}\ A$, in Fall (3) $C\ \breve{\mathtt{d}}\ A$ und in Fall 4 ist nur $C \{\mathtt{s},\breve{\mathtt{s}},\equiv\} A$ möglich. D.h. die Relation $C \{\mathtt{f},\breve{\mathtt{f}},\mathtt{o},\breve{\mathtt{o}}\} A$ ist niemals möglich, obwohl der Allensche Abschluss diese als Möglichkeiten herleitet.

Damit ist die Allensche Vervollständigung bezüglich der Semantik einer reellen Zeitachse nicht vollständig. Es können nicht alle Relationen exakt hergeleitet werden. □

Der Allensche Kalkül kann damit auch nicht immer erkennen, ob eine eingegebene Menge von Relationen inkonsistent ist.

Satz 4.5.2. *Der Allensche Kalkül ist nicht widerlegungsvollständig.*

Beweis. Ein Gegenbeispiel kann man durch Abwandlung des Gegenbeispiels zur Vollständigkeit gewinnen, man fügt die Relation $A \{\mathtt{f},\breve{\mathtt{f}}\} C$ hinzu, d.h. man erhält das Constraintnetzwerk:

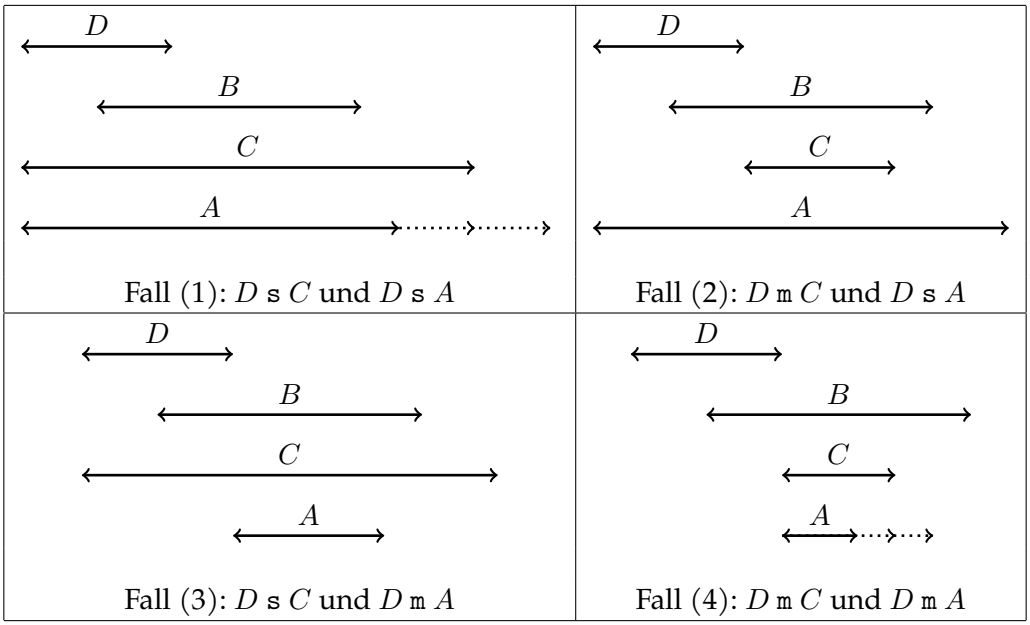
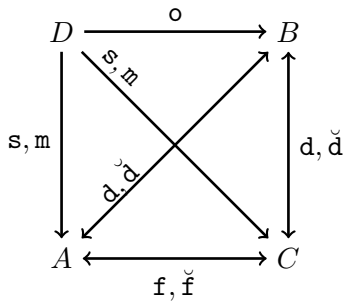


Abbildung 4.4: Vier Fälle zur Herleitungs-Unvollständigkeit



Man sieht, dass die Allensche Vervollständigung hier keine weiteren Erkenntnisse bringt: Man muss 12 Möglichkeiten der Komposition prüfen. Jede ergibt nur allgemeinere Bedingungen als die schon vorhandenen. Somit kann der Kalkül nichts Neues schließen. Insbesondere findet er keinen Widerspruch, obwohl die Menge der Constraints widersprüchlich ist. □

Bemerkung 4.5.3. *Man kann sich bei Anfragen an den Allen-Kalkül nur darauf verlassen, dass die Vervollständigung richtig ist, aber evtl. nicht optimal. Wenn man die Frage stellt: Ist das Constraint C widersprüchlich, so kann man sich nur bei „JA“ (d.h. Herleitung der 0) auf die Antwort verlassen, aber micht bei „NEIN“. Umgekehrt heisst das auch, dass man sich bei der Frage nach der Erfüllbarkeit dementsprechend nur auf die Antwort „NEIN“ verlassen kann.*

4.6 Einge Analysen zur Implementierungen

Ein Allensches Constraint kann man versuchen vollständig auf Erfüllbarkeit bzw Unerfülllbarkeit zu testen, indem man Fallunterscheidungen macht.

Wir schauen genauer auf Allensche Constraints, wobei wir annehmen, dass diese normalisiert sind, d.h. zwischen zwei Intervallkonstanten gibt es nur ein Constraint.

Definition 4.6.1. *Ein Allensches Constraint nennt man* eindeutig, *wenn für alle Paare A, B von Intervallkonstanten gilt: Das Constraint enthält genau eine Beziehung $A\ r\ B$, wobei r eine der dreizehn Basisrelationen ist.*

Es gilt:

Satz 4.6.2. *Der Allensche Abschluss eines eindeutigen Allenschen Constraints F ist entweder 0, oder wiederum F.*

Beweis. Es reicht zu zeigen, dass die Anwendung der Transitivitätsregel nur eindeutige Relationen erzeugen kann (nach Anwendung weiterer Regeln): Seien $A\ r_1\ B$, $B\ r_2\ C$ Teilformeln des eindeutigen Allenschen Constraint, dann muss es bereits eine Beziehung $A\ r_3\ C$ geben (sonst wäre das Constraint nicht eindeutig). Die Transitivitätsregel ergibt:

$$A\ r_1\ B \wedge B\ r_2\ C \wedge A\ r_3\ C \rightarrow A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2)\ C \wedge A\ r_3\ C.$$

Anschließend können wir eine weitere Regel anwenden:

$$A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2)\ C \wedge A\ r_3\ C \rightarrow A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2) \cap \{r_3\}\ C$$

. Da der Schnitt $(r_1\ \circ\ r_2) \cap \{r_3\}$ entweder r_3 oder $\emptyset$ (Unerfüllbarkeit) ergeben muss, folgt die Aussage. □

Definition 4.6.3. *Zu jedem Allenschen Constraint C kann man die Menge aller zugehörigen eindeutigen Allenschen Constraints D definieren, wobei gelten muss: Wenn $A\ r\ B$ in D vorkommt und $A\ R\ B$ in C, dann gilt $r \in R$.*

Lemma 4.6.4. *Ein Allensches Constraint ist erfüllbar, gdw., es ein zugehöriges eindeutiges Constraint gibt, das erfüllbar ist.*

Beweis. Das ist klar, da die Einzelconstraints ja Disjunktionen sind, und bei einer erfüllenden Belegung genau eine der Relationen in der Relationsmenge gilt. □

Satz 4.6.5. *Ein eindeutiges Allensches Constraint ist erfüllbar, gdw. der Allensche Kalkül bei Vervollständigung das Constraint nicht verändert, d.h. wenn es ein Fixpunkt ist.*

Beweis. Wenn der Kalkül einen Widerspruch entdeckt, dann ist das Constraint natürlich widersprüchlich. Für den Fall, dass der Kalkül keinen Widerspruch entdeckt, kann man

Algorithmus **Erfüllbarkeitstest für konjunktive Allensche Constraints**

Eingabe: $(n \times n)$-Array R, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$
Ausgabe: `True` (Widerspruch) oder `False` (erfüllbar)

function AllenSAT(R):
  R' := AllenAbschluss(R);
  if $\exists R'_{i,j}$ mit $R'_{i,j} = \emptyset$ **then return** `True` **endif**;// Widerspruch
  if $\forall R'_{i,j}$ gilt: $|R'_{i,j}| = 1$ **then return** `False` // eindeutig und erfüllbar
  else
    wähle $R'_{i,j}$ mit $R'_{i,j} = \{r_1, r_2, \ldots\}$;
    $R^l := R'$; $R^l_{i,j} := \{r_1\}$;// kopiere R' und setze (i,j) auf r_1
    $R^r := R'$; $R^r_{i,j} := R'_{i,j} \setminus \{r_1\}$;// kopiere R' und setze (i,j) auf $r_2, \ldots$
    return (AllenSAT(R^l) $\wedge$ AllenSAT(R^r));
  endif

Abbildung 4.5: Vollständiger Erfüllbarkeitstest für konjunktive Allen-Constraints

zeigen, dass eine totale Ordnung der Intervallenden möglich ist. Einen entsprechenden Beweis findet man bspw. in (Valdés-Pérez, 1987) □

Daraus ergibt sich ein (im worst-case exponentieller) Algorithmus, der die Erfüllbarkeit eines Allenschen Constraints testet:

- Sei C ein Allensches Constraint.
- Sei D die Menge aller eindeutigen Allenschen Constraints zu C.
- Berechne den Allenschen Abschluss jedes Constraints $C' \in D$.
- Wenn dabei kein Widerspruch auftritt, ist C erfüllbar, anderenfalls C widersprüchlich.

Hierbei kann man noch etwas geschickter vorgehen: Sobald ein Modell gefunden wurde, kann man komplett aufhören (die Erfüllbarkeit gilt) und sobald ein Widerspruch gefunden wurde braucht man die dazugehörigen eindeutigen Allen-Constraints nicht weiter zu betrachten (da der Allensche Abschluss auch auf mehrdeutigen Constraints korrekt ist). Abbildung 4.5 zeigt den so optimierten Algorithmus.

Die durchschnittliche Verzweigungsrate dieses Algorithmus ist 6.5, da man im schlimmsten Fall pro Beziehung $A\ R\ B$ dreizehn Fallunterscheidungen machen muss, und im besten Fall R leer ist.

4.7 Komplexität

Komplexität des Problems und des Algorithmus sind zu unterscheiden:

Die Komplexität des Problems ist schlechter, denn es gilt:

Satz 4.7.1. *Die Erfüllbarkeit einer Konjunktion von Allenschen Relationen ist $\mathcal{NP}$-vollständig, bzw. die Erfüllbarkeit eines konjunktiven Allenschen Constraints ist $\mathcal{NP}$-vollständig.*

Beweis. Es ist leicht einzusehen, dass die Erfüllbarkeit eines Allenschen Constraints in $\mathcal{NP}$ ist. Man muss nur ein lineare Reihenfolge aller Werte X_a, X_e angeben (bzw. raten), wobei X eine Intervallkonstante ist und X_a der Anfang und X_e das Ende. Der Test, ob diese lineare Reihenfolge das Constraint erfüllt, ist dann polynomiell.

Zum Beweis der $\mathcal{NP}$-härte nehme das $\mathcal{NP}$-vollständige Problem der Drei-Färbbarkeit eines Graphen:

> Gegeben ein Graph mit Knotenmenge N und Kantenmenge K. Gibt es eine Färbung der Knoten mit drei Farben, so dass benachbarte Knoten verschiedene Farbe haben?

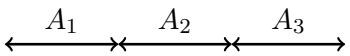
Sei eine Instanz gegeben, d.h. ein Graph (N, K). Dann erzeuge ein Allensches Constraint:

Seien A_1, A_2, A_3 Intervallkonstanten. Für jeden Knoten $v_i \in N$ erzeuge eine Intervallkonstante. B_i. Erzeuge die folgenden Allenschen Relationen (d.h. verunde sie zu einem konjunktiven Allenschen Constraint):

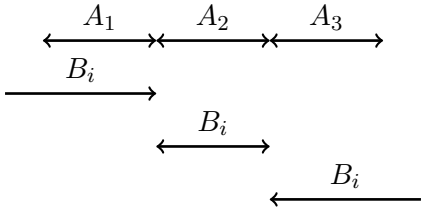
$$\begin{array}{lllll} & A_1 & \mathtt{m} & A_2 & \\ & A_2 & \mathtt{m} & A_3 & \\ \forall v_i \in N: & B_i & \{\mathtt{m}, \equiv, \breve{\mathtt{m}}\} & A_2 & \\ \forall (v_i, v_j) \in K: & B_i & \{\mathtt{m}, \breve{\mathtt{m}}, \prec, \succ\} & B_j & \end{array}$$

Die Idee dabei ist gerade: Die Lage jedes B_i zu A_2 bestimmt die Farbe des Knotens v_i. Das kann man sich klar machen, indem man die Constraints als Bild veranschaulicht:

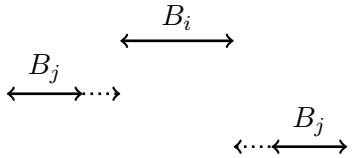
Die ersten beiden Constraints ergeben gerade



Die Constraints für die Knoten ergeben, dass B_i entweder mit A_2 übereinstimmt ($B_i \equiv A_2$), oder direkt links davon ($B_i \ \mathtt{m} \ A_2$, überschneidend mit A_1) oder direkt rechts davon ($B_i \ \breve{\mathtt{m}} \ A_2$, überschneidend mit A_3) liegt:



Die Constraints für die Kanten ergeben, dass die Intervalle B_i, B_j keine Überschneidungen haben:



Nun ist zu zeigen: Der Graph ist dreifärbbar, gdw. die Allenschen Relationen erfüllbar sind. Hierfür reicht die Zuordnung:

- v_i hat Farbe 2 gdw. $B_i \equiv A_2$
- v_i hat Farbe 1 gdw. $B_i \; \mathtt{m} \; A_2$
- v_i hat Farbe 3 gdw. $B_i \; \breve{\mathtt{m}} \; A_2$

Diese Übersetzung kann in polynomieller Zeit abhängig von der Größe des Graphen gemacht werden. Diese Übersetzung zeigt die $\mathcal{NP}$-Härte der Allenschen Constraints.

Damit ist die Erfüllbarkeit von Allenschen Constraints $\mathcal{NP}$-vollständig. □

Als Schlussfolgerung kann man sagen, dass ein vollständiger Algorithmus nach aktuellem Wissenstand mindestens exponentiell sein muss, während der polynomiell Allensche Vervollständigungs-Algorithmus unvollständig sein muss.

4.8 Eine polynomielle und vollständige Variante

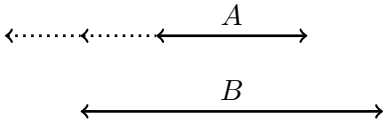
Man kann Varianten und Einschränkungen der Allenschen Constraints suchen, die einen sowohl vollständigen als auch polynomiellen Erfüllbarkeitstest erlauben. Eine solche Variante haben wir bereits gesehen: Für eindeutige Allensche Constraints ist der Allensche Kalkül vollständig.

Eine Variante ist die Menge der Relationen, die man durch eine Konjunktion von atomaren Ungleichungen der Form $x < y$ oder $x = y$ repräsentieren kann, wobei x, y Endpunkte von Intervallen sind. Hier kommt es darauf an, dass man keine Disjunktionen hat, die eine Fallunterscheidung erzwingen.

Einen passenden Satz von Relationen gibt es:

Alle Basisrelationen, $\{\mathtt{d}, \mathtt{o}, \mathtt{s}\}$, und $\{\breve{\mathtt{o}}, \mathtt{f}, \mathtt{d}\}$ und deren Konverse. d.h. $\{\breve{\mathtt{d}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}\}$, und $\{\mathtt{o}, \breve{\mathtt{f}}, \breve{\mathtt{d}}\}$. Beachte, dass man noch weitere dazu nehmen kann.

Die Relation $A\{\mathtt{d}, \mathtt{o}, \mathtt{s}\}B$ z.B. kann man als Ungleichung über den Endpunkten ausdrücken:



Wenn $A = [A_a, A_e], B = [B_a, B_e]$, dann entspricht obige Relation der Konjunktion der Ungleichungen: $A_a < A_e, B_a < B_e, A_e < B_e, B_a < A_e$

Eine Konjunktion von solchen Relationsbeziehungen ergibt in der Summe ein Constraint, das eine Konjunktion von Ungleichungen der Endpunkte von Intervallen ist. Diese Konjunktion kann man mittels transitivem Abschluss über die Endpunkte von Intervallen vervollständigen. Dies geht in polynomieller Zeit. Wenn man eine Relation der Form $X < X$ erzeugt hat, dann ist das Constraint unerfüllbar. Ansonsten ist er erfüllbar, denn die Endpunkte kann man mit topologischem Sortieren in eine lineare Reihenfolge bringen.

Insgesamt hat man gezeigt, dass der so definierte Kalkül auf den eingeschränkten Constraints vollständig und polynomiell ist. Es gilt sogar, dass der Allensche Kalkül selbst auf diesen Constraints vollständig ist (siehe z.B. (Nebel & Bürckert, 1995)).

Der Hintergrund der speziellen Klasse von Allenschen Constraints ist, dass sich diese Klasse als Grund-Hornklauseln darstellen lassen. Hornklauseln sind Klauseln, die maximal ein positives Literal haben. Hierbei ist ein Literal positiv, wenn es ein unnegiertes Atom ist.

Für aussagenlogische Hornklauselmengen und auch für Grund-Hornklauselmengen gilt, dass deren Erfüllbarkeit in polynomieller Zeit testbar ist.

Die notwendige Menge von Axiomen und Fakten ist in dem eingeschränkten Fall eine Hornklauselmenge,:
Man hat Fakten in der Form $a < b$ und $c = d$, wobei a, b, c, d unbekannte Konstanten sind. Es gibt auch Hornklauseln, die von der Symmetrie und Transitivität stammen:

$$\begin{array}{lcl} x < y \wedge y < z & \Rightarrow & x < z \\ x = y \wedge y = z & \Rightarrow & x = z \\ x = y & \Rightarrow & y = x \\ x < y \wedge y = z & \Rightarrow & x < z \\ x = y \wedge y < z & \Rightarrow & x < z \end{array}$$

Tatsächlich kann man weitere Allensche Constraints zulassen, und behält die Vollständigkeit des Allen-Kalküls: Dies ist genau die Klasse der Constraints deren Übersetzung in Constraints über Endpunkten Hornklauseln ausschließlich mit Literalen $a \leq b$, $a = b$ und $\neg(a = b)$ erzeugt (siehe (Nebel & Bürckert, 1995)). Ein interessanter Aspekt dabei ist, dass von den $2^{13} = 8192$ möglichen Relationen, 868 Relationen diese Eigenschaft aufweisen.

Für den vollständigen (aber exponentiellen) Algorithmus für beliebige Allensche Constraints, kann man dies ausnutzen, indem man nicht in eindeutige Relationen, sondern in Relationen entsprechend dieser Klasse Fallunterscheidungen durchführt, die durchschnittliche Verzweigungsrate kann dadurch von 6,5 auf 2,533 gesenkt werden (Nebel, 1997).

5 Modallogik (aussagenlogisch)

5.1 Einführung

Die Modallogik erlaubt Formulierung und Repräsentation von Aussagen, die über die Aussagenlogik hinausgehen: Beispielsweise kann man Aussagen über Aussagen machen: „bald wird es regnen“ oder „möglicherweise ist die Erde eine Kugel“ . Damit kann man eine Form des Meta-Schließens in die Aussagenlogik und in die Logik erster Ordnung einbringen. Die Aussage „bald wird es regnen“ kann man sich zusammengesetzt vorstellen aus einem Modaloperator „bald wird gelten“ und der Aussage „es regnet“.

Es gibt verschiedene Interpretationen und Anwendungen dieser modalen Erweiterungen von Logik in ganz unterschiedlichen Bereichen, die wir kurz in Beispielen vorstellen wollen. Wie alle Modellierungen, haben diese logischen Modellierungen auch ihre Schwächen. Diese wollen wir eher am Rande diskutieren, und uns mehr auf das Verständnis der Formalisierung konzentrieren. Zudem kann man die möglichen Schwächen bei praktischen Anwendung erst herausfinden, wenn man diese Formalisierung verstanden hat.

Die *philosophische Logik* verwendet und betrachtet ebenfalls modale Logiken, wobei hier die Diskussion im Vordergrund steht, welche Variante für welchen Zweck adäquat ist und die richtigen Schlüsse zulässt.

Beispiele:

1. Modellierung der Zeit. Eine bekannte Modellierung ist die diskrete Zeit, diese hat keine Zeitmessung, sondern kann nur „vorher“, „nachher“, „immer“, „manchmal“ usw. formulieren.
   $\Box F$: F gilt immer in der Zukunft
   $\Diamond F$: F gilt irgendwann in der Zukunft.
   Intuitiv gilt in dieser Modellierung : $\Box\Box F \iff \Box F$

   Man kann deterministische oder nicht-deterministische Prozesse zeitlich modellieren. Deterministischer Ablauf entspricht einer linearen Ordnung der möglichen Zustände auf der Zeitachse, ein nichtdeterministischer Ablauf ergibt verzweigende Zustände auf der Zeitachse, d.h. einen Baum bzw einen azyklischen gerichteten Graphen.

Bei linearer Zeit bzw. deterministischen Prozessen sollte gelten:

$$(\Diamond\Box F) \implies (\Box\Diamond F)$$

D.h. irgendwann gilt F immer impliziert: es gilt für jeden zukünftigen Zeitpunkt, dass irgendwann danach F gilt.

Dies gilt bei verzweigender Zeit nicht mehr.
Es gibt eine Logik CTL* (computation tree logic), die eine Verallgemeinerung einer Modallogik zu einer verzweigenden Zeit entspricht, und die z.B. in der Hardwareverifikation verwendet wird. Diese Logik und ihre Spezialisierungen dienen in diesem Bereich als logische Basis von verschiedenen „model checking“ -Verfahren.

2. Logik des Erlaubten und Verbotenen (Normative Logik, Deontic Logic)
   Aus Wikipedia: „Deontische Logik ist der Bereich der Logik, der die logischen Verhältnisse von Begriffen, die sich auf das Sollen beziehen, untersucht. Begriffe, die sich auf das Sollen beziehen, sind Gebot, Verbot, Erlaubnis und andere mehr.“
   $\Box F$: F ist geboten; F muss gelten; bzw. F ist obligatorisch.
   $\Diamond F$: F ist erlaubt.
   In dieser Logik sollte gelten, dass obligatorisches auch erlaubt sein sollte. D.h.

$$\Box F \implies \Diamond F$$

3. Logik des Wissens (epistemische Logik)
   $\Box F$: F ist bekannt.
   $\Diamond F$: F ist glaubhaft.
   In dieser Logik sollte gelten, dass bekannte Fakten richtig sind: D.h.

$$\Box F \implies F$$

   Dies wäre in einer Zeitlogik mit $\Box$ = nächster Zustand nicht immer richtig, denn das würde bedeuten: Wenn F in der Zukunft immer gilt, dann auch heute.

4. Logik des Beweisens
   $\Box F$: F ist beweisbar
   $\Diamond F$: F ist nicht widerlegbar

   In dieser Logik sollte gelten, dass bekannte Fakten richtig sind:
   D.h.

$$\Box F \implies F$$

   .

5. Variante einer Logik des Wissens (autoepistemische Logik)

   Diese modelliert den Glauben an gewisse Fakten.

   $\Box F$: F wird geglaubt.
   $\Diamond F$: F ist möglich ($\neg F$ wird nicht geglaubt).

   Die Axiome, die in einem Artikel verwendet werden (schwaches S5, s.u.) sind:

$$\begin{array}{lll} \Box(F \implies G) & \implies & (\Box F \implies \Box G) \\ \Box F & \implies & \Box\Box F \\ \neg\Box F & \implies & \Box(\neg\Box F) \end{array}$$

   Damit kann man z.B. eine sogenannte „nichtmonotone“ Logik modellieren, die folgende Schlussweise verwenden: Wenn ich etwas nicht glaube, dann gilt diese Tatsache auch nicht:

$$\neg\Box F \implies \neg F$$

6. Erweiterungen sind parametrisierte Modaloperatoren (oder Multimodallogiken), die dann im Extremfall zur *dynamischen Logik* führen. Diese erlaubt z.B. $\boxed{\text{P}}F$ mit der Bedeutung: nach (jeder) Ausführung des Programms P gilt, falls das Programm terminiert, die Formel F.

7. parametrisierte Modaloperatoren in einer Logik des Wissens mit mehreren Akteuren: $\boxed{\text{A}}$ F bedeutet A weiß, dass F gilt, usw. Damit kann man auch formulieren: $\boxed{\text{A}}\,(\neg\boxed{\text{B}}\,F)$: A weiß, dass B nicht weiß, dass F wahr ist.

   Oder: ich weiß etwas was Du nicht weißt: $(\boxed{\text{ich}}F) \wedge (\neg\boxed{\text{du}}F)$. Aber man kann nicht schreiben: $\exists F : \ldots$, da man über Formeln nicht quantifizieren kann.

5.2 Beispiel: Wise man puzzle

Drei Weise sitzen sich gegenüber, so dass jeder den anderen von vorne sieht. Jedem wird ein roter oder blauer Punkt auf die Stirn gemalt. Mindestens einer der Punkte ist rot, was den Weisen bekannt ist. Jeder sieht die Stirn der anderen aber nicht seine eigene.
Nach einer Zeit des Überlegens sagt der erste Weise: „ Ich weiß nicht, welche Farbe der Punkt auf meiner Stirn hat.“Nach weiterem Überlegen sagt der zweite Weise: „ Ich weiß auch nicht, welche Farbe der Punkt auf meiner Stirn hat.“Kurz danach sagt der dritte Weise: „ Jetzt weiß ich, welche Farbe der Punkt auf meiner Stirn hat“. Welche?

Der Punkt ist rot. Wie kann man argumentieren?

Der erste Weise kann nur RR oder RB sehen, sonst wüßte er seine Farbe.
Der zweite Weise kann diesen Schluss nachvollziehen, aber auch er sieht RR oder RB, und kann somit nichts schließen. Der dritte Weise kann jetzt alle Wissenstände nutzen:

- Wenn er BB sieht, dann weiß er dass er R hat.
- Wenn er RB sieht, und er hat selbst B, dann muss ein anderer Weise BB gesehen haben, und somit seine Farbe wissen. Also hat er auch in diesem Fall R.
- Wenn er RR sieht, und selbst B hat, dann hätte aber spätestens der zweite Weise schon sagen können, was er selbst hat, also muss er R haben. Der Grund ist: Wenn der dritte Weise B hat dann kennt der erste Weise seine eigene Farbe nicht. Aber der zweite kann dann schließen, dass er deswegen selbst nicht B haben kann.
  Also weiß der dritte Weise, dasser R hat. Da das in allen Fällen so war.

Das kann man mit Modallogik formal modellieren, siehe Beispiel 5.3.2

5.3 Syntax der aussagenlogischen Modallogik

Die Basis ist die Aussagenlogik. Es gibt zwei zusätzliche einstellige Modal-Operatoren, die in Präfixschreibweise verwendet werden: $\Box$ und $\Diamond$, die man auf Aussagen anwenden kann, aber auch schachteln kann, man kann z.B. die Formel $\Box((\Diamond X) \vee Y)$ bilden. Ansonsten ist alles wie in der zweiwertigen Aussagenlogik. Diese Modal-Operatoren sind jedoch semantisch gesehen grundsätzlich verschieden von Junktoren, da sie nicht wahrheitsfunktional sind. D.h. man kann keine Wahrheitstabelle angeben. [1]

5.3.1 Kripke-Semantik

Da man über verschiedene Belegungen (Interpretationen) der Wahrheitswerte von Variablen spricht, (z.B. jetzt und in der Zukunft) führt man explizit eine Entsprechung ein. Man spricht von „möglichen Welten“ (Zuständen). Die (nichtleere) Menge dieser Welten nennen wir W. Ferner sind die Welten miteinander (evtl.) verbunden mittels einer Erreichbarkeitsrelation R. Wenn $w_1\ R\ w_2$ für zwei Welten w_1, w_2 gilt, dann sagen wir, w_2 ist von w_1 aus erreichbar.
Das Paar (W, R) aus Menge W der Welten und binärer (Erreichbarkeits-) Relation R nennen wir Kripke-Rahmen (Kripke-Frame). Wenn jeder Welt eine (klassische) Interpretation zugeordnet wird, dann bezeichnen wir dies mit (W, R, I), wobei I jeder Welt eine Interpretation zuordnet. Das Tripel $K = (W, R, I)$ nennt man (*aussagenlogische*) *Kripke-Struktur*.

Um die Gültigkeit von modallogischen Aussagen F in einer Kripke-Struktur $K = (W, R, I)$ zu definieren, benötigen wir zuerst den semantischen Begriff

[1] Das wurde zwar teilweise mit mehrwertigen Logiken versucht, ist aber im wesentlichen gescheitert.

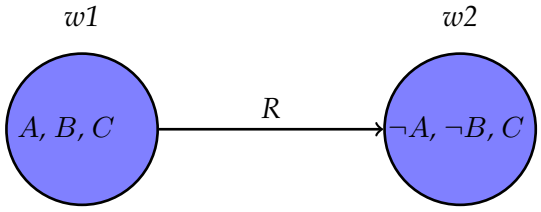
„ist gültig in einer Welt w“

Definition 5.3.1. *Sei $w \in W$ eine der möglichen Welten. Dann definieren wir die Gültigkeit $\models$ wie folgt, wobei wir statt $I(w) \models A$ einfach $w \models A$ schreiben, wobei A eine Formel ist.*

$$
\begin{array}{lll}
w \models 1 & & \\
w \not\models 0 & & \\
w \models A & \textit{gdw.} & I(w)(A) \textit{ für eine Variable } A \\
w \models F \vee G & \textit{gdw.} & w \models F \textit{ oder } w \models G \\
w \models F \wedge G & \textit{gdw.} & w \models F \textit{ und } w \models G \\
w \models F \implies G & \textit{gdw.} & \textit{nicht } (w \models F) \textit{ oder } w \models G \\
w \models \neg F & \textit{gdw.} & \textit{nicht } w \models F \\
w \models \Box F & \textit{gdw.} & \forall w' : w \; R \; w' \implies w' \models F \\
w \models \Diamond F & \textit{gdw.} & \exists w' : w \; R \; w' \wedge w' \models F
\end{array}
$$

Eine Formel F heißt gültig in einer Kripke-Struktur *$K = (W, R, I)$, falls für alle $w \in W$ gilt: $w \models F$. Wir schreiben das als $K \models F$.*
Eine Formel F heißt Tautologie, *falls für alle Kripke-Strukturen $K = (W, R, I)$ gilt: $K \models F$.*
Eine Formel F heißt erfüllbar, *falls es eine Kripke-Struktur $K = (W, R, I)$ gibt, so dass gilt: $K \models F$.*
Wir definieren auch Gültigkeit in einem Kripke-Rahmen: (W, R). Eine Formel F heißt gültig in einem Kripke-Rahmen (W, R), falls für alle I gilt: $(W, R, I) \models F$.

Beispiel 5.3.2. *Ein Beispiel zur Kripke Semantik, Welten und Interpretationen.*



In dieser Kripkestruktur sind die Wahrheitswerte der folgenden Beispielformeln in den Knoten / Welten $w1, w2$ notiert, was man leicht nachvollziehen kann anhand der Definitionen.

Formel	*w1*	*w2*
A	*1*	*0*
$\Box A$	*0*	*1*
$\Diamond A$	*0*	*0*
$\Diamond C$	*1*	*0*

Formel	*w1*	*w2*
$A \implies \Box A$	*0*	*1*
$C \implies \Box C$	*1*	*1*
$\Box\Box A$	*1*	*1*
$\Box 0$	*0*	*1*

Definition 5.3.3. *Sei $\mathcal{G}$ eine Menge von Formeln und F eine Formel. Dann ist F eine* **semantische Folgerung** *(logische Konsequenz) von $\mathcal{G}$, falls für alle Kripke-Strukturen K gilt:*
Wenn $K \models \mathcal{G}$, dann auch $K \models F$. Dies schreiben wir als $\mathcal{G} \models_K F$.

Beachte, dass das nicht pro Welt gemeint ist, sondern für den ganzen Rahmen.

Lemma 5.3.4. *Sei Wenn* $F_1 \wedge \ldots \wedge F_n \implies F$ *eine Tautologie ist, dann* $F_1, \ldots, F_n \models_K F$.

Beweis. Das ist eine Übung zur Verwendung der Semantik.
$F_1 \wedge \ldots \wedge F_n \implies F$ sei eine Tautologie. Sei K eine Kripke-Struktur, und $F_1 \wedge \ldots \wedge F_n$ gelten in K. Da dann in jeder Welt sowohl $F_1 \wedge \ldots \wedge F_n$ als auch $F_1 \wedge \ldots \wedge F_n \implies F$ gilt, muss F gelten, und zwar in jeder Welt. Damit gilt $F_1, \ldots, F_n \models_K F$. □

Satz 5.3.5. $F_1 \wedge \ldots \wedge F_n \implies F$ *ist eine Tautologie* $\implies F_1, \ldots, F_n \models F$
Aber die Umkehrung ist falsch.
D.h. das Deduktionstheorem gilt nur noch in einer Richtung.

Satz 5.3.6. *Die Umkehrung von Lemma 5.3.4 gilt nicht, da das Deduktionstheorem falsch ist in der Modallogik.*

Beweis. Sei P eine aussagenlogische Variable. Betrachte

$$P \models_K \Box P$$

Dies gilt: Wenn P in einer Kripkestruktur K gilt, dann gilt es in jeder Welt, also gilt auch $\Box P$ in jeder Kripkestruktur K.
Aber $P \implies \Box P$ ist keine Tautologie: Seien w_1, w_2 zwei Welten mit $w_1 \; R \; w_2$, so dass P in w_1 gilt, aber nicht in w_2. Dann ist $P \implies \Box P$ falsch in der Welt w_1. □

Damit hat man gezeigt, dass das Deduktionstheorem **nicht gilt**.

Man erhält modallogische Tautologien aus schon bekannten, wenn man die aussagenlogischen Variablen durch beliebige Formeln ersetzt:

Lemma 5.3.7. (*Ersetzung*).
Sei F eine modallogische Tautologie, sei A eine aussagenlogische Variable in F und sei G eine beliebige Formel. Dann ist $F[G/A]$ ebenfalls eine modallogische Tautologie.

Lemma 5.3.8. *Wenn F eine Tautologie in einem Rahmen ist, dann ist auch $\Box F$ eine Tautologie.*

Beispiel 5.3.9. *Die Aussage $\Box 0$ bzgl einer Welt w bedeutet, dass es von w aus keine erreichbare Welt mehr gibt, denn es darf in keiner Interpretation der Wert 0 wahr sein.*

Wenn $\Box\Box\Box 0$ in einer Welt w wahr ist, dann bedeutet das, dass man von w aus höchstens 2 R-Schritte machen kann.

Die Aussage $\Box 1$ ist eine Tautologie in allen Modallogiken.

5.3.2 Multimodallogiken, Schlüsse und die drei Weisen

5.3.2.1 Folgerungssystem

Da Folgerungen über die Prüfung von Tautologien nicht allgemein genug ist, brauchen wir etwas anderes: ein Herleitungssystem. Das ist aber in Logiken der Normalfall und war historisch der Standard.

Wir modellieren hier ein Folgerungssystem für *normale* Modallogiken, d.h. solche, deren Semantik mittels Kripkestrukturen begründet wird.

Gegeben eine Menge von Formeln $\mathcal{F} = \{F_1, \ldots, F_n\}$, dann können folgende weiteren Formeln hergeleitet werden:

1. Alle aus $\mathcal{F}$ aussagenlogisch herleitbaren Formeln.
2. Formeln, die durch Transformationen $\neg\Diamond F \iff \Box\neg F$ aus Formeln aus $\mathcal{F}$ entstehen.
3. (eingeschränkte) Notwendigkeitseinführung: Wenn $F \in \mathcal{F}$, und F wurde ohne Varaussetzungen hergeleitet (d.h. es ist eine Tautologie), dann kann man $\Box F$ herleiten.
4. (Axiom K): Wenn $\Box(F \implies G) \in \mathcal{F}$, dann kann man $\Box F \implies \Box G$ herleiten.
5. **Beweis durch Widerspruch**:
   Wenn man aus $\mathcal{F} \cup \{G\}$ einen Widerspruch herleiten kann, dann kann man $\neg G$ aus $\mathcal{F}$ herleiten.

Bemerkung 5.3.10. *Man kann mit diesen Mitteln alle Tautologien der Modallogik K (d.h. die Modallogik zum Axiom K) herleiten, wenn man mit der leeren Menge $\mathcal{F}$ startet. Den Nachweis oder weitere Hinweise dazu kann man in Büchern finden, z.B. (Bauer & Wirsing, 1987) Kapitel VI.*
Damit sind auch die folgenden Formeln (Tautologien) herleitbar:

- $\Box 1$
- $(\Box A) \wedge (\Box B) \iff (\Box(A \wedge B)$

Die Negationen der Tautologien sind dann Widersprüche, wie z.B $\Diamond 0$.

Widerspruch bedeutet, dass man aussagenlogisch 0 herleiten kann, oder: für eine Formel H, sowohl H also auch $\neg H$, das ergibt dann aber sofort 0, oder die Negation einer Tautologie.

Es gibt für verschiedene Varianten der Modallogik Folgerungssysteme wobei die Axiome variiert werden (oben z.B. K). Die Vollständigkeit unserer Folgerungsmethode gilt vermutlich, jedenfalls reicht es aus, nach Verallgemeinerung, um z.B. die richtigen Folgerungen im Beispiel der drei Weisen zu ziehen:

5.3.3 Beispiel der Drei Weisen

Wir modellieren das Problem der die Weisen in einer sogenannten Multi-Modallogik, die verschiedene Erreichbarkeitsrelationen hat. Für jede Erreichbarkeitsrelation gibt es eigene Operatoren: $\Box, \Diamond$ mit Parametern. Wir verwenden $\boxed{\text{A}}, \boxed{\text{B}}, \boxed{\text{C}}$ für die drei Weisen und deren aktuelles Wissen. Der Operator $\Box$ ohne Parameter könnte für globales Wissen verwendet werden, ist aber nicht nötig.

Wir nehmen an, dass man für gegebene Annahmen F alle $\Box$-Formeln ebenfalls herleiten kann: $\boxed{\text{A}}F, \boxed{\text{B}}F, \boxed{\text{C}}F$ herleiten kann. Schließen durch Widerspruch kann man gut einsetzen, da es zielgerichteter ist als direktes Herleiten.

Seien RA, RB, RC die drei aussagenlogischen Variablen zu „A hat roten Punkt“, entsprechend für B und C.

Die Formulierung besteht aus den Formeln:

$$
\begin{array}{lcl}
RA \vee RB \vee RC & & \\
\boxed{\text{A}}(RA \vee RB \vee RC) & & \\
\boxed{\text{B}}(RA \vee RB \vee RC) & & \\
\boxed{\text{C}}(RA \vee RB \vee RC) & & \\
RA & \Longrightarrow & (\boxed{\text{B}}RA) \wedge (\boxed{\text{C}}RA) \\
\neg RA & \Longrightarrow & (\boxed{\text{B}}\neg RA) \wedge (\boxed{\text{C}}\neg RA) \\
RB & \Longrightarrow & (\boxed{\text{A}}RB) \wedge (\boxed{\text{C}}RB) \\
\neg RB & \Longrightarrow & (\boxed{\text{A}}\neg RB) \wedge (\boxed{\text{C}}\neg RB) \\
RC & \Longrightarrow & (\boxed{\text{A}}RC) \wedge (\boxed{\text{B}}RC) \\
\neg RC & \Longrightarrow & (\boxed{\text{A}}\neg RC) \wedge (\boxed{\text{B}}\neg RC)
\end{array}
$$

Wir erlauben das Axiom K in den drei Varianten:

$$
\begin{array}{lcl}
\boxed{\text{A}}(F \Longrightarrow G) & \Longrightarrow & (\boxed{\text{A}}F \Longrightarrow \boxed{\text{A}}G) \\
\boxed{\text{B}}(F \Longrightarrow G) & \Longrightarrow & (\boxed{\text{B}}F \Longrightarrow \boxed{\text{B}}G) \\
\boxed{\text{C}}(F \Longrightarrow G) & \Longrightarrow & (\boxed{\text{C}}F \Longrightarrow \boxed{\text{C}}G)
\end{array}
$$

Aus den obigen Formeln (Axiomen) kann man noch nichts Sinnvolles schließen.
Die beiden Aussagen, die zusätzlich gemacht werden, sind:

$$
\begin{array}{ll}
\neg\boxed{\text{A}}RA & \text{(A kennt die Farbe seines Punktes nicht.)} \\
\neg\boxed{\text{B}}RB & \text{(B kennt die Farbe seines Punktes nicht.)}
\end{array}
$$

Hieraus kann man jetzt folgendes herleiten:
Beweis durch Widerspruch:

- Angenommen $\neg RB \wedge \neg RC$ gilt.
  Die Formel $(\boxed{\text{A}}(RA \vee RB \vee RC))$ ist in der Formulierungs-Menge enthalten.

Aus den Annahmen kann man jetzt $\boxed{\text{A}}(\neg RB)$ und $\boxed{\text{A}}(\neg RC)$ herleiten:
Wir nehmen K in der A-Variante: $(\boxed{\text{A}}(\neg RB \implies (RA \vee RC)))$ und $\boxed{\text{A}}(\neg RB)$ impliziert mit K: $\boxed{\text{A}}(RA \vee RC)$. Analog nochmal K benutzen ergibt: $\boxed{\text{A}}(RA)$. Das ist aber ein Widerspruch zu der Annahme $\neg\boxed{\text{A}}(RA)$.
Also hat man $RB \vee RC$ hergeleitet.

- Symmetrisch kann man herleiten: $RA \vee RC$.

- Das ist noch unklar: Insbesondere kann man jetzt herleiten: $\boxed{\text{B}}(RB \vee RC)$. Das ist das gleiche wie $\boxed{\text{B}}(\neg RC \implies RB)$:
  Erneuter Beweis durch Widerspruch:
  Angenommen $\neg RC$ gilt. Dann kann man herleiten: $\boxed{\text{B}}\neg RC$, und damit, wieder mittels Verwendung der K-Axiom-Varianten: $\boxed{\text{B}}RB$, im Widerspruch zu $\neg\boxed{\text{B}}RB$.
  Also ist RC hergeleitet durch Widerspruch.

Damit gilt auch $\boxed{\text{C}}RC$.

5.4 Modallogiken und Relationsvarianten

Zur Modellierung der verschiedenen Modallogiken werden i.a. die Eigenschaften der Erreichbarkeitsrelation eingeschränkt. Historisch ererbt sind einige Namen für bestimmte Modallogiken, wobei diese Namensgebung auch z.T. nicht ganz eindeutig ist.
Die Vorgehensweise ist so, dass man eine Eigenschaft E der Erreichbarkeitsrelation fordert. Die Semantik der zugehörigen Modallogik ML_E beschränkt sich auf die Menge der Kripke-Rahmen, deren Erreichbarkeitsrelation die Eigenschaft E hat.

Folgende Tabelle gibt einige Namen und zugehörige Eigenschaften an:

Name der Modallogik (des Axioms) (bzw. der Axiome)	Eigenschaft der Relation im Rahmen (W, R)
K	R ist beliebig
T	R ist reflexiv
K4	R ist transitiv
S4	R ist reflexiv und transitiv
B	R ist symmetrisch
S5	R ist reflexiv, transitiv und symmetrisch
5	R ist euklidisch
D	R ist seriell (unbeschränkt)
D4	R ist seriell und transitiv
S4.2	R ist reflexiv, transitiv und konfluent
Grz	R: in jeder unendlichen R-Kette $x_1\ R\ x_2 \ldots$ gibt es ein i mit $x_j = x_{j+1}$ für alle $j \geq i$
Aussagenlogik	es gibt genau eine Welt w_0, und $w_0\ R\ w_0$ gilt

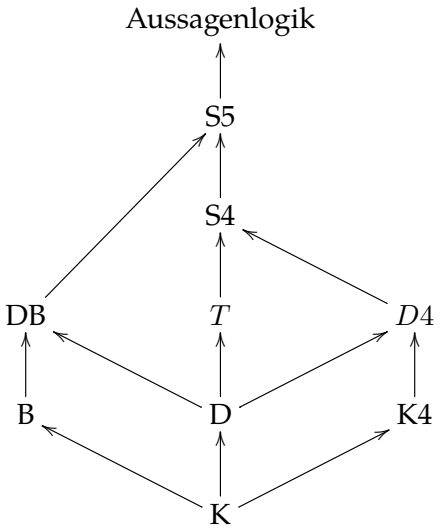
Hierbei sind die Eigenschaften so definiert:

reflexiv	$\forall w : R(w, w)$
transitiv	$\forall w_1, w_2, w_3 : w_1\ R\ w_2 \wedge w_2\ R\ w_3 \implies w_1\ R\ w_3$
symmetrisch	$\forall w_1, w_2 : w_1\ R\ w_2) \implies w_2\ R\ w_1$
seriell	$\forall w \exists w' : w\ R\ w')$
euklidisch	$\forall w_1, w_2, w_3 : w_1\ R\ w_2 \wedge w_1\ R\ w_3 \implies (w_2\ R\ w_3)$
konfluent	(falls R reflexiv und transitiv)
	$\forall w_1, w_2, w_3 : (w_1\ R\ w_2) \wedge w_1\ R\ w_3) \implies (\exists w_4 : w_2\ R\ w_4) \wedge w_3\ R\ w_4$

Beachte:	reflexiv	$\implies$	seriell
	seriell, transitiv, symmetrisch	$\implies$	reflexiv

Wir werden im folgenden die Begriffe Tautologie, Kripke-Struktur, Erfüllbarkeit usw. relativieren, indem wir je nach betrachteter Modallogik den entsprechenden Präfix davor schreiben, z.B. K-Tautologie, S4-Tautologie, usw.

Es gilt folgendes Abhängigkeitsdiagramm.



Die transitive Hülle der Pfeile gilt ebenfalls. Dieses Diagramm der Abhängigkeiten sagt etwas über die Gültigkeit von Tautologien aus. Falls $L_1 \to L_2$ im Diagramm enthalten ist, und F eine L_1-Tautologie ist, dann ist F auch eine L_2-Tautologie.
Die Aussagenlogik kann man in das Diagramm einfügen, wobei wir bei der Übertragung der Tautologien einfach die Extra-Zeichen streichen.

Die folgende Tabelle gibt die Namen der definierenden Axiome (Axiomenschemata) an. Leider sind die Namen in der Literatur nicht einheitlich und werden z.T mit verschiedener Bedeutung benutzt. Der Name G wird in mindestens 3 verschiedenen Bedeutungen benutzt.

Name (alt. N)	Formel	Anwendung
K	$\Box(F \implies G) \implies (\Box F \implies \Box G)$	
T (M)	$\Box F \implies F$	Logik des Wissens (Wissen ist korrekt)
D	$\Box F \implies \Diamond F$	Deontisch (Nichtglaube an das Gegenteil)
4	$\Box F \implies \Box\Box F$	Zeit ; Positive Introspektion)
B	$F \implies \Box\Diamond F$	
5 (E)	$\Diamond F \implies \Box\Diamond F$	(negative Introspektion)
M (G)	$\Box\Diamond F \implies \Diamond\Box F$	
L (H, Lem0)	$\Box((A \wedge \Box A) \implies B) \vee \Box((B \wedge \Box B) \implies A)$	
3 (H, H_0^+, Lem)	$\Box(\Box A \implies B) \vee \Box(\Box B \implies A)$	
2 (G)	$\Diamond\Box F \implies \Box\Diamond F$	
G (W)	$\Box(\Box F \implies F) \implies \Box F$	
Dum	$\Box(\Box(F \implies \Box F) \implies F) \implies (\Diamond\Box F \implies \Box F)$	
Grz	$\Box(\Box(F \implies \Box F) \implies F) \implies F)$	

Einige in epistemischen Logiken (auch deontischen[2]) verwendeten Axiome mit ihrer intuitiven Bedeutung.

Name	Formel	Erklärung
T	$\Box F \implies F$	Was ich weiss, das gilt (Axiom des perfekten Wissens)
D	$\Box F \implies \Diamond F$	Wissen ist nicht widersprüchlich (nach Umformung äquivalent zu: $\neg(\Box A \wedge \Box(\neg A)$
	$\Box F \implies \neg\Box\neg F$	Wenn ich F glaube, dann glaube ich das Gegenteil von F nicht
4	$\Box F \implies \Box\Box F$	Wenn ich F weiß, dann weiß ich dass ich F weiß
5	$\Diamond F \implies \Box\Diamond F$	
	$\neg\Box F \implies \Box\neg\Box F$	wenn ich F nicht weiß, dann weiß ich dass ich F nicht weiß.

Man kann Modallogiken auch mittels ihrer axiomatischen Basis und eines dazugehörigen Kalküls beschreiben. Dazu gehören dann die Schlussweisen Ersetzung, Modus Ponens, Notwendig-Einführung: $\frac{F}{\Box F}$ und die Axiome. Diese Methode wird z.B. im Buch von Hughes und Cresswell eingehend diskutiert. Wir wenden uns nur mit der Beschreibungsmethode mittels Rahmen und der Erreichbarkeitsrelation befassen. Die Benutzung der Kripke-Semantik schließt einige in der Literatur untersuchten Modallogiken von der Betrachtung aus, wenn K in diesen Logiken nicht gilt. Diese nennt man auch nicht-normale Modallogiken.

Die Namensgebung der verschiedenen Modallogiken wird mittels der Namen der notwendigen Axiome durchgeführt. Es gibt einige festgelegte Namen, z.B. S4, S5 bedeuten 4. System (5. System) in einer von Lewis und Langford beschriebenen Serie von modallogischen Systemen. Die Namen der Logiken sind am einfachsten im System von Lemmon hinzuschreiben, wobei einfach eine axiomatische Basis genügt. Der Name der Logik $S4$ wäre $KT4$ (da $K + T + 4$ als Axiome ausreichen) für S5 wäre dies KT5.

Wie wir schon gesehen haben, gibt es für einige Logiken einen einfachen Zusammenhang zwischen axiomatischer Basis und Eigenschaft der Erreichbarkeitsrelation des Rahmens. Diese Zusammenhänge sind nicht immer so einfach. Die (modallogische) Korrespondenztheorie untersucht diese Zusammenhänge. Einige interessante Modallogiken können mittels der Eigenschaften der Erreichbarkeitsrelation beschrieben werden. Manchmal gibt es keine Beschreibung mittels Axiomen erster Ordnung. Es gibt auch ein Beispiel (S4.2), für dass es einen Rahmen gibt, so dass dieser Rahmen gerade alle richtigen Tautologien liefert, aber die axiomatische Basis eigentlich eine unendliche Menge von Rahmen beschreibt, die sich nur mit einer Eigenschaft zweiter Ordnung beschreiben

[2] Vorsicht bei deontic logic: diese führt in mehreren Formulierungenleicht zu Paradoxien; es scheint keine brauchbare Formulierung zu geben.

lassen.

5.4.1 Tautologien in Varianten der Modallogik

Einige K-Tautologien können wir jetzt schon angeben.

Lemma 5.4.1. *In K, d.h. in allen Modallogiken, die eine Kripkesemantik haben, gelten folgende Tautologien:*

1. $\Box(F \implies G) \implies (\Box F \implies \Box G)$
2. $\Box F \iff \neg(\Diamond \neg F)$
3. $\Diamond F \iff \neg\Box(\neg F)$
4. $\Box(F \wedge G) \iff (\Box F \wedge \Box G)$
5. $\Diamond(F \vee G) \iff (\Diamond F \vee \Diamond G)$

Beweis. Begründung
Wir begründen nur 1. und 2. 1. Sei $K = (W, R, I)$ eine Kripkestruktur und w die Welt, in der die Formel ausgewertet wird. Falls $\Box(F \implies G)$ falsch ist unter I, dann gilt Axiom K. Also betrachten wir den Fall, dass $\Box(F \implies G)$ wahr ist. Dann ist in allen von w erreichbaren Welten $(F \implies G)$ wahr unter I. Jetzt betrachten wir die rechte Seite. Es genügt den Fall zu betrachten dass $\Box F$ wahr ist. Dann ist F in allen von w aus erreichbaren Welten wahr. Da dies auch für $(F \implies G)$ gilt, haben wir auch, dass G in allen erreichbaren Welten gilt, also gilt $\Box G$ unter I.
2. Wenn F in allen von w erreichbaren Welten gilt, dann ist dies äquivalent dazu, dass $\neg F$ in von w aus erreichbaren Welten nicht gelten kann. □

Weiterhin können wir schon begründen, dass folgende Schlussregel gilt:

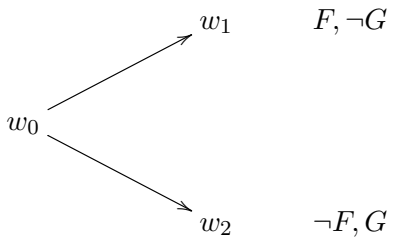
Wenn F Tautologie, dann ist auch $\Box F$ eine Tautologie.

Begründung:
Sei (W, R, I) eine Kripkestruktur. Wenn F eine Tautologie ist, dann gilt F in allen Welten w. Somit gilt auch, dass für jede Welt w die Formel F in allen von w aus erreichbaren Welten gilt, d.h. $\Box F$ gilt in (W, R, I). Da dies für alle Kripkestrukturen gilt, ist $\Box F$ dann eine Tautologie.

Vorsicht: Die Formel $F \implies \Box F$ ist keine Tautologie. Dies widerlegt eine einfache Kripke-Struktur:

Es gilt nicht: $P \implies \Box P$ in der Welt a.

Beachte: $\Box(F \vee G) \implies (\Box F \vee \Box G))$ ist keine K-Tautologie.
Um dies zu untermauern, konstruieren wir eine Kripke-Struktur, in der diese Aussage nicht gilt. Dazu nehme als Welten w_0, w_1, w_2 mit:



Dann gilt $\Box(F \vee G)$ in der Welt w_0, aber weder $\Box F$ noch $\Box G$ gilt.

Wir zeigen am Beispiel von T, wie die Struktur der Welten mit dem Axiom $\Box F \implies F$ zusammenhängt. Zu beachten ist hier, dass diese Korrespondenz nur gilt, wenn man Gültigkeit in Kripke-Rahmen betrachtet.

Lemma 5.4.2. *Gegeben ein Kripke-Rahmen (W, R). Dann ist R reflexiv gdw. $\Box F \implies F$ im Kripke-Rahmen (W, R) für alle Formeln F gültig ist*

Beweis. Sei (W, R, I) eine Kripke-Struktur mit reflexivem R. Dann ist zu zeigen, dass $\Box F \implies F$. Sei w_0 die aktuelle Welt und sei weiterhin $\Box F$ wahr. Dann ist auch F wahr in w_0, denn es gilt $w_0 \; R \; w_0$.
Nun sei $\Box F \implies F$ gültig im Rahmen (W, R).
Sei w_0 eine Welt, so dass nicht $w_0 \; R \; w_0$ gilt. Dann können wir ein I so definieren, dass für eine aussagenlogische Variable X, diese nicht in w_0 gilt, aber in allen w mit $w_0 \; R \; w$. Dann gilt X in w_0. Da $\Box F \implies F$ im Rahmen (W, R) gilt, gilt auch X in w_0. Dies ist ein Widerspruch. □

Lemma 5.4.3. *Gegeben eine Kripke-Rahmen (W, R). Dann ist R transitiv gdw. $\Box F \implies \Box\Box F$ im Kripke-Rahmen (W, R) gültig ist.*

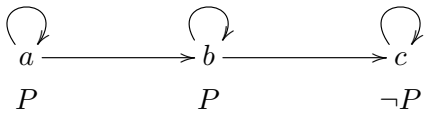
Beweis. Sei (W, R, I) eine Kripke-Struktur mit transitivem R. Dann ist zu zeigen, dass $\Box F \implies \Box\Box F$ gilt. Sei w_0 die aktuelle Welt und sei weiterhin $\Box F$ wahr. Seien w_1 und w_2 Welten mit $w_0 \; R \; w_1$ und $w_1 \; R \; w_2$. Dann gilt auch $w_0 \; R \; w_2$ und wegen Voraussetzung ist F in w_2 gültig.
Sei $\Box F \implies \Box\Box F$ gültig im Rahmen (W, R). Seien w_0, w_1, w_2 Welten, so dass $w_0 \; R \; w_1$ und $w_1 \; R \; w_2$, aber nicht $w_0 \; R \; w_2$. Dann können wir ein I so definieren, dass für eine aussagenlogische Variable X, diese in w_0 gilt, in allen von w_0 aus erreichbaren Welten, aber nicht in w_2. Dann gilt $\Box X$ in w_0, aber nicht $\Box\Box X$. Da $\Box F \implies \Box\Box F$ im Rahmen (W, R) gilt,ist dies ein Widerspruch. □

In analoger Weise kann man andere Beziehungen zwischen Axiomen und Eigenschaften von R in einem Rahmen verifizieren.

Im folgenden wollen wir auch nachweisen, dass einige der angegebenen Logiken verschieden sind.

$K \neq T$: wurde implizit oben schon gezeigt.

$T \neq S4$:



In dieser Kripkestruktur gilt in allen Welten: $\Box P \implies P$, aber in der Welt a gilt nicht $\Box P \implies \Box\Box P$. Beachte, dass die Relation $a \; R \; c$ nicht gilt, da R nicht als transitiv definiert wurde.

Bemerkung zu Tableaukalkülen:

Die Tableaukalküle für die verschiedenen Modallogiken können in der jetzigen Form Tautologien nachweisen. Da aber das Deduktionstheorem nicht gilt (siehe 5.3.6), muss man beim Nachweis von Folgerungsrelationen $F_1, \ldots, F_n \models G$, folgende Erweiterung machen:

- Die Annahmen $F_1, \ldots, F_n$ können im Beweis als $\Box F_i$ nochmal eingeführt werden. Genauer: sogar als $\Box^k F_i$ für jedes k. Die anderen Statements dürfen nicht mit ${}^B ox$ wieder eingeführt werden.
- Beweis durch Widerspruch im Tableau ist ansonsten genauso: jede Aussage bezieht sich auf eine bestimmte Welt.

Diese Hinweise zum Tableauverfahren für die Konsequentrelations sind informell! Für einen kompletten Nachweis der Korrektheit diese(r) Tableauverfahren ist die wissenschaftliche Literatur dazu zu befragen und zu recherchieren.

5.4.2 Zum Skript

Im Rest des Kapitels sind Tableaumethoden beschrieben, die ab dem Jahr 2016 weggelassen wurden.
Wer mehr zur Modallogik wissen will, kann z.B. weiterlesen.
Das Skript ohne die Tableaukalküle geht weiter auf Seite 105

5.5 Ein Tableaukalkül für die einfachste Modallogik K

Die Tableaukalküle zu Modallogiken in diesem Kapitel lösen die Aufgabe: ist F eine Tautologie!

Die Modallogik K entspricht der angegebenen Kripke-Semantik ohne weitere Bedingungen. D.h. die Tautologien bzgl. K sind genau die geschlossenen Formeln, die in allen Kripke-Interpretationen gelten.

Folgende Herleitungsregeln für K-Tautologien sind ausreichend:

1. Alle klassischen Tautologien
2. $\Box(A \implies B) \implies (\Box A \implies \Box B)$
3. $\dfrac{A \implies B \quad A}{B}$
4. $\dfrac{A}{\Box A}$

Definition 5.5.1. *Ein Tableaukalkül für die Logik K kann man so definieren:*

Ein Tableau ist ein Baum mit zwei Arten von Knoten:

$w; F$ *w ist ein Name für eine Welt.*

$R(w_1, w_2)$ *Erreichbarkeitsrelation zwischen w_1, w_2.*

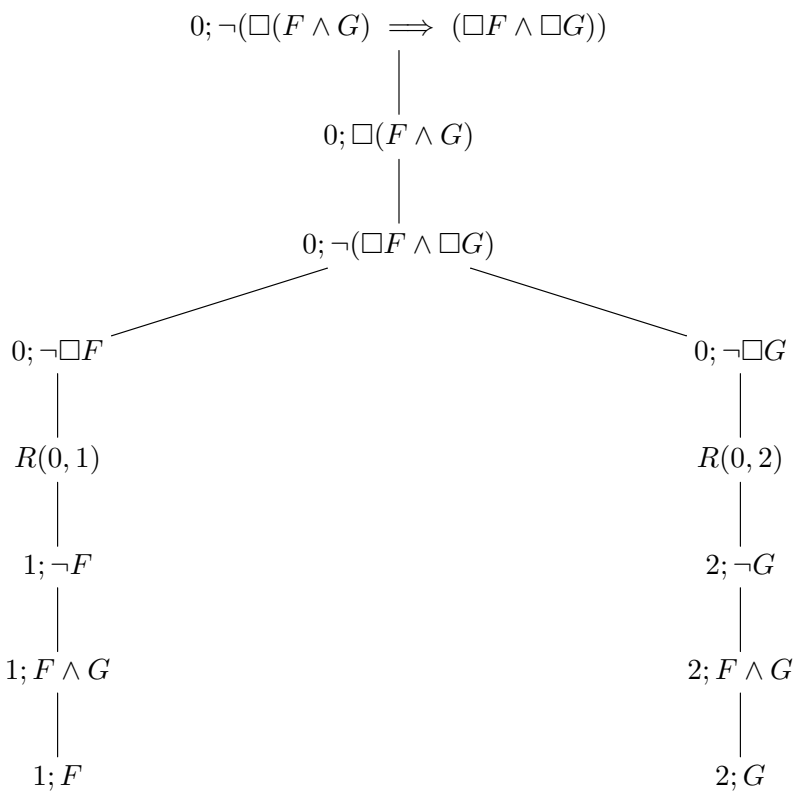
Ein Pfad P ist geschlossen, *wenn $w; 0$ auf dem Pfad P oder w, A und $w; \neg A$ ist auf dem Pfad P (mit gleichem w)*

Ein Tableau ist geschlossen, *wenn alle Pfade geschlossen sind.*

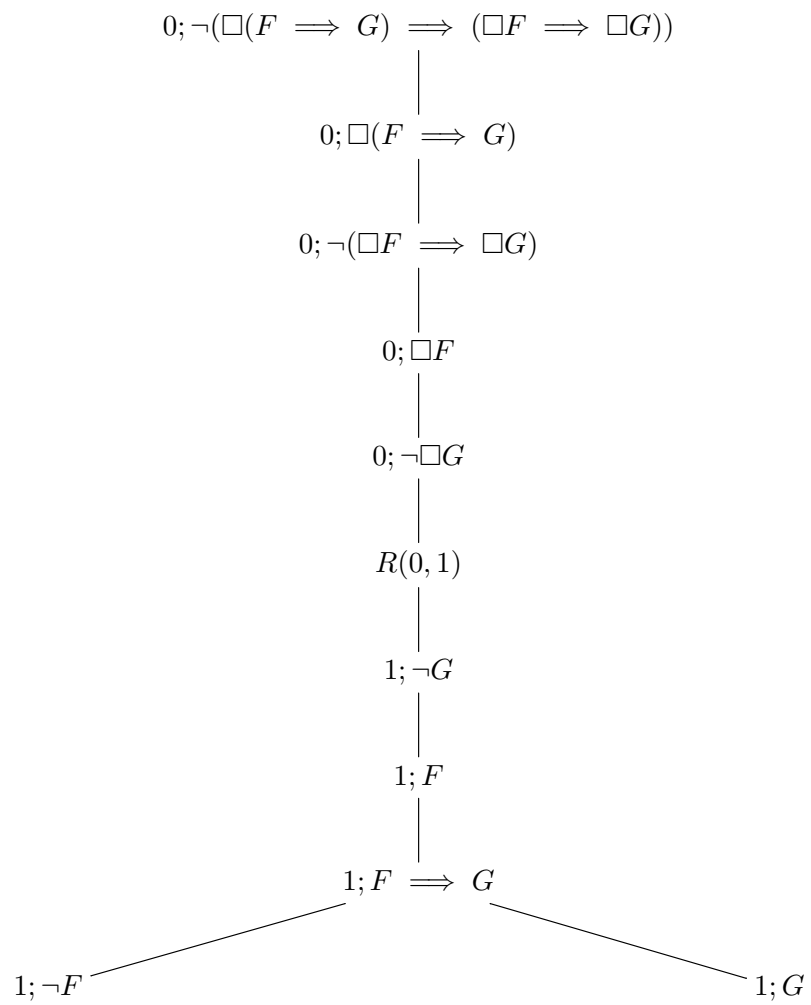
Die Expansionsregeln des K-Tableaukalküls sind:

- *aussagenlogische Tableauregeln; w bleibt erhalten*
- $\dfrac{w; \Diamond F}{R(w, w') \text{ und } w'; F}$
  werden an allen Pfaden durch $w; \Diamond F$ angehängt (w' ist neue Welt)
- $\dfrac{w; \Box F}{w'; F}$
  wird an allen Pfaden durch $w; \Box F$ angehängt, die $R(w, w')$ enthalten, aber noch nicht $w'; F$.
- *$w; \neg\Box F$ wird wie $w; \Diamond(\neg F)$ behandelt.*
  und $w; \neg\Diamond F$ wird wie $w; \Box(\neg F)$ behandelt.

Beispiel 5.5.2. *Der $\Box$-Operator ist distributiv über $\wedge$:*



Beispiel 5.5.3. *Das K-Axiom gilt:*



Eigenschaften des Tableaukalkül für K:

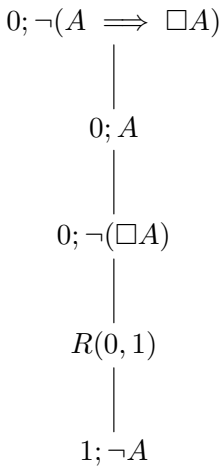
Theorem 5.5.4. *Der K-Tableaukalkül terminiert, ist korrekt und vollständig*

D.h. Das vervollständigte Tableau ist geschlossen genau dann wenn man einen K-Widerspruch eingibt.

Begründung: die neu eingeführten Formeln sind stets kleiner als die expandierte Formel. Man kann jedem Vorkommen eines modallogischen Symbols $\Box, \Diamond$ eine Tiefe zuordnen, die der Schachtelungstiefe entspricht. Man sieht dann, dass die die Länge der R-Ketten is beschränkt ist.

Eine gut Eigenschaft von Tableaukalkülen ist auch hier erfüllt: Wenn das vollständige Tableau nicht geschlossen ist, dann repräsentieren die nicht-geschlossene Pfade Modelle der eingegebenen Formel, bzw. Gegenbeispiele der Formel vor der Negation.

Beispiel 5.5.5. *Betrachte* $A \implies \Box A$*:*



Das Modell kann man direkt ablesen.

5.6 Allgemeiner Tableau-Kalkül für L mit Hornklauseltheorie für R

Wenn R-Axiome für L Hornklauseln sind, dann kann man folgenden einfachen Tableaukalkül verwenden, der den einfachen Tableaukalkül für K verallgemeinert:

Die Expansionsregeln sind:

- aussagenlogische Tableauregeln; w bleibt erhalten
- $\dfrac{w; \Diamond F}{R(w, w') \text{ und } w'; F}$
  werden an allen Pfaden durch $w; \Diamond F$ angehängt
  (w' ist neue oder bereits im Pfad vorkommende Welt)
- $\dfrac{w; \Box F}{w'; F}$
  wird an allen Pfaden durch $w; \Box F$ angehängt, die
  $R(w, w')$ enthalten, aber noch nicht $w'; F$.
- $R(w_1, w_2)$ wird am Pfad P angehängt,
  wenn es aus den R-Axiomen und den R-Relationen im Pfad folgt.
- $w; \neg\Box F$ wird wie $w; \Diamond(\neg F)$ behandelt.
  und $w; \neg\Diamond F$ wird wie $w; \Box(\neg F)$ behandelt.

Dieser Tableaukalkül terminiert möglicherweise nicht, da man in bestimmten Fällen immer weitere neue Welten hinzufügen kann.

Es gilt aber noch, dass der Tableaukalkül korrekt und vollständig ist.

5.6.1 Generische Tableaukalkül(e) für verschiedene Modallogiken

Im folgenden betrachten wir einen verbesserten Tableaukalkül, der weniger Fallunterscheidungen machen muss und kleinere Tableaus erzeugt, dafür aber eine ompliziertere Datenstruktur und ein kompliziertes Verfahren benötigt.

Die Knoten der Tableaus sind jetzt Paare aus einer R-Sequenz und einer Formel.

Eine *R-Sequenz* ist ein String aus Variablen und Termen, wobei diese Variablen, Funktionssymbole und Konstanten nicht in den Formeln vorkommen und den Weg von der Ursprungswelt zur aktuellen formal notieren sollen. D.h. das sind eigentlich Variablen und Terme, die modallogische Welten bezeichnen. Wir nennen diese Symbole auch Welten-Variablen, Welten-Konstanten, Welten-Funktionen. Weiterhin gilt, dass für jede Welten-Variable x der R-Sequenz von Anfang bis x in jedem Pfad des Tableaus, in dem x vorkommt, gleich ist. Im Tableaukalkül werden Pfade erzeugt, die einer Menge von markierten Formeln entsprechen. Die Menge der R-Sequenzen, die in einem Pfad P vorkommen, sei Pfad(P). Ein Formel im Tableau wird als $p; F$ notiert, wobei p die R-Sequenz und F die eigentliche Formel ist. Der Begriff L-Instanz meint eine Einsetzung von Weltentermen in Welten-Variablen, wobei nur im Pfad benutzte Weltensymbole verwendet werden dürfen, und wobei die Axiome der Logik L benutzt werden dürfen (siehe auch unten).

5.6.2 Kalkül TK_L

Eine Logik L wird fixiert. Folgende Regeln werden zur Expansion des Tableaus verwendet:

$$\frac{p; \neg\neg F}{p; F}$$

$$\frac{p; \alpha}{\begin{array}{l} p; \alpha_1 \\ p; \alpha_2 \end{array}}$$

$$\frac{p; \beta}{p; \beta_1 \mid p; \beta_2} \qquad p \text{ enthält keine Variablen}$$

$$\frac{p; \beta}{p'; \beta} \qquad p' \text{ ist L-Instanz von } p \text{ im Pfad}$$

$$\frac{p; \Diamond F}{p \cdot f(x_1, \ldots, x_n); F} \qquad f \text{ ist neu und } x_i \text{ sind die Variablen in } p$$

$$\frac{p; \Box F}{p \cdot x; F} \qquad x \text{ ist neue Variable}$$

Die Formeln $\neg\Box F$ und $\neg\Diamond F$ werden wie $\Diamond\neg F$ bzw. $\Box\neg F$ behandelt. An Steuerung ist nur zu beachten, dass jede Formel nur einmal pro Pfad expandiert wird, bis auf die in-

stanziierbaren Formeln vom Typ p, β. Hier müssen alle Instanziierungen von p ausprobiert werden; leider für alle Pfade, auf denen das Paar liegt. Dabei kann es vorkommen, dass neue Instanziierungen entstehen, und die Formel oft (evtl. unbegrenzt) in instanziierter Form in einem Pfad hinzugefügt werden muss. Hier kann es zu Nichtterminierung des Tableaukalküls kommen, abhängig von der benutzten Modallogikvariante.

Wir nehmen an, dass für jede Modallogik L Algorithmen existieren, die für eine Menge von R-Sequenzen bzgl eines Pfades folgendes leisten:

1. Entscheiden, ob eine R-Sequenz p L-erfüllbar in einem Pfad ist.
2. Entscheiden, ob zwei R-Sequenzen p_1 und p_2 L-kompatibel (in einem der gemeinsamen Pfade) sind.
3. Berechnen einer L-Instanz einer R-Sequenz.

Jedes Paar von (p, F) im Tableau kann man übersetzen in eine PL1- Formel. Das ist einfacher zu klären, wenn man auf die Erzeugung der R-Sequenzen zurückgeht. Durch $\Box$ werden $\forall$-quantifizierte Variablen eingeführt und durch $\Diamond$ werden $\exists$-quantifizierte Variablen eingeführt. Z.B das Paar $(x \cdot y \cdot f(x, y); F)$ entspricht der Formel

$$\forall x : 0\ R\ x \implies (\forall y : x\ R\ y \implies \exists z : y\ R\ z \wedge (\text{Gilt}(z, F)))$$

wobei Gilt(z, F) ein Prädikatensymbol ist, das $z \models F$ formalisiert. Skolemisiert man das z in der Formel, ergibt sich die R-Sequenz $x \cdot y \cdot f(x, y)$.

Definition 5.6.1. *Sei L eine der oben angegebenen Logiken, die eine Klasse von Rahmen festgelegt. Sei P ein Pfad in einem Tableau und $PT(P)$ die Menge der R-Sequenzen von P. Um die oben angegebenen Begriffe genauer zu definieren, interpretieren wir die R-Sequenzen folgendermaßen: Die* initiale Welt *bezeichnen wir mit* 0.
$PT(P)$ hat die Eigenschaft, dass mit jeder R-Sequenz auch der Präfix in $PT(P)$ ist, da dieser vorher im Pfad aufgebaut wurde.

Jede R-Sequenz, die als letztes Zeichen keine Variable hat, kann in eine Formel erster Ordnung übersetzt werden, die etwas über die Relation R im Pfad aussagt. Z.B. hat $0 \cdot x \cdot y \cdot f(x, y)$ folgende Auswirkung: wenn im Pfad P schon bekannt ist, dass $0\ R\ a_1, a_1\ R\ a_2$, dann gilt auch $a_2\ R\ f(a_1, a_2)$. Die Sequenz $0 \cdot a \cdot b$ wird in $R(0, a) \wedge R(a, b)$ übersetzt.

Damit kann man jedem Pfad P eine Menge $Q(P)$ von R-Relationen zuordnen:
Für R-Sequenzen $0 \cdot t_1 \cdot \ldots \cdot t_n$ aus $PT(P)$ gilt folgende Erzeugungsregel für $Q(P)$ falls t_n keine Variable ist:

1. *Falls $0 \cdot t_1 \cdot \ldots \cdot t_n \in PT(P)$ und t_n keine Variable ist und $0\ R\ \sigma(t_1), \ldots, \sigma(t_{n-2})\ R\ \sigma(t_{n-1}) \in Q(P)$:*
   Füge $\sigma(t_{n-1})\ R\ \sigma(t_n)$ zu $Q(P)$ hinzu.
2. *Im Falle, dass die Logik Rahmenaxiome $\subseteq$ { reflexiv, symmetrisch, transitiv } hat: Wenn $Q(P) \models s\ R\ t$ für Grundterme s, t, füge $s\ R\ t$ zu $Q(P)$ hinzu.*

$Q(P)$ ergibt sich als kleinster Fixpunkt der Einfügeoperationen. Diese Berechnung ist für den Tableaukalkül nur dann sinnvoll, wenn $Q(P)$ endlich ist.

In D würde das Verfahren unendlich viele neue Konstanten bzw. Funktionen neu erzeugen, die nicht im Pfad vorkommen. Hier verwendet man eine Ergänzung von $PT(P)$, die ausreicht: Wenn man zu einer R-Sequenz $p = 0 \cdot t_1 \ldots t_n$, die mit einer Variablen endet, keine Instanz in $Q(P)$ findet, dann darf man die R-Sequenz $p' = 0 \cdot t_1 \ldots t_{n-1} \cdot f(x_1, \ldots, x_k)$ hinzufügen, wobei f neu ist, und x_i die Variablen in $0 \cdot t_1 \cdot \ldots \cdot t_{n-1}$ sind. Danach kann man die Hülle berechnen wie in K.

1. *Eine L-Instanz bzgl. P einer R-Sequenz p ist $\sigma(p)$, so dass alle Relationen aus $\sigma(p)$ in $Q(P)$ sind.*

2. *Eine R-Sequenz p ist L-erfüllbar bzgl P, wenn es eine L-Instanz gibt.*

3. *Zwei R-Sequenzen p und q sind L-kompatibel bzgl. P, wenn es Grundinstanzen p' und q' von p und q bzgl $Q(P)$ gibt, so dass p und q das gleiche Ende haben.*

Diese Definitionen gehen von der Annahme aus, dass verschiedene Namen auch etwas verschiedenes bedeuten (unique names assumption, freie Termalgebra). In bestimmten Logikvarianten kann es sein, dass man auf Gleichheit von syntaktisch verschiedenen Welttermen schließen kann. Intuitiv bedeutet L-erfüllbar, dass die R-Sequenz als Folge von Welten bzgl R gedeutet werden kann,
L-kompatibel bedeutet, dass beide R-Sequenzen Instanzen haben, die mit der gleichen Welt enden.

Definition 5.6.2. *Ein Pfad P eines Tableaus ist L-*geschlossen*, wenn eine der folgenden Bedingungen erfüllt ist:*

1. *Es gibt zwei komplementäre Formeln $p_1; F$ und $p_2; \neg F$ auf dem Pfad, so dass p_1 und p_2 L-kompatibel bzgl $Q(P)$ sind.*

2. *Es gibt eine Formel $p; 0$ so dass die R-Sequenz p L-erfüllbar ist bzgl $Q(P)$.*

Das Tableau ist L-geschlossen, wenn alle Pfade des gesamten Tableaus L-geschlossen sind.

Ein Pfad P ist erschöpft, *wenn er nicht geschlossen ist, und man kein neues Blatt an P mehr anhängen kann. Entsprechend ist das Tableau* erschöpft, *wenn es nicht geschlossen ist, aber man keine neuen Blätter mehr anhängen kann*

Ein *Tableaubeweiser für die Logik L*, der die Formel F beweisen will, startet mit $\varepsilon; \neg F$ an der Wurzel und versucht durch Anwenden der Expansionsregeln alle Pfade zunächst zu erweitern und dann zu schließen.

Was man jetzt noch pro Logik finden muss, ist ein Algorithmus, der die Eigenschaften der Pfade und $Q(P)$ effizient überprüft.

5.6.2.1 Algorithmen für verschiedene Logiken K, D, B, T, S4, S4.2, S5

Logik K : Es gibt keine Bedingungen an die Relation.

Sei P eine nichtleere Menge von Pfaden an pfadmarkierten Formeln. Wir fügen eine Anfangswelt 0 am Anfang eines jeden Pfades ein. Dann berechnen wir zunächst die Hülle $Q(P)$ der Relationen für P. Dann ist p bezüglich P *K-erfüllbar*, wenn es eine R-Sequenz s gibt, so dass s Instanz von p ist. Leider ist die Hüllenberechnung exponentiell, da man exponentiell viele Instanzen berechnen muss.

1. p ist bezüglich P *K-erfüllbar*, wenn eine Instanz von p in $Q(P)$ ist.
2. p_1 und p_2 sind K-kompatibel, wenn es R-Sequenzen s_1 und s_2 gibt, so dass s_i Instanz von p_i ist, und s_1 und s_2 den gleichen Endpunkt haben. D.h. dass diese Pfade syntaktisch gleich sind. Dies kann man prüfen durch eine syntaktische Unifikation der Pfade als n-Tupel, und danach durch eine Prüfung, ob das Ergebnis K-erfüllbar ist.

Die Prüfung, ob eine R-Sequenz p K-erfüllbar ist, oder ob zwei R-Sequenzen p_2, p_2 K-kompatibel sind, kann in polynomieller Zeit gemacht werden.

Logik D : Es gibt stets eine erreichbare Welt.

In diesem Fall sind alle Pfade D-erfüllbar, da man stets die Variablen mit instanzieren kann. D.h. p_1 und p_2 sind D-kompatibel, wenn diese unifizierbar sind. Diese Unifikation erfolgt als Unifikation von zwei n-Tupeln, der Test auf D-Erfüllbarkeit entfällt. Diese Berechnung kann linear gemacht werden.

Die Hüllenberechnung $Q(P)$ erfolgt analog zur Logik K, mit einer Ausnahme: Wenn man zu einer R-Sequenz $p = t_1 \dots t_n$, die mit einer Variablen endet, keine Instanz in $Q(P)$ findet, dann darf man die R-Sequenz $p' = t_1 \dots t_{n-1} \cdot f(x_1, \dots, x_k)$ hinzufügen, wobei f neu ist, und x_i die Variablen in $t_1 \cdot \dots \cdot t_{n-1}$ sind.

Die Berechnung der D-Instanzen erfolgt dann bzgl $Q(P)$.

Logik T : Die Relation ist reflexiv.

Die Menge der Relationen $Q(P)$, die aus $PT(P)$ und der Reflexivität folgt, wird zunächst berechnet. Offenbar sind alle Pfade T-erfüllbar. D.h. p_1 und p_2 sind T-kompatibel, wenn diese zwei Grundinstanzen haben mit gleichen Endpunkten. Dies kann man auch durch einen Unifikationsalgorithmus testen, der die Reflexivität von R benutzen darf. Z.B. würde $0 \cdot x \cdot y$ mit $0a$ unifizieren durch $x = y = a$. Eine T-Instanz von p ist eine R-Sequenz, die syntaktische Instanz von p ist und eine R-Kette bezüglich der oben berechneten Hülle ist.

Logik B : Die Relation ist symmetrisch.

Sei P eine nichtleere Menge von pfadmarkierten Formeln. Dann berechnen wir zunächst die symmetrische Hülle der Relationen zwischen der Menge aller Konstanten

und der zusätzlichen Konstanten 0. Die Algorithmen sind dann wie bei K. Dann ist p bezüglich P B-erfüllbar, wenn es eine R-Sequenz s gibt, so dass s Instanz von p ist. p_1 und p_2 sind B-kompatibel, wenn es R-Sequenzen s_1 und s_2 gibt, so dass s_i Instanz von p_i ist, und s_1 und s_2 gleichen Endpunkt haben. Die Berechnung einer B-Instanz eines Pfades p kann durch Bilden aller variablenfreier Instanzen erfolgen. Da man bei der Instanziierung von Pfaden keine Zyklen erhält, gibt es nur endlich viele Instanzen eines Pfades.

Logik S4 : Die Relation ist reflexiv und transitiv.

Die Hüllenberechnung von R erfolgt unter Ausnutzung von Reflexivität und Transitivität. Alle Pfade sind $S4$-erfüllbar. p_1 und p_2 sind $S4$-kompatibel, wenn es R-Sequenzen s_1 und s_2 gibt, so dass s_i Instanz von p_i ist, und s_1 und s_2 gleichen Endpunkt haben. Eine $S4$-Instanz eines Pfades ist eine variablenfreie Instanz.

Logik S5 : Die Relation ist reflexiv, symmetrisch und transitiv.

Die Hüllenberechnung von R erfolgt unter Ausnutzung von Reflexivität, Symmetrie und Transitivität. Alle Pfade sind $S5$-erfüllbar. p_1 und p_2 sind $S5$-kompatibel, wenn es R-Sequenzen s_1 und s_2 gibt, so dass s_i Instanz von p_i ist, und s_1 und s_2 gleichen Endpunkt haben. Dies kann leicht syntaktisch überprüft werden: Wenn die letzten Welten die Form $(a, x), (x, y)$, oder (a, a) haben, dann sind die Pfade $S5$-kompatibel. Eine Instanz eines Pfades ist eine variablenfreie Instanz. Wenn kein geeigneter Term für die letzte Welt existiert, so kann man einen neuen hinzufügen. Es gibt nur endlich viele Instanzen eines Pfades.

Logik S4.2 : Die Relation ist reflexiv, transitiv und konfluent.

Die Hüllenberechnung von R erfolgt unter Ausnutzung von Reflexivität, und Transitivität. Alle Pfade sind $S4.2$-erfüllbar. p_1 und p_2 sind $S4.2$-kompatibel, wenn es R-Sequenzen s_1 und s_2 gibt, so dass s_i Instanz von p_i ist, und s_1 und s_2 gleichen Endpunkt haben. Dies ist syntaktisch leicht zu prüfen, da R eine Äquivalenzrelation ist: Wenn die Paare der Endwelten die Form $(a, x), (x, y)$ oder (b, y) haben. Eine Instanz eines Pfades ist eine variablenfreie Instanz.

5.6.2.2 Beispiele

Logik K

Beispiel 5.6.3. *Wir konstruieren ein geschlossenes Tableau für die Distributivität von $\Box$über $\wedge$. Dazu werden beide Richtungen getrennt bewiesen.*

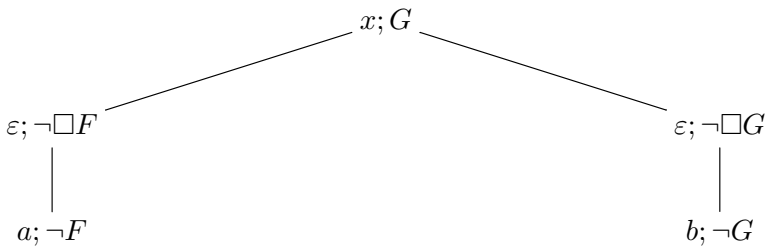
$$\varepsilon; \neg(\Box(F \wedge G) \implies (\Box F \wedge \Box G))$$

$$\varepsilon; \Box(F \wedge G)$$

$$\varepsilon; \neg(\Box F \wedge \Box G)$$

$$x; F \wedge G$$

$$x; F$$



Das Tableau ist geschlossen.

$$\varepsilon; \neg((\Box F \wedge \Box G) \implies \Box(F \wedge G))$$

$$\varepsilon; (\Box F \wedge \Box G)$$

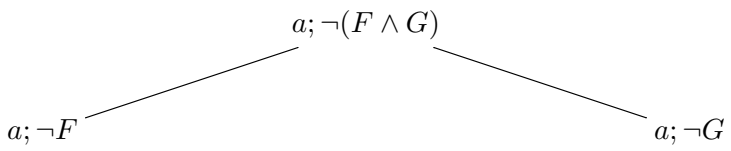
$$\varepsilon; \neg\Box(F \wedge G)$$

$$\varepsilon; \Box F$$

$$\varepsilon; \Box G$$

$$x; F$$

$$y; G$$



Das Tableau ist geschlossen.

Beispiel 5.6.4. *In K gilt die Distributivität von $\Box$ über $\vee$ nur in einer Richtung.*

$$\varepsilon; \neg(\Box(F \vee G) \implies (\Box F \vee \Box G))$$

$$\varepsilon; \Box(F \vee G)$$

$$\varepsilon; \neg(\Box F \vee \Box G)$$

$$x; F \vee G$$

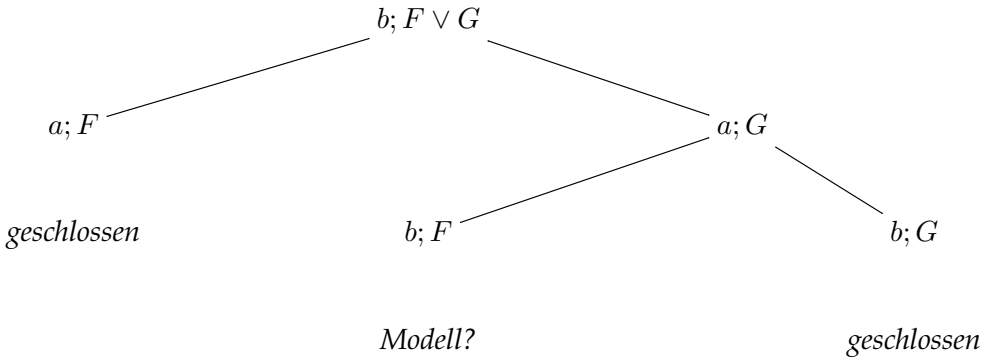
$$\varepsilon; \neg\Box F$$

$$\varepsilon; \neg\Box G$$

$$a; \neg F$$

$$b; \neg G$$

$$a; F \vee G$$



Die andere Richtung gilt:

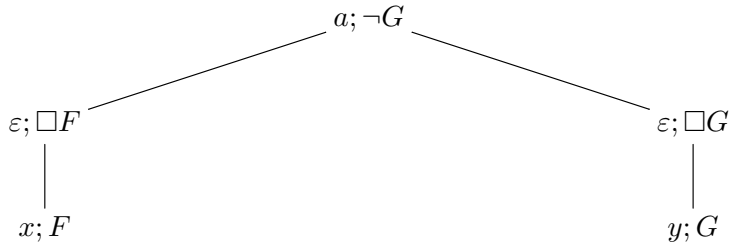
$$\varepsilon; \neg((\Box F \vee \Box G) \implies \Box(F \vee G))$$

$$\varepsilon; (\Box F \vee \Box G)$$

$$\varepsilon; \neg\Box(F \vee G)$$

$$a; \neg(F \vee G)$$

$$a; \neg F$$



Beispiel 5.6.5. *Es läßt sich in K nicht zeigen, dass $\Box F \implies \Diamond F$. D.h. in K lässt sich nicht zeigen, dass von jeder Welt aus eine weitere Welt erreichbar ist:*

$$\begin{array}{l}\varepsilon; \neg(\Box F \implies \Diamond F)\\ \varepsilon; \Box F\\ \varepsilon; \neg\Diamond F\\ x, \neg F\\ y, F\end{array}$$

Das Tableau ist nicht geschlossen, da x und y nicht mit einer Konstanten instanziiert werden können.

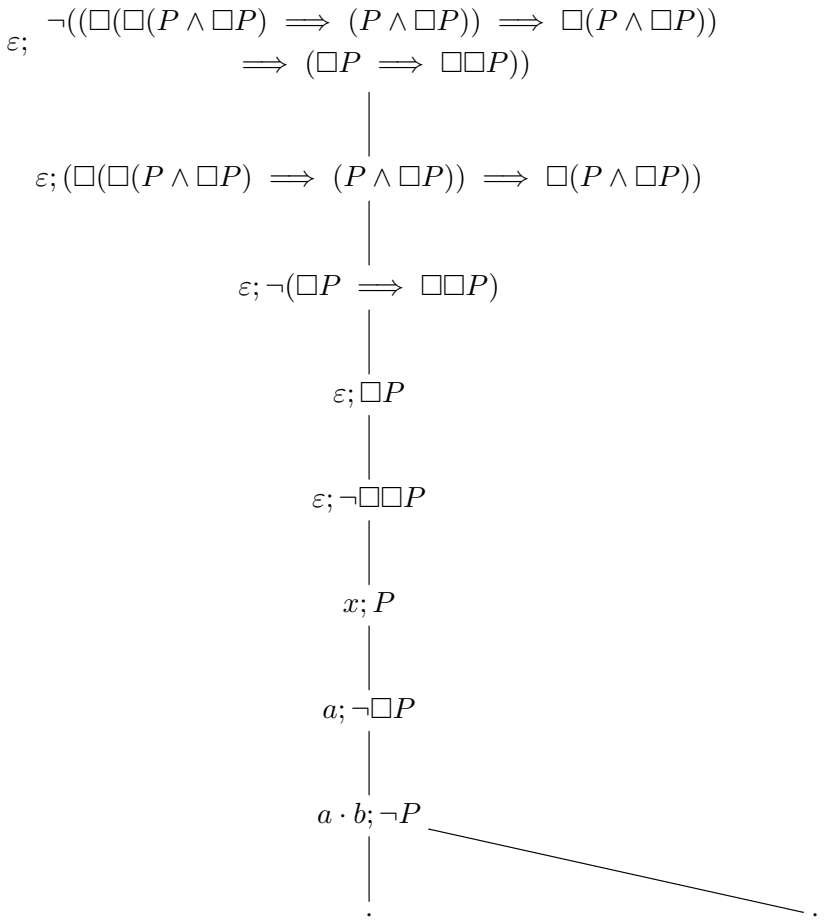
Es kann in K auch nicht bewiesen werden, dass $\neg\Box 0$ eine Tautologie ist:

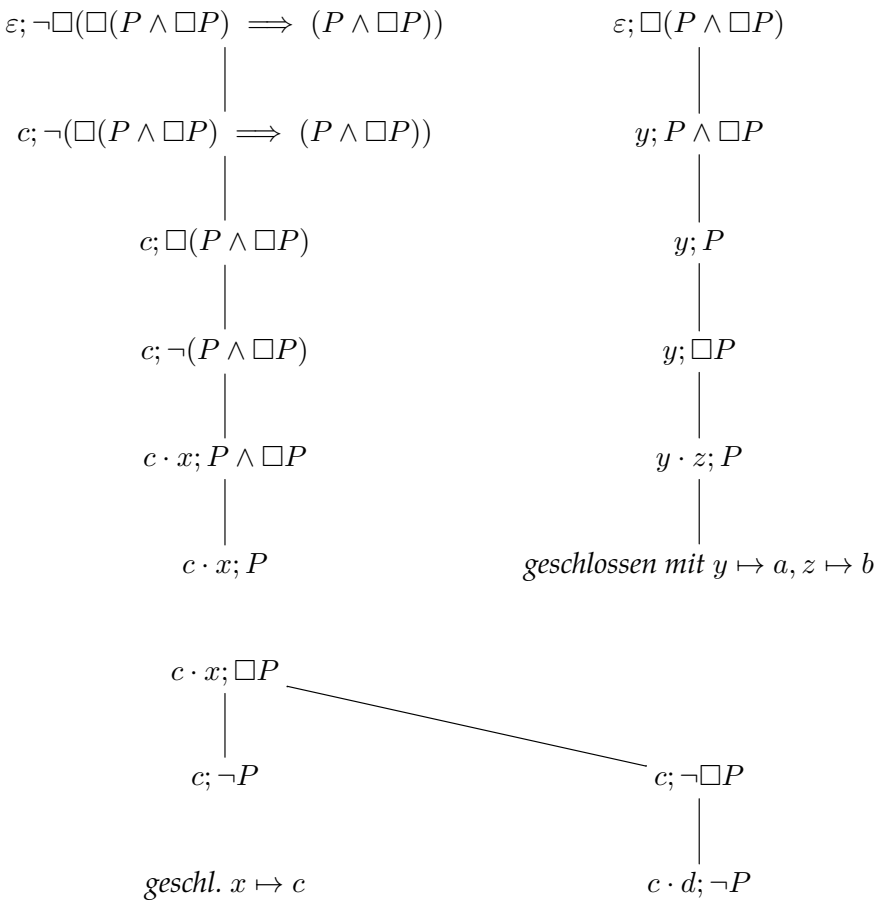
$$\begin{array}{l}\varepsilon; \neg\neg\Box 0\\ \varepsilon; \Box 0\\ x; 0\end{array}$$

Das Tableau ist erschöpft, aber nicht geschlossen.

Beispiel 5.6.6. *Löbs Axiom ist $\Box(\Box F \implies F) \implies \Box F$. Hier mit Instanz $F = (P \wedge \Box P)$*

Löbs Axiom impliziert Transitivität:





geschlossen

Logik D: seriell Einige Beispiele

Beispiel 5.6.7. *Das Axiom* $\Box F \implies \Diamond F$ *gilt in D*

$$\begin{array}{c} \varepsilon; \neg(\Box F \implies \Diamond F) \\ \varepsilon; \Box F \\ \varepsilon; \neg\Diamond F \\ x; \neg F \\ y; F \end{array}$$

Das Tableau ist geschlossen, da x *und* y *mit* $x \mapsto y$ *gleichgemacht werden können, und da es in D von jeder Welt aus, insbesondere von der initialen Welt* 0 *aus eine weitere gibt.*

Beispiel 5.6.8. $\neg\Box 0$ *ist eine D-Tautologie:*

$$\varepsilon; \neg\neg\Box 0$$
$$\varepsilon; \Box 0$$
$$x; 0$$

Ist geschlossen, da in D alle Pfadterme erfüllbar sind.

Beispiel 5.6.9. *Reflexivität ist nicht zu beweisen:*

$$\varepsilon; \neg(\Box F \implies F)$$
$$\varepsilon; \Box F$$
$$\varepsilon; \neg F$$
$$x; F$$

Das ist nicht geschlossen, da $0 \cdot 0$ i.a. keine D-Instanz von $0 \cdot x$ ist.

Logik S4

Beispiel 5.6.10. *In S4 ist nicht beweisbar: $\Box\Diamond F \implies \Diamond\Box F$ (das M-Axiom)*

$\varepsilon; \neg(\Box\Diamond F \implies \Diamond\Box F)$
$\varepsilon; \Box\Diamond F$
$\varepsilon; \neg\Diamond\Box F$ *dies entspricht* $\Diamond\neg F$
$x, \Diamond F$
$y, \Diamond\neg F$
$x \cdot a(x), F$
$x \cdot b(x), \neg F$

Die letzten beiden Welten $a(x), b(x)$ lassen sich nicht gleichmachen, sind also nicht kompatibel.

Beispiel 5.6.11. *In S4 gilt: $\Diamond\Box F \implies \Diamond\Box\Diamond F$:*

$$\varepsilon; \neg(\Diamond\Box F \implies \Diamond\Box\Diamond F)$$
$$\varepsilon; \Diamond\Box F$$
$$\varepsilon; \neg\Diamond\Box\Diamond F$$
$$a; \Box F$$
$$a \cdot x; F$$
$$y; \neg\Box\Diamond F$$
$$y \cdot b(y); \neg\Diamond F$$
$$y \cdot b(y) \cdot z; \neg F$$

Diese Tableau lässt sich schließen mit folgender S4-Instanziierung der Pfadterme: $y \mapsto a, x \mapsto b(a), z \mapsto b(a)$. Dann sind $a \cdot x; F$ und $y \cdot b(y) \cdot z, \neg F$ widersprüchlich.

Beispiel 5.6.12. *Zeige, dass in S4 $\Diamond\Box\Diamond F \implies \Diamond F$ gilt.*

$$\begin{gathered}\varepsilon; \neg(\Diamond\Box\Diamond F \implies \Diamond F)\\ \varepsilon; \Diamond\Box\Diamond F\\ \varepsilon; \neg\Diamond F\\ x, \neg F\\ a, \Box\Diamond F\\ a \cdot y, \Diamond F\\ a \cdot y \cdot b(y), F\end{gathered}$$

Ist geschlossen, da $b()$ von 0 aus erreichbar, bzw. $0 \cdot x$ und $0 \cdot a \cdot y \cdot b(y)$ S4-kompatibel sind.

Beispiel 5.6.13. *Zeige, dass $\Diamond\Box F \implies \Box F$ nicht in S4 beweisbar ist:*

$$\begin{gathered}\varepsilon; \neg(\Diamond\Box F \implies \Box F)\\ \varepsilon; \Diamond\Box F\\ \varepsilon; \neg\Box F\\ a; \neg F\\ b; \Box F\\ b \cdot x; F\end{gathered}$$

Ist nicht geschlossen, da die Pfadterme a und $b \cdot x$ nicht kompatibel in diesem Pfad.
$S4$ ist transitiv und reflexiv, aber nicht mehr. Daher kann man die offensichtliche Instanziierung $x \mapsto a$ nicht machen, da b R a nicht gelten muss.

Logik S5

Beispiel 5.6.14. *In S5 gilt: $\Diamond(F \vee \Diamond G) \implies \Diamond F \vee \Diamond G$:*

$$\varepsilon; \neg(\Diamond(F \vee \Diamond G) \implies \Diamond F \vee \Diamond G)$$

$$\varepsilon; \Diamond(F \vee \Diamond G)$$

$$\varepsilon; \neg(\Diamond F \vee \Diamond G)$$

$$a; F \vee \Diamond G$$

$$\varepsilon; \neg\Diamond F$$

$$\varepsilon; \neg\Diamond G$$

$$x; \neg F$$

$$y; \neg G$$

$$a; F \qquad\qquad a; \Diamond G$$

geschlossen $\qquad\qquad a \cdot b, G$

geschlossen, da R transitiv

D.h. diese Formel gilt auch in S4.

Beispiel 5.6.15. *In S5 gilt:* $\Diamond(F \vee \Box G) \implies \Diamond F \vee \Box G$

$$\varepsilon; \neg(\Diamond(F \vee \Box G) \implies \Diamond F \vee \Box G)$$

$$\varepsilon; \Diamond(F \vee \Box G)$$

$$\varepsilon; \neg(\Diamond F \vee \Box G)$$

$$a; F \vee \Box G$$

$$\varepsilon; \neg\Diamond F$$

$$\varepsilon; \neg\Box G$$

$$x; \neg F$$

$$b; \neg G$$

$a; F$ $\qquad$ $a; \Box G$

geschlossen $\qquad$ $\&a \cdot y; G$

geschlossen, da R Äquivalenzrelation

d.h. $y = b$ ist erlaubt

In S4 gilt diese Formel nicht.

Beispiel 5.6.16. *Zeige:* $\Diamond\Box F \implies \Box F$*;*

$$\varepsilon; \Diamond\Box F \implies \Box F$$
$$\varepsilon; \Diamond\Box F$$
$$\varepsilon; \neg\Box F$$
$$a; \neg F$$
$$b; F$$
$$b \cdot x; F$$
geschlossen

Beispiel 5.6.17. *Zeige, dass $\Diamond F \implies \Box F$ nicht beweisbar ist in S5:*

$$\varepsilon; \neg(\Diamond F \implies \Box F)$$
$$\varepsilon; \Diamond F$$
$$\varepsilon; \neg\Box F$$
$$b; \neg F$$
$$a; F$$

Ist nicht geschlossen, da verschiedene Endpunkte.
Die sind zwar verbunden, aber nicht gleich.

5.6.3 Theoretische Eigenschaften des Kalküls

Definition 5.6.18. *Gegeben eine pfadmarkierte Formel $p; F$ und eine Kripkestruktur M. dann ist $p; F$ in einer Welt w gültig, wenn:*

1. *Für jedes Funktionssymbol f gibt es eine Funktion f_M auf den Welten.*
2. *Falls $p = \varepsilon$, dann gilt $w \models F$.*
3. *Für Formeln $x \cdot p$: für alle Welten w' mit $w\ R\ w'$ gilt $w' \models p[x \mapsto w']; F$*
4. *Für Formeln $f(w_1, \ldots, w_n) \cdot p$ gibt es eine Welt $w' = f_M(w_1, \ldots, w_n)$ mit wRw' mit $w' \models p; F$.*

*Eine Menge von pfad-markierten Formeln ist L-*erfüllbar*, wenn es eine L-Kripkestruktur M gibt, so dass alle Formeln in M erfüllbar sind. Ein Pfad ist L-erfüllbar, wenn die Menge der pfad-markierten Formeln L-erfüllbar ist. Ein Tableau ist L-erfüllbar, wenn ein Pfad L-erfüllbar ist.*

Lemma 5.6.19. *Jede Tableauregel überführt L-erfüllbare Tableaus in L-erfüllbare und umgekehrt.*

Beweis. Wir gehen die Tableauregeln durch: Die $\neg\neg$ -Regel und die α-Regeln sind unproblematisch Betrachte die β-Regel. Wenn die Instanziierungsregeln einen Term t einsetzen, dann ist β_1 oder β_2 wahr in der Welt t. d.h., man kann beide Pfade anhängen. Bei den anderen ist das auch klar. □

Lemma 5.6.20. *Vollständigkeit des Kalküls: Jeder erschöpfte Pfad ist erfüllbar.*

Beweis. Zu jedem erschöpften Pfad ist ein Modell zu konstruieren. Dies hängt naturgemäß von der betrachteten Logik ab.

1. Welten konstruieren.

2. Wenn Endwelten nicht existieren, dann ist die Formel wahr.

3. Wenn am Ende eine Variable steht, und der Pfad erfüllbar ist, dann ist die Formel wahr. Ansonsten betrachte die Fälle, in denen die Formel von der Form A oder $\neg A$ ist wobei A ein Atom ist und in denen der Pfad erfüllbar ist. Dann sei die Interpretation so, dass für alle Instanzen des Pfades die entsprechende Formel wahr ist. Diese Interpretation macht keine widersprüchlichen Zuweisungen, da die Algorithmen für diese Logiken korrekt sind (noch zu definieren).

□

Lemma 5.6.21. *Für die Logiken K,D,T,B,S4, S5, S4.2 gibt es Algorithmen die die Kompatibilität und Erfüllbarkeit von Pfaden entscheiden.*

Beweis. ? □

Lemma 5.6.22. *Kalkül TK_{ML} terminiert für Logiken K, T, B, D*

Beweis. Es gilt, dass für diese Logiken die Algorithmen für Kompatibilität und Erfüllbarkeit Entscheidungsverfahren sind. Alle Formeln außer px, β werden nur einmal pro Pfad expandiert. Formeln werden immer kleiner. Wir betrachten den Fall, dass eine Formel px, β instanziiert und hinzugefügt wird. Das kann für K und D nur endlich oft geschehen, da es keine Verkürzungsmöglichkeit für Pfade gibt. Für die Logik T und B muss man anders argumentieren: Die Anzahl der nichtvariablen Terme in einem Pfad, die für eine Welt-variable eingesetzt werden können, bleibt endlich, denn für jeden solchen Term kann man einen „Abstand“ von der initialen Welt definieren. □

Theorem 5.6.23. *Der oben angegebene Tableaukalkül entscheidet Tautologien in den folgenden aussagenlogischen Modallogiken: K, T, B, D.*

Beispiel 5.6.24. *Der Kalkül terminiert nicht für $S4$. Versuche Grz: $(\Box(\Box(F \implies \Box F) \implies F) \implies F)$ in $S4$ zu zeigen:*

$$
\begin{array}{lll}
0; \neg(\Box(\Box(F \implies \Box F) \implies F) \implies F) & & \\
0; \Box(\Box(F \implies \Box F) \implies F) & & \\
0; \neg F & & \\
0x; (\Box(F \implies \Box F) \implies F) & & \\
00; (\Box(F \implies \Box F) \implies F) & & \\
00; \neg\Box(F \implies \Box F) & & 00; F \\
00a; \neg(F \implies \Box F) & & \textit{geschlossen} \\
00a; F & & \\
00a; \neg\Box F & & \\
00ab; \neg F & & \\
(\textit{Transitivität und Instanz}) & & \\
0b; \neg\Box(F \implies \Box F) & & 0b; F \\
0bc; \neg(F \implies \Box F) & & \textit{geschlossen} \\
0bc; F & & \\
0bc; \neg\Box F & & \\
0bcd; \neg F & & \\
\ldots & &
\end{array}
$$

Man kann mit d statt b das ganze wiederholen ... Dies terminiert nicht, da man immer wieder neue Konstanten erzeugt.

Allerdings gilt:

Theorem 5.6.25. *In S4 und S5 ist die Tautologieeigenschaft entscheidbar.*

Beweis. (siehe Buch von Hughes und Cresswell). □

Abhilfe im Falle unseres Tableaukalküls.

Die Logiken $S4$ und $S5$ haben die endliche Rahmeneigenschaft. D.h. Wenn eine Formel nicht gilt, so existiert ein endlicher Rahmen, der die Formel widerlegt. Z.B. oben: mit b = 0 erhält man:

$$
\begin{array}{lll}
0 & \longrightarrow & \\
\neg F & & \\
 & & \\
\neg\Box(F \implies \Box F) & & \\
(\Box(F \implies \Box F)) \implies F & & \\
\neg\Box((\Box(F \implies \Box F)) \implies F) \implies F & &
\end{array}
\qquad
\begin{array}{lll}
a & \longrightarrow & 0 \\
F & & \neg F \\
\neg\Box F & & \neg(F \implies \Box F)
\end{array}
$$

Die Idee zur Abhilfe wäre alle Möglichkeiten durchzuspielen, wie man syntaktisch verschiedene Welten identifizieren könnte. Dies erfordert Tableau-Regeln, die keine Formeln abbauen, sondern die Möglichkeiten aufzählen, wie Weltenterme gleich gemacht werden können.

Dies erlaubt auch einen Tableaukalkül für die Logik S4.3.1, die Logik der deterministischen Zeit, die auf dem Kripke-Rahmen $(N, <)$ basiert, d.h. die Welten sind die natürlichen Zahlen und die Relation ist die $<$-Relation. (Beispiel siehe unten)

Allerdings ist dadurch die Terminierung nicht garantiert.

5.6.3.1 Logische Folgerung

Die Tableaukalküle für die verschiedenen Modallogiken können in der jetzigen Form Tautologien nachweisen, aber nur unvollständig die logische Folgerung $F \models G$, da das Deduktionstheorem nicht gilt (siehe 5.3.6). Ein weiteres Gegenbeispiel zum Deduktionstheorem ist:

Beispiel 5.6.26. *Es gilt $X \models \Box X$, aber $X \implies \Box X$ ist keine Tautologie, wenn X eine Variable ist:*
$X \models \Box X$ ist die Regel der $\Box$-Einführung. Es ist leicht zu sehen, dass das immer gilt. Die Formel $X \implies \Box X$ wird falsch in einer Kripke-struktur (bzw. in einem Rahmen oder einer Logik), wenn es möglich ist, zwei Welten w_1, w_2 zu haben mit $w_1 \; R \; w_2$ und $w_1 \models X$, aber $w_2 \not\models X$. Solche Kripkestrukturen gibt es in allen Logiken, die wir bis jetzt betrachtet haben. Also gilt in keiner dieser Logiken das Deduktionstheorem.

Um $F \models G$ nachzuweisen, kann man versuchen, stattdessen $F \implies G$ nachzuweisen, und hat im Erfolgsfalle auch die Folgerung nachgewiesen. Da das aber nicht immer geht, muss man die Schlussweisen bzw. die entsprechenden Tableaukalküle erweitern:

Zum Nachweis der Folgerung $F \models G$ starte mit $\neg G$ an der Wurzel und benutze die üblichen Tableauregeln.

Folgende Regel kann man ebenfalls verwenden:

(ModFolg) Sei $F_1, \ldots, F_n$ die Menge der Formeln, die man als Voraussetzung hat. Wenn p ein Pfadterm auf dem Tableaupfad P ist, dann erweitere den Pfad mit p, F_i für ein i.

Beispiel 5.6.27. *Zu zeigen ist die Folgerung $\Diamond X \models \Diamond\Diamond X$.*

$$\begin{gathered}
\varepsilon; \neg\Diamond\Diamond X \\
x; \neg\Diamond X \\
x \cdot y; \neg X \\
\varepsilon; \Diamond X \\
a; X \\
a; \Diamond X \\
a \cdot b; X \\
\textit{geschlossen}
\end{gathered}$$

Nach zweimaliger Verwendung der Regel (ModFolg) ist das Tableau geschlossen

5.6.3.2 Normalformen und iterierte Modalitäten.

Die Logik $S5$ hat die interessante Eigenschaft, dass sich alle Formeln in eine Normalform bringen lassen, in der Modaloperatoren nicht im Bereich von anderen Modaloperatoren vorkommen.

Die folgenden Transformationen bewirken diese Normalform:

1. Schiebe alle Negationszeichen nach innen unter Benutzung folgender Vereinfachungen:

$$\begin{array}{lcl} \neg\neg F & \rightarrow & F \\ \neg\Box F & \rightarrow & \Diamond\neg F \\ \neg\Diamond F & \rightarrow & \Box\neg F \\ \neg(F \vee G) & \rightarrow & \neg F \wedge \neg G \\ \neg(F \wedge G) & \rightarrow & \neg F \vee \neg G \end{array}$$

2. Wende folgende Transformationen an:

   a)

$$\begin{array}{lcl} \Box\Box F & \rightarrow & \Box F \\ \Diamond\Box F & \rightarrow & \Box F \\ \Box\Diamond F & \rightarrow & \Diamond F \\ \Diamond\Diamond F & \rightarrow & \Diamond F \end{array}$$

   b)

$$\begin{array}{lcl} \Diamond(F \vee G) & \rightarrow & \Diamond F \vee \Diamond G \\ \Diamond(F \wedge \Diamond G) & \rightarrow & \Diamond F \wedge \Diamond G \\ \Diamond(F \wedge \Box G) & \rightarrow & \Diamond F \wedge \Box G \end{array}$$

   c)

$$\begin{array}{lcl} \Box(F \wedge G) & \rightarrow & \Box F \wedge \Box G \\ \Box(F \vee \Diamond G) & \rightarrow & \Box F \vee \Diamond G \\ \Box(F \vee \Box G) & \rightarrow & \Box F \vee \Box G \end{array}$$

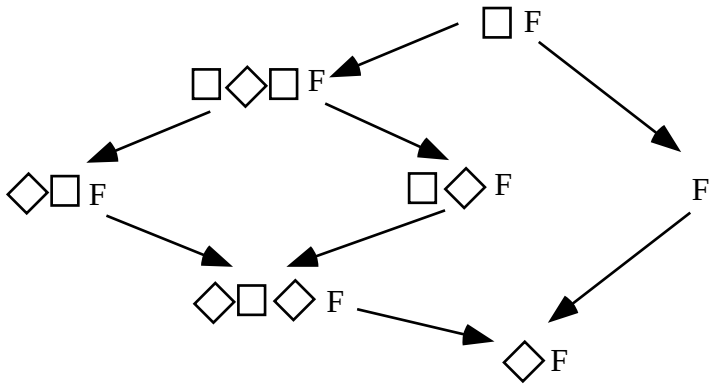
   Wende auch die symmetrischen Regeln an für $\wedge, \vee$

Wenn keine der Transformationen anwendbar und F nicht in modaler Normalform ist, dann liegt einer der Fälle $\Diamond(F \wedge G)$ oder $\Box(F \vee G)$ vor und F und G haben keinen Modaloperator als obersten Operator, aber in F oder G befinden sich noch Modaloperatoren. Es ist keine Einschränkung wenn wir im Falle, dass $\Diamond(F \wedge G)$ vorkommt, annehmen, dass der Operator in F vorkommt und F eine Disjunktion $F_1 \vee F_2$ ist. Wir können ein Distributivgesetz verwenden und erhalten für $\Diamond((F1 \vee F2) \wedge G)$ die Formel $\Diamond((F_1 \wedge G) \vee (F_2 \wedge G))$. Anwenden der Distributivität von $\Diamond$ über $\vee$ ergibt $\Diamond(F_1 \wedge G) \vee \Diamond(F2 \wedge G)$. Im anderen Fall können wir genauso verfahren.

Die Normalformen können unter den Modaloperatoren allerdings beliebige aussagenlogische Ausdrücke enthalten.

In anderen Logik-Varianten wie $S4, D, K$, usw. gibt es keine vergleichbare Normalform, allerdings lassen sich teilweise die iterierten Modalitäten reduzieren.

In $S4$ beispielsweise gibt es nur 7 Modalitäten, für die folgendes Implikationsdiagramm gilt:



5.6.3.3 Anwendungen in der Zeitlogik (temporale Logik)

Die Modaloperatoren bedeuten:

1. in deterministischer (linearer) Zeit: Logik S4.3.1.
   - $\Box F$ von jetzt ab gilt stets F.
   - $\Diamond F$ es gibt einen Zeitpunkt in der Zukunft, an dem F gilt.
2. verzweigende Zeit (nichtdeterministische Prozesse)
   - $\Box F$ von jetzt ab gilt stets F
   - $\Diamond F$ es ist möglich, dass F in der Zukunft gilt. (bzw. Es gibt einen möglichen Ablauf des Prozesses, so dass irgendwann in der Zukunft F gilt.

5.6.3.4 lineare, diskrete, deterministische Zeit

Wir können zum Nachweis von Tautologien den $S4$-Kalkül nehmen.

In linearer Zeit gilt z.B. folgendes Axiom:

$$(\Diamond F \wedge \Diamond G) \implies (\Diamond(F \wedge \Diamond G) \vee \Diamond(F \wedge G) \vee \Diamond(\Diamond F \wedge G)$$

Das entspricht der Aussage, dass zwei Ereignisse E_1, E_2 entweder gleichzeitig, oder E_1 vor E_2 oder E_2 nach E_1.

Normalerweise hat eine Zeitlogik weitere Modaloperatoren für die Vergangenheit, die dual zu $\Box, \Diamond$ sind.

$\Box$	$=$	G	gilt ab jetzt immer
$\Diamond$	$=$	F	gilt irgendwann in der Zukunft
$\Box_P$	$=$	H	galt immer in der Vergangenheit
$\Diamond_P$	$=$	P	galt irgendwann in der Vergangenheit

Als Zusammenhang zwischen Zukunft und Vergangenheit braucht man Axiome, wie z.B. die „K-Axiome der Zeit“:

$$\begin{array}{lcl} \Box(F \implies G) & \implies & (\Box F \implies \Box G) \\ \Box_P(F \implies G) & \implies & (\Box_P F \implies \Box_P G) \\ F & \implies & \Box\Box_P F \\ F & \implies & \Box_P\Box F \end{array}$$

Für lineare Zeit braucht man dann auch das duale Axiom:

$$(\Diamond_P F \wedge \Diamond_P G) \implies (\Diamond_P(F \wedge \Diamond_P G) \vee \Diamond_P(F \wedge G) \vee \Diamond_P(\Diamond_P F \wedge G)$$

Beispiel 5.6.28. *Als Beispiel zeigen wir in einer deterministischen Zeit: Wenn nach Ostern Weihnachten kommt und nach Weihnachten wieder Ostern und ab irgendwann ist immer Weihnachten, dann ist irgendwann danach Ostern:*

$\Box(O \implies \Diamond W) \wedge \Box(W \implies \Diamond O) \implies (\Diamond\Box W \implies \Diamond O)$

Wie nehmen den S4-Tableaukalkül:

$\varepsilon; \neg(\Box(O \implies \Diamond W) \wedge \Box(W \implies \Diamond O) \implies (\Diamond\Box W \implies \Diamond O))$
$\varepsilon; \Box(O \implies \Diamond W)$
$\varepsilon; \Box(W \implies \Diamond O)$
$\varepsilon; \neg(\Diamond\Box W \implies \Diamond O)$
$x, O \implies \Diamond W$
$y, W \implies \Diamond O)$
$\varepsilon; \Diamond\Box W$
$\varepsilon; \neg(\Diamond O)$
$a; \Box W$
$az; W$
$u; \neg O$
$a; W \implies \Diamond O$

$a; \neg W$	$a; \Diamond O$
geschlossen	$ab; O$
da S4 reflexiv	*geschlossen, da S4 transitiv*

Beispiel 5.6.29. $(\Diamond\Box F \wedge \Diamond\Box G) \implies \Diamond\Box(F \wedge G)$

Intuitiv: Wenn ab einem Zeitpunkt stets F gilt, und ab einem Zeitpunkt stets G gilt, dann gibt es auch einen Zeitpunkt, ab dem stets beides, nämlich $F \wedge G$ gilt. Dies gilt nicht für verzweigende Zeit, da dort der Operator $\Diamond$ eine andere Bedeutung hat.

$$
\begin{array}{l}
\varepsilon; \neg((\Diamond\Box F \wedge \Diamond\Box G) \implies \Diamond\Box(F \wedge G)) \\
\varepsilon; \Diamond\Box F \\
\varepsilon; \Diamond\Box G \\
\varepsilon; \neg\Diamond\Box(F \wedge G) \\
a; \Box F \\
b; \Box G \\
ax; F \\
by; G \\
u; \neg\Box(F \wedge G) \\
uc; \neg(F \wedge G)
\end{array}
$$

$a = b$		$a < b$		$b < a$	
$ac; \neg F$	$ac, \neg G$	$bc, \neg F$	$bc, \neg G$	$ac, \neg F$	$ac, \neg G$
geschl.	*geschl.*	*geschl.*	*geschl.*	*geschl.*	*geschl.*

Axiome für dichte Zeit

$$\Diamond F \implies \Diamond\Diamond F$$

kann man ansehen als Formulierung von: zwischen zwei Zeitpunkten gibt es immer noch einen weiteren. Dieses Axiom beschreibt auch gerade die Kripke-Rahmen, die diese Eigenschaft haben.

Axiome für diskrete Zeit

$$
\begin{array}{lcl}
F \wedge \Box_P F & \implies & \Diamond\Box_P F \\
F \wedge \Box F & \implies & \Diamond_P \Box F
\end{array}
$$

6

Konzeptbeschreibungssprachen

In diesem Kapitel behandeln wir Ansätze zur Wissensrepräsentation und Wissensverarbeitung. Der Schwerpunkt liegt dabei auf sogenannten Beschreibungslogiken (Description Logics). Diese haben insbesondere den Vorteil, dass sie eine eindeutig definierte Semantik (innerhalb der Prädikatenlogik) besitzen und viele Inferenzmechanismen und -algorithmen existieren. Zunächst werden wir jedoch kurz auf die historische Entwicklung eingehen, und vorhergehende Ansätze der Semantischen Netze und sogenannter Frames betrachten. Im Anschluss beschäftigen wir uns eingehend mit den Beschreibungslogiken.

6.1 Ursprünge

In diesem Abschnitt stellen wir kurz die (älteren) Formalismen der Semantischen Netze und Frames vor, aus denen (und deren Nachteile) die Beschreibungslogiken entwickelt wurden. Wir geben hier keinen umfassenden Überblick über diese Ansätze (dafür sei auf entsprechende Literatur verwiesen), sondern der Abschnitt ist eher als kurzer Einblick zu sehen.

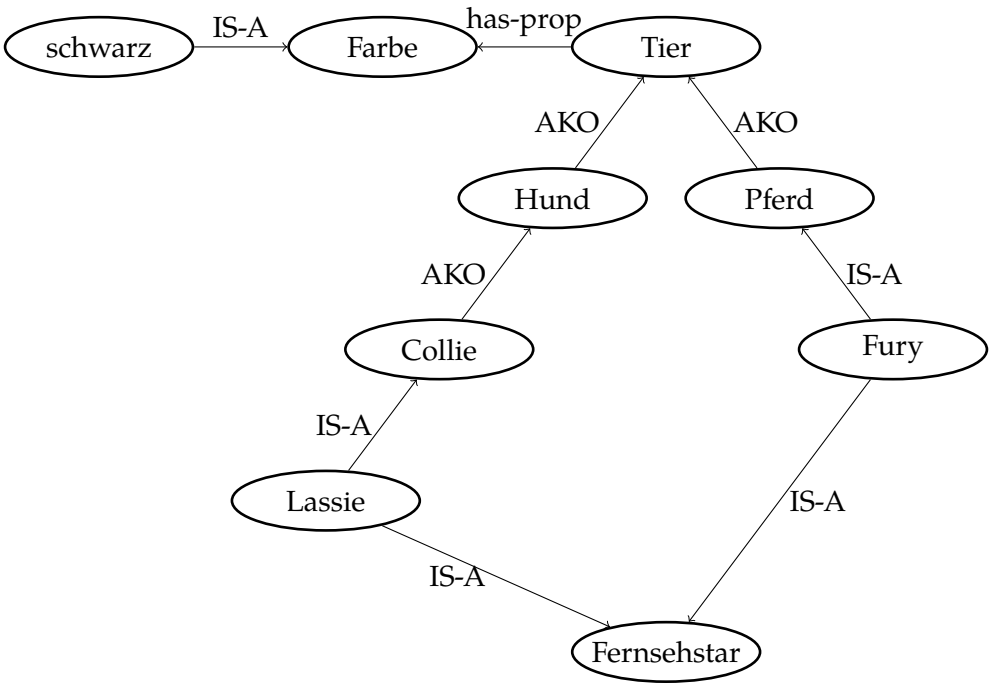
6.1.1 Semantische Netze

Semantische Netze sind ein Formalismus zur Darstellung von Unterbeziehungen und Enthaltensseinsbeziehungen mittels (gerichteten) Graphen. Dabei sind

Knoten markiert mit *Konzepten*, Eigenschaften, (tlw. auch Individuen)

Gerichtete Kanten (Links) geben Beziehungen (Relationen) zwischen Konzepten an. Wesentliche Kanten sind Instanzbeziehungen (IS-A), Unterkonzeptbeziehungen (A-KIND-OF, AKO), Eigenschaften (Prop), Part-of, connected-to, ...

Ein Beispiel ist das folgende Netz zu Tieren:



Allerdings gibt es viele verschiedene Möglichkeiten und Notationen, je nach festgelegtem Formalismen.

Besonderheiten der Semantischen Netze sind:

- Repräsentieren von Beziehungen und Eigenschaften ist möglich.
- Vererbung von Eigenschaften über AKO-links und über IS-A links, je nach Eigenschaft.

Man kann mit semantischen Netzen auch logische Formeln darstellen: UND-Verzweigungen, ODER-Verzweigungen, Quantifizierte Ausdrückebei letzteren entsteht ein Problem der Eindeutigkeit und des Bindungsbereichs.

6.1.1.1 Operationen auf semantischen Netzen:

- Anfragen der Form „Was kann fliegen?“können mittels Matching von Teilgraphen und Beschreibungen bzw. Angaben von Teilnetzen beantwortet werden. D.h. die Suche orientiert sich an bestimmten Knoten oder Kanten oder an ganzen Unterstrukturen.
- Veränderung: Eintragung, Entfernen, Ändern von Kanten und Knoten.
- Operation: Suche nach Verbindungswegen im Netz.

Bemerkung 6.1.1. *Probleme der semantischen Netze:*

- *Die Semantik war nur ungenau definiert.*

- *Jedes Programm arbeitete auf einer anderen semantischen Basis. z.B. was bedeuten jeweils zirkuläre Links ?*
- *Darstellung als Graph wird sehr unübersichtlich für große Netze.*

6.1.2 Frames

Frames sind ein Konzept innerhalb von Repräsentationssprachen, zunächst ohne prozedurale Komponente. Die Grundlagen dazu stammen aus der Kognitionsforschung (Minsky, 1975) und den Scripts (Schenk & Abelson, 1975). Die Frame-Sprachen haben eine Analogie zu Klassen in Objektorientierten Programmiersprachen, allerdings ist die Absicht von Frames nicht ein Programm zu strukturieren, sondern einen (Wissens-)Bereich strukturiert darzustellen.

- Frames beschreiben Klassen oder Instanzen
  - Namen
  - Oberklasse(e)
  - Eigenschaften (Slots)
- Vererbung von Eigenschaften (Slot-Werten) von Oberklassen auf Unterklassen
- Slot:
  - Klasse (Wertebereich)
  - Defaultwerte
  - generische Werte (gilt für alle Instanzen)
  - Bedingungen (z.B. Wertebereichseinschränkungen)
  - Prozedurale Zusätze (z.B. Dämonen, die bei Eintragung eines Slotwertes aktiv werden)

Dies ergibt eine implizite Klassenhierarchie (sog. Prototypen). Zum Beispiel:

Vogel	(Oberklasse: Wirbeltiere)
	(Farbe: Farbe , Gewicht: Zahl, kann-fliegen: Bool, …
	#Beine: 2)
grüne-Vögel	(Oberklasse: Vogel)
	(Farbe:grün)

Bemerkung 6.1.2. *Diese Darstellung hat einige inhärente Probleme zu lösen:*

- *multiple Vererbung*
- *semantische Unterscheidung: Prototyp / individuelles Objekt*
- *logische Operatoren*
- *Semantik von Defaults und überschriebenen Defaults*

6.2 Attributive Konzeptbeschreibungssprachen (Description Logic)

Wir betrachten im folgenden eine Sprachfamilie, die man als Nachfolger von KL-ONE sehen kann. Empfehlenswert als sehr guter Überblick ist das Handbuch (Baader et al., 2010) und zur Komplexität der Artikel (Donini et al., 1997).

Man nennt die Konzeptbeschreibungssprachen auch *Terminologische Sprachen*. Sie haben sich entwickelt aus einer Sprache KL-ONE (Brachman, ca. 1980), die zur Repräsentation und Verarbeitung der Semantik von natürlichsprachlichen Ausdrücken entwickelt wurde. Es gibt ähnliche Strukturen bei der automatischen Verarbeitung der Syntax von natürlichsprachlichen Ausdrücken: Features (Attribute, bzw. Merkmalsstrukturen) (siehe Shieber: unification grammars).

Man kann die Konzeptbeschreibungssprachen als Weiterentwicklung von semantischen Netzen und Frames sehen, wobei sie den Vorteil einer eindeutigen und deklarativ definierten Semantik haben und nicht zu mächtige Konstrukte (wie z.B. Prozeduren) zulassen.

Neuere Entwicklungen sind (relativ schnelle) und vollständige Schlussfolgerungsmechanismen, Erweiterung in viele Richtungen wie Zeitrepräsentation, Kombination mit anderen Formalismen.

Ein auf Beschreibungslogik basierendes Wissensverarbeitungssystem muss Möglichkeiten bieten, die Wissensbasis darzustellen und zu verändern, sowie neue Schlüsse mithilfe von Inferenzmechanismen aufgrund der Wissenbasis zu ziehen.

Die Wissensbasis besteht dabei aus den beiden Komponenten T-Box und A-Box. Die T-Box legt *Terminologie* des Anwendungsbereichs fest, d.h. das Vokabular, das verwendet werden soll. Die A-Box legt *Annahmen* (Assertions) über die Individuen (unter Verwendung der in der T-Box gegebenen Terminologie) fest. Verglichen mit Datenbanksystemen, legt die T-Box das Datenbankschema fest, und die A-Box die eigentlichen Daten.

Das Vokabular der T-Box besteht aus *Konzepten* und *Rollen*. Konzepte repräsentieren dabei Menge von Individuen, Rollen stehen für binäre Relationen zwischen Individuen. Atomare Konzepte und atomare Rollen sind nur Bezeichner, d.h. sie werden durch ihren Namen repräsentiert und später semantisch als Mengen bzw. Relationen interpretiert. Neben atomaren Konzepten und Rollen gibt es in allen Beschreibungslogiken Konstrukte, um *komplexe* Beschreibungen von Konzepten und Rollen auszudrücken (dies sind Formeln, die als Atome dann gerade atomare Konzepte und Rollen verwenden). Innerhalb der T-Box werden diese komplexen Konzepte und Rollen definiert und mit neuen Namen versehen. D.h. im Grunde besteht die T-Box aus *atomaren* Bezeichnern und *definierten* Namen.

Je nach verfügbaren Konstrukten zur Konstruktion von komplexen Konzepten und Rollen werden die verschiedenen Beschreibungslogiken unterschieden. Im Wesentlich gibt es hierbei zwei Abwägungen: Je mehr Konstrukte verfügbar sind, desto einfacher (oder gar mehr) lässt sich Wissen ausdrücken, andererseits gilt jedoch auch: Je mehr Konstrukte

vorhanden sind, umso komplexer sind die Inferenzmechanismen. Schon in relativ ausdrucksschwachen Beschreibungslogiken können Probleme untentscheidbar oder nichtpolynomiell sein.

Typische Fragestellungen, die ein Wissensverarbeitungssystem beantworten sollte, sind: Sind die Beschreibungen innerhalb der T-Box erfüllbar (also nicht widersprüchlich), gibt es Beschreibungen die in anderen Beschreibungen enthalten sind (üblicherweise als Subsumption bezeichnet).

Bezüglich der A-Box ist eine wichtige Fragestellung, ob die Daten konsistent sind, d.h. gibt es ein Modell, dass die Annahmen über die Individuen einhält.

Bevor wir uns wieder mit T-Box und A-Box im speziellen beschäftigen, führen wir verschiedene Konzeptbeschreibungssprachen ein. Hierfür konzentrieren wir uns auf die Konstruktion und Semantik von Konzept- und Rollenbeschreibungen.

6.2.1 Die Basis-Sprache $\mathcal{AL}$

Atomare Beschreibungen sind atomare Konzepte und atomare Rollen. Komplexe Beschreibungen werden durch Konzeptkonstruktoren erstellt. Wir verwenden A, B für atomare Konzepte, R für atomare Rollen und C, D für komplexe Konzepte. Die Sprache $\mathcal{AL}$ gilt als eine minimale Sprache von praktischem Interesse. Komplexe Konzeptbeschreibungen werden durch *Konzept-Terme* gebildet (siehe (Schmidt-Schauß & Smolka, 1991)).

Definition 6.2.1 (Syntax von Konzept-Termen in $\mathcal{AL}$)**.** *Die Syntax von Konzepttermen von $\mathcal{AL}$ ist durch die folgende Grammatik definiert, wobei A atomares Konzept, C, D Konzepte und R atomare Rolle:*

$$
\begin{array}{rcll}
C, D \in \mathcal{AL} & ::= & A & \textit{atomares Konzept} \\
& | & \top & \textit{universelles Konzept} \\
& | & \bot & \textit{leeres Konzept} \\
& | & \neg A & \textit{atomares Komplement} \\
& | & C \sqcap D & \textit{Schnitt} \\
& | & \forall R.C & \textit{Wert-Einschränkung} \\
& | & \exists R.\top & \textit{beschränkte existentielle Einschränkung}
\end{array}
$$

Beachte das in $\mathcal{AL}$ die Komplementbildung nur für atomare Konzepte erlaubt ist, und dass für die existentielle Einschränkung nur das universelle Konzept erlaubt ist (d.h. $\exists R.C$ für beliebiges Konzept C ist nicht erlaubt).

Beispiel 6.2.2. *Nehmen wir an* Person *und* Weiblich *seinen atomare Konzepte. Dann drückt das Konzept* Person $\sqcap$ Weiblich *gerade weibliche Personen aus, während das Konzept* Person $\sqcap \neg$Weiblich *alle nicht weiblichen Personen ausdrückt.*

Sei hatKind *eine atomare Rolle. Dann können wird die Konzepte* Person $\sqcap \exists$hatKind.$\top$ *und* Person $\sqcap \forall$hatKind.Weiblich *konstruieren. Das erste Konzept beschreibt Personen, die Kinder ha-*

ben. Evtl. sollten wir das Konzept verfeinern zu Person $\sqcap \exists$hatKind.$\top \sqcap \forall$hatKind.Person, *um sicherzustellen, dass alle Kinder auch Personen sind.*

Das zweite Konzept beschreibt Personen, deren Kinder alle weiblich sind. Anstelle des ersten Konzepts können kinderlose Personen durch das Konzept Person$\sqcap\forall$hatKind.$\bot$ *beschrieben werden.*

Eigentlich können wir die Beispiele jedoch noch gar nicht interpretieren, da uns noch die *Semantik* der Beschreibungen fehlt. Das holen wir jetzt nach. Wie üblich definieren wir eine *Interpretation*:

Definition 6.2.3 (Semantik von $\mathcal{AL}$)**.** *Eine* Interpretation I *einer $\mathcal{AL}$-Formel legt folgendes fest:*

- *Eine nichtleere Menge Δ von* Objekten.
- *Für jedes atomare Konzept A: $I(A)$ als Teilmenge von Δ, d.h. $I(A) \subseteq \Delta$.*
- *Für jede atomare Rolle R: $I(R)$ als binäre Relation über Δ, d.h. $I(R) \subseteq \Delta \times \Delta$.*

Die Erweiterung einer Interpretation I auf komplexe Konzeptbeschreibungen ist induktiv durch die folgenden Fälle definiert.

$$\begin{array}{lcl} I(C_1 \sqcap C_2) & = & I(C_1) \cap I(C_2) \\ I(\bot) & = & \emptyset \\ I(\top) & = & \Delta \\ I(\neg A) & = & \Delta \setminus I(A) \\ I(\exists R.\top) & = & \{x \in \Delta \mid \exists y.(x,y) \in I(R)\} \\ I(\forall R.C) & = & \{x \in \Delta \mid \forall y.(x,y) \in I(R) \Rightarrow y \in I(C)\} \end{array}$$

Zwei Konzepte C, D sind äquivalent, *geschrieben als $C \equiv D$, genau dann wenn $I(C) = I(D)$ für alle Interpretationen I gilt*

Beispiel 6.2.4. *Sei I die Interpretation mit $\Delta = \{Marie, Horst, Susi, Fritz, Lassie\}$, I*(Person) $= \Delta \setminus \{Lassie\}$, I(Weiblich) $= \{Marie, Susi, Lassie\}$ *und I*(hatKind) $=$

$\{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\}$. *Dann können wir ausrechnen:*

$$I(\mathsf{Person} \sqcap \mathsf{Weiblich}) = I(\mathsf{Person}) \cap I(\mathsf{Weiblich}) = \{Marie, Susi\}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \exists \mathsf{hatKind}.\top) \\
&= I(\mathsf{Person}) \cap I(\exists \mathsf{hatKind}.\top) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \exists y.(x,y) \in I(\mathsf{hatKind})\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \exists y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\}\} \\
&= I(\mathsf{Person}) \cap \{Fritz, Marie\} \\
&= \{Fritz, Marie\}
\end{aligned}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \forall \mathsf{hatKind}.\mathsf{Weiblich}) \\
&= I(\mathsf{Person}) \cap I(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Weiblich})\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\} \\
&\qquad\qquad \Rightarrow y \in \{Marie, Susi, Lassie\}\} \\
&= I(\mathsf{Person}) \cap \{Marie, Horst, Susi, Lassie\} \\
&= \{Marie, Horst, Susi\}
\end{aligned}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \forall \mathsf{hatKind}.\bot) \\
&= I(\mathsf{Person}) \cap I(\forall \mathsf{hatKind}.\bot) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\bot)\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\} \Rightarrow y \in \emptyset\} \\
&= I(\mathsf{Person}) \cap \{Horst, Susi, Lassie\} \\
&= \{Horst, Susi\}
\end{aligned}$$

Beispiel 6.2.5. *Es gilt* $(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \sqcap (\forall \mathsf{hatKind}.\mathsf{Student})$ *und* $\forall \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Student})$ *sind äquivalente Konzepte. Das lässt sich mit der Semantik nachweisen:*

$$\begin{aligned}
& I(\forall \mathsf{hatKind}.\mathsf{Weiblich} \sqcap \forall \mathsf{hatKind}.\mathsf{Student}) \\
= \; & I(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \cap I(\forall \mathsf{hatKind}.\mathsf{Student}) \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Weiblich})\} \\
& \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Student})\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow (y \in I(\mathsf{Weiblich}) \wedge y \in I(\mathsf{Student}))\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in (I(\mathsf{Weiblich}) \cap I(\mathsf{Student}))\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in (I(\mathsf{Weiblich} \sqcap \mathsf{Student}))\} \\
= \; & I(\forall \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Student}))
\end{aligned}$$

Beachte, der Beweis ist völlig unabhängig von den atomaren Konzepten und Rollen, d.h. es gilt allgemein:

$$(\forall R.C) \sqcap (\forall R.D) \;\equiv\; \forall R.(C \sqcap D)$$

6.2.2 Allgemeinere Konzept-Definitionen

Wir betrachten nun mögliche Erweiterungen der Basissprache $\mathcal{AL}$. Eine erste Erweiterung ist das Verwenden von *atomaren Funktionen*, die auch als *Attribute*, *Features*, *Attributsbezeichner* oder *funktionale Rollen* bezeichnet werden. Ihre Verwendung entspricht der Verwendung von Rollen, d.h. sie dürfen überall dort verwendet werden, wo auch Rollen erlaubt sind. Der Unterschied liegt in der Semantik: Jede Interpretation darf eine atomare Funktion nur als (partielle) *einstellige Funktion* auf Δ, also als Funktion $f : \Delta \to \Delta$ definieren, was äquivalent dazu ist, dass es f eine *rechtseindeutige* binäre Relation ist[1]. Eine echte Änderung der Syntax beim Einführen der atomaren Funktionen ist nicht notwendig, lediglich ist eine disjunkte Aufteilung der Namen in: Namen für Konzepte, Namen für Rollen und Namen für atomare Funktionen notwendig.

Beispiel 6.2.6. *Beispiele für atomare Funktionen sind* istMutterVon (*was wohl einer totalen Funktion entsprechen sollte*), *und* istEhepartner (*was i.A. einer partiellen Funktion entspricht*).

Wir betrachten weitere Konstrukte, die in Konzeptbeschreibungssprachen (neben den bereits eingeführten) verwendet werden, um komplexe Konzeptbeschreibungen C zu bilden.

Definition 6.2.7. *Die Syntax zur Bildung von komplexen Konzeptbeschreibungen aus Definition 6.2.1 kann erweitert werden:*

$C ::=$	$\ldots$	
$\mid$	$\neg C$	*Komplement*
$\mid$	$(C_1 \sqcup C_2)$	*Vereinigung*
$\mid$	$(\exists R.C)$	*existenzielle Einschränkung*
$\mid$	$(\leq\ n\ R)$, $(\geq\ n\ R)$	*Anzahlbeschränkungen* (*number restrictions*)
$\mid$	$(\leq n\ R.C)$, $(\geq\ n\ R.C)$	*qualifizierte Anzahlbeschränkung.*
$\mid$	$(R_1; R_2; \ldots; R_n = R'_1; R'_2; \ldots; R'_m)$	*Pfadgleichungen, Attributsübereinstimmung*

Es gibt auch Konstrukte, die es erlauben *komplexe Rollen* zu konstruieren, ein solches Konstrukt ist

$$(\text{RESTRICT } R\ C) \quad \text{Rolleneinschränkung}$$

Wir werden später weitere solcher Konstrukte behandeln.

In der Literatur werden teilweise andere Symbole verwendet, z.B.:

[1] D.h. für alle $x, y, z \in \Delta$: Wenn $(x, y) \in f$ und $(x, z) \in f$ muss gelten $y = z$

AND	entspricht	$\sqcap$
OR	entspricht	$\sqcup$
NOT	entspricht	$\neg$
SOME	entspricht	$\exists$
ALL	entspricht	$\forall$
...		

Beispiel 6.2.8. *Wir geben einige Beispiele, was man in diesen Sprachen ausdrücken kann. Wenn man die Konzeptnamen* Mensch, Frau, Mann *und keine weiteren Axiome hat, sind die Beziehungen zwischen* Mensch, Frau *und* Mann *ziemlich frei. Als Axiome würde man sich wünschen* $\mathsf{Frau} \sqcap \mathsf{Mann} = \bot$, *d.h. die beiden Konzepte sollen disjunkt sein; und* $\mathsf{Mensch} \equiv \mathsf{Frau} \sqcup \mathsf{Mann}$, *d.h. Menschen sind entweder Mann oder Frau.*

Mittels der Relationen kann man weitere Konzepte definieren:

$$\mathsf{Eltern} := \mathsf{Mensch} \sqcap (\exists \mathsf{hatKind}.\mathsf{Mensch})$$

Da das noch Freiheiten lässt, ist folgende Definition genauer

$$\mathsf{Eltern} := \mathsf{Mensch} \sqcap (\exists \mathsf{hatKind}.\mathsf{Mensch}) \sqcap (\forall \mathsf{hatKind}.\mathsf{Mensch})$$

Desweiteren kann man definieren:

$$\mathsf{Mutter} := \mathsf{Frau} \sqcap \mathsf{Eltern}$$

Hier kann das Wissensrepräsentationssystem in Prinzip herausfinden, dass gilt $I(\mathsf{Mutter}) \subseteq I(\mathsf{Frau})$. *Eine Studentin könnte man definieren durch*

$$\mathsf{Studentin} := \mathsf{Frau} \sqcap (\exists \mathsf{studiertFach}.\top)$$

Anzahlbeschränkungen kann man verwenden z.B. in

$$\begin{aligned} \mathsf{Bigamist} \quad := \quad & \mathsf{Mann} \sqcap (\geq\ 2\ \mathsf{verheiratetMit}) \\ & \sqcap (\leq\ 2\ \mathsf{verheiratetMit}) \sqcap (\forall \mathsf{verheiratetMit}.\mathsf{Frau}) \end{aligned}$$

wenn verheiratetMit *eine Rolle ist. Ist* verheiratetMit *sowieso eine atomare Funktion, dann sollte die Semantik (die wir gleich definieren) liefern:* $I(\mathsf{Bigamist}) = \emptyset$ *für jede Interpretation* I.

Sei isstGerne *eine Rolle, dann beschreibt das Konzept*

$$\mathsf{Mensch} \sqcap (\mathsf{verheiratetMit}; \mathsf{isstGerne} = \mathsf{isstGerne})$$

gerade alle Menschen deren Ehepartner das gleiche Essen mögen, wie sie selbst.

Verwendet man atomare Rollen und Pfadgleichungen, so beschreibt das Konzept

$$\mathsf{Mann} \sqcap (\mathsf{hatNachnamen} = \mathsf{hatMutter}; \mathsf{hatNachnamen})$$

alle Männer, die den selben Nachnamen wie ihre Mutter haben (*wobei* hatNachnamen *und* hatMutter *atomare Funktionen sind*).

Man erkennt, dass der Formalismus die Konzeptbildung mittels Vereinigung, Schnitt, Komplement, und über Eigenschaften bzw. Relationen erlaubt.

6.2.3 Semantik von Konzepttermen

Für Konzeptterme in der erweiterten Syntax erweitern wir die modulare, deklarative Semantik:

Definition 6.2.9 (Semantik von Konzepttermen)**.** *Eine Interpretation ist analog zu Definition 6.2.3 definiert, wobei I neben atomaren Konzepten und atomaren Rollen, die Bedeutung der atomaren Funktionen als* partielle Funktionen $f : \Delta \to \Delta$ *festlegt.*

Die Interpretation wird auf die neuen Konstrukte wie folgt erweitert:

$$
\begin{aligned}
I(C_1 \sqcup C_2) &= I(C_1) \cup I(C_2) \\
I(\exists R.C) &= \{x \in \Delta \mid \exists y.(x,y) \in I(R) \text{ und } y \in I(C)\} \\
I(\leq\ n\ R) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R)\}| \leq n\} \\
I(\geq\ n\ R) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R)\}| \geq n\} \\
I(\leq\ n\ R.C) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R) \wedge y \in I(C)\}| \leq n\} \\
I(\geq\ n\ R.C) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R) \wedge y \in I(C)\}| \geq n\} \\
I(R_1; R_2; \ldots; R_n = R'_1; R'_2; \ldots; R'_m) &= \left\{ x \in \Delta \mid \begin{array}{r} \{y \in \Delta \mid (x,y) \in (I(R_1) \circ \ldots \circ I(R_n))\} \\ = \{y \in \Delta \mid (x,y) \in (I(R'_1) \circ \ldots \circ I(R'_m))\} \end{array} \right\} \\
&\quad \text{wobei } \circ \text{ die Komposition von Relationen ist}^1 \\
I(\mathit{RESTRICT}\ R\ C) &= \{(x,y) \in \Delta \times \Delta \mid (x,y) \in I(R) \text{ und } y \in I(C)\}
\end{aligned}
$$

Diese Semantik erlaubt es, Gleichheiten bzw. Untermengenbeziehungen von Konzepten zu zeigen.

Beispiel 6.2.10. *Wir zeigen* $(\exists R.C) \equiv (\exists(\mathit{RESTRICT}\ R\ C).\top)$*. Sei I eine Interpretation. Dann gilt für die linke Seite:*

$$I(\exists R.C) = \{x \mid \exists y.(x,y) \in I(R) \wedge y \in I(C)\}$$

Die rechte Seite hat als Interpretation:

$$
\begin{aligned}
& \{x \mid \exists y.(x,y) \in I(\mathit{RESTRICT}\ R\ C)\} \\
= \ & \{x \mid \exists y.(x,y) \in \{(a,b) \mid (a,b) \in I(R) \wedge b \in I(C)\} \\
= \ & \{x \mid \exists y.(x,y) \in \{(a,b) \in I(R) \mid b \in I(C)\} \\
= \ & \{x \mid \exists y.(x,y) \in I(R) \wedge y \in I(C)\}
\end{aligned}
$$

[1] D.h. $R_1 \circ R_2 = \{(x,z) \mid (x,y) \in R_1 \wedge (y,z) \in R_2\}$

Das Beispiel zeigt auch, dass man voll existentielle Einschränkung nicht braucht, wenn RESTRICT und die beschränkte existentielle Einschränkung vorhanden sind.

6.2.4 Namensgebung der Familie der $\mathcal{AL}$-Sprachen

Die Namensgebung vieler Erweiterung der Basissprache $\mathcal{AL}$ erfolgt durch Anhängen weiterer Buchstaben, d.h. man verwendet als mögliche Sprachbezeichner

$$\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}]$$

wobei die Buchstaben für die entsprechenden zu $\mathcal{AL}$ hinzugefügten Konstrukte stehen:

$\mathcal{U}$	Union: Vereinigung ($\sqcup$)
$\mathcal{E}$	Volle existenzielle Beschränkung ($\exists R.C$)
$\mathcal{N}$	number restriction: Anzahlbeschränkung. $((\leq\ n\ R), (\geq\ n\ R))$
$\mathcal{C}$	Volles Komplement ($\neg C$)

Z.B. ist $\mathcal{ALEN}$ die Sprache, die auf $\mathcal{AL}$ aufbaut und noch allgemeine existenzielle Beschränkung und Anzahlbeschränkung hat.

Nicht alle diese Namen bezeichnen verschiedene Sprachen aus semantischer Sicht. Denn z.B. ist semantisch $\mathcal{ALUE} = \mathcal{ALC}$, denn es gilt

$$C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$$

und

$$\exists R.C \equiv \neg(\forall R.\neg C).$$

D.h. alles was in $\mathcal{ALUE}$ ausgedrückt werden kann, kann auch in $\mathcal{ALC}$ ausgedrückt werden und umgekehrt.

Wir beweisen die letzte Gleichung:

$$\begin{aligned}
I(\neg(\forall R.\neg C)) &= \Delta \setminus \{a \in \Delta \mid \forall b.(a,b) \in I(R) \Rightarrow y \in I(\neg C)\} \\
&= \Delta \setminus \{a \in \Delta \mid \forall b.(a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C))\} \\
&= \{a \in \Delta \mid \neg(\forall b.(a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.\neg((a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.(a,b) \in I(R) \wedge \neg(b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.(a,b) \in I(R) \wedge (b \in I(C))\} \\
&= I(\exists R.C)
\end{aligned}$$

Deshalb kann man Vereinigung $\mathcal{U}$ und volle existenzielle Beschränkung durch $\mathcal{AL}$ mit Komplementen ausdrücken. Umgekehrt kann man volle Negation durch Vereinigung $\mathcal{U}$ und volle existenzielle Beschränkung ausdrücken, da man die Normalform darstellen kann.

Insgesamt erhält man 8 Sprachen: $\mathcal{AL}, \mathcal{ALC}, \mathcal{ALCN}, \mathcal{ALN}, \mathcal{ALU}, \mathcal{ALE}, \mathcal{ALUN}, \mathcal{ALEN}$. Wir geben eine Tabelle der Konstrukte an, die in diesen Sprachen erlaubt ist:

Name	Konstrukte	implizite Konstrukte
$\mathcal{AL}$	$\sqcap, \bot, \top, \neg A, \forall R.C, \exists R.\top$	
$\mathcal{ALC}$	$\mathcal{AL}$, $\neg C$	$\sqcup$,$\exists R.C$
$\mathcal{ALCN}$	$\mathcal{AL}$, $\neg C$, $(\leq\ n\ R)$	$\sqcup$, $\exists R.C$, $(\geq\ n\ R)$
$\mathcal{ALN}$	$\mathcal{AL}$, $(\leq\ n\ R)$ $(\geq\ n\ R)$	
$\mathcal{ALU}$	$\mathcal{AL}$, $\sqcup$	
$\mathcal{ALE}$	$\mathcal{AL}$, $\exists R.C$	
$\mathcal{ALUN}$	$\mathcal{AL}$, $\sqcup$, $(\leq\ n\ R)$, $(\geq\ n\ R)$	
$\mathcal{ALEN}$	$\mathcal{AL}$, $\exists R.C$, $(\leq\ n\ R)$, $(\geq\ n\ R)$	

Die Sprachen $\mathcal{ALC}$ und $\mathcal{ALCN}$ spielen eine besondere Rolle, da sie sehr allgemein sind. Man kann z.B. so viel repräsentieren wie Aussagenlogik, denn Schnitte, Vereinigungen und Komplemente sind erlaubt (was gerade zum Kodieren von Konjunktion, Disjunktion und Negation in der Aussagenlogik verwendet werden kann).

Es gibt noch weitere Konzeptsprachen, die aufgrund der Historie andere Namen (ungleich vom Schema $\mathcal{AL}$+Erweiterung) haben. Diese werden mit $\mathcal{FL}$ (für frame-based description language) abgekürzt (Levesque & Brachman, 1987) (Levesque, und Brachman Expressiveness and tractability in knowledge representation and reasoning)
Es gibt drei Varianten:

Name	Konstrukte
$\mathcal{FL}_0$	$\sqcap, \forall R.C$
$\mathcal{FL}^-$	$\sqcap, \forall R.C, \exists R.\top$
$\mathcal{FL}$	$\sqcap, \forall R.C, \exists R.C$

Es gelten folgende Beziehungen zwischen den $\mathcal{FL}$-Sprachen und den $\mathcal{AL}$-Sprachen:

$$\begin{aligned} \mathcal{AL} &\equiv \mathcal{FL}^- \cup \{\neg A, \top\} \\ \mathcal{ALC} &\equiv \mathcal{FL}^- \cup \{\neg C\} \\ \mathcal{ALC} &\equiv \mathcal{FL} \cup \{\neg C\} \end{aligned}$$

Ebenso gibt es noch eine Erweiterung der $\mathcal{ALUEN}$-Sprachen um den Schnitt von Rollen, d.h. die Syntax ist erweitert, so dass man dort, wo sonst nur elementare Rollen verwendet werden, den Schnitt von mehreren elementaren Rollen verwenden darf.

$$\mathcal{R}: \quad R_1 \sqcap R_2 \quad \text{Schnitt von Rollen}$$

Die Namemsgebung ist dann entsprechend:

$$\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}][\mathcal{R}]$$

Dies ergibt 8 weitere Sprachen in Erweiterung der $\mathcal{ALUEN}$-Sprachen. Es gelten weitere Sprach-Äquivalenzen: Da $\mathcal{C}$ immer $\mathcal{U}$ und $\mathcal{E}$ impliziert,
ist z.B. $\mathcal{ALUCR} = \mathcal{ALCR} = \mathcal{ALECR}$

6.2.5 Inferenzen und Eigenschaften

Wir definieren die interessanten Eigenschaften für Konzepte und zwischen Konzepte etwas formaler. Diese Eigenschaften möchte man mit Inferenzverfahren nachweisen.

Definition 6.2.11. *Seien C, D (evtl. komplexe) Konzepte*

- *Ein Konzept D* subsumiert *ein Konzept C (geschrieben als $C \sqsubseteq D$), gdw. für alle Interpretationen I gilt: $I(C) \subseteq I(D)$.*
- *Ein Konzept C ist* konsistent *gdw. es eine Interpretation I gibt, so dass gilt: $I(C) \neq \emptyset$. Gibt es keine solche Interpretation, so nennt man C* inkonsistent.
- *Zwei Konzepte C und D sind* disjunkt, *gdw. für alle Interpretationen I gilt: $I(C) \cap I(D) = \emptyset$.*
- *Zwei Konzepte C und D sind* äquivalent *($C \equiv D$) gdw. für alle Interpretationen I gilt: $I(C) = I(D)$.*

Beachte, dass C konsistent nicht bedeutet, dass C in allen Interpretationen nicht leer ist, sondern dass eine Interpretation I mit $I(C) \neq \emptyset$ ausreichend ist.

Man kann je nach Sprache die verschiedenen Fragestellungen ineinander überführen:

- C ist inkonsistent, gdw. $C \equiv \bot$.
- $C \equiv D$ gdw. $C \sqsubseteq D$ und $D \sqsubseteq C$.
- C ist disjunkt zu D gdw. $C \sqcap D$ inkonsistent.
- C inkonsistent, gdw. C wird von $\bot$ subsumiert ($C \sqsubseteq \bot$)
- Wenn allgemeine Komplemente erlaubt, dann gilt:
  $C \sqsubseteq D$ gdw. $C \sqcap \neg D$ inkonsistent ist.
- Wenn allgemeine Komplemente erlaubt, dann gilt:
  $C \sqsubseteq D$ gdw. C und $\neg D$ disjunkt sind.

Wenn die Sprache allgemeine Komplemente erlaubt, dann sind Subsumtion, Disjunktheitstest und Konsistenztest äquivalent und haben auch gleiche Komplexität. Dies gilt z.B. in der Sprache $\mathcal{ALC}$.

In der Sprache $\mathcal{FL}$ sind das aber verschiedene Fragestellungen.

6.2.6 Anwendungen der Beschreibungslogiken

In Ontologien der Life-Sciences (Medizin, Biologie) gibt es Anwendungsmöglichkeiten von Beschreibungslogiken zur Konsistenzprüfung. Einige Beispiele sind:

- SNOMED CT (ein Akronym für Systematized Nomenclature of Medicine – Clinical Terms) ist eine medizinische Nomenklatur, die ca. 500.000 Konzepte umfasst
- The National Cancer Institute Thesaurus , 45.000 Konzepte
- GO: gene ontology: ca. 20.000 Konzepte
- „ GALEN was concerned with the computerisation of clinical terminologies."

Insbesondere gibt es folgende Anwendungen einer ausdrucksschwachen Beschreibungslogik $\mathcal{EL}$:

- Überprüfung der Konsistenz einer Menge von Konzepten
- Überprüfung der Erweiterbarkeit einer Menge von Konzepten.

Die Beschreibungslogik $\mathcal{EL}$ hat als Syntax der Konzepte:

$$C ::= \top \mid A \mid C \sqcap D \mid \exists R.C$$

Die Subsumption in $\mathcal{EL}$ ist polynomiell und ist damit anwendungsgeeignet.
Ein Beispiel für eine Subsumtionsbeziehung ist:

$$\exists \mathsf{child}.\mathsf{Rich} \sqcap \exists \mathsf{child}.(\mathsf{Woman} \sqcap \mathsf{Rich}) \equiv \exists \mathsf{child}.(\mathsf{Woman} \sqcap \mathsf{Rich})$$

Als weiteres (eigentlich falsches) Beispiel wurde mittels Subsumption erkannt:

$$\text{„}\mathsf{Finger\text{-}Amputation} \sqsubseteq \text{Arm-Amputation"},$$

was von den anderen Konzepten impliziert wurde, aber so nicht gemeint war und zu Korrekturen führte.

6.3 T-Box und A-Box

6.3.1 Terminologien: Die T-Box

Eine Terminologie ist normalerweise eine Vereinbarung von Namen für Konzepte. In der allgemeinsten Form in $\mathcal{AL}$ ist es eine Menge von terminologischen Axiomen der Formen

$$C \sqsubseteq D \text{ oder } C \equiv D$$

wobei C, D Konzepte sind.
Wenn es Rollenterme gibt, dann können die Axiome auch die Form

$$R \sqsubseteq S \text{ oder } R \equiv S$$

haben, wobei R, S Rollen sind. Es handelt sich um Inklusionen und um Gleichheiten.

Definition 6.3.1. *Gegeben eine Menge T von terminologischen Axiomen. Dann* erfüllt *eine Interpretation I diese Axiome, wenn für alle Axiome*

- *$C \sqsubseteq D$ (bzw. $R \sqsubseteq S$) $\in T$ die Gleichung $I(C) \subseteq I(D)$ (bzw. $I(R) \subseteq I(S)$) gilt und*
- *für alle Axiome $C \equiv D$ (bzw. $R \equiv S$) die Gleichung $I(C) = I(D)$ (bzw. $I(R) = I(S)$) gilt.*

Wenn I die Menge der Axiome T erfüllt, dann sagen wir I ist ein Modell *für T.*

Die einfachste Variante einer Menge von terminologischen Axiomen ist eine Menge von *Definitionen*. D.h. Gleichungen der Form $A = C$ (bzw. (R = S), wobei A (bzw. R) ein (symbolischer) Name ist und C ein Konzeptterm (bzw. S ein Rollenterm. Wenn zusätzlich jeder definierte Name höchstens einmal auf der linken Seite einer Definition auftaucht, dann spricht man von einer **T-Box**. Dies entspricht dem Sprachgebrauch von KL-ONE[2].

Eine T-Box nennt man *zyklisch*, wenn es einen Namen gibt, der (evtl. implizit) durch sich selbst definiert ist. Ansonsten nennt man die T-Box azyklisch.

Eine azyklische T-Box ist dadurch gekennzeichnet, dass man alle definierten Namen in rechten Seiten von Axiomen durch Definitionseinsetzung eliminieren kann. D.h. man kann die T-Box selbst eliminieren und nur mit Konzepttermen arbeiten. Der Nachteil, den man sich erkauft, ist die mögliche exponentielle Vergrößerung der Konzeptterme durch die Einsetzung.

Wenn keine zyklischen Definitionen vorliegen, kann man den symbolischen Namen eine eindeutige Interpretation I geben, wenn man bereits eine Interpretation I_0 der (nicht-definierten) atomaren Symbole gegeben hat.

Wenn zyklische Definitionen vorliegen, geht das nur noch unter einschränkenden Bedingungen.

Beispiel 6.3.2. *Beispiel für eine Terminologie (T-Box):*

[2] Ein von Brachman beschriebenes Wissensrepräsentationssystem für die Verarbeitung natürlichsprachlicher Ausdrücke

$$
\begin{array}{lcl}
\mathsf{Frau} & \equiv & \mathsf{Person} \sqcap \mathsf{Weiblich} \\
\mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\
\mathsf{Mutter} & \equiv & \mathsf{Frau} \sqcap \exists\mathsf{hatKind.Person} \\
\mathsf{Vater} & \equiv & \mathsf{Mann} \sqcap \exists\mathsf{hatKind.Person} \\
\mathsf{Eltern} & \equiv & \mathsf{Vater} \sqcup \mathsf{Mutter} \\
\mathsf{Grossmutter} & \equiv & \mathsf{Mutter} \sqcap \exists\mathsf{hatKind.Eltern} \\
\mathsf{MutterMitVielenKindern} & \equiv & \mathsf{Mutter} \sqcap (\geq\ 3\ \mathsf{hatKind}) \\
\mathsf{MutterMitTochter} & \equiv & \mathsf{Mutter} \sqcap (\forall\mathsf{hatKind.Frau}) \\
\mathsf{Ehefrau} & \equiv & \mathsf{Frau} \sqcap (\exists\mathsf{hatEhemann.Mann})
\end{array}
$$

Da diese T-Box azyklisch ist und nur aus Definitionen besteht, kann man stets aus einer Interpretation I_0, die nur die Konzeptnamen Person, Weiblich, hatKind *und* hatEhemann *interpretiert, ein Modell I für die T-Box erzeugen, indem man setzt $I(\mathsf{Person}) := I_0(\mathsf{Person})$, $I(\mathsf{Weiblich}) := I_0(\mathsf{Weiblich}$, $I(\mathsf{hatKind}) := I_0(\mathsf{hatKind})$ und $I(\mathsf{hatEhemann}) := I_0(\mathsf{hatEhemann})$ und für alle anderen Konzeptnamen (d.h. den definierten Namen), die Interpretation einfach „ausrechnet", z.B. $I(\mathsf{Frau}) = I_0(\mathsf{Person}) \cap I_0(\mathsf{Weiblich})$.*

Beispiel 6.3.3. *Ein Beispiel für eine sinnvolle zyklische T-Box ist:*

$$\mathsf{Mensch'} \equiv \mathsf{Tier} \sqcap \forall\mathsf{HatEltern.Mensch'}$$

Menschen sind alle Tiere, die deren Elteren nur Menschen sind.

Das Finden eines Modells ist in diesem Fall nicht so einfach, denn selbst wenn man eine Interpretation I_0 hat die Tier *und* HatEltern *festlegt, können wir nicht ohne weiteres ein Modell finden, denn $I(\mathsf{Mensch'}) := I_0(\mathsf{Tier}) \cap \{x \mid \forall y.(x,y) \in I_0(\mathsf{HatEltern}) \implies y \in I(\mathsf{Mensch'})\}$ ist eine rekursiv definierte Menge. Man benötigt eine Interpretation (als Menge) $I(\mathsf{Mensch'}) \subseteq \Delta$, die ein Fixpunkt der rekursiven Gleichung ist.*

6.3.2 Fixpunktsemantik für zyklische T-Boxen

Zwei weitere Beispiele für sinnvolle zyklische T-Box-Definitionen sind:

$$\mathsf{MnurS} \equiv \mathsf{Mann} \sqcap \forall\mathsf{hatKind.MnurS}$$

das entspricht dem Konzept: „Mann, der nur Söhne hat, und für dessen Söhne das gleiche gilt". Zyklen können auch bei Datenstrukturen auftreten, z.B. für binäre Bäume:

$$\mathsf{BinBaum} \equiv \mathsf{Baum} \sqcap (\leq\ 2\ \mathsf{hatAst}) \sqcap \forall\mathsf{hatAst.BinBaum}$$

Eine Fixpunktsemantik hilft hier weiter. Im letzten Beispiel muss man auf jeden Fall einen kleinsten Fixpunkt nehmen (sonst wären unendlich tiefe Bäume auch enthalten!) Zunächst ist klar, dass eine Modell I einer T-Box, alle Axiome erfüllen muss. Für jedes

Axiom $A \equiv C_A$ muss $I(A) = I(C_A)$ gelten. Wenn A in C_A als Name vorkommt, dann ist diese Bedingung nichttrivial.

Beispiel 6.3.4. *Eine solche Interpretation muss nicht immer existieren, wenn die T-Box zyklisch ist:*

$$A \equiv \neg A$$

kann man zwar hinschreiben, aber keine Interpretation dafür angeben, denn $\Delta \neq \emptyset$ war vorausgesetzt, und $I(A) = \Delta \setminus I(A)$ ist daher nie erfüllt.

Man kann eine **Fixpunktsemantik** für eine zyklische T-Box mit Gleichungen als Grenzwert unter gewissen Bedingungen konstruieren: Zunächst hat man eine Basisinterpretation I_B der nicht-definierten Namen.
Man definiert eine Folge von Interpretationen $I_i, i = 0, 1, 2, \ldots$, und

1. erweitert I_B: sei $I_0(A) = \emptyset$ für alle definierten Namen A.
2. Danach definiert man $I_{i+1}(A) := I_i(C_A)$ für alle i und jede Definition $A \equiv C_A$.
3. Wenn die Folge monoton steigend ist, d.h. für alle Namen stets $I_i(A) \subseteq I_{i+1}(A)$ gilt, dann kann man $I_\infty(A) = \bigcup_i I_i(A)$ definieren.
   Das ergibt einen kleinsten Fixpunkt.

Einen größten Fixpunkt erhält man mit folgendem Vorgehen:

1. Man startet mit I_B und erweitert diese so: $I_0(A) = \Delta$ für alle definierten Namen A.
2. Danach definiert man $I_{i+1}(A) := I_i(C_A)$ für jede Definition $A \equiv C_A$.
3. Wenn die Folge monoton fallend ist, d.h. für alle Namen stets $I_{i+1}(A) \subseteq I_i(A)$ gilt, dann kann man $I_\infty(A) = \bigcap_i I_i(A)$ definieren.
   Das ergibt einen größten Fixpunkt.

Beispiel 6.3.5. *Betrachte erneut die T-Box $A \equiv \neg A$. Der Versuch einen Fixpunkt zu erzeugen, scheitert, sei Δ beliebig als nicht-leere Menge festgelegt. Dann gibt es keinen kleinsten Fixpunkt:*

$$\begin{aligned} I_0(A) &= \emptyset \\ I_1(A) &= I_0(\neg A) = \Delta \setminus I_0(A) = \Delta \\ I_2(A) &= I_1(\neg A) = \Delta \setminus I_1(A) = \emptyset \end{aligned}$$

Jetzt sieht man schon die Nichtmonotonie, da $\Delta = I_1(A) \not\subseteq I_2(A) = \emptyset$.
Für den größten Fixpunkt geht es analog:

$$\begin{aligned} I_0(A) &= \Delta \\ I_1(A) &= I_0(\neg A) = \Delta \setminus I_0(A) = \emptyset \\ I_2(A) &= I_0(\neg A) = \Delta \setminus I_1(A) = \Delta \end{aligned}$$

Da $I_2(A) \not\subseteq I_1(A)$ ist die Monotonie verletzt.

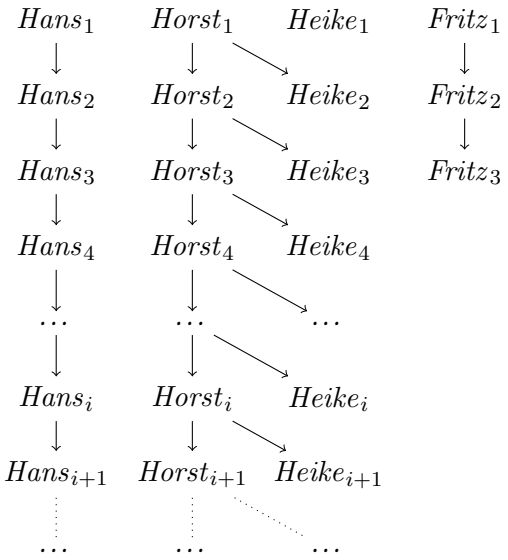
Beispiel 6.3.6.

$$\mathsf{MnurS} \equiv \mathsf{Mann} \sqcap \forall \mathsf{hatKind}.\mathsf{MnurS}$$

Sei I_B eine Interpretation, die Δ, $I(\mathsf{Mann})$ und $I(\mathsf{hatKind})$ festlegt als

$$\begin{aligned}
\Delta &= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Horst_i \mid i \in \mathbb{N}\} \cup \{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_B(\mathsf{Mann}) &= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Horst_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_B(\mathsf{hatKind}) &= \{(Hans_i, Hans_{i+1}) \mid i \in \mathbb{N}\} \cup \{(Horst_i, Horst_{i+1}) \mid i \in \mathbb{N}\} \\
&\quad \cup \{(Horst_i, Heike_{i+1}) \mid i \in \mathbb{N}\} \cup \{(Fritz_i, Fritz_{i+1}) \mid i \in \{1,2\}\}
\end{aligned}$$

Als Graph wobei eine gerichtete Kante für „hatKind“ steht



Intuitiv erwartet man, dass die Interpretation von MnurS *alle drei $Fritz_i$ und alle $Hans_i$ enthält. Wir zeigen, dass man dafür die größte Fixpunkt-Semantik nehmen muss, während die kleinste Fixpunkt-Semantik zuwenig liefert.*

Wir berechnen den kleinsten Fixpunkt:

$$\begin{aligned}
I_0(\mathsf{MnurS}) &= \emptyset \\
I_1(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_0(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_3\}) = \{Fritz_3\} \\
I_2(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_1(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_2, Fritz_3\}) = \{Fritz_2, Fritz_3\} \\
I_3(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_2(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_1, Fritz_2, Fritz_3\}) = \{Fritz_1, Fritz_2, Fritz_3\} \\
I_4(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_3(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_1, Fritz_2, Fritz_3\}) = \{Fritz_1, Fritz_2, Fritz_3\} \\
I_j(\mathsf{MnurS}) &= \{Fritz_1, Fritz_2, Fritz_3\} \text{ für alle weiteren } i
\end{aligned}$$

Das ergibt $\bigcup_i I_i(\mathsf{MnurS}) = \{Fritz_1, Fritz_2, Fritz_3\}$.

Nimmt man den größten Fixpunkt, werden auch die „unendlichen Pfade“ beachtet:

$$
\begin{array}{lcl}
I_0(\mathsf{MnurS}) & = & \Delta \\
I_1(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_0(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\Delta) \\
 & = & I_B(\mathsf{Mann}) \\
I_2(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_1(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_B(\mathsf{Mann})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i, Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}) \\
 & = & \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_3(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_2(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i, Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}) \\
 & = & \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_j(\mathsf{MnurS}) & = & \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \textit{ für alle weiteren } i
\end{array}
$$

Das ergibt $\bigcap_i I_i(\mathsf{MnurS}) = \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\}$.

Theorem 6.3.7. *Ist die Terminologie ohne Komplemente definiert, dann kann man sowohl einen kleinsten als auch einen größten Fixpunkt der Interpretationen als Erweiterung einer Basisinterpretation definieren.*

Der Grund dafür, dass dieses Verfahren funktioniert, ist, dass man folgende Monotonie in diesem Fall hat:

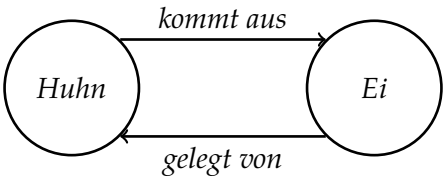
$$I \subseteq I' \Rightarrow I(C) \subseteq I'(C)$$

wobei $I \subseteq I'$ definiert ist als: $\forall$atomare Konzepte $A : I(A) \subseteq I'(A)$. Das wiederum folgt daraus, dass $\sqcap, \sqcup, \forall R.C, \exists R.C, (\geq\ n\ R)$ alle monoton im Konzept-Argument sind (falls es eines gibt). Wenn man über $\mathcal{ALCN}$ hinausgeht, ist das Konstrukt $(\geq\ n\ R.C)$ monoton, während $(\leq\ n\ R.C)$ nicht monoton ist.

Theorem 6.3.8. *Ist die Terminologie so definiert, dass jeder zyklische Pfad durch die Terme durch eine gerade Anzahl Negationen geht, denn kann man sowohl einen kleinsten als auch einen größten Fixpunkt der Interpretationen als Erweiterung einer Basisinterpretation definieren.*

Der Grund für das Funktionieren ist in diesem Fall ebenfalls die Monotonie bzgl. Interpretationen, wobei jedes Komplement die Monotonie in eine Antimonotonie verwandelt und nach zwei solchen Übergängen das Verhalten wieder monoton ist.

Beispiel 6.3.9. *Wir betrachten ein Beispiel für eine zyklische T-Box, die nicht nur aus Definition besteht. „Jedes Huhn kommt aus einem Ei. Jedes Ei wurde von einem Huhn gelegt.“ Die graphische Darstellung kann man so notieren:*



Die zugehörigen Axiome in der T-Box sind:

$$\begin{array}{lcl} \mathsf{Huhn} & \sqsubseteq & (\exists \mathsf{kommtAus}.\mathsf{Ei}) \\ \mathsf{Ei} & \sqsubseteq & (\exists \mathsf{gelegtVon}.\mathsf{Huhn}) \end{array}$$

Versuche ein Modell zu finden, so dass $Clarissa \in I(\mathsf{Huhn})$*:*

1. *Dann muss gelten: Es gibt ein Objekt* $ClarissaEi$ *mit* $(Clarissa, ClarissaEi) \in I(\mathsf{kommtAus})$*. Das reicht jedoch nicht, denn es muss noch sichergestellt werden, dass* $ClarissaEi \in I(\mathsf{Ei})$*.*

2. *Also wird eine Mutter von Clarissa benötigt, die das Ei gelegt hat, d.h. es gibt* $ClarissaEi$ *mit* $(ClarissaEi, ClarissaMutter) \in I(\mathsf{gelegtVon})$*. Jetzt muss aber sichergestellt werden, dass* $ClarissaMutter$ *in* $I(\mathsf{Huhn})$ *ist, usw.*

D.h. in allen endlichen Modellen mit $I(\mathsf{Huhn}) \neq \emptyset$ *gibt es nur nur Hühner, die ihre eigene Vorfahren sind. Wenn das Modell unendlich sein darf, dann kann es auch Hühner geben, die nicht ihre eigenen Vorfahren sind.*

6.3.3 Inklusionen in T-Boxen

Als terminologische Axiome sind auch Inklusionen $A \sqsubseteq C$ erlaubt (wie wir sie gerade beim Huhn- / Ei-Beispiel gesehen haben). Dabei ist A ein Name. Wenn man diese Namen einführt und nur durch eine Inklusion spezifiziert, dann kann man diese Axiome auf eine normale T-Box (die nur Äquivalenzen enthält) zurückführen, wobei man den Freiheitsgrad in einen neuen Namen kodiert:

Enthalten die terminologischen Axiome Inklusionen und kommen alle Namen auf linken Seite von Definitionen und Inklusionen nur jeweils einmal auf linken Seiten vor, dann kann man daraus eine äquivalente T-Box machen durch folgendes Verfahren:

- Zu jeder Inklusion $A \sqsubseteq C_A$ erfinde einen neuen Namen $\hat{A}$.
- Ersetze die Inklusion $A \sqsubseteq C_A$ durch die Definition $A \equiv \hat{A} \sqcap C_A$.

Damit hat man aus Inklusionen Gleichungen gemacht. Als Preis muss man neue Namen einführen. Die Modelle vorher und nachher sind die gleichen, wenn man sich beim Vergleich der Interpretationen auf die gemeinsamen Namen beschränkt.

D.h. Inklusionen sind nur dann kritisch, wenn man von bereits definierten Namen noch eine extra Inklusion verlangt.

Beispiel 6.3.10. *Hat man die T-Box*

$$\mathsf{Mann} \sqsubseteq \mathsf{Person}$$

, so kann man mit dem eben beschrieben Verfahren, daraus die T-Box

$$\mathsf{Mann} \equiv \widehat{\mathsf{Mann}} \sqcap \mathsf{Person}$$

machen (dabei ist $\widehat{\mathsf{Mann}}$ *der neu eingeführte Konzeptname).*
Anders verhält es sich bei der T-Box:

$$\begin{array}{lcl} \mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\ \mathsf{Mann} & \sqsubseteq & \mathsf{Tier} \end{array}$$

Hier genügt es nicht, daraus die T-Box

$$\begin{array}{lcl} \mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\ \mathsf{Mann} & \equiv & \widehat{\mathsf{Mann}} \sqcap \mathsf{Tier} \end{array}$$

zu erzeugen, da diese nicht einfach interpretiert werden kann, denn es gibt jetzt zwei Definition für Mann.

6.3.4 Beschreibung von Modellen: Die A-Box

Von terminologischen Axiomen bzw. im Spezialfall von T-Boxen wird nur eine Struktur auf den Konzepten erzwungen. Was fehlt, ist eine konkretere Beschreibung von Modellen (Interpretationen). In diesem Sinn entspricht eine T-Box einem Datenbankschema, während eine A-Box einer Datenbank dazu entspricht. Der Name A-Box ist von Assertion hergeleitet.

Definition 6.3.11. *Gegeben eine T-Box* $\mathcal{T}$.
Eine A-Box $\mathcal{A}$ *zu* $\mathcal{T}$ *ist definiert als Menge von Annahmen über Individuen (Objekte) der Formen*

- $C(a)$ *wobei* C *ein Konzeptterm ist und* a *ein Individuenname.*
- $R(a, b)$ *wobei* R *eine Rolle ist (evtl. ein Rollenterm) und* a, b *sind Individuennamen.*

Man kann eine A-Box auch allgemeiner auf Basis einer Menge von terminologischen Axiomen definieren.

Beispiel 6.3.12. *Beispiele für Einträge in einer A-Box sind*

$$\begin{array}{l} \mathsf{MutterMitTochter}(\texttt{Maria}) \\ \mathsf{Vater}(\texttt{Peter}) \\ \mathsf{hatKind}(\texttt{Maria}, \texttt{Paul}) \\ \mathsf{hatKind}(\texttt{Maria}, \texttt{Peter}) \\ \mathsf{hatKind}(\texttt{Peter}, \texttt{Harry}) \end{array}$$

Dabei sind `Peter`, `Harry`, `Maria` *und* `Paul` *Konstanten.*

Das hat Ähnlichkeiten mit einer Datenbank in Prolog, bzw. mit Datalog und deduktiven Datenbanken. Allerdings sind in Konzeptbeschreibungssprachen allgemeinere Konzeptterme möglich. z.B. ist

$$(\exists \mathsf{hatKind}.\mathsf{Person})(\mathtt{Michael})$$

ebenfalls ein gültiger Eintrag.

Definition 6.3.13 (Semantik der A-Box)**.** *Die Semantik I einer A-Box $\mathcal{A}$ zur T-Box $\mathcal{T}$ erweitert die Semantik einer T-Box: Sei I eine Interpretation zur T-Box $\mathcal{T}$. Die Interpretation kann auf die A-Box $\mathcal{A}$ erweitert werden durch:*

- *Jedem Individuennamen a wird ein Objekt $I(a) \in \Delta$ zugeordnet. Hierbei wird die sogenannte* **unique names assumption** *beachtet: Verschiedenen Individuennamen werden verschiedenen Objekte zugeordnet, d.h. $I(a) = I(b)$ gdw. $a = b$.*
- *Für $C(a) \in \mathcal{A}$, ist $I(C(a)) = 1$ gdw. $I(a) \in I(C)$.*
- *Für $R(a, b) \in \mathcal{A}$ ist $I(R(a, b)) = 1$ gdw. $(I(a), I(b)) \in I(R)$ gilt.*

6.3.5 T-Box und A-Box: Terminologische Beschreibung

Eine *terminologische Beschreibung* (oder auch terminologische Datenbank) besteht aus

- Menge von terminologischen Axiomen.
- Mengen von Annahmen über Existenz und Eigenschaft von Objekten (A-Box)

Das kann als Spezialfall eine Kombination von T-Box und A-Box sein.

Beispiel 6.3.14. *Ein Beispiel für eine T-Box und eine zugehörige A-Box ist:*
T-Box:

$$\begin{array}{lcl}
\mathsf{Motor} & \sqsubseteq & \mathsf{Komponente} \\
\mathsf{Lampe} & \sqsubseteq & \mathsf{Komponente} \sqcap (\neg \mathsf{Motor}) \\
\mathsf{Stecker} & \sqsubseteq & \mathsf{Komponente} \sqcap (\neg \mathsf{Lampe}) \sqcap (\neg \mathsf{Motor}) \\
\mathsf{Gerät} & \sqsubseteq & (\forall \mathsf{hatTeil}.\mathsf{Komponente}) \sqcap (\neg \mathsf{Komponente}) \\
\mathsf{ElektroGerät} & \equiv & \mathsf{Gerät} \sqcap (\exists \mathsf{hatTeil}.\mathsf{Stecker})
\end{array}$$

A-Box:

$$\begin{array}{l}
\mathsf{Motor}(\mathtt{Motor1234}) \\
\mathsf{Komponente}(\mathtt{Lichtmaschine320}) \\
\mathsf{hatTeil}(\mathtt{Motor1234}, \mathtt{Lichtmaschine320}) \\
\ldots
\end{array}$$

Definition 6.3.15. *Gegeben eine T-Box $\mathcal{T}$ und eine A-Box $\mathcal{A}$.*

- *Dann ist $\mathcal{A}$* konsistent, *wenn es Modell I von $\mathcal{T}$ gibt, das alle Einträge in der A-Box wahr macht.*

- *Wir schreiben $\mathcal{A} \models C(a)$ gdw. für alle Modelle I von $\mathcal{T}$, die alle Einträge der A-Box wahr machen, auch $I(C(a))$ gilt.*

Folgende Inferenzen und Anfragen an T-Box und A-Box sind von praktischem Interesse:

- *Konsistenztest:* Prüfen, ob definierte Konzepte konsistent sind.
  $\mathsf{Motor} \sqcap (\neg\mathsf{Motor})$ ist inkonsistent, d.h. es gibt keine Objekte in diesem Konzept.

  Damit kann man auch erkennen, ob eine A-Box widersprüchliche Annahmen enthält. Z.B ist $(\mathsf{Motor} \sqcap \neg\mathsf{Motor})(\mathtt{Motor123})$ nicht erfüllbar.

- *Subsumtionstest:* Prüfen, ob ein Konzept eine Untermenge eines anderen ist oder ob Konzepte disjunkt sind. Dadurch kann man z.B. die Struktur der Terminologie angeben.

- *Retrieval Problem* Berechne alle Instanzen eines Konzepts, wobei nur auf die Konstanten in der A-Box zugegriffen werden darf. z.B. „Welche Motoren gibt es ?“

  Das kann man formal schreiben als die Frage: Zu gegebenem Konzept C, finde alle a mit $\mathcal{A} \models C(a)$.

- *Pinpointing (bzw. Realization Problem)* Das ist die Einordnung von Objekten in Konzepte. Z.B. die Frage: „Ist $\mathtt{Staubsauger1}$ ein $\mathsf{ElektroGerät}$?“ Genauer kann man diese Anfrage spezifieren als: Gegeben ein Individuum a, finde das spezifischste Konzept C, so dass $\mathcal{A} \models C(a)$ gilt. D.h. finde das kleinste Konzept in der Subsumtionsordnung.

Es gibt folgende Zusammenhänge zwischen den Inferenzproblemen der A-Box und T-Box:

Satz 6.3.16. *Es gilt:*

- *$\mathcal{A} \models C(a)$ gdw. $\mathcal{A} \cup \{\neg C(a)\}$ ist inkonsistent.*

- *C ist konsistent gdw. $C(a)$ konsistent ist für einen neuen Namen a.*

Beweis. Der erste Teil gilt, da ein Modell für die T-Box, das alle Axiome aus $\mathcal{A}$ wahr macht, auch $C(a)$ wahr machen muss, damit $\mathcal{A} \models C(a)$ gilt; jedes solche Modell muss daher $\neg C(a)$ falsch machen, und daher ist $\mathcal{A} \cup \{\neg C(a)\}$ inkonsistent.

Der zweite Teil gilt, da $C(a)$ nur dann wahr werden kann, wenn $I(C)$ nicht leer ist. □

Bemerkung 6.3.17. *Subsumtionstest in der Beschreibungslogik $\mathcal{ALC}$ und auch in einigen anderen hat einen direkte Beziehung zu Folgerungsfragen in der (aussagenlogischen) Modallogik K; das ist die einfachste Kripke-Modallogik. Die Algorithmen und deren Komplexitäten sind analog.*

6.3.6 Erweiterung der Terminologie um Individuen

Man kann in der T-Box auch Konzepte erlauben, die man als Aufzählungskonzepte ansehen kann: Die Syntax ist:

$$A \equiv \{a_1, \ldots . a_n\}$$

wobei a_i Individuennamen sind. Die Semantik ist fixiert so dass $I(\{a_1, \ldots, a_n\}) = \{I(a_1), \ldots, I(a_n)\}$ gilt.

Damit kann man z.B. das Konzept Grundfarben $\equiv \{\texttt{rot}, \texttt{blau}, \texttt{gelb}\}$ definieren.

Man kann den Effekt allerdings auch kodieren, wenn man $\sqcap, \sqcup, \neg$ in der Beschreibungssprache hat.

6.3.7 Open-World und Closed-World Semantik

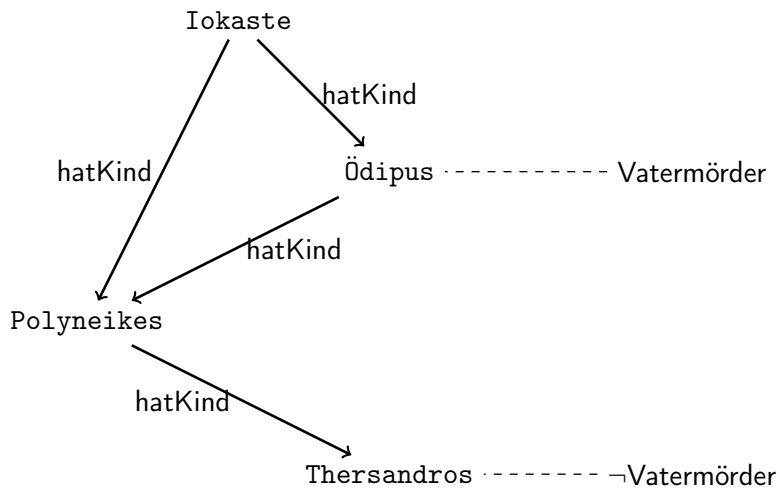
Im Gegensatz zu Datenbanken bei denen die Closed-World-Semantik angenommen wird, die eine eindeutige Semantik bei Datenbanken garantiert, benutzt man bei A-Boxen der Beschreibungssprachen die Open-World-Semantik. D.h., man geht davon aus, dass man unvollständiges Wissen hat. Der Schlussfolgerungsmechanismus wird somit fehlende Einträge nicht als negative Einträge werten. Da die Logik nicht auf Hornklauseln beruht, ist das auch die bessere Wahl, denn CWA + nicht-Hornklauseln können zu Inkonsistenzen führen.

In der Open-World Semantik kann man neue Fakten hinzufüge, ohne dass alte Schlüsse ungültig werden; d.h. Schlussfolgern ist monoton.

Den Unterschied kann man sich klarmachen am Beispiel: hatKind(Maria, Peter). Wenn das der einzige Eintrag in einer Datenbank ist, dann hat Peter keine Geschwister. Wenn es der einzige Eintrag in einer A-Box ist, dann ist es durchaus möglich, dass Peter noch weitere Geschwister hat, nur hat die A-Box darüber keine Information. Man kann aber mit der Angabe $(\leq \ 1 \ \text{hatKind})(\texttt{Maria})$ in der A-Box die Information eintragen, dass es keine Geschwister gibt.

Beispiel 6.3.18. *Dieses Beispiel dient der Illustration des Verhaltens der Inferenzen durch die Open-World-Annahme.*
Eine A-Box $\mathcal{A}_{oed}$ zum Problem des Ödipus:

hatKind(Iokaste, Ödipus)	hatKind(Iokaste, Polyneikes)
hatKind(Ödipus, Polyneikes)	hatKind(Polyneikes, Thersandros)
Vatermörder(Ödipus)	¬Vatermörder(Thersandros)



Frage: kann man aus dieser A-Box folgendes schließen?

$$\mathcal{A}_{oed} \models (\exists \mathsf{hatKind}.(\mathsf{Vatermörder} \sqcap (\exists \mathsf{hatKind}.\neg \mathsf{Vatermörder})))(\mathtt{Iokaste})$$

Schaut man sich das an, dann könnte man folgendermaßen schließen:
`Iokaste` *hat zwei Kinder,* `Ödipus` *und* `Polyneikes`*. Bei* `Ödipus` *ist der erste Teil erfüllt, aber man kann nichts über den zweiten Teil der Konjunktion sagen, da es kein Wissen über* Vatermörder(`Polyneikes`) *gibt. Nimmt man* `Polyneikes` *als Kandidat, so weiss man nicht, dass er* Vatermörder *ist, also scheint man nichts schließen zu können.*

Korrekt ist aber, dass man eine Fallunterscheidung machen kann: `Polyneikes` *ist entweder* Vatermörder *oder nicht. Im ersten Fall ist er der Kandidat, der die Formel als* hatKind *von* `Iokaste` *erfüllt. Im zweiten Fall ist der Kandidat* `Ödipus`*. Somit kann man obige Formel aus der A-Box schließen.*

Wie dieses Beispiel zeigt, kann die Open-World-Annahme Fallunterscheidungen erforderlich machen, um korrekt und vollständig schließen zu können.

6.4 Inferenzen in Beschreibungslogiken: Subsumtion

6.4.1 Struktureller Subsumtionstest für die einfache Sprache $\mathcal{FL}_0$

Wir wiederholen: Die Beschreibungssprache $\mathcal{FL}_0$ hat als Konstrukte nur Konjunktion und Wertbeschränkungen, d.h. nur atomare Konzepte, $C \sqcap D$ und $\forall R.C$ sind möglich. Diese Sprache ist eine Untersprache von $\mathcal{AL}$. Wir betrachten sie hier, um einen sogenannten strukturellen Subsumtionstest zu illustrieren.

Da Komplemente fehlen, gibt es in $\mathcal{FL}_0$ keinen direkten Zusammenhang zwischen Inkonsistenz von Konzepten und Subsumtion.

Es gilt:

Lemma 6.4.1. *Alle Konzepte in $\mathcal{FL}_0$ sind konsistent.*

Beweis. Das kann man ganz einfach dadurch zeigen, dass man eine Interpretation I angibt, die folgendes erfüllt:

$$\begin{aligned} I(A) &= \Delta \quad \text{für alle atomaren Konzepte } A \\ I(R) &= \Delta \times \Delta \quad \text{für alle Rollen } R \end{aligned}$$

Dann werden sogar alle zusammengesetzten Konzepte C immer als $I(C) = \Delta$ interpretiert. □

D.h. der Konsistenztest in $\mathcal{FL}_0$ ist trivial. Das kommt daher, dass man keine Möglichkeit hat, Konzepte mittels Negation einzuschränken.

Trotzdem ist der Subsumtionstest für $\mathcal{FL}_0$ nichttrivial. Z.B. gilt sicher $A \sqcap B \sqsubseteq A$, aber $A \not\sqsubseteq A \sqcap B$ für atomare Konzepte A, B.

6.4.1.1 Struktureller Subsumtionstest

Allgemein (nicht nur in $\mathcal{FL}_0$) lässt sich der strukturelle Subsumtionstest, der $C \sqsubseteq D$ testet, durch die folgenden beiden Schritte beschreiben:

1. Bringe C und D in eine Normalform C' bzw. D'.
2. Vergleiche C' und D' syntaktisch.

Beide Teile variieren von Sprache zu Sprache.

In $\mathcal{FL}_0$ ist eine Normalform folgendermaßen definiert:

$$A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

wobei die Kommutativität und Assoziativität von $\sqcap$ ausgenutzt wird, um Klammern wegzulassen, und A_i verschiedene atomare Konzepte sind, die R_i verschiedene Rollensymbole sind, und C_i Konzepte in Normalform sind.

D.h. ein Normalformalgorithmus für $\mathcal{FL}_0$ wird folgendes tun:

- assoziativ Ausklammern, dann Umsortieren, und dann gleiche A_i in Konjunktionen eliminieren.
- Falls es einen Unterausdruck $\forall R.C \sqcap \forall R.D$ gibt, nutzt man aus, dass dieser äquivalent zu $\forall R.C \sqcap D$ ist, da das Rollensymbol das gleiche ist.

  Als Begründung folgende Gleichungskette:

$$\begin{aligned} & \{x \mid \forall y.xI(R)y \Rightarrow y \in I(C_1) \cap I(C_2)\} \\ = \; & \{x \mid \forall y.\neg(x\ I(R)\ y \vee y \in I(C_1) \cap I(C_2)\} \\ = \; & \{x \mid \forall y.(\neg(x\ I(R)\ y) \vee y \in I(C_1) \wedge (\neg(x\ I(R)\ y) \vee y \in I(C_2)\} \\ = \; & \{x \mid \forall y.(\neg(x\ I(R)\ y) \vee y \in I(C_1)) \wedge \forall y.\neg(x\ I(R)\ y) \vee y \in I(C_2)\} \\ = \; & I((\forall\ R.C_1) \sqcap\ (\forall\ R.C_2)) \end{aligned}$$

- Es wird rekursiv in den rechten Seiten C aller Ausdrücke $\forall R.C$ dasselbe Verfahren durchgeführt.

Der Vergleich zweier Normalformen wird folgendermaßen durchgeführt: Angenommen, wir haben zwei Ausdrücke D, D':

$$D \equiv A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

und

$$D' \equiv A'_1 \sqcap \ldots \sqcap A'_{m'} \sqcap \forall R'_1.C'_1 \sqcap \ldots \sqcap \forall R'_{n'}.C'_{n'}$$

Dann ist $D \sqsubseteq D'$ gdw:

- Jedes atomare Konzept A'_i kommt unter den A_j vor und
- zu jedem Ausdruck $\forall R'_i.C'_i$ gibt es einen Ausdruck $\forall R_j.C_j$, so dass die Rollennamen gleich sind, d.h. $R'_i = R_j$ und C_j von C'_i subsumiert wird. Dieser Test wird rekursiv durchgeführt.

Dieser Subsumtionstest ist korrekt und vollständig. Dabei bedeutet „korrekt“ , dass bei Antwort „ja“auch wirklich $I(D) \subseteq I(D')$ in allen Modellen I gilt. „Vollständig“ bedeutet, dass die Antwort „nein“ impliziert, dass es ein Modell I gibt, das ein Gegenbeispiel ist, d.h. $I(D) \not\subseteq I(D')$ für ein Modell I.

Will man einen Beweis führen, so ist es relativ einfach zu begründen, dass er korrekt ist, da man die Mengensemantik verwenden kann:

- Für die atomaren Konzepte: Da jedes atomare Konzept A'_i unter den A_j vorkommt, können wir $A_1 \sqcap \ldots \sqcap A_m$ auch umschreiben zu $A'_1 \sqcap \ldots \sqcap A'_{m'} \sqcap A''_1 \sqcap A''_k$ wobei A''_i irgendwelche der A_i sind. Nimmt man nun die Interpretation so sieht man leicht:

  $$I(A'_1) \cap \ldots \cap I(A'_{m'}) \cap I(A''_1) \cap I(A''_k) \subseteq I(A'_1) \cap \ldots \cap I(A'_{m'})$$

  da Schnitte die Mengen nur verkleinern.

- Für die Wertbeschränkungen reicht es zu zeigen, dass $I(\forall R_j.C_j) \subseteq I(\forall R'_i.C'_i)$ wenn R_j und R'_i die gleichen Rollennamen sind und $C_j \sqsubseteq C'_i$. Wir schreiben zur vereinfachten Darstellung R anstelle von R_j, R'_i: Aus $C_j \sqsubseteq C'_i$ dürfen wir schließen $I(C_j) \subseteq I(C'_i)$ für jedes Modell I. Da

  $$I(\forall R.C_j) = \{x \in \Delta \mid \forall y.(x,y) \in I(R) \implies y \in I(C_j)\}$$

  und

  $$I(\forall R.C'_i) = \{x \in \Delta \mid \forall y.(x,y) \in I(R) \implies y \in I(C'_i)\}$$

  sieht man, dass die erste Menge kleiner (oder gleich) sein muss.

- Schließlich muss man sich noch klar machen, dass $S_1 \subseteq S_1'$ und $S_2 \subseteq S_2'$ impliziert $S_1 \cap S_2 \subseteq S_1' \cap S_2'$ (damit wir die beiden Teile: atomare Konzepte und Wertbeschränkungen getrennt voneinander betrachten durften).

Die Vollständigkeit kann man z.B. zeigen, indem man im Falle, dass der Algorithmus scheitert, ein Modell angibt, in dem die Subsumtionsbeziehung tatsächlich nicht gilt.

Die *Komplexität dieses Algorithmus* kann man wie folgt abschätzen:
Die Normalformherstellung arbeitet auf der Termstruktur und sortiert sinnvollerweise, d.h. man hat einen Anteil $O(n*log(n))$ für die Herstellung der Normalform. Das rekursive Vergleichen ist analog und könnte in einer sortierten Darstellung sogar linear gemacht werden. Insgesamt hat man $O(n * log(n))$ als Größenordnung des Zeitbedarfs.

Beispiel 6.4.2. *Betrachte die $\mathcal{FL}_0$-Konzeptdefinitionen:*

$$\begin{array}{lcl} C_1 & \equiv & (\forall R_1.A_1) \sqcap A_2 \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_2 \sqcap A_3 \sqcap A_4)) \\ C_2 & \equiv & ((\forall R_2.\forall R_1.A_4) \ \sqcap \ A_2 \ \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \end{array}$$

Wir testen, ob $C_1 \sqsubseteq C_2$ und ob $C_2 \sqsubseteq C_1$ mithilfe des strukturellen Subsumtionstests. Zunächst berechnen wir die Normalformen $NF(C_1), NF(C_2)$:
Normalformberechnung für C_1:

$$\begin{array}{lcl} C_1 & \equiv & (\forall R_1.A_1) \sqcap A_2 \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_2 \sqcap A_3 \sqcap A_4)) \\ & \rightarrow & A_2 \sqcap (\forall R_1.A_1) \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \rightarrow & A_2 \sqcap (\forall R_1.A_1) \sqcap (\forall R_2.A_5 \sqcap (\forall R_1.(A_2 \sqcap A_3 \sqcap A_4))) = NF(C_1) \end{array}$$

Normalformberechnung für C_2:

$$\begin{array}{lcl} C_2 & \equiv & ((\forall R_2.\forall R_1.A_4) \ \sqcap \ A_2 \ \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \rightarrow & A_2 \sqcap ((\forall R_2.\forall R_1.A_4) \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \rightarrow & A_2 \sqcap (\forall R_2.((\forall R_1.A_4) \sqcap \forall R_1.(A_3 \sqcap A_4))) \\ & \rightarrow & A_2 \sqcap (\forall R_2.(\forall R_1.(A_4 \sqcap A_3 \sqcap A_4)) \\ & \rightarrow & A_2 \sqcap (\forall R_2.(\forall R_1.(A_3 \sqcap A_4)) = NF(C_2) \end{array}$$

Nun müssen wir die Normalformen vergleichen: Die atomaren Konzepte sind genau gleich, da beide Normalformen A_2 als atomares Konzept in der Konjuktion haben. Für die Wertbeschränkungen: Da $\forall R_1 \ldots$. nur in der $NF(C_1)$ vorkommt, können wir sofort schließen $C_2 \not\sqsubseteq C_1$. Für $C_1 \sqsubseteq C_2$ müssen wir für $\forall R_2 \ldots$. rekursiv zeigen Das erfordert als rekursiven Vergleich: $A_5 \sqcap (\forall R_1.(A_2 \sqcap A_3 \sqcap A_4))) \sqsubseteq (\forall R_1.(A_3 \sqcap A_4))$ Für die atomaren Konzepte ist der Vergleich erfolgreich, da A_5 nur links vorkommt. Für die Wertbeschränkungen müssen wir rekursiv vergleichen: $(A_2 \sqcap A_3 \sqcap A_4) \sqsubseteq (A_3 \sqcap A_4)$ was offensichtlich erfüllt ist. Daher dürfen wir schließen $C_1 \sqsubseteq C_2$.

6.4.2 Struktureller Subsumtionstest für weitere DL-Sprachen

6.4.2.1 Subsumtionstest für die Sprache $\mathcal{FL}^-$

Die Sprache $\mathcal{FL}^-$ hat $\sqcap, \forall R.C, (\exists R.\top)$ als Konstrukte. Der Subsumtionsalgorithmus für $\mathcal{FL}^-$ geht vor wie der Subsumtionstest für $\mathcal{FL}_0$:

1. Bringe die Konzeptterme in eine $\mathcal{FL}^-$-Normalform:

$$\begin{array}{ll} & A_1 \sqcap \ldots \sqcap A_m \\ \sqcap & \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n \\ \sqcap & \exists R'_1.\top \sqcap \ldots \sqcap \exists R'_k.\top \end{array}$$

   Analog wie in $\mathcal{FL}_0$ fasst man $\forall R.C$-Ausdrücke zusammen, wenn das gleiche Rollensymbol vorkommt.

2. Jetzt vergleicht man strukturell, wobei man genauso wie bei $\mathcal{FL}_0$ vorgeht, nur dass man $\exists R.\top$-Ausdrücke wie atomare Konzepte behandelt.

Auch der $\mathcal{FL}^-$-Subsumtions-Algorithmus ist korrekt und vollständig.
Der Zeitaufwand des Algorithmus ist $O(n * \log(n))$.

6.4.2.2 Weitere Sprachen

Erweitert man $\mathcal{FL}_0$ um das konstante Konzept $\bot$, dann kann man ebenfalls einen Subsumtionstest durchführen, nur muss man beachten, dass man Konjunktionen, die $\bot$ enthalten zu $\bot$ vereinfacht, und dass $\bot$ von allen Konzepten subsumiert wird.

Erweitert man $\mathcal{FL}_0$ um $\bot$ und atomare Negation, d.h. zusätzlich ist noch $\neg A$ für atomare Konzepte A erlaubt, dann kann man ebenfalls einen strukturellen Subsumtionsalgorithmus angeben.

Zusätzlich muss nur beachtet werden:

- kommt in einer Konjunktion A und $\neg A$ vor, dann wird die gesamte Konjunktion durch $\bot$ ersetzt.
- $\bot$ wird von allen Konzepten subsumiert.
- Beim Vergleich von Normalformen werden negierte Atome wie neue Namen behandelt (das man ersetzt $\neg A$ durch $NOTA$ vor dem Vergleich.

6.4.2.3 Subsumtions-Algorithmus für $\mathcal{AL}$

Wir können somit auch einen strukturellen Subsumtions-Algorithmus für $\mathcal{AL}$ angeben: Es fehlen an Konstrukten nur: $\top$ und $\exists R.\top$.

Zu beachten ist:

- In Konjunktionen, die $\top$ enthalten, kann man das $\top$ streichen.
- Das Konzept $\forall R.\top$ kann man durch $\top$ ersetzen. Ebenso kann man $\neg\top$ und $\neg\bot$ direkt vereinfachen. D.h. bei der Normalformherstellung spielt $\top$ keine Rolle. Es kann nur als Gesamtergebnis vorkommen (abgesehen von der Syntax $\exists R.\top$).
- Bei der Subsumtion ist nur zu beachten, dass $\top$ alles subsumiert.

6.4.2.4 Konflikte bei Anzahlbeschränkungen

Wenn Anzahlbeschränkungen mit betrachtet werden, dann ergeben sich weitere Möglichkeiten für Konflikte, die in strukturellen Subsumtionsalgorithmen zu beachten sind:

Z.B. gibt es eine Interferenz zwischen $\forall R.\bot$ und $(\geq\ 1\ R)$. Das kann bei Schnitten ein Konflikt sein: $\forall R.\bot \sqcap (\geq\ 1\ R)$ ist äquivalent zu $\bot$. Es gibt auch neue Subsumtionen zu beachten: $(\geq\ n\ R) \sqsubseteq (\geq\ m\ R)$ gdw. $n \geq m$.

Es gibt weitere Sprachen, die einen strukturellen Subsumtionsalgorithmus haben:

$\mathcal{ALE}$	$\mathcal{AL}$ erweitert um $\exists R.C$.
$\mathcal{ALU}$	$\mathcal{AL}$ erweitert um $\sqcup$.
$\mathcal{ALN}$	$\mathcal{AL}$ erweitert um Anzahlbeschränkungen $(\leq\ n\ R)$.

Die Sprache $\mathcal{ALN}$ hat einen polynomiellen und strukturellen Subsumtionsalgorithmus. Die Sprache $\mathcal{FL}$ hingehgen hat ein PSPACE-vollständiges Subsumtionsproblem.

Allerdings sind strukturelle Subsumtionsalgorithmen i.a. ungeeignet, wenn Disjunktion (d.h. Vereinigung) oder volle Negation vorkommt, z.B. in der Sprache $\mathcal{ALC}$.

6.4.3 Subsumtion und Äquivalenzen in $\mathcal{ALC}$

$\mathcal{ALC}$ ist die Konzeptbeschreibungssprache, die $\sqcap, \sqcup, \neg, \bot, \top, \forall R.C, \exists R.C$ erlaubt, aber nur atomare Rollen und keine Anzahlbeschränkungen.

In dieser Sprache kann man mittels der Semantik zeigen, dass folgende Äquivalenzen gelten:

$$\begin{array}{lcl}
\neg(C_1 \sqcap C_2) & \equiv & (\neg C_1) \sqcup (\neg C_2) \\
\neg(C_1 \sqcup C_2) & \equiv & (\neg C_1) \sqcap (\neg C_2) \\
\neg(\neg C) & \equiv & C \\
\neg(\forall R.C) & \equiv & (\exists R.(\neg C)) \\
\neg(\exists R.C) & \equiv & (\forall R.(\neg C)) \\
\neg\bot & \equiv & \top \\
\neg\top & \equiv & \bot
\end{array}$$

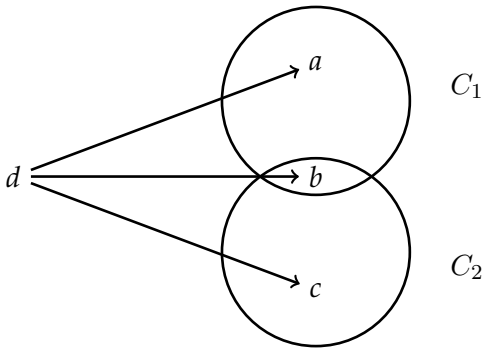
Beispiel 6.4.3. *Analog zur Prädikatenlogik gilt die folgende Gleichung* nicht*!*

$$(\forall R.(C_1 \sqcup C_2)) = (\forall R.C_1) \sqcup (\forall R.C_2)$$

Wir zeigen, dass die Gleichung nicht gilt, indem wir ein Gegenbeispiel angeben. Sei I die Interpretation mit

$$\begin{array}{lcl} \Delta & = & \{a, b, c, d\} \\ I(C_1) & = & \{a, b\} \\ I(C_2) & = & \{b, c\} \\ I(R) & = & \{(d, b), (d, a), (d, c)\} \end{array}$$

Als Bild:



Es gilt

$$\begin{aligned} I(\forall R.(C_1 \sqcup C_2)) &= \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{a, b, c\}\} \\ &= \{a, b, c, d\} \end{aligned}$$

$$\begin{aligned} I((\forall R.C_1) \sqcup (\forall R.C_2)) &= \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{a, b\}\} \\ &\quad \cup \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{b, c\}\} \\ &= \{a, b, c\} \end{aligned}$$

Da $d \in I(\forall R.(C_1 \sqcup C_2))$ aber $d \notin I((\forall R.C_1) \sqcup (\forall R.C_2))$ sind die beiden Konzepte nicht äquivalent.

Lemma 6.4.4. *Der Subsumtionstest in $\mathcal{ALC}$ lässt sich als Konsistenztest formulieren und umgekehrt: $C_1 \sqsubseteq C_2$ gilt gdw. $(C_1 \sqcap \neg C_2) \equiv \bot$*

Will man $C \not\equiv \bot$ testen, dann kann man auch $C \sqsubseteq \bot$ testen. Da ein Entscheidungsverfahren dann ja oder nein sagt, kann man daraus die Konsistenz bzw. Inkonsistenz von C schließen.

6.4.4 Ein Subsumtionsalgorithmus für $\mathcal{ALC}$

Der Algorithmus zum Subsumtionstest von zwei Konzepten in $\mathcal{ALC}$ ist ein Tableauverfahren. Es verwendet den Konsistenztest für ein Konzept und prüft daher ob eine Interpretation mit $I(C) \neq \emptyset$ existiert. Die Idee dabei ist: Konstruiere eine solche Interpretation oder zeige, dass jede Konstruktion einer solchen Interpretation scheitern muss. Das Tableauverfahren arbeitet dabei mit einer neuen Struktur, sogenannten *Constraint-Systemen*. Der Algorithmus ist so allgemein, dass er leicht erweiterbar ist, um auch die Konsistenz

von A-Boxen zu prüfen. Er ist auch leicht erweiterbar auf allgemeinere Konzeptbeschreibungssprachen.

Die Idee des Algorithmus ist es, eine Interpretation aufzubauen, die $I(C)$ nicht leer macht. Dabei geht man vorsichtig vor, so dass der Algorithmus entweder bemerkt, dass ein Widerspruch aufgetreten ist, der anzeigt, dass das Konzept C leer ist, oder es sind ausreichend viele Objekte und Beziehungen gefunden, die ein Modell darstellen.

Wir definieren zunächst die Struktur des Constraint-Systems:

Definition 6.4.5. *Ein* Constraint *ist eine Folge von Ausdrücken der Form:*

$$x : X \qquad xRy \qquad X \sqsubseteq C \qquad X \sqsubseteq Y \sqcup Z \qquad X(\forall R)Y \qquad X(\exists R)Y$$

wobei x, y, z Elemente (von Δ), X, Y, Z Konzeptnamen und C (auch komplexe) Konzepte sind. [3]

Wir beschreiben informell, was die einzelnen Constraints bedeuten: Diese entsprechen *Anforderungen* an das Modell I: $x : X$ entspricht der Bedingung $x \in I(X)$, xRy entspricht der Bedingung $(x, y) \in I(R)$, $X \sqsubseteq C$ entpsricht der Bedingung $I(X) \subseteq I(C)$, $X \sqsubseteq Y \sqcup Z$ entspricht der Bedingung $I(X) \subseteq I(Y) \cup I(Z)$, $X(\forall R)Y$ entspricht der Bedingung $\forall a \in I(X) : \forall b : (a, b) \in I(R) : b \in I(Y)$, d.h. für alle Elemente a in $I(X)$, die links in $I(R)$ vorkommen, muss jedes rechte Element b auch in $I(Y)$ sein, $X(\exists R)Y$ entspricht der der Bedingung $\forall a \in I(X) : \exists b.(a, b) \in I(R) : b \in I(Y)$, d.h. für alle Element a in $I(X)$ muss es mindestens ein Paar $(a, b) \in I(R)$ geben, wobei $b \in I(Y)$ ist.

Wie wir gleich sehen werden, wird im ersten Schritt (bevor das Tableau aufgebaut wird) das Constraint-System aufgefaltet. Hierfür werden die folgenden Regeln verwendet:

$$\begin{array}{lcl}
X \sqsubseteq (\forall R.C) & \rightarrow & X(\forall R)Y, Y \sqsubseteq C, \text{ wobei } Y \text{ ein neuer Name} \\
X \sqsubseteq (\exists R.C) & \rightarrow & X(\exists R)Y, Y \sqsubseteq C, \text{ wobei } Y \text{ ein neuer Name} \\
X \sqsubseteq C \sqcap D & \rightarrow & X \sqsubseteq C, X \sqsubseteq D \\
X \sqsubseteq C \sqcup D & \rightarrow & X \sqsubseteq Y \sqcup Z, Y \sqsubseteq C, Z \sqsubseteq D, \\
 & & \text{wenn und } C \text{ oder } D \text{ kein atomares Konzept ist,} \\
 & & \text{und } Y, Z \text{ sind neue Namen sind.} \\
X \sqsubseteq \top & \rightarrow & \text{nichts zu tun}
\end{array}$$

Es ist relativ leicht einzusehen, dass diese Regeln terminieren und als Normalform ein Constraint-System liefern, dass nur noch Konzeptnamen und negierte Konzeptnamen für C in Constraints $X \sqsubseteq C$ enthält.

Der zweite Schritt des Verfahrens baut ein Tableau auf, indem folgende Regeln benutzt werden, um Individuen-Variablen korrekt einzufügen und das System zu vervollständigen:

[3] Bei Verallgemeinerung auf $\mathcal{ALCN}$ muss man darauf achten, dass Objektvariablen verschiedene Objekte bezeichnen

1. Wenn $x : X$ und $X(\exists R)Y$ da sind, aber keine Variable y mit xRy und $y : Y$, dann füge eine neue Variable y mit den Constraints xRy und $y : Y$ ein.

2. Wenn $x : X, X(\forall R)Y, xRy$ da ist, dann füge $y : Y$ hinzu.

3. Wenn $x : X, X \sqsubseteq Y \sqcup Z$ da sind, aber weder $x : Y$ noch $x : Z$, dann füge $x : Y$ oder $x : Z$ hinzu.

Beachte nur die letzte Regel stellt eine Verzweigung im Tableau dar. Beachte auch, dass Blätter im Tableau mit Constraint-System markiert sind, die vervollständigt sind, d.h. keine der drei Regeln ist mehr anwendbar.

Schließlich kann für jedes solche Blatt (teilweise auch schon früher, dann kann man die Vervollständigung auf dem entsprechenden Pfad stoppen) wird geprüft, ob das Constraintsystem *widersprüchlich* ist:

Definition 6.4.6. *Ein Constraint-System ist* widersprüchlich, *wenn die Konstellation* $x : X, X \sqsubseteq A, x : Y, Y \sqsubseteq \neg A$, *oder* $x : X, X \sqsubseteq \neg\top$, *oder* $x : X, X \sqsubseteq \bot$ *vorkommt.*

Jetzt haben wir alle Teilschritte beisammen, um den Algorithmus zur Konsistenzprüfung zu beschreiben:

Definition 6.4.7. *Der Algorithmus zur Konsistenzfeststellung von C arbeitet folgendermaßen: Starte mit dem Constraint $x : X, X \sqsubseteq C$ (das entspricht gerade der Bedingung, dass es eine Interpretation I gibt, die C nicht leer interpretiert (denn $x \in I(X) \subseteq I(C)$)).*

Als erster Schritt wird $x : X, X \sqsubseteq C$ aufgefaltet.

Anschließend wir das Tableau aufgebaut, indem das aufgefaltete Constraintsystem (nichtdeterministisch) vervollständigt wird (Verzweigungen ergeben sich aus Regel 3 der Vervollständigunsregeln). Wenn es möglich ist, ein Constraint-System zu erzeugen, das nicht widersprüchlich ist (d.h. es gibt einen Pfad dessen Blatt nicht widersprüchlich ist), gebe „konsistent“ aus; ansonsten, wenn alle erzeugbaren Constraint-Systeme (d.h. alle Blätter des Tableaus) widersprüchlich sind, gebe „inkonsistent“ aus.

Beachte, dass der Algorithmus nicht mit einer T-Box als Eingabe arbeitet, sondern nur ein (evtl. komplexes) Konzept C erhält. D.h. für den Fall einer T-Box muss diese zuvor entfaltet werden.

Beispiel 6.4.8. *Seien* Mann *und* hatKind *atomare Konzepte. Wir prüfen die Konsitenz des Konzepts:*

$$\mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann})$$

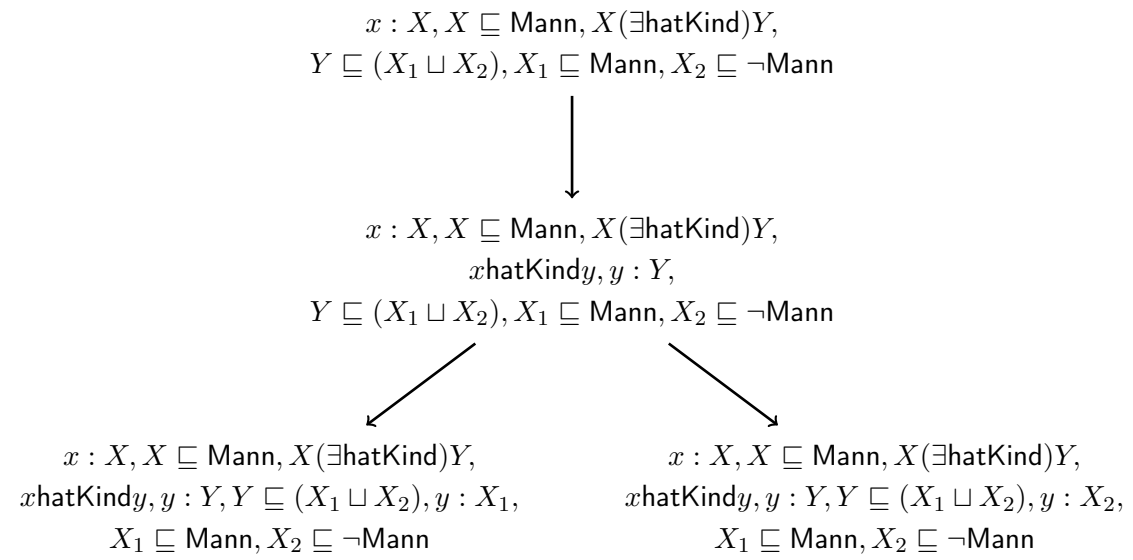
Wir starten daher mit

$$x : X, X \sqsubseteq \mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}).$$

Entfalten des Constraintsystems:

$$
\begin{aligned}
& x : X, X \sqsubseteq \mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\
\rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X \sqsubseteq \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\
\rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X(\exists \mathsf{hatKind})Y, Y \sqsubseteq (\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\
\rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X(\exists \mathsf{hatKind})Y, Y \sqsubseteq (X_1 \sqcup X_2), X_1 \sqsubseteq \mathsf{Mann}, X_2 \sqsubseteq \neg \mathsf{Mann}
\end{aligned}
$$

Vervollständigung ergibt das Tableau:



Beide Blätter sind vervollständigt, aber nicht widersprüchlich. Tatsächlich kann man die Modelle ablesen:

- *Für das linke Blatt kann ablesen:* $\Delta = \{x, y\}$, $I(X) = \{x\}$, $I(Y) = \{y\}$, $I(X_1) = \{x, y\}$, $I(\mathsf{hatKind}) = \{(x, y)\}$ *und* $I(\mathsf{Mann}) = \{x, y\}$, $I(X_2) = \emptyset$.
- *Für das rechte Blatt kann man ablesen:* $\Delta = \{x, y\}$, $I(X) = \{x\}$, $I(Y) = \{y\}$, $I(X_2) = \{y\}$, $I(\mathsf{hatKind}) = \{(x, y)\}$, $I(\mathsf{Mann}) = \{x\}$, $I(X_1) = \{x\}$.

Das Konzept ist daher konsistent (einer der beiden Pfad hätte dafür gereicht).

Da man zeigen kann, dass dieser Algorithmus terminiert und im Falle der Terminierung die richtige Antwort gibt, hat man ein Entscheidungsverfahren zur Feststellung der Konsistenz von $\mathcal{ALC}$-Konzepten.

Theorem 6.4.9. *Subsumtion und Konsistenz in $\mathcal{ALC}$ sind entscheidbar. Der Algorithmus kann in polynomiellem Platz durchgeführt werden.*

Aber es gilt:

Theorem 6.4.10. *Konsistenz in $\mathcal{ALC}$ ist PSPACE-hart. D.h Konsistenz in $\mathcal{ALC}$ ist PSPACE-vollständig.*

Der Nachweis kann direkt geführt werden, indem man zeigt, dass die Frage nach der Gültigkeit von quantifizierten Booleschen Formeln direkt als $\mathcal{ALC}$-Konsistenzproblem kodiert werden kann.

Dazu nimmt man quantifizierte Booleschen Formeln, die einen Quantorprefix haben und als Formel eine Klauselmenge (Konjunktion von Disjunktionen). Da die Kodierung recht technisch ist, geben wir als Beispiel die Kodierung der gültigen Formel $\forall x.\exists y.(x \vee \neg y) \wedge (\neg x \vee y)$ an:
Man benötigt nur ein Konzept A und eine Rolle R: Die Kodierung ist wie folgt:

$$\begin{array}{rl|l} & \exists R.A \sqcap \exists R.\neg A & \forall x \\ \sqcap & \forall R.(\exists R.A \sqcup \exists R.\neg A) & \exists y \\ \sqcap & (\forall R.(\neg\, A \sqcap (\forall R.A))) & (\neg x \vee y) \\ \sqcap & (\forall R.(A \sqcap (\forall R.\neg\, A))) & (x \vee \neg y) \end{array}$$

6.4.5 Subsumtion mit Anzahlbeschränkungen

Auch für die Konzeptsprache $\mathcal{ALCN}$, die $\mathcal{ALC}$ um Anzahlbeschränkungen erweitert, kann man mit Constraintsystemen einen Algorithmus zum Testen der Konsistenz angeben.

Die Erweiterung betreffen die Konzepterme der Form $(\geq\ n\ R.C)$ und $(\leq\ n\ R.C)$. In dem Fall erweitert man zunächst die Constraints um Terme $X(\leq\ n\ R)Y$ und $X(\geq\ n\ R)Y$. Das Auffalten erhält als zusätzliche Regeln

$$X \sqsubseteq (\geq\ n\ R.C) \rightarrow X(\geq\ n\ R)Y, Y \sqsubseteq C$$
$$X \sqsubseteq (\leq\ n\ R.C) \rightarrow X(\leq\ n\ R)Y, Y \sqsubseteq C$$

und für die Vervollständigung mit Elementen muss man zu $x : X, X(\leq\ n\ R)Y$ die Variablen y zählen mit xRy und entweder Variablen y hinzufügen, oder zwei Variablen gleichsetzen. Diese Schritte müssen evtl. mehrfach durchgeführt werden.

Auch hier kann man zeigen, dass man das so durchführen kann, dass das Verfahren terminiert, aber es bleibt $PSPACE$-hart, so dass jeder Algorithmus exponentiell ist.

6.4.5.1 Konsistenztest von A-Boxen

Wie schon erwähnt, kann man den Subsumtionstest bzw. den Konsistenztest von Konzepten auf Konsistenz von A-Boxen ausdehnen. Allerdings können A-Boxen etwas allgemeiner starten, so dass es tatsächlich passieren kann, dass der naive Algorithmus der Vervollständigung nicht terminiert. Es ist aber möglich, auf bestimmte Vervollständigungen zu verzichten, so dass er immer terminiert, ohne dass der Algorithmus unvollständig wird.

Es gilt:

Theorem 6.4.11. *Konsistenz von $\mathcal{ALCN}$-A-Boxen ist entscheidbar und PSPACE-complete.*

6.4.6 Komplexität der Subsumtions-Inferenzen in der T-Box/A-Box

Um die Konzeptsprachen zu bewerten, untersucht man die Ausdrucksstärke und die zugehörige Komplexität der Inferenzprobleme wie der Subsumtion. Hier gibt es, je nach Sprachumfang, eine Hierarchie von Sprachen und Komplexitäten.

- Die Beschreibungssprache $\mathcal{FL}^-$, die nur $\sqcap, \forall$ und $(\exists R)$ erlaubt, hat einen Subsumtionstest, der polynomiell ist.
- Die Beschreibungssprache $\mathcal{FL}$, die nur $\sqcap, \forall R.C$, $(\exists R.C)$ erlaubt, hat ein PSPACE-vollständiges Subsumtionproblem. Beachte, dass analog zur Sprache $\mathcal{FL}^-$ in der Konzeptsprache $\mathcal{FL}$ alle Konzepte konsistent sind.
- Erlaubt man nur $\sqcap, \sqcup, \neg$, dann ist der Subsumtionstest co-NP-vollständig, da dies gerade das Komplement von SAT ist.
- Die Sprache $\mathcal{ALC}$ erlaubt $\sqcap, \neg, \sqcup, \forall, (\exists R.C)$. Der Konsistenztest $\mathcal{ALC}$ ist genauso schwer wie der Subsumtionstest. Beide sind PSPACE-complete.
- erlaubt man zuviel, z.B. $\mathcal{ALC}$ und zusätzlich den Vergleich von Verknüpfungen von Relationen:
  Alle, deren Kinder die gleichen Fächer wie sie selbst studieren (hatKind; studiertFach) = studiertFach) Dann wird Subsumtion sogar unentscheidbar.

6.5 Erweiterungen, weitere Fragestellungen und Anwendungen

In diesem letzten Abschnitt betrachten wir kurz einige weitere Erweiterungen und andere Fragestellungen, die im Zusammehang mit Beschreibungslogiken auftreten.

6.5.1 Konstrukte auf Rollen: Rollenterme

Es gibt noch weitere Konstrukte auf Rollen: Schnitt, Vereinigung, Komplement, transitiver Abschluss, die man verwenden kann um die Ausdruckskraft einer Beschreibungssprache zu erhöhen.

$R \sqcap S$	Schnitt von Rollen	$I(R \sqcap S) = I(R) \cap I(S)$
$R \sqcup S$	Vereinigung von Rollen	$I(R \sqcup S) = I(R) \cup I(S)$
$\neg R$	Komplement einer Rolle	$I(\neg R) = \Delta \times \Delta \setminus I(R)$
(R^{-1})	Rolleninversion	$I(R^{-1}) = \{(b,a) \mid (a,b) \in I(R)\}$
$(R \circ S)$	Rollenkomposition	$I(R \circ S) = \{(a,c) \mid$ $\exists b.(a,b) \in I(R), (b,c) \in I(S)\}$
(R^+)	transitiver Abschluss	$I(R^+) =$ transitiver Abschluss von $I(R)$.

Die Ergebnisse, die es in der Literatur gibt, sind Entscheidungsverfahren und Komplexitäten von Subsumtionsalgorithmen.

z.B.: Die Sprache $\mathcal{ALCN}$ mit Rollenschnitt hat ein PSPACE-vollständiges Subsumtionsproblem, wenn man Zahlen in Strichcode schreibt.

Die Sprache $\mathcal{ALC}_{trans}$, die $\mathcal{ALC}$ um transitive Rollen erweitert, hat ein entscheidbares Subsumtionsproblem. Interessanterweise ergeben sich hier Brücken und Übergänge zu anderen Logiken, beispielsweise zur propositional dynamic logic, die eine erweiterte Modallogik ist

6.5.2 Unifikation in Konzeptbeschreibungssprachen

Das Unifikationsproblem für Konzeptbeschreibungssprachen lässt sich wiefolgt beschreiben:

Problem:	Gegeben zwei Konzeptterme C, D.
Frage:	Gibt es eine Einsetzung σ von Konzepttermen für die atomare Konzepte, so dass $\sigma(C) \equiv \sigma(D)$?

Matching von Konzepttermen ist das eingeschränkte Unifikationsproblem, wenn Ersetzungen nur in einem Term erlaubt ist:

Eine mögliche Anwendung des Matching ist das Debugging einer Wissensbasis, die Description Logic verwendet. Dies eröffnet die Möglichkeiten zu erkennen ob man verschiedene Kodierungen des gleichen intendierten Konzepts eingegeben hat.

Zum Beispiel: Alice und Bob defnieren Konzepte für: Frauen, die Töchter haben:

Alice:	$\mathsf{Frau} \sqcap \exists \mathsf{hatKind}.\mathrm{Frau}$
Bob:	$\mathsf{Weiblich} \sqcap \mathsf{Mensch} \sqcap \exists \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Mensch})$

Die Einsetzung $\{\mathsf{Frau} \mapsto \mathsf{Weiblich} \sqcap \mathsf{Mensch}\}$ macht die beiden Konzepte gleich, und ist daher ein Unifikator.

6.5.2.1 Unifikation in $\mathcal{EL}$

: Unifikation in $\mathcal{EL}$ ist entscheidbar, genauer: ist NP-vollständig (siehe (Baader & Morawska, 2009)).

Ein Algorithmus dazu hat Anwendungen in der Ontologiedatenbank Snomed. Eine unschöne Eigenschaft dieses Problems ist, dass es Unifikationsprobleme gibt, deren Lösungsmenge unendliche viele Substitutionen enthalten muss. z.B. $X \sqcap \exists R.Y \equiv^? \exists R.Y$. In der Anwendung braucht man evtl. nicht die volle Lösungsmenge.

6.5.2.2 Unifikation in $\mathcal{ALC}$

Beachte, dass die Entscheidbarkeit der Unifikation in $\mathcal{ALC}$ im Moment ein offenes Problem ist. Das Problem ist äquivalent zur Entscheidbarkeit der Unifikation in der Basis-(Multi-)Modallogik K.

Da Aussagenlogik mit $\wedge, \vee, \neg$ und propositionalen Konstanten eine Untermenge der Sprache ist, ist die Boolesche Unifikation ein Subproblem, d.h. das Problem is NP-hart.

6.5.3 Beziehung zu Entscheidungsbäumen, Entscheidungslisten

Wir vergleichen die Ausdrucksmöglichkeiten der Beschreibungslogik mit den Möglichkeiten von attributierten Objekten wie z.B. Entscheidungsbäumen und -listen (siehe dazu auch das nächste Kapitel dieses Skripts).

Um die verschiedenen Attribut-Begriffe hier zu unterscheiden, sprechen wir von EB-Attributen, wenn die Darstellung der Objekte bei Entscheidungsbäumen gemeint ist.

Objekte mit nur diskreten EB-Attributen sind in DL direkt darstellbar, wenn man statt Rollen Attribute nimmt und die Attributwerte mittels Aufzählungsmengen darstellt:

Das Konzept `Farbe = rot` kann man dann als $\exists$`hatFarbe`.{`rot`} wobei man schon definiert haben muss, dass Farbe ein Attribut und keine Rolle ist.

Damit sind die EB-Konzepte alle darstellbar, wenn man Vereinigung, Schnitt und Negation erlaubt. Aber: man hat in DL noch etwas mehr, denn es ist auch darstellbar, dass die Farbe unbekannt ist bzw. noch nicht definiert ist. Aus den DL erhält man einen Subsumtionsalgorithmus auch für die etwas allgemeineren Konzepte.

6.5.4 Merkmals-strukturen (feature-structures)

Wenn keine Rollen, sondern nur Attribute erlaubt sind, d.h. statt binären Relationen hat man Attribute (partielle Funktionen), dann erhält man sogenannte *Merkmalsstrukturen*.

Erste Beobachtung ist folgende:
Da es keine echten Relationen mehr gibt, kann man $\forall R.C$ ersetzen durch einen $\exists$-Ausdruck, denn es gilt:

$$(\exists A.C) \sqcup \neg(\exists A.\top) = (\forall A.C)$$

da das Attribut A funktional sein muss.

Weiterhin gilt jetzt:

$$(\exists A.(C_1 \sqcup C_2)) = (\exists A.C_1) \sqcup (\exists A.C_2)$$

denn:

$$
\begin{aligned}
& \{x \mid \exists y.(I(A)(x) = y \wedge (y \in I(C_1)) \wedge y \in I(C_2))\} \\
= \; & \{x \mid \exists y.(I(A)(x) = y \wedge y \in I(C_1)) \wedge I(A)(x) = y \wedge y \in I(C_2)\} \\
= \; & \{x \mid (\exists y.(I(A)(x) = y \wedge y \in I(C_1))) \wedge \exists y.I(A)(x) = y \wedge y \in I(C_2)\} \\
= \; & I((\exists A.K_1) \sqcup (\exists A.C_2))
\end{aligned}
$$

Entsprechende Aussagen gelten für $\sqcap$. Hierbei ist der Definitionsbereich zu beachten.

Es gilt: Die Subsumtion in $\mathcal{ALC}$ mit funktionalen Rollen, d.h. Attributen, ist entscheidbar, auch wenn agreements (role value maps) hinzugenommen werden.

6.5.5 Verarbeitung natürlicher (geschriebener) Sprache

Man kann Merkmalsstrukturen im Bereich (Unifikations-)Grammatiken zur Verarbeitung natürlicher Sprache in der Computerlinguistik verwenden. Man schreibt die Anwendung von Funktionen auch in der Dot-Notation. D.h. statt $f(g(x))$ schreibt man $x.f.g$

Beispiel 6.5.1. *Beispielsweise kann man das Wort „sie“im Deutschen wie folgt charakterisieren:*

$$
\begin{aligned}
sie \;\; := \;\; & (Syn.cat = PP) \sqcap (Syn.agr.person = 3) \sqcap (Syn.agr.sex = W) \\
& \sqcap (Syn.agr.num = sg) \\
\textit{oder} & \\
sie \;\; := \;\; & (Syn.cat = PP) \sqcap (Syn.agr.person = 3) \sqcap (Syn.agr.num = pl)
\end{aligned}
$$

Grammatikregeln werden um Attributgleichungen (feature agreements, Pfadgleichungen) erweitert:
NP = Det Adj N $\quad (Det.person) = (N.person) \sqcap (Det.casus) = (N.casus)$

Anwendungen für Merkmalsstrukturen sind:

- Lexikoneinträge von Wörtern, oder Wortklassen:
- Semantische Zuordnung von Wörtern aus dem Lexikon
  z.B. Bank: Gebäude oder Sitzmöbel
- Kontextfreie Grammatikregeln mit Bedingungen

Der Test, ob eine Regel anwendbar ist, erfordert dann einen Konsistenztest für eine Merkmalsstruktur

Dieser Test wird auch Unifikation von Merkmalsstrukturen genannt („feature unification“).

Die entsprechenden Grammatiken heißen auch Unifikationsgrammatiken (PATR-II, STUF, ...)

Die Ausdrucksstärke von Merkmalsstrukturen kann je nach erlaubten Konstrukten ebenfalls unterschiedlich sein.

6.6 OWL – Die Web Ontology Language

Die Web Ontology Language (kurz OWL in analogie zu owl (Engl. Eule)) ist eine durch das W3C standardisierte formale Beschreibungssprache zur Erstellung von Ontologien[4], d.h. der maschinell verarbeitbaren Erstellung von Konzepten und Beziehungen. OWL trägt dabei eine bedeutsame Rolle zum semantischen Web, der Weiterentwicklung des Internet, in der sämtliche Information auch semantisch dargestellt, erfasst und entsprechend verarbeitet werden kann. Als Syntax baut OWL auf der RDF-Syntax[5] auf und verwendet insbesondere XML als maschinell verarbeitbare Syntax.

Es gibt drei Varianten von OWL: OWL Lite, OWL DL, und OWL Full. Jede der Sprachen ist eine Erweiterung der vorhergehenden, d.h. OWL Lite lässt am wenigsten zu, OWL DL erweitert OWL Lite, und OWL Full erweitert OWL DL. Die beiden ersten Varianten OWL Lite und OWL DL entsprechen Beschreibungslogiken und lassen sich entsprechend durch Beschreibungslogiken formalisieren, hingegen passt OWL Full nicht mehr in den Rahmen der Beschreibungslogiken.

OWL Lite entspricht der Beschreibungslogik $\mathcal{SHIN}(\mathbf{D})$ und OWL DL der Beschreibungslogik $\mathcal{SHOIN}(\mathbf{D})$.

Wir erklären diese Namen:

- $\mathcal{S}$ steht für $\mathcal{ALC}$ erweitert um transitive Rollen, d.h. neben atomaren Konzepten und Rollen, Werteinschränkung, existentieller Werteinschränkung, Vereinigung, Schnitt und Negation stehen spezielle Rollen zur Verfügung, die stets als transitive Relation interpretiert werden müssen, d.h. für jede Interpetation I und transitive Rolle R muss stets gelten $(x, y) \in I(R), (y, z) \in I(R) \Rightarrow (x, z) \in I(R)$. Der Name $\mathcal{S}$ wurde verwendet, da diese Logik in enger Beziehung zur Modallogik S_4 steht.
- $\mathcal{H}$ steht für Rollenhierarchien, d.h. Axiome der Form $R \sqsubseteq S$ für Rollen R, S sind erlaubt. und werden semantisch als $I(R) \subseteq I(S)$ interpretiert.
- $\mathcal{I}$ steht für inverse Rollen, d.h. R^- ist als Rollenkonstruktor erlaubt (wenn R eine Rolle ist) und wird als $I(R^-) = \{(y, x) \mid (x, y) \in I(R)\}$ interpretiert.
- $\mathcal{N}$ kennen wir bereits: number restrictions $(\leq\ n\ R)$, $(\geq\ n\ R)$
- $\mathcal{Q}$ steht für qualifizierte Anzahlbeschränkungen $(\leq n\ R.C)$ und $(\geq\ n\ R.C)$
- $\mathcal{O}$ steht für "nominal": Dies ist der Konstruktor $\{o\}$, wobei o ein Individuenname ist. Er konstruiert einelementige Mengen von Individuen, die in der T-Box verwendet werden dürfen.
- $\mathbf{D}$ meint, dass konkrete Datentypen verwendet werden dürfen (dies ist eine Variante sogenannter *concrete domains*). In OWL dürfen die XML Schema Datentypen (Integer, Strings, Float, ...) verwendet werden.

[4] siehe http://www.w3.org/2004/OWL/
[5] RDF = Resource Description Framework

Eine OWL Lite- oder OWL DL- Ontologie entspricht einer TBox zusammen mit einer Rollenhierarchie, die den Wissenbereich mittels Konzepten und Rollen beschreiben. Allerdings wird in OWL statt Konzept der Begriff *Klasse* (Class) und statt Rolle der Begriff *Eigenschaft* (Property) verwendet. Eine Ontologie besteht aus einer Menge von Axiomen, die beispielsweise Subsumtionsbeziehungen zwischen Konzepten (oder Rollen) zusichern.

Genau wie in den bereits gesehenen Beschreibungslogiken werden OWL Klassen durch einfachere Klassen und Konstruktoren gebildet. Die in OWL verfügbaren Konstrukte und ihre Entsprechung in Beschreibungslogik zeigt die folgende Tabelle:

Konstruktor	Syntax in DL
`owl:Thing`	$\top$
`owl:Nothing`	$\bot$
`intersectionOf`	$C_1 \sqcap \ldots \sqcap C_n$
`unionOf`	$C_1 \sqcup \ldots \sqcup C_n$
`complementOf`	$\neg C$
`oneOf`	$\{a_1, \ldots, a_m\}$
`allValuesFrom`	$\forall R.C$
`someValuesFrom`	$\exists R.C$
`hasValue`	$\exists R.\{a\}$
`minCardinality`	$\geq nR$
`maxCardinality`	$\leq nR$
`inverseOf`	R^-

Die RDF/XML-Darstellung ist dabei noch nicht gezeigt, da sie ziemlich unübersichtlich ist. Z.B. kann das Konzept Mensch $\sqcap$ Weiblich in XML-Notation als

```
<owl:Class>
 <owl:intersectionOf rdf:parseType="Collection">
 <owl:Class rdf:about="#Mensch"/>
 <owl:Class rdf:about="#Weiblich"/>
 </owl:intersectionOf>
</owl:Class>
```

geschrieben werden, und ≤ 2hatKind$\top$ kann als

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasChild"/>
  <owl:minCardinality rdf:datatype="&xsd;NonNegativeInteger">
    2
  </owl:minCardinality>
</owl:Restriction>
```

geschrieben werden.

Konstrukte zum Formulieren von Axiomen zeigt die folgende Tabelle

Konstruktor	Syntax in DL
`subClassOf`	$C_1 \sqsubseteq C_2$
`equivalentClass`	$C_1 \equiv C_2$
`subPropertyOf`	$R_1 \sqsubseteq R_2$
`equivalentProperty`	$R_1 \equiv R_2$
`disjointWith`	$C_1 \sqcap C_2 \equiv \bot$ bzw. $C_1 \sqcap \neg C_2$
`sameAs`	$\{a_1\} \equiv \{a_2\}$
`differentFrom`	$\{a_1\} \equiv \neg\{a_2\}$
`TransitiveProperty`	definiert eine transitive Rolle
`FunctionalProperty`	definiert eine funktionale Rolle
`InverseFunctionalProperty`	definiert eine inverse funktionale Rolle
`SymmetricProperty`	definiert eine symmetrische Rolle

Ein Grund für die beiden Varianten OWL Lite und OWL DL ist, dass OWL DL als zu umfangreich angesehen wird, so dass sich unerfahrene Benutzer schwer tun beim Verwenden. Auch aus Komplexitätssicht sind die Sprachen verschieden: Subsumption und Konsistenztest sind in beiden Sprachen entscheidbar, aber in $\mathcal{SHOIN}(\mathbf{D})$ sind die Tests NEXPTIME-vollständig, während in sie in $\mathcal{SHIN}(\mathbf{D})$ „nur“ EXPTIME-vollständig sind.

Es gibt einige Inferenzwerkzeuge für die OWL-Ontologien, eine Auswahl ist: Racer (http://www.racer-systems.com/), FaCT++ (http://owl.man.ac.uk/factplusplus/), Pellet (http://pellet.owldl.com/).

Literatur

Allen, J. F. (**1983**). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F. (**2010**). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, New York, NY, USA, 2nd edition.

Baader, F. & Morawska, B. (**2009**). Unification in the description logic $\mathcal{EL}$. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, RTA '09, pages 350–364. Springer-Verlag, Berlin, Heidelberg.

Bauer, F. L. & Wirsing, M. (**1987**). *Elementare Aussagenlogik*. Springer-Verlag, Berlin.

Bratko, I. (**1990**). *Prolog – Programming for Artificial Intelligence*. Addison Wesley.

Donini, F. M., Lenzerini, M., Nardi, D., & Nutt, W. (**1997**). The complexity of concept languages. *Inf. Comput.*, 134(1):1–58.

Ebbinghaus, H.-D., Flum, J., & Thomas, W. (**1986**). *Einführung in die mathematische Logik*. Wissenschaftliche Buchgesellschaft Darmstadt.

Eder, E. (**1992**). *Relative Complexities of First order Calculi*. Vieweg, Braunschweig.

Gal, A., Lapalme, G., Saint-Dizier, P., & Somers, H. (**1991**). *Prolog for Natural Language Processing*. John Wiley & sons.

Haken, A. (**1985**). The intractability of resolution. *Theoretical Computer Science*, 39:297–308.

Levesque, H. J. & Brachman, R. J. (**1987**). Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93.

Minsky, M. (**1975**). A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, London.

Nebel, B. (**1997**). Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ord-horn class. *Constraints*, 1(3):175–190.

Nebel, B. & Bürckert, H.-J. (**1995**). Reasoning about temporal relations: a maximal tractable subclass of allen's interval algebra. *J. ACM*, 42(1):43–66.

Pereira, F. C. & Shieber, S. M. (**1987**). *Prolog and Natural-Language Analysis*. CSLI.

Robinson, J. A. (**1965**). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.

Russell, S. J. & Norvig, P. (**2010**). *Artificial Intelligence - A Modern Approach* (*3. internat. ed.*). Pearson Education.

Schenk, R. & Abelson, R. P. (**1975**). Scripts, Plans and Knowledge. In *International Joint Conference on Artificial Intelligence*, pages 151–157.

Schmidt-Schauß, M. & Smolka, G. (**1991**). Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26.

Valdés-Pérez, R. E. (**1987**). The satisfiability of temporal constraint networks. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, pages 256–260. AAAI Press.