

Unterlagen zur Vorlesung

Deduktionssysteme: Grundlagen und Anwendungen

Sommersemester 1999

Professor Dr. Manfred Schmidt-Schauß

Fachbereich Informatik
J.W.Goethe-Universität Frankfurt

1 Einführung und Überblick: Grundlagen und Anwendungen von Automatischen Deduktionssystemen

1.1 Literatur

Folgende Bücher und Handbuchartikel können als Grundlage für ein weiteres Studium dienen:

[And86,Duf91,Bib83,Bib92,BS92,BB87,RB79] [Bun83,CL73,EFT86,Gal86,GN87,DH89], [ki-98,Kow79,Lov78,Pau94,SA94,WOLB84,BS94,BS99]

Für spezieller Themenbereiche können auch folgende Bücher verwendet werden: [Ede92,FLTN93,Tha88,Smu71]

Literatur

- [And:81] P. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, 1981.
- [And:86] Peter B. Andrews. *An Introduction to mathematical logic and type theory: to truth through proof*. Academic Press, 1986.
- [BB:87] K.-H. Bläsius and H.-J. Bürckert. *Deduktionssysteme - Automatisierung des logischen Denkens*. Oldenbourg Verlag, 1987.
- [Bib:83] W. Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 1983.
- [Bib:92] W. Bibel. *Deduktion, Automatisierung der Logik*. Oldenbourg, 1992.
- [BS:92] Wolfgang Bibel and Peter H. Schmitt, editors. volume I – III. Kluwer Academic Publishers, 1992.
- [BS:94] Franz Baader and Jörg H. Siekmann. Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Oxford Science Publications, 1994.
- [BS:99] Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. to appear.
- [Bun:83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, London, 1983.
- [CL:73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.
- [DH:89] Ralf-Detlef Kutsche Dieter Hofbauer. *Grundlagen des maschinellen Beweisens*. Vieweg, 1989.
- [Duf:91] David A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.
- [Ede:92] Elmar Eder. *Relative Complexities of First order Calculi*. Vieweg, Braunschweig, 1992.
- [EFT:86] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Wissenschaftliche Buchgesellschaft Darmstadt, 1986.
- [FLTN:93] C. Fermüller, A. Leitsch, Tammet T., and Zamov N. *Resolution methoda for the decision problem*, volume 679 of *LNCS*. Springer-Verlag, 1993.
- [Gal:86] Jean H. Gallier. *Logical for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, New York, 1986.
- [GN:87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.
- [Göd:31] K Gödel. über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Mh. Math. Phys*, 38:173–198, 1931.

- [ki:98] Themenheft Deduktion und Anwendung, 1998. in german.
- [Kow:79] R. Kowalski. *Logic for Problem Solving*. North Holland, Amsterdam, 1979.
- [Lov:78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, New York, 1978.
- [Pau:94] Larry C. Paulson. *Isabelle: a Generic Theorem prover*. Springer-Verlag, 1994.
- [RB:79] J S. Moore R. Boyer. *A Computational Logic*, volume 29. Academic Press, London, 1979.
- [Rob:65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [SA:94] Rolf Socher-Ambrosius. *Deduktionssysteme*. BI-Wissenschaftsverlag, 1994. in german.
- [Smu:71] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, 1971.
- [Sti:86] Mark E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. In *Proc. of 8th Int. Conf. on Automated Deduction*, number 230 in *Lecture Notes in Computer Science*, pages 573–587. Springer-Verlag, 1986.
- [Tar:53] A. Tarski. *Der Wahrheitsbegriff in den formalisierten Sprachen*. Studia Philosophica I. 1953.
- [Tha:88] André Thayse, editor. *From Standard Logic to Logic Programming*. John Wiley & Sons, 1988.
- [WOLB:84] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning – Introduction and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

1.2 Vorbemerkung

Das Gebiet der Automatischen Deduktionssysteme ist relativ jung im Vergleich zum Gebiet der Logik.

Insbesondere bedeutet dies, dass Notationen und Begriffe sich z.T. nur im Englischen mit einer festen Bedeutung etabliert haben. Benutzt man Lehrbücher oder weiterführende Literatur (Handbücher, Artikel), so wird man mit verschiedenen Notationen und verschiedenen Namen für gleiche Begriffe konfrontiert. So werden manche englischen Begriffe auf verschiedene Weise ins deutsche übersetzt. Glücklicherweise sind die Begriffsbildungen diesselben, so daß beim Literaturstudium nach etwas Einarbeitung keine wesentlichen Schwierigkeiten auftauchen.

1.3 Behandelte Themen

Da diese Unterlagen mit der Vorlesung mitwachsen und nicht vorgefertigt vorliegen, hier ein vorläufiger Plan der Themen:

Prädikatenlogik erster Stufe

Resolution

- Klauselnormalformen und Resolution
- Unifikation
- Verfeinerungen der Resolution: Elimination von Redundanz.

- logisches Programmieren: SLD-Resolution

Analytische Tableaunkalküle

Aussagenlogik Einfache automatische Beweismethoden.

Implementierte Deduktionssysteme

Gleichheitslogik Kalküle dazu.

Deduktionssysteme für Logik höherer Ordnung

1.4 Teilgebiete

Im folgenden eine Liste von Teilgebieten:

Beweiserbau Die Softwaretechnik der Automatischen Deduktionssysteme. Hier werden Fragen zur Softwaretechnik von Deduktionssystemen, effizienten Implementierungen, Datenstrukturen, Heuristiken untersucht. Ebenso Fragen der Benutzung: interaktiv, vollautomatisch, halbautomatisch, verifizierend, Erzeugung von lesbaren Beweisen.

Gleichheitsbehandlung Theorie und Verfahren zur Behandlung der Gleichheitsaxiomen. Für Herleitung reichen naiven Suchstrategien nicht aus. Aber: diese Relation kommt häufig in mathematischen Disziplinen und in der theoretischen Informatik vor.

Termersetzungssysteme Themengebiet sind gerichtete Gleichungen, deren Benutzung als Transformationssystem oder zum Entscheiden des Wortproblems. Kann man auch als Untergebiet der Gleichheitsbehandlung ansehen.

Unifikation Zentrale Fragestellung ist, ob zwei vorgegebene Strukturen mit offenen Stellen (Variablen) gleichgemacht werden können. Die Unifikationstheorie liefert spezielle Algorithmen und das theoretische Rüstzeug für die Untersuchung solcher Probleme. Diese spezielle Fragestellung tritt verbreiteter auf, als man annimmt: Frage der gemeinsamen Instanzen von Pattern, Lösungen von Gleichungen, Logiken erster Ordnung, Gleichungstheorien, Lambda-Kalkül.

Nichtmonotone Logiken Eine Logik heißt monoton, wenn durch die Hinzunahme neuer Annahmen nicht weniger folgt als ohne diese Annahmen: $\Phi \models TH$ impliziert $\Phi \cup \Xi \models TH$. Die klassischerweise untersuchten Logiken sind monoton, viele Herleitungsprobleme der Künstlichen Intelligenz jedoch nicht.

Logische Methoden wissensbasierter Systeme Die wichtigsten Probleme sind die Entwicklung von Inferenzmethoden für wissensbasierte Systeme und die Mechanisierung nichtklassischer Logiken.

Modallogik Dies sind Gegenstand älterer logische/philosophische Untersuchungen. Erst durch den Versuch, geeignete Formulierungssprachen zur computergerechten Eingabe von verschiedenen Problemen zu finden, wurden diese Logiken wiederbelebt und für die Computerverarbeitung entdeckt.

Deduktive Programmsynthese Idee: Mit Hilfe deduktiver Verfahren ein Programm automatisch zu synthetisieren. Z.B.: In der konstruktiven Logik höherer Ordnung entsprechen Existenzbeweise einem ablauffähigen Programm.

Logisches Programmieren Die Prädikatenlogik, insbesondere das Fragment der Hornklauseln, wird als (Grundlage einer) Programmiersprache (Prolog) aufgefaßt. Implementierungen, Erweiterungen (Negation, Constraints, ..), werden untersucht.

Anwendungen Automatischer Deduktionssysteme

- logische Programmiersprachen, Constraints + logische Programmiersprachen.
- deduktiven Datenbanken
- Planen, Plangenerierung in Systemen der KI.
- Diagnose.
- Nachweis der Korrektheit von Hard- oder Software:
 - Programmverifikation
 - Programmanalyse
 - Untersuchung von Sicherheitsprotokollen
- Inferenzkomponenten für wissensbasierte Systeme
- Wissensrepräsentation formalismen

Wichtigste Methoden

Resolution

Tableaukalküle

Gleichheit Termersetzung, Paramodulation, Superposition

Modelchecking

1.5 Einige computergeführte Beweise

Einige lange Zeit offene Problemen, die mit Computerhilfe gelöst wurden, (allerdings nicht mit allgemeinen automatischen Deduktionssystemen), sind:

- **Der Vierfarbensatz:**
jede ebene Landkarte läßt sich mit vier Farben färben. Die Lösungsmethode war ein spezielles Programm, das einige tausend Konfigurationen zu untersuchen hatte.

- **Nichtexistenz einer endlichen projektiven Ebene der Ordnung 10.**
Hintergrund ist die Axiomatisierung der projektiven Ebenen:

1. Durch je zwei Punkte geht genau eine Gerade.
2. Je zwei Geraden schneiden sich in einem Punkt.
3. Es gibt 4 verschiedene Punkte von denen keine drei kollinear sind.

Stichwortartig einige Eigenschaften: Ordnung sei n ; $n + 1 =$ Anzahl der Punkte auf einer Geraden $=$ Anzahl der Geraden durch einen Punkt. Alle Geraden haben gleichviele Punkte ($n + 1$). Durch alle Punkte gehen gleich viele Geraden ($n + 1$). Anzahl aller Punkte $=$ Anzahl aller Geraden $= n^2 + n + 1$. Man kennt zu jeder Primzahlpotenz p^n (mindestens) eine projektive Ebene dieser Ordnung. Aber: Man weiß nicht, ob es eine endliche projektive Ebene zu einer Ordnung gibt, die keine Primzahlpotenz ist. Die kleinste Ordnung für die dies lange Zeit unbekannt war, ist 10. Die nächste offene Ordnung ist 12.

- Die “Kepler-Vermutung“: Diese Vermutung ist (war) bereits über dreihundert Jahre alt.

Sie besteht darin, daß die dichteste Kugelpackung im Raum die ist, die man durch Probieren sofort findet:

Nämlich die Art der Stapelung von Melonen oder von Kanonenkugeln, die zu einer Pyramide führt: In der ersten Ebene sind die Kugeln wabenförmig aneinandergelegt. Die nächste Ebene sieht genauso aus, nur leicht versetzt, so daß die Kugeln der zweiten Ebene in die Täler der ersten Ebene fallen, usw. In diesem Fall werden $\frac{\pi}{\sqrt{18}} \sim 0,74048$ des Raumes von Kugeln gefüllt.

Der mathematische Fallstrick ist offenbar, daß die lokale Minimierung des Volumens, das von einer Kugel und den (maximal) 12 umgebenden Kugeln benötigt wird, eine scheinbar bessere Packung ergibt; allerdings läßt sich diese nicht fortsetzen.

Die Kepler-Vermutung wurde gelöst von Tom Hales und seinem Doktoranden Sam Ferguson (University of Michigan) (weitere Info unter <http://www.math.lsa.umich.edu/~hales>, siehe ebenso einen aktuellen “Zeit“-Artikel)

Die Lösung erforderte umfangreiche mathematische Vorarbeit, bis man das Problem darauf reduziert hat, Kugelhaufen mit 53 Kugeln zu untersuchen. Hiervon gibt es 5000 Typen, die alle untersucht werden müssen. Dies wurde mit Hilfe eines Computers durchgeführt.

Bei diesen Problemen wurden spezialisierte Systeme entwickelt, die eine Art logisches Numbercrunching durchgeführt haben. Beim Vierfarbensatz und beim Kepler-Problem nach einer erheblichen mathematischen Vorarbeit, beim projektiven-Ebenen-Problem-10 ist es geordnetes, optimiertes Durchprobieren von sehr vielen Möglichkeiten bei der Suche nach einem Modell.

Automatische Deduktionssysteme, die allgemeiner konstruiert sind, haben auch schon Spitzenleistungen erzielt: z.B. Lösung des sogenannten Problems der Robbins-Algebren:

1.6 Das Problem der Robbins Algebren

Das Robbins Problem “sind alle Robbins-Algebren auch Boolesche Algebren?” wurde von William McCune, Automated Deduction Group, Mathematics and Computer Science Division, Argonne National Laboratory gelöst und zwar mit Hilfe des automatischen Beweisers EQP für Gleichungstheorien. Die Lösung ist:

Jede Robbins-Algebra ist eine Boolesche Algebra.

Zunächst: Eine Boolesche Algebra hat zwei zweistellige, kommutative, assoziative Operatoren **und**, **oder**, einen einstelligen Operator **not**, Konstanten 0,1, so daß folgendes gilt: die De-Morganschen Gesetze, Distributivität, **not** ist ein Komplementoperator: $x \text{ und } (\text{not } x) = 0$, $x \text{ oder } (\text{not } x) = 1$, $\text{not}(\text{not } x) = x$.

Etwas Geschichte:

Im Jahre 1933 gab E. V. Huntington die folgende (alternative) Axiomatisierung einer Booleschen Algebra:

$$\begin{array}{ll} x + y = y + x & [\text{kommutativ}] \\ (x + y) + z = x + (y + z) & [\text{assoziativ}] \\ n(n(x) + y) + n(n(x) + n(y)) = x & [\text{Huntington Gleichung}] \end{array}$$

In dieser Axiomatisierung kann man **und** mittels **oder** und **not** definieren, und nachweisen, dass alle Axiome einer Booleschen Algebra dafür gelten.

Danach vermutete Robbins, dass man das letzte Axiom durch eine anderes ersetzen kann:

$$n(n(x + y) + n(x + n(y))) = x \text{ [Robbins equation]}$$

Robbins und Huntington konnten keinen Beweis finden, später arbeitete Tarski mit Studenten an diesem Problem, ebenfalls ohne Erfolg.

Die Algebren mit den Axiomen Kommutativität, Assoziativität und Robbins Axiom wurden bekannt als Robbins Algebren. Man sieht leicht, dass Boolesche Algebren auch Robbins Algebren sind; die interessante offene Frage war die Umkehrung.

Seit 1979 arbeitete daran: Steve Winker, Larry Wos mit dem Beweiser “Otter” Die Methode war schrittweise immer stärkere Bedingungen (d.h. zusätzliche Axiome) zu finden, unter denen eine Robbins Algebra auf jeden Fall auch eine Boolesche Algebra ist. Hierbei wurden die Beweise vollautomatisch gesucht. Winker fand viele solche Bedingungen und bewies dies unter Benutzung von Otter : U.a.:

$$\begin{array}{ll} \forall x : n(n(x)) = x & \\ \exists 0 : \forall x : x + 0 = x & \\ \forall x : x + x = x & \\ \exists C : \exists D : C + D = C & [\text{erste Winker-Bedingung, 1992}] \\ \exists C : \exists D : n(C + D) = n(C) & [\text{zweite Winker-Bedingung, 1996}] \end{array}$$

Der automatische Beweis der Lösung des Robbins Problem wurde am 10 Oktober 1996 mittels EQP geführt. EQP benutzt assoziative-kommutative Unifikation und kann nur Gleichheitslogik.
Die Gleichungen

$$\begin{array}{ll} n(n(y) + x) + n(x + y) = x & \text{[Robbins Axiom]} \\ n(x + y) \neq n(x) & \text{[Negation von Winker-2]} \end{array}$$

wurden von EQP zum Widerspruch geführt. Die Suche dauerte 8 Tage. Ein Problem war die Prüfung des von EQP gefundenen Beweises. Hierzu mußte mittels Otter ein (maschinen-)lesbarer Beweis erzeugt werden, der dann wieder von einem anderen Beweisprüfprogramm nochmal geprüft wurde.

Für weitere Informationen, siehe die zugehörigen Web-Seiten:

`www-unix.mcs.anl.gov/mccune/papers/robbins/lemmas.html`
`www.mcs.anl.gov/AR/eqp`
`www.mcs.anl.gov/AR/otter`

1.7 Methodenwahl: menschliche oder maschinelle?

Was wird erfolgreicher sein?

- Die Nachahmung der menschlichen Vorgehensweise, oder
- Die (intelligente) brute-force (numbercrunching)-Methode

Die bisherige Erfahrung zeigt, daß, (wenn man einmal vom Einsatz eines Rechners wie beim 4-Farben-Problem absieht), der Einsatz von Computern erfolgreicher ist, wenn man nicht das menschliche Problemlösungsverhalten nachahmt, sondern die zweite Variante, die (intelligente) brute-force Methode verwendet. Z. B. sind die Erfolge der Schachprogrammierung eher auf solche Methoden als auf rational planende und nachdenkende Methoden zurückzuführen.

Genauso verhält es sich z.B. mit der Lösung des Robbins-Algebra-Problems. Wobei man hier einwenden könnte, daß ohne einen Menschen, der das System EQP intelligent einsetzt, der Beweis nicht gefunden worden wäre.

Es gibt einen Zweig des “Automated Reasoning“, der gute Gründe dafür anführt, daß eine Analogie zum menschlichen Planen notwendig sei; allerdings sind die großen Erfolge hier bisher ausgeblieben.

1.8 Einführende Beispiele

Aussagen über die reale Welt oder über fiktive oder abstrakte Welten sind manchmal nicht unabhängig voneinander, sondern es kann der Fall sein, daß eine Aussage aus einer oder mehreren anderen Aussage folgt. Zum Beispiel würde man bei den Aussagen

“Jede Katze ißt Fische“
 “Garfield ist eine Katze“
 “Garfield ißt Fische“

sicherlich zustimmen, daß die dritte Aussage aus den beiden anderen folgt. Wann immer man annimmt, daß die ersten beiden Aussagen wahr sind, muß man notwendigerweise auch annehmen, daß die dritte Aussage wahr ist. Das gleiche gilt für folgende Aussagen

“Alle Menschen sind sterblich“
 “Sokrates ist ein Mensch“
 “Sokrates ist sterblich“.

Wieder gilt, daß die dritte Aussage aus den ersten beiden folgt. Das interessante dabei ist, daß der Grund weshalb die dritte Aussage aus den beiden anderen folgt offensichtlich in beiden Fällen derselbe ist. Es hängt gar nicht davon ab, ob wir über Fische essende Katzen oder sterbliche Menschen oder über irgendwelche unbekannten Objekte mit unbekannten Eigenschaften sprechen:

“Alle X haben die Eigenschaft E“
 “A ist ein X“
 “A hat die Eigenschaft E“

Offensichtlich folgt die dritte Aussage aus den beiden anderen. Die Beobachtung, daß die “folgt aus“ Beziehung zwischen verschiedenen Aussagen gar nicht notwendigerweise aus deren Bedeutung definiert werden muß, sondern nur aus deren syntaktischer Form geht zurück auf Aristoteles und andere antike griechische Philosophen. Die verblüffende Konsequenz ist, daß eine an und für sich semantische, d.h. von der Bedeutung her, definierte Beziehung, syntaktisch, d.h. über rein mechanische Symbolmanipulation, realisiert werden kann. In diesem Fall kann der Test auf die Folgerungsbeziehung zwischen Aussagen auch Computerprogrammen übertragen werden. Programme mit diesen Fähigkeiten heißen **Deduktionssysteme**. Unter einem Deduktionssystem kann man daher ganz allgemein ein programmierbares Verfahren verstehen, das aus vorhandenen Daten Schlußfolgerungen zieht und damit neue Daten ableitet. Welche Art von Schlußfolgerungen gemeint ist, illustriert “Modus Ponens“, eine typische Regel, der auch die beiden obigen Beispiele zum Teil zugrundeliegen, und die schon auf Aristoteles zurückgeht.

Fakt 1:	P	(P und Q sind beliebige Aussagen.)
Fakt 2:	Aus P folgt Q	
Schlußfolgerung:	Q	

Die Mechanismen, die einen Rechner zu solchen Ableitungen befähigen, sind im Prinzip sehr ähnlich zu denen, die ihn zum Rechnen mit Zahlen befähigen. Die Fakten müssen als Datenobjekte repräsentiert und die Schlußfolgerungen als Operationen auf diesen Datenobjekten programmiert werden.

1.9 Motivationsbeispiele

Um einen allerersten Eindruck von der Arbeitsweise eines Deduktionssystems zu bekommen, sollen zwei Beispiele zu deren Anwendung vorgestellt werden. Das

erste ist das Sokrates-Beispiel, und das zweite ist ein logisches Rätsel. Die beiden Beispiele sollen nur einen ungefähren Eindruck geben und demonstrieren, daß man offensichtlich nicht nur Fische fressende Garfields oder sterbliche Philosophen, sondern wirklich nichttriviale logische Zusammenhänge mit automatisierbaren Methoden behandeln kann.

Beispiel 1.1. Wir wollen beweisen: Aus

“Alle Menschen sind sterblich“
 “Sokrates ist ein Mensch“

folgt

“Sokrates ist sterblich“.

Zunächst benötigt man eine formale Sprache, in der man diesen Sachverhalt ausdrücken kann. Dafür bietet sich in diesem Fall die Prädikatenlogik an. Das ist die formale Sprache, in der die meisten mathematischen Zusammenhänge einfach aufgeschrieben werden können, und die daher den meisten Lesern sowieso mehr oder weniger vertraut sein sollte. Da es in diesem Beispiel um zwei Eigenschaften geht, führen wir **Mensch** und **sterblich** als Prädikatensymbole ein. **Sokrates** ist ein Konstantensymbol in der Sprache. Die drei Aussagen sind dann:

- 1.) $\forall x : \text{Mensch}(x) \Rightarrow \text{sterblich}(x)$
- 2.) $\text{Mensch}(\text{Sokrates})$
- 3.) $\text{sterblich}(\text{Sokrates})$

Der herzuleitende Aussage wird jetzt, um einen Beweis durch Widerspruch zu demonstrieren, verneint. Damit hat man eine Menge von Aussagen, auf die man das mechanische Verfahren der Resolution anwenden kann. Das Ziel ist es einen Widerspruch herzuleiten. Hat man dies erreicht, dann kann man sagen, daß die verneinte Aussage aus den anderen folgt.

- 1.) $\forall x : \text{Mensch}(x) \Rightarrow \text{sterblich}(x)$
- 2.) $\text{Mensch}(\text{Sokrates})$
- 3.) $\neg \text{sterblich}(\text{Sokrates})$

Zunächst muß man die Formeln noch etwas abändern:

- 1.) $\forall x : \neg \text{Mensch}(x) \vee \text{sterblich}(x)$
- 2.) $\text{Mensch}(\text{Sokrates})$
- 3.) $\neg \text{sterblich}(\text{Sokrates})$

Das mechanische Verfahren der Resolution erlaubt es jetzt, aus 1) und 3) durch instantiieren von x mit **Sokrates** die Formel

$\neg \text{Mensch}(\text{Sokrates})$.

herzuleiten. Diese Formel und die zweite erlauben es jetzt, den endgültigen Widerspruch direkt zu erkennen.

Beispiel 1.2. Wise Men Puzzle Es war einmal ein König, der hatte drei weise Männer. Eines Tages beschloss er, herauszufinden, wer von den dreien der weiseste ist. Er stellt sie einander gegenüber, macht jedem von ihnen einen weißen oder schwarzen Punkt auf die Stirn und sagt ihnen, daß mindestens einer der Punkte weiß ist. Nach einer Weile, in der niemand was sagte, fragte er den ersten, ob er die Farbe seines Punktes wüßte. Der verneinte. Dann fragte er den zweiten. Der verneinte ebenfalls. Daraufhin sagte der dritte prompt, daß sein Punkt weiß ist. Woher wußte der das?

Zur Lösung dieses Rätsels ist es notwendig, Schlüsse über den Wissensstand der Beteiligten zu ziehen. Zur Formulierung dieses Rätsels ist die prädikatenlogische Sprache daher nicht sehr gut geeignet. Einfacher wird es, wenn man die logische Sprache um neue Operatoren erweitert, mit denen man ausdrücken kann, daß jemand eine bestimmte Sache weiß oder nicht weiß. Damit macht man aus der Prädikatenlogik eine sogenannte *epistemische Logik*. Dies führt zur sogenannten Modallogik und deren Varianten. Es gibt Methoden, die Aussagen dieser Logik wieder in Prädikatenlogik übersetzen.

2 Aussagenlogik (propositional calculus)

Bevor wir im nächsten Kapitel den Prädikatenkalkül und die zugehörigen Deduktionsmethoden vorstellen, wollen wir zunächst die Methoden und Konzepte in der etwas einfacheren Variante der Logik, nämlich in Aussagenlogik, uns genauer anschauen. Die Verallgemeinerung auf Prädikatenlogik ist dann nicht mehr ein so großer Schritt.

Dieser Abschnitt geht zwar detailliert auf Aussagenlogik, Kalküle und Eigenschaften ein, aber das Ziel ist es nicht, optimale und effiziente Verfahren für die Aussagenlogik vorzustellen.

Definition 2.1. Syntax

Die Syntax ist gegeben durch die Grammatik:

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

Hierbei ist X ein Nichtterminal für aussagenlogische Variablen und A ein Nichtterminal für Aussagen. Die Konstanten $0, 1$ entsprechen falsch bzw. wahr. Üblicherweise werden bei der Notation von Aussagen Klammern weggelassen, wobei man die Klammerung aus den Prioritäten der Operatoren wieder rekonstruieren kann: Die Prioritätsreihenfolge ist: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$.

Die Zeichen $\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$ nennt man Junktoren. Aussagen der Form $0, 1$ oder x nennen wir Atome. Literal sind entweder Atome oder Aussagen der Form $\neg A$, wobei A ein Atom ist.

- $A \wedge B$: Konjunktion (Verundung).
- $A \vee B$: Disjunktion (Veroderung).
- $A \Rightarrow B$: Implikation .
- $A \Leftrightarrow B$: Äquivalenz.
- $\neg A$: negierte Formel.
- A Atom, falls A eine Variable ist.
- A Literal, falls A Atom oder negiertes Atom.

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 2.2. Semantik Zunächst definiert man für jede Operation $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$ Funktionen $\{0, 1\} \rightarrow \{0, 1\}$ bzw. $\{0, 1\}^2 \rightarrow \{0, 1\}$. 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 2.3. Eine Interpretation I ist eine Funktion: $I : \{\text{aussagenlogische Variablen}\} \rightarrow \{0, 1\}$.

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

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

Wenn für eine Aussage F und eine Interpretation I gilt: $I(F) = 1$, dann schreiben wir auch: $I \models F$. Sprechweisen: I ist ein Modell für F , F gilt in I , I macht F wahr.

Definition 2.4. Sei A ein Aussage.

- A ist eine Tautologie (Satz, allgemeingültig) gdw. für alle Interpretationen I gilt: $I \models A$.
- A ist ein Widerspruch (widersprüchlich, unerfüllbar) gdw. für alle Interpretationen I gilt: $I(A) = 0$.
- A ist erfüllbar (konsistent) gdw. es eine Interpretationen I gibt mit: $I \models A$.
- ein Modell für eine Formel A ist eine Interpretation I mit $I(A) = 1$.

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

Beispiel 2.5.

- $X \vee \neg X$ ist eine Tautologie.
- $(X \Rightarrow Y) \Rightarrow ((Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z))$ ist eine Tautologie.
- $X \wedge \neg X$ ist ein Widerspruch.
- $X \vee Y$ ist erfüllbar.
- I mit $I(X) = 1, I(Y) = 0$ ist ein Modell für $X \wedge \neg Y$

Satz 2.6.

- Es ist entscheidbar, ob eine Aussage eine Tautologie (Widerspruch, erfüllbar) ist.
- Die Frage “Ist A erfüllbar?” ist \mathcal{NP} -vollständig.
- Die Frage “Ist A Tautologie (Widerspruch)?” ist $\text{co-}\mathcal{NP}$ -vollständig.

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

Zur Erläuterung: \mathcal{NP} -vollständig bedeutet, daß jedes Problem der Problemklasse mit einem nicht-deterministischen Verfahren in polynomialer Zeit gelöst werden kann, und daß es keine bessere obere Schranke gibt.

Eine Problemklasse ist $\text{co-}\mathcal{NP}$ -vollständig, wenn die Problemklasse die aus den Negationen gebildet wird, \mathcal{NP} -vollständig 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.

Die Klasse der \mathcal{NP} -vollständigen Probleme ist vom praktischen Standpunkt aus leichter als $\text{co-}\mathcal{NP}$ -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 $\text{co-}\mathcal{NP}$ -vollständige Probleme muß man i.a. viele Möglichkeiten testen: Z.B. muß man i.a. alle Interpretation einer Formel ausprobieren.

2.1 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 2.7. 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: wenn für alle Formeln $F \in \mathcal{F}$ die Auswertung $I(F) = 1$ ergibt, dann auch $I(G) = 1$.

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

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

Satz 2.8. $\{F_1, \dots, F_n\} \models G$ gdw. $F_1 \wedge \dots \wedge F_n \Rightarrow G$ ist Tautologie.

Zwei Aussagen F, G nennen wir *äquivalent*, gdw. wenn $F \Leftrightarrow G$ eine Tautologie ist. Beachte, daß F und G genau dann äquivalent sind, wenn für alle Interpretationen I : $I \models F$ 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 2.9. Sei \mathcal{F} eine Menge von (aussagenlogischen) Formeln und G eine weitere Formel. Gegeben sei ein (nicht-deterministischer) Algorithmus \mathcal{A} , 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 $\mathcal{H} \vdash_{\mathcal{A}} H$ oder auch $\mathcal{H} \rightarrow_{\mathcal{A}} H$).

- Der Algorithmus \mathcal{A} ist korrekt (sound), gdw. $\mathcal{H} \rightarrow_A H$ impliziert $\mathcal{H} \models H$
- 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 $\{F_1, \dots, F_n\}$, zähle alle Formeln F auf, prüfe, ob $F_1 \wedge \dots \wedge F_n \Rightarrow F$ 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; Davis-Putnam Verfahren: Fallunterscheidung mit Simplifikationen. Tableaueinkalkül, der Formeln syntaktisch analysiert (analytic tableau).

In dieser Vorlesung sind die letzten beiden Verfahren von Interesse, da sie auf den prädikatenlogischen Fall verallgemeinerbar sind und wir diese Verallgemeinerungen auch betrachten wollen.

2.2 Tautologien und einige einfache Verfahren

Wir wollen im folgenden Variablen in Aussagen nicht nur durch die Wahrheitswerte T, F 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: $A[B_1/x_1, \dots, B_n/x_n]$, 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 $A[B_1/x_1, \dots, B_n/x_n]$ die gleiche Funktion wie $A \circ (B_1, \dots, B_n)$.

Satz 2.10. Sind A_1, A_2 äquivalente Aussagen, und B_1, \dots, B_n weitere Aussagen, dann sind $A_1[B_1/x_1, \dots, B_n/x_n]$ und $A_2[B_1/x_1, \dots, B_n/x_n]$ ebenfalls äquivalent.

Beweis. Man muß zeigen, daß alle Zeilen der Wahrheitstabellen für die neuen Ausdrücke gleich sind. Seien y_1, \dots, 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. \square

Satz 2.11. Sind A, B äquivalente Aussagen, und F eine weitere Aussage, dann sind $F[A]$ und $F[B]$ ebenfalls äquivalent.¹

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

$$\begin{aligned}\mathcal{F} \wedge \mathcal{G} &\Leftrightarrow \mathcal{G} \wedge \mathcal{F} \\ \mathcal{F} \wedge \mathcal{F} &\Leftrightarrow \mathcal{F} \\ \mathcal{F} \wedge (\mathcal{G} \wedge \mathcal{H}) &\Leftrightarrow (\mathcal{F} \wedge \mathcal{G}) \wedge \mathcal{H} \\ \mathcal{F} \vee \mathcal{G} &\Leftrightarrow \mathcal{G} \vee \mathcal{F} \\ \mathcal{F} \vee (\mathcal{G} \vee \mathcal{H}) &\Leftrightarrow (\mathcal{F} \vee \mathcal{G}) \vee \mathcal{H} \\ \mathcal{F} \vee \mathcal{F} &\Leftrightarrow \mathcal{F}\end{aligned}$$

Weiterhin gibt es für Aussagen noch Rechenregeln und Gesetze, die wir im folgenden auflisten wollen. Alle lassen sich mit Hilfe der Wahrheitstablen 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.

Lemma 2.12 (Äquivalenzen:).

$$\begin{aligned}\neg(\neg A) &\Leftrightarrow A \\ A \Rightarrow B &\Leftrightarrow \neg A \vee B \\ A \Leftrightarrow B &\Leftrightarrow A \Rightarrow B \wedge B \Rightarrow A \\ \neg(A \wedge B) &\Leftrightarrow \neg A \vee \neg B && (DeMorgansche\ Gesetze) \\ \neg(A \vee B) &\Leftrightarrow \neg A \wedge \neg B \\ A \wedge (B \vee C) &\Leftrightarrow (A \wedge B) \vee (A \wedge C) && Distributivität \\ A \vee (B \wedge C) &\Leftrightarrow (A \vee B) \wedge (A \vee C) && Distributivität \\ (A \Rightarrow B) &\Leftrightarrow (\neg B \Rightarrow \neg A) && Kontraposition \\ A \vee (A \wedge B) &\Leftrightarrow A && Absorption \\ A \wedge (A \vee B) &\Leftrightarrow A && Absorption\end{aligned}$$

Diese Regeln sind nach Satz 2.10 nicht nur verwendbar, wenn $A; B; C$ Aussagevariablen sind, sondern auch, wenn A, B, C für Aussagen stehen.

2.3 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.

¹ Hierbei ist $F[B]$ die Aussage, die dadurch entsteht, daß in F die Subaussage A durch B ersetzt wird.

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

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

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

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

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

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

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

Satz 2.13. *Eine Klausel C ist eine Tautologie genau dann wenn es eine Variable A gibt, so daß sowohl A als auch $\neg A$ in der Klausel vorkommen*

Beweis. Übungsaufgabe.

Es gilt:

Satz 2.14. *Zu jeder Aussage kann man eine äquivalente CNF finden und ebenso eine äquivalente DNF. Allerdings nicht in eindeutiger Weise.*

Satz 2.15.

- *Eine Aussage in CNF kann man in linearer Zeit auf Tautologie-eigenschaft testen.*
- *Eine Aussage in DNF kann man in linearer Zeit auf Unerfüllbarkeit testen.*

Beweis. Eine CNF $C_1 \wedge \dots \wedge C_n$ 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. \square

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) Definition, 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.

Satz 2.16. *Sei F eine Formel. Dann ist F allgemeingültig gdw. $\neg F$ ein Widerspruch ist*

Bemerkung 2.17. Aus Lemma 2.15 kann man Schlußfolgerungen ziehen über die zu erwartende Komplexität eines Algorithmus, der eine Formel (unter Äquivalenzerhaltung) in eine DNF (CNF) transformiert. Wenn dieser Algorithmus polynomial wäre, dann könnte man einen polynomialen Tautologietest durchführen:

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

Dies wäre ein polynomieller Algorithmus für ein co-NP -vollständiges Problem.

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

Definition 2.18. Transformation in Klauselnormalform

Folgende Prozedur wandelt jede aussagenlogische Formel in konjunktive Normalform (CNF, Klauselnormalform) um:

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

$$F \Leftrightarrow G \rightarrow F \Rightarrow G \wedge G \Rightarrow F$$

und

$$F \Rightarrow G \rightarrow \neg F \vee G$$

2. *Negation ganz nach innen schieben:*

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

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:

$$\begin{array}{l} (L_{1,1} \vee \dots \vee L_{1,n_1}) \\ \wedge (L_{2,1} \vee \dots \vee L_{2,n_2}) \\ \wedge \dots \\ \wedge (L_{k,1} \vee \dots \vee L_{k,n_k}) \end{array}$$

oder in (Multi-)Mengenschreibweise:

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

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:

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

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

Beispiel 2.19.

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

2.4 Lineare CNF

Wir geben einen Algorithmus an, der eine aussagenlogische Formel F in polynomialer Zeit in eine DNF F_{DNF} umwandelt, wobei die Formel F Tautologie ist gdw. F_{DNF} eine Tautologie ist. Es ist hierbei nicht gefordert, daß F und F_{DNF} 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 $(G \Leftrightarrow A) \Rightarrow F[A]$, wobei A eine neue aussagenlogische Variable ist.

Satz 2.20. $F[G]$ ist eine Tautologie gdw. $(G \Leftrightarrow A) \Rightarrow F[A]$ eine Tautologie ist. Hierbei muß A eine Variable sein, die nicht in $F[G]$ vorkommt.

Beweis. “ \Rightarrow ” Sei $F[G]$ eine Tautologie und sei I eine beliebige Interpretation. Werte die Formel $(G \Leftrightarrow A) \Rightarrow F[A]$ unter I aus: Wenn $I(G \Leftrightarrow A) = 0$, dann ist die Formel unter I wahr. Wenn $I(G \Leftrightarrow A) = 1$, dann ist $I(F[A]) = I(F[G]) = 1$, da $F[G]$ eine Tautologie ist.

“ \Leftarrow “ Sei $(G \Leftrightarrow A) \Rightarrow F[A]$ eine Tautologie und I eine Interpretation der Variablen von $F[G]$. Wähle eine Interpretation I' , die wie I ist, aber für A so gewählt ist, daß $I(A) = I(G)$. Da $(G \Leftrightarrow A) \Rightarrow F[A]$ eine Tautologie ist, erhält man $I'((G \Leftrightarrow A) \Rightarrow F[A]) = 1$. Da $I'(G \Leftrightarrow A) = 1$, kann nur $I'(F[A]) = 1$ gelten. Da außerdem $I'(G) = I(A)$, erhält man $I'(F[G]) = I'(F[A]) = 1$. D.h. Für alle Interpretationen I ist $F[G]$ gültig. \square

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.

Definition 2.21. 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 $H \equiv (G_1 \text{ op } G_2)$, dann sind G_1, G_2 Subformeln der Tiefe $n + 1$, wobei op einer der Junktoren $\vee, \wedge, \Rightarrow, \Leftrightarrow$ sein kann.

Die Tiefe einer Formel sei die maximale Tiefe einer Subformel.

Wir wollen dieses Lemma jetzt auf folgende Art und Weise ausnutzen: Wegen der Dualität genügt es, den Algorithmus für DNF zu formulieren.

Definition 2.22. Schneller DNF-Algorithmus

Wenn eine Formel bereits in der Form $H_1 \vee \dots \vee H_n$ ist, und H_j eine Tiefe ≥ 4 hat, dann ersetze H_j folgendermaßen: Ersetze alle Subformeln G_1, \dots, G_m von H_j die in Tiefe 3 auftauchen, durch neue Variablen A_i : D.h. Ersetze $H_j[G_1, \dots, G_m]$ durch $\neg(G_1 \Leftrightarrow A_1) \vee \dots \vee \neg(G_m \Leftrightarrow A_m) \vee H_j[A_1, \dots, A_m]$ in der Formel $H_1 \vee \dots \vee H_n$.

Iteriere diesen Schritt, bis er nicht mehr durchführbar ist.

Danach wandle die verbliebenen Teile der Disjunktion mit Tiefe ≤ 3 in DNF um.

Die Begründung der verbesserten Komplexität ist folgendermaßen: Aus einer Formel F der Tiefe n entsteht im ersten Schritt eine Formel der Tiefe höchstens 3 und weitere Disjunktionsglieder der Form $\neg(G \Leftrightarrow A)$ (mit kleinerer Tiefe als F). D.h. nur in G kann wieder ersetzt werden. D.h. die Anzahl der neu hinzugefügten Formel ist kleiner als die Anzahl aller Subformeln der ursprünglichen Formel F . Den Aufwand zur Umformung der kleinen Formeln kann man als konstant ansehen. Da die Größe der Formel F gerade die Anzahl der Subformeln ist, ist die Anzahl der durchzuführenden Schritte linear. Je nach Datenstruktur (zur Vermeidung von Suche) kann man den Algorithmus linear formulieren und implementieren.

Dual dazu ist die wieder die schnelle Herstellung einer CNF, wobei hierbei die Eigenschaft der Unerfüllbarkeit erhalten bleibt.

Beispiel 2.23. Wandle die Formel

$$((((X \Leftrightarrow Y) \Leftrightarrow Y) \Leftrightarrow Y) \Leftrightarrow X)$$

um. Das ergibt:

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

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

Bemerkung 2.24. Eine Implementierung eines Algorithmus, der Formeln in Klauselmengen verwandelt, kann z.B. auf der www-Seite <http://spass.mpi-sb.mpg.de> gefunden werden. Das Programm heißt **FLOTTER**. Es benötigt einen gewissen Vorspann: Symbole, ...; es verlangt eine standardisierte Syntax und erzeugt dann als Ausgabe-datei eine Klauselmenge. Diese kann dann in einen weiteren Beweiser (**SPASS**) eingegeben werden. Allerdings hat **FLOTTER** einen aussagenlogischen check als Teilsystem eingebaut, so daß sich aussagenlogische Formeln sofort damit entscheiden lassen. Dieses System gibt im erfüllbaren Fall ein Modell aus.

2.5 Resolution für Aussagenlogik

Das Resolutionsverfahren dient zum Erkennen von Widersprüchen, wobei statt dem Test 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:

Satz 2.25. Eine Formel $A_1 \wedge \dots \wedge A_n \Rightarrow F$ ist allgemeingültig gdw. $A_1 \wedge \dots \wedge A_n \wedge \neg F$ widersprüchlich ist.

Beweis. Übungsaufgabe □

Die semantische Entsprechung ist:

Satz 2.26. $\{A_1, \dots, A_n\} \models F$ gdw. es keine Interpretation I gibt, so daß $I \models \{A_1, \dots, A_n, \neg F\}$

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

Resolution:

$$\frac{A \vee B_1 \vee \dots \vee B_n \quad \neg A \vee C_1 \vee \dots \vee C_m}{B_1 \vee \dots \vee B_n \vee C_1 \vee \dots \vee C_m}$$

Man nennt die ersten beiden Klauseln auch Elternklauseln und die neu hergeleitete Klausel *Resolvente*.

Auf der Ebene der Klauselmengen sieht das Verfahren so aus:

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

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.

Satz 2.27. *Wenn $C \rightarrow C'$ mit Resolution, dann ist C äquivalent zu C' .*

Beweis. Wir zeigen den nichttrivialen Teil:

Sei I eine Interpretation, die sowohl $A \vee B_1 \vee \dots \vee B_n$ und $\neg A \vee C_1 \vee \dots \vee C_m$ wahrmacht. Wenn $I(A) = 1$, dann gibt es ein C_j , so daß $I(C_j) = 1$. Damit ist auch die Resolvente unter I wahr. Analog für den Fall $I(A) = 0$. □

Satz 2.28. *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 Klauseln, da Resolution keine neuen Variablen einführt. □

Übungsaufgabe 2.29. Gebe ein Beispiel an, so daß R sich aus C_1, C_2 mit Resolution herleiten läßt, aber $C_1 \wedge C_2 \Leftrightarrow R$ 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 $\{\{A, B\}, \{A, \neg A\}, \{B, \neg B\}, \{\neg A, \neg B\}\}$. Resolution ergibt:

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

D.h. es wurden zwei tautologische Klauseln hinzugefügt. Ein Modell läßt sich direkt nicht ablesen. Zum Generieren von Modellen ist die Davis-Putnam-Prozedur (siehe 2.6) oder ein Tableau-kalkül (siehe 2.7) geeigneter.

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

Satz 2.30. *Für eine unerfüllbare Klauselmengen findet Resolution nach endlich vielen Schritten die leere Klausel.*

Beweis. Dazu genügt es anzunehmen, daß eine unerfüllbare Klauselmengen C existiert, die keine Herleitung der leeren Klausel mit Resolution erlaubt. Wir können annehmen, daß es eine kleinste Klauselmengen gibt bzgl. des Maßes $lcn(C) = \text{Anzahl der Literale} \Leftrightarrow \text{Anzahl der Klauseln}$ (Anzahl der überschüssigen Literale).

Im Basisfall (d.h. $lcn(C) = 0$) gibt es nur noch 1-Klauseln. Damit diese Klauselmengen unerfüllbar ist, muß es eine Variable A geben, so daß sowohl $\{A\}$ als auch $\{\neg A\}$ als Klausel vorkommt. Dann ist aber noch eine Resolution möglich, die die leere Klausel herleitet.

Sei also $lcn(C) > 0$. Dann betrachte eine Klausel $K \in C$, die mehr als ein Literal hat. Ersetzt man K durch K' , wobei ein Literal gestrichen ist, d.h. $K = K' \cup \{L\}$, so erhält man ebenfalls eine unerfüllbare Klauselmengen C' : Wäre C' erfüllbar mit der Interpretation I , dann auch $I \models C$. Da $lcn(C') < lcn(C)$, gibt es für C' eine Herleitung der leeren Klausel. Wenn diese Herleitung K' nicht (als Elternklausel) benötigt, dann kann man diese Herleitung bereits in C machen, also benötigt diese Herleitung die Klausel K . Übersetzt man diese Herleitung der leeren Klausel in eine Herleitung unter Benutzung von K , so erhält man eine Herleitung der 1-Klausel $\{L\}$.

Betrachtet man jetzt noch die Klauselmengen C'' , die aus C entsteht, wenn man K durch $\{L\}$ ersetzt, so sieht man wie oben: C'' ist unerfüllbar und $lcn(C'') < lcn(C)$. Damit existiert eine Herleitung der leeren Klausel in C'' . Zusammengesetzt erhält man eine Herleitung der leeren Klausel in C . \square

Satz 2.31. *Resolution erkennt unerfüllbare Klauselmengen.*

Was wir hier nicht mehr durchführen wollen, sondern auf die Behandlung der allgemeinen Resolution verschieben, ist die Verwendung von Redundanzkriterien. Z.B. Löschung von Tautologien, unnötigen Klauseln usw.

Die Komplexität im schlimmsten Fall wurde von A. Haken [Hak85] (siehe auch [Ede92] 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 polynomial lange Beweise für die Schubfachformeln haben. Es ist theoretisch offen, ob es ein Beweisverfahren gibt, das für alle Aussagen polynomial lange Beweise hat: Dies ist äquivalent zum offenen Problem $\mathcal{NP} = \text{co-}\mathcal{NP}$.

2.6 Davis-Putnam-Verfahren

Die Prozedur von Davis und Putnam zum Entscheiden der Erfüllbarkeit (und Unerfüllbarkeit) von aussagenlogische Klauselmengen beruht auf Fallunter-

scheidung und Ausnutzen und Propagieren der Information. Wenn man die Vollständigkeit des Resolutionskalküls für Prädikatenlogik inklusive einiger Redundanzregeln voraussetzt, (Subsumptionsregel, Isolationsregel), kann man die Korrektheit leicht begründen.

Definition 2.32. *Resolutionsverfahren für Aussagenlogik (Davis-Putnam Prozedur)*

Sei C eine aussagenlogische Klauselmenge. Dann wird die Davis-Putnam Entscheidungsprozedur DP folgendermaßen (rekursiv) definiert: Es ist ein Algorithmus, der eine Klauselmenge als Eingabe hat, und genau dann **true** als Ausgabe hat, wenn die Klauselmenge unerfüllbar ist.

Als Vorverarbeitungsschritt können wir annehmen, daß keine Tautologien in der Klauselmenge enthalten sind. D.h. es gibt keine Klausel, die gleichzeitig P und $\neg P$ für eine Variable P enthält.

1. (a) Wenn die leere Klausel in C ist: RETURN **true**.
 (b) Wenn C die leere Klauselmenge ist: RETURN **false**.
2. wenn es in eine 1-Klausel $\{P\}$ ($\{\neg P\}$) gibt, wobei P eine Variable ist, dann verändere C wie folgt:
 - (a) Lösche alle Klauseln in denen P ($\neg P$) als Literal vorkommt.
 - (b) Lösche alle Vorkommen des Literals $\neg P$ (P) in anderen Klauseln.
 Die resultierende Klauselmenge sei C' . RETURN $DP(C')$
3. Wenn es isolierte Literale² gibt, wende die Löschregel für isolierte Literale an. D.h. lösche die Klauseln, in denen isolierte Literale vorkommen. Die resultierende Klauselmenge sei C' .
 RETURN $DP(C')$
4. Wenn keiner der obigen Fälle zutrifft, dann wähle eine noch in C vorkommende aussagenlogische Variable P aus.
 RETURN $DP(C \cup \{P\}) \wedge DP(C \cup \neg P)$

Dies ist ein vollständiges, korrektes Entscheidungsverfahren für die (Un-)Erfüllbarkeit von aussagenlogischen Klauselmengen. Punkt 2a) entspricht der Subsumption, Punkt 2b) ist Resolution mit anschließender Subsumption. Punkt 3) ist der Spezialfall der isolierten Literale und 4)) ist eine Fallunterscheidung, ob P wahr oder falsch ist. Diese DP-Prozedur ist im allgemeinen sehr viel besser als eine vollständige Fallunterscheidung über alle möglichen Variablenbelegungen, d.h. besser als die Erstellung einer Wahrheitstafel. Die DP-Prozedur braucht im schlimmsten Fall exponentiell viel Zeit, 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 DP-Algorithmus ist erstaunlich schnell, wenn man bei Punkt 4) 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.

² P ist isoliert, wenn $\neg P$ nicht mehr vorkommt, entsprechend $\neg P$ ist isoliert, wenn P nicht mehr vorkommt

Der DP-Algorithmus kann leicht so erweitert werden, daß im Falle der Erfüllbarkeit der Klauselmenge auch ein Modell 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.

Beispiel 2.33. Betrachte folgende Klauselmenge, wobei jede Zeile einer Klausel entspricht.

$$\begin{array}{l}
 P, \quad Q \\
 \neg P, \quad Q, \quad R \\
 P, \quad \neg Q, \quad R \\
 \neg, \quad P \neg Q, \quad R \\
 P, \quad Q, \quad \neg R \\
 \neg P, \quad Q, \quad \neg R \\
 P, \quad \neg Q, \quad \neg R \\
 \neg P, \quad \neg Q, \quad \neg R
 \end{array}$$

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

$$\begin{array}{l}
 Q, \quad R \\
 \neg Q, \quad R \\
 Q, \quad \neg R \\
 \neg Q, \quad \neg R
 \end{array}$$

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:

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

Weitere Schritte für Q ergeben

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

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

Übungsaufgabe 2.34. Wende die DP-Prozedur auf weitere Klauselmengen an. Z.B.

$$\begin{array}{l}
\neg P \quad Q \quad R \\
P \quad \neg Q \quad R \\
\neg P \quad \neg Q \quad R \\
P \quad Q \quad \neg R \\
\neg P \quad Q \quad \neg R \\
P \quad \neg Q \quad \neg R \\
\neg P \quad \neg Q \quad \neg R
\end{array}$$

Beispiel 2.35. Wir nehmen uns ein Rätsel von Raymond Smullyan vor: Die Fragen nach dem Pfefferdieb. Es gibt drei Verdächtige: Den Hutmacher, den Schnapphasen und die Maus. Folgendes ist bekannt:

- Genau einer von ihnen ist der Dieb.
- Unschuldige sagen immer die Wahrheit
- Schnappphase: der Hutmacher ist unschuldig.
- Hutmacher: die Haselmaus ist unschuldig

Kodierung: H, S, M sind Variablen für Hutmacher, Schnappphase, Haselmaus 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 resoliert ergibt: S . Nach Prüfung, ob $\{\neg H, \neg M, S\}$ ein Modell ergibt, können wir sagen, daß der Schnappphase schuldig ist.
2. Davis Putnam: 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 2.36. 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:

höchstens einer sagt die Wahrheit:

$\neg FRW \Rightarrow FIW$
 $\neg FRW \Rightarrow HBW$
 $\neg FIW \Rightarrow FRW$
 $\neg FIW \Rightarrow HBW$
 $\neg HBW \Rightarrow FRW$
 $\neg HBW \Rightarrow FIW$

genau einer ist der Dieb:

$FID \vee FRD \vee HBD$
 $FID \Rightarrow \neg FRD$
 $FID \Rightarrow \neg HBD$
 $FRD \Rightarrow \neg FID$
 $FRD \Rightarrow \neg HBD$
 $HBD \Rightarrow \neg FID$
 $HBD \Rightarrow \neg FRD$

Die Aussagen:

$FRW \Rightarrow FID$
 $FIW \Rightarrow \neg FID$
 $HBW \Rightarrow HBD$

Die Lösung kann mit dem Davis-Putnam Verfahren gefunden werden (eine Implementierung in Haskell ist auf der [www-Seite](#) der Vorlesung).

$$FRW = 1, FIW = 2, HBW = 3, FID = 4, FRD = 5, HBD = 6$$

Da kein Parser implementiert ist: Die Kodierung als eine Klauselmengen, wobei die Variablen durch positive Zahlen und die Negierung als negative Zahlen dargestellt sind, so daß n und $\neg n$ komplementäre Literale sind.

$$\begin{aligned} &[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2], \\ &[\Leftrightarrow 4, 5, 6], [\Leftrightarrow 4, \Leftrightarrow 5], [\Leftrightarrow 4, \Leftrightarrow 6], [\Leftrightarrow 5, \Leftrightarrow 4], [\Leftrightarrow 5, \Leftrightarrow 6], [\Leftrightarrow 6, \Leftrightarrow 4], [\Leftrightarrow 6, \Leftrightarrow 5], \\ &[\Leftrightarrow 1, 4], [\Leftrightarrow 2, \Leftrightarrow 4], [\Leftrightarrow 3, 6]] \end{aligned}$$

Die berechnete Lösung ist:

$$[\Leftrightarrow 5, 6, \Leftrightarrow 4, 3, 2, \Leftrightarrow 1]$$

D.h FRW ist falsch, d.h. der Fisch hat gelogen und der Herzbube war der Dieb.

Beispiel 2.37. Anwendung auf ein Suchproblem: das n-Damen Problem.

Es 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.

Ein Programm zum Erzeugen der Klauselmengen erzeugt im Fall $n = 4$:

$$\begin{aligned} &[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16], [1, 5, 9, 13], \\ &[2, 6, 10, 14], [3, 7, 11, 15], [4, 8, 12, 16], \\ &[-1, -5], [-1, -9], [-1, -13], \\ &[-1, -2], [-1, -6], [-1, -3], [-1, -11], [-1, -4], [-1, -16], [-5, -9], [-5, -13], \\ &[-5, -2], [-5, -6], [-5, -10], [-5, -7], [-5, -15], [-5, -8], [-9, -13], \\ &[-9, -6], [-9, -10], [-9, -14], [-9, -3], [-9, -11], [-9, -12], [-13, -10], [-13, -14], \\ &[-13, -7], [-13, -15], [-13, -4], [-13, -16], [-2, -6], [-2, -10], [-2, -14], \\ &[-2, -3], [-2, -7], [-2, -4], [-2, -12], [-6, -10], [-6, -14], [-6, -3], \\ &[-6, -7], [-6, -11], [-6, -8], [-6, -16], [-10, -14], [-10, -7], [-10, -11], [-10, -15], \\ &[-10, -4], [-10, -12], [-14, -11], [-14, -15], [-14, -8], [-14, -16], [-3, -7], \\ &[-3, -11], [-3, -15], [-3, -4], [-3, -8], [-7, -11], [-7, -15], [-7, -4], [-7, -8], \\ &[-7, -12], [-11, -15], [-11, -8], [-11, -12], [-11, -16], [-15, -12], \\ &[-15, -16], [-4, -8], [-4, -12], [-4, -16], [-8, -12], [-8, -16], [-12, -16]] \end{aligned}$$

Das Ergebnis der DP-Prozedur sind zwei Interpretationen:

$$\begin{aligned} &[[-4, -8, -15, 5, -13, 14, -6, -2, 12, -9, -1, 3, -16, -10, -7, -11], \\ &[-4, 2, 8, -6, -1, 9, -12, -14, -13, -5, 15, -3, -16, -10, -7, -11]] \end{aligned}$$

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

2.7 Tableaunkalkül für Aussagenlogik

Im folgenden betrachten wir einen Kalkül, der in verschiedenen Formen und Ausprägungen viele Einsatzbereiche hat: u.a. Aussagenlogik, Prädikatenlogik, Modallogik, mehrwertige Logik, und Programmanalysen.

Ein Tableau ist i.a. eine baumförmig organisierte Datenstruktur, die mit (beliebigen) Formeln markiert ist, und die mit geeigneten Regeln aufgebaut wird. Die Formel an der Wurzel ist bewiesen (ein Tautologie), wenn das Tableau (der Baum) bestimmte Bedingungen erfüllt. Wir betrachten hier eine Variante des sogenannten analytischen Tableaunkalküls, der eine komplexe Formel Schritt für Schritt vereinfacht und am Ende Literale an den Blättern hat. Die zu überprüfende Bedingung betrifft jeweils die Formeln auf den Pfaden.

Grundbegriffe für den aussagenlogischen Tableaunkalkül sind:

- α -Formeln (konjunktive Formeln) und
- β -Formeln (disjunktive Formeln)

Beachte, daß die Negationszeichen nicht nach innen gezogen sind.

Die direkten Unterformeln der α -Formeln sind:

α	α_1	α_2
$X \wedge Y$	X	Y
$\neg(X \vee Y)$	$\neg X$	$\neg Y$
$\neg(X \Rightarrow Y)$	X	$\neg Y$
$(X \Leftrightarrow Y)$	$X \Rightarrow Y$	$Y \Rightarrow X$

Die direkten Unterformeln der β -Formeln sind:

β	β_1	β_2
$X \vee Y$	X	Y
$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$X \Rightarrow Y$	$\neg X$	Y
$\neg(X \Leftrightarrow Y)$	$\neg(X \Rightarrow Y)$	$\neg(Y \Rightarrow X)$

Es gilt: α ist äquivalent zu $(\alpha_1 \wedge \alpha_2)$, und β ist äquivalent zu $(\beta_1 \vee \beta_2)$.

Tableau-Kalkül für Aussagenlogik Ziel des Tableaunkalküls ist: Beweise, daß eine Aussage A eine Tautologie ist. Damit kann man auch zeigen, daß eine Aussage B aus einer Menge von Aussagen A_1, \dots, A_n folgt: nämlich durch den Nachweis, daß $A_1 \wedge \dots \wedge A_n \Rightarrow B$ eine Tautologie ist.

Idee beim Tableau-Kalkül: Zeige, daß die Verneinung eine inkonsistente Aussage ist.

Definition 2.38. Ein (aussagenlogisches) Tableau ist ein markierter Baum, wobei die Knoten mit aussagenlogischen Formeln markiert sind.

- Ein Pfad ist geschlossen, wenn 0 oder $\neg 1$ vorkommt, oder eine Formel X existiert, so daß auch $\neg X$ auf diesem Pfad ist.

- Ein Pfad ist (atomar) geschlossen, wenn 0 oder $\neg 1$ vorkommt, oder ein Atom A existiert, so daß auch $\neg A$ auf diesem Pfad ist. Ein Tableau ist (atomar) geschlossen, wenn alle Pfade (atomar) geschlossen sind.

Man kann die Regeln ansehen als Tableau-aufbauregeln oder als Transformationsregeln auf Tableaus. Wir werden die Sichtweise der Tableau-aufbauregeln verfolgen, denn dann gibt es keinen Unterschied zwischen der Situation beim Aufbau und der Situation des fertigen (geschlossenen) Tableaus. Im Falle von Transformationsregeln könnte es sein, daß das fertige Tableau keine Überprüfung der Aufbauregeln zuläßt.

Definition 2.39. Der Tableaukalkül TK_A hat als Eingabe eine Formel F . Initial wird ein Tableau mit einem Knoten und der Formel $\neg F$ erzeugt. Danach werden ausgehend vom initialen Tableau weitere Tableaus erzeugt mit folgenden Expansionsregeln:

$$\frac{\neg\neg X}{X} \quad \frac{\alpha}{\alpha_1} \quad \frac{\beta}{\beta_1 \mid \beta_2} \quad \frac{\neg 0}{1} \quad \frac{\neg 1}{0}$$

α_2

Diese Regeln sind wie folgt zu verstehen: Sei θ ein Pfad im Tableau T , und die obere Formel F_o eine Markierung eines Knotens auf diesem Pfad, dann erweitere das Tableau durch Verlängern des Pfades θ (d.h. Anhängen an das Blatt des Pfades) um einen mit der unteren Formel F_u markierten Knoten. Stehen unten zwei oder mehrere durch \mid getrennte Formeln, dann sollen entsprechend viele Blätter als Töchter angehängt werden, mit der jeweiligen Formel markiert. Danach verzweigt der Pfad θ am alten Blatt zu mehreren Pfaden.

Stehen zwei oder mehr Formeln untereinander, dann sollen in Folge an den Pfad θ zwei oder mehrere Blätter (mit den jeweiligen Formeln markiert) angehängt werden.

Wenn oben eine α -Formel steht, erweitere erst um α_1 , dann den neuen Knoten um α_2 . Wenn oben eine β -Formel steht, hänge zwei markierte Knoten als Töchter an, eine mit β_1 eine mit β_2 markiert.

Als Einschränkung wird verwendet, daß jede Formel auf jedem Pfad nur einmal analysiert (bzw. expandiert) wird.

Ein Formel F ist bewiesen, wenn aus dem Tableau mit einem Knoten und der Formel $\neg F$ ein geschlossenes Tableau erzeugt worden ist.

Im allgemeinen wird man die Formel direkt am Blatt markieren. Allerdings kommt es auch vor, daß eine Formel, die nicht das Blatt ist, expandiert wird.

Diese Regeln sind nicht-deterministisch, d.h. es gibt keine genaue Angabe, welche Formel zu expandieren ist. Diese Formulierung ist gewählt, um eine möglichst große Freiheit bei der Anwendung zu haben, mit der Garantie, daß die Anwendung korrekt bleibt. Allerdings sollte man in einer effizienten Implementierung zunächst die besten Schritte machen: D.h. möglichst wenig verzweigen. Dies wird durch Bevorzugung der α -Regeln erreicht. Außerdem sollte man Formeln nicht zweimal auf dem gleichen Pfad expandieren.

Beispiel 2.40. Wir zeigen ein (das) Tableau für $X \wedge \neg X$:

$$\begin{array}{c} X \wedge \neg X \\ | \\ X \\ | \\ \neg X \end{array}$$

Beispiel 2.41. Ein Tableau für $\neg(X \wedge Y \Rightarrow X)$:

$$\begin{array}{c} \neg(X \wedge Y \Rightarrow X) \\ | \\ X \wedge Y \\ | \\ \neg X \\ | \\ X \\ | \\ Y \end{array}$$

Beide Tableaus sind geschlossen.

Zur Optimierung der Analyse von Aussagen der Form $A \Leftrightarrow B$, gibt es eine bessere Alternative:

Erfinde neue Tableau-expansionsregeln

$$\frac{A \Leftrightarrow B}{\begin{array}{c|c} A & \neg A \\ B & \neg B \end{array}}$$

$$\frac{\neg(A \Leftrightarrow B)}{\begin{array}{c|c} A & \neg A \\ \neg B & B \end{array}}$$

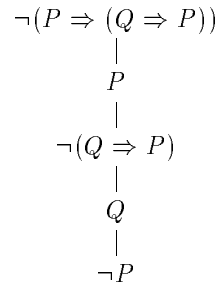
Übungsaufgabe 2.42. Gebe die Tableauregeln für XOR an.

Beachte: $A \text{ XOR } B$ ist äquivalent zu $\neg(A \Leftrightarrow B)$.

Hat man die Aufgabe zu zeigen, daß B aus A_1, \dots, A_n folgt, so kann man daraus sofort ein Tableau machen:

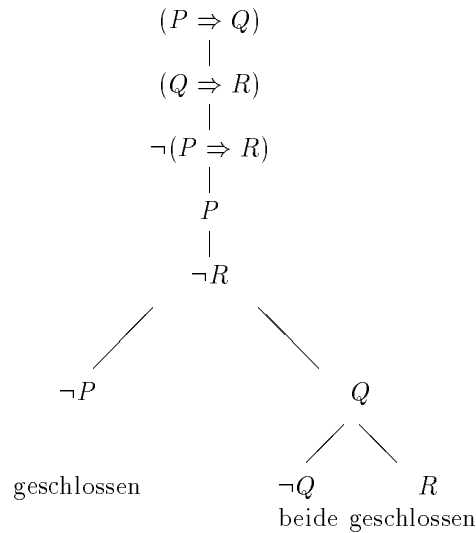
$$\begin{array}{c} A_1 \\ | \\ \dots \\ | \\ A_n \\ | \\ \neg B \end{array}$$

Beispiel 2.43. Zeige, daß $P \Rightarrow (Q \Rightarrow P)$ eine Tautologie ist:



Das Tableau ist geschlossen, da P und $\neg P$ auf dem einen Pfad liegen.

Beispiel 2.44. $((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \Rightarrow (P \Rightarrow R)$:



Der Nachweis der *algorithmischen Korrektheit des Tableaurekalküls* für Aussagenlogik besteht aus zwei Teilen:

1. Korrektheit (Soundness): Der Kalkül erzeugt geschlossene Tableaus nur für Tautologien
2. Vollständigkeit (completeness): Für jede Tautologie kann der Tableaurekalkül ein geschlossenes Tableau erzeugen.

Im folgenden verwenden wir “Korrektheit“ im Sinne der soundness.

Definition 2.45. Ein Pfad eines Tableaus ist erfüllbar, wenn die Konjunktion aller Formeln auf dem Pfad erfüllbar ist. Ein Tableau ist erfüllbar, wenn es einen Pfad gibt, der erfüllbar ist.

Beachte: Wenn eine Menge die Formel 0 oder $\neg 1$ enthält, dann ist sie nicht erfüllbar.

Ein geschlossenes Tableau ist nicht erfüllbar.

Lemma 2.46. *Für die Tableau-Expansionsregeln gilt: Wenn T zu T' expandiert wird, dann ist T erfüllbar gdw. T' erfüllbar.*

Beweis. “ \Rightarrow “ Es genügt, sich einen erfüllbaren Pfad θ anzuschauen, und eine Interpretation I der aussagenlogischen Variablen zu wählen, die alle Formeln des Pfades wahr macht und alle Fälle der Expansionsregeln durchzugehen.

- Wenn $\neg\neg X$ in θ , dann ist $I(\neg\neg X) = 1$ wahr, also auch $I(X) = 1$
- Wenn für die α -Formel $X \wedge Y$ gilt, daß $I(X \wedge Y) = 1$, dann auch $I(X) = I(Y) = 1$.
- Wenn für die β -Formel $X \vee Y$ gilt daß $I(X \vee Y)$, dann gilt auch entweder $I(X) = 1$ oder $I(Y) = 1$. Somit ist einer der Pfade, die θ fortsetzen, erfüllbar, entweder der mit dem Blatt X , oder der mit dem Blatt Y .
- Analog für die anderen α und β -Formeln.

“ \Leftarrow “: Wenn ein Pfad in T' erfüllbar ist, dann ist auch der verkürzte Pfad in T , der den Pfad T' erzeugt hat, erfüllbar. \square

Korollar 2.47. *Der Tableaurekalkül TK_A ist sound.*

Beweis. Gegeben eine Formel F , die keine Tautologie ist. Dann ist $\neg F$ erfüllbar. Der Tableaurekalkül startet mit $\neg F$, also ist das initiale Tableau erfüllbar, also auch alle daraus erzeugten, insbesondere kann kein geschlossenes Tableau erzeugt werden. \square

Die Vollständigkeit kann im Fall der Aussagenlogik auf relativ einfache Weise gezeigt werden, allerdings ist diese Beweis-methode nicht auf allgemeinere Logiken übertragbar.

Zwischenziel: Zeige die Terminierung des Tableaurekalküls für Aussagenlogik.

Bemerkung 2.48. Erinnerung:

Ein *fundierte (well-founded)* Ordnung ist eine partielle Ordnung \geq auf einer Menge M , so daß es keine unendlich absteigenden Ketten $a_1 > a_2 > \dots$ in M gibt.

Es gilt: Die *lexikographische Kombination* von fundierten Ordnungen ist wieder fundierte Ordnung. D.h. Seien M_1, \geq_1 und M_2, \geq_2 fundierte Ordnungen. Dann ist $M_1 \times M_2$ mit der Ordnung $(m_1, m_2) >_{12} (m'_1, m'_2)$ gdw. $m_1 >_1 m'_1$ oder $(m_1 =_1 m'_1$ und $m_2 >_2 m'_2)$ fundiert.

Eine weitere nützliche Konstruktion von fundierten Ordnungen gibt es mittels Multimengen, sogenannte Multimengenordnungen: Sei $(M, >)$ eine Menge mit fundierter Ordnung, dann kann man auf Multimengen (Mengen bei denen mehrfaches Vorkommen von Elementen erlaubt ist) über M eine Ordnung erklären:

Seien A und B Multimengen über M , dann definiert man $A \gg B$, wenn es weitere Multimengen X und Y gibt, so daß $B = (A \setminus X) \cup Y$ und es zu jedem Element von Y ein echt größeres Element in X gibt.

Es gilt: Die Multimengenordnung \gg ist eine partielle Ordnung. Sie ist fundiert, gdw. $>$ fundiert ist.

Z. B. Nimmt man die natürlichen Zahlen mit der $>$ -Ordnung, dann gilt $\{3, 3, 2, 1\} \gg \{3, 2, 2, 2\}$, denn $\{3, 2, 2, 2\} = \{3, 3, 2, 1\} \setminus \{3, 1\} \cup \{2, 2, 2\}$.

Mit folgender *Steuerung* terminiert der Tableauealkül: Jede Formel wird auf jedem Pfad nur einmal expandiert.

Beachte, daß eine Formel die als Markierung eines Knoten auftritt, möglicherweise mehrfach expandiert werden muß, da mehrere Pfade hindurchgehen können. Dies ist unabhängig von der Art der Formel.

Jeden Pfad können wir einem Blatt zuordnen, dem Endknoten des Pfades.

Lemma 2.49. *Der Tableauealkül für Aussagenlogik terminiert, wenn man jede Formel auf jedem Pfad höchstens einmal expandiert.*

Beweis. Wir konstruieren ein fundiertes Maß für die Größe eines Tableaus als Multimenge, so daß jede Expansionsregel dieses Maß verkleinert.

1. Die Größe einer Formel sei eine gewichtete Anzahl der Zeichen: Das Zeichen \Leftrightarrow wird doppelt gezählt, Klammern werden nicht gezählt, alle anderen einfach.
2. Die Größe eines Pfades wird gemessen durch die Multimenge der Größe der Formeln an seinen Knoten, wobei der Expansions-status für diesen Pfad noch berücksichtigt werden muß: Ein Knoten wird nur dann in das Maß aufgenommen, wenn er auf diesem Pfad noch nicht expandiert worden ist.
3. Die Größe des Tableaus ist die Multimenge der Größe aller nicht geschlossenen Pfade.

Betrachte typische Fälle der Expansionsregeln:

- $\neg\neg X \rightarrow X$ macht einen Pfad kleiner, unabhängig von der Art der Formel X , da danach $\neg\neg X$ in diesem Pfad schon expandiert wurde. D.h. im Maß des Pfades wird die Größe von $\neg\neg X$ durch die Größe von X ersetzt.
- $X \wedge Y$ wird durch X, Y ersetzt. D.h. $gr(X) + gr(Y) + 1$ wird im Maß des Pfades durch $\{gr(X), gr(Y)\}$ ersetzt. Analog für die anderen α -Formeln.
- $X \vee Y$ wird durch $X \mid Y$ ersetzt. D.h. es werden zwei Pfade erzeugt. D.h. ein Pfad θ wird durch zwei andere ersetzt. Im Maß wirkt sich das wie folgt aus: Da $gr(X \vee Y) > gr(X), gr(Y)$ für alle Formeln X, Y . Dadurch wird $gr(\theta) \gg gr(\theta_1)$ und $gr(\theta) \gg gr(\theta_2)$. Damit wird die Multimenge aller Pfade kleiner.
- $X \Leftrightarrow Y$ wird durch $X \Rightarrow Y$ und $Y \Rightarrow X$ ersetzt. Da $gr(X \Leftrightarrow Y) \Leftrightarrow 1 = gr(X \Rightarrow Y)$, kann man die gleiche Argumentation wie oben anwenden.
- Andere Fälle analog.

Das zugehörige Maß für das Tableau ist fundiert, also terminiert das Verfahren. \square

Definition 2.50. *Eine (aussagenlogische) Hintikka-Menge ist eine Menge H von aussagenlogischen Formeln, für die folgendes gilt:*

1. Es kann nicht gleichzeitig $A \in H$ und $\neg A \in H$ für ein Atom A gelten.
2. $0 \notin H, \neg 1 \notin H$
3. $\neg\neg X \in H \Rightarrow X \in H$
4. $\alpha \in H \Rightarrow \alpha_1 \in H$ und $\alpha_2 \in H$
5. $\beta \in H$ impliziert $\beta_1 \in H$ oder $\beta_2 \in H$

Lemma 2.51. (Hintikka) *Jede aussagenlogische Hintikka-Menge ist erfüllbar.*

Beweis. Wir zeigen, daß es eine Interpretation gibt, so daß jede Formel in H den Wert 1 erhält. Wenn eine Variable $A \in H$, so definiere $I(A) := 1$, wenn $\neg A \in H$, dann definiere $I(A) := 0$, wenn weder A noch $\neg A$ in H , dann definiere $I(A)$ beliebig, z.B. $I(A) := 1$. Induktion über die Termstruktur zeigt jetzt, daß für alle Formeln $F \in H$, $I(F) = 1$ gilt. \square

Lemma 2.52. *Sei T ein Tableau, auf das keine Expansionsregeln mehr angewendet werden können. Dann ist jeder Pfad entweder geschlossen oder entspricht einer aussagenlogischen Hintikka-Menge.*

Beweis. Wenn ein nicht geschlossener Pfad existiert, dann zeigen die Expansionsregeln, daß dieser Pfad dann eine Hintikka-Menge ist. Hierbei genügt entsprechend der oben angegebenen Steuerung daß jede Formel nur einmal expandiert wird. \square

Satz 2.53. *Der Tableaurekalkül für Aussagenlogik terminiert, ist korrekt und vollständig.*

Beweis. Wir haben schon die Korrektheit gezeigt. Wir betrachten den Fall, daß eine Tautologie eingegeben wird. In diesem Falle terminiert das Verfahren. Wenn ein Pfad existiert, der nicht geschlossen ist, so ist die Menge seiner Markierungen eine Hintikka-Menge, also erfüllbar, und somit auch die eingegebene Formel, was der Voraussetzung widerspricht. Also ist jeder Pfad geschlossen. \square

Lemma 2.54. *Der Tableaurekalkül konstruiert für erfüllbare Formeln ein Modell. Ein Modell kann man ablesen an jedem Pfad, der nicht geschlossen ist, aber auch nicht weiter expandiert werden kann.*

Begründung. Da erfüllbare Tableaus solange expandiert werden, bis alle Pfade entweder geschlossen sind oder eine Hintikka-Menge darstellen, erhält man zumindest einen Pfad mit einer Hintikka-Menge, wenn man das Tableauverfahren startet mit einer erfüllbaren Formel an der Wurzel. Dies ergibt ein Modell für einen Pfad. Beachtet man, daß im Beweis oben für erfüllbare Tableaus T , die zu T' expandiert werden, die Interpretation erhalten bleibt, kann man schließen, daß die in diesem Pfad enthaltenen Atome ein Modell der Formel an der Wurzel definieren.

3 Überblick über den Aufbau und Funktionsweise eines Automatischen Deduktionssystems am Beispiel eines Resolutionsbeweisers

Ein implementiertes Automatisches Deduktionssystem baut sich im wesentlichen aus vier Schichten auf – Logik, Kalkül, logisches Zustandsübergangssystem und Steuerung. Im folgenden sollen die Ideen informell skizziert werden. Später werden diese in Definitionen präzisiert.

Logik Die unterste Schicht eines Deduktionssystems wird durch eine Logik gebildet, die mit der Festlegung der Syntax einer formalen Sprache und deren Semantik die zulässige Struktur und die Bedeutung von Aussagen vorgibt. Aussagen entsprechen Formeln der Logik. Die gewählte Logik bestimmt ganz konkret, welche Arten von Aussagen erlaubt und welche verboten sind. Beispielsweise kann sie festlegen, daß eine Quantifizierung “Für alle Zahlen gilt ...” erlaubt, eine Quantifizierung “Für alle Funktionen über natürlichen Zahlen gilt ...” dagegen verboten sein soll. Die Definition einer Bedeutung (Semantik) für die Formeln liefert darüberhinaus eine Beziehung zwischen Aussagen, die als “aus A folgt B ” gedeutet werden kann. Damit ist ein *semantischer Folgerungsbegriff* etabliert, der zunächst jedoch in keiner Weise hilft, für gegebene A und B algorithmisch zu bestimmen, ob B wirklich aus A folgt. Wie in den Motivationsbeispielen schon angedeutet wurde, gibt es jedoch nicht die eine Logik, sondern ähnlich wie bei Programmiersprachen gibt es eine ganze Hierarchie von Logiken mit vielen unterschiedlichen Varianten und Erweiterungen nach verschiedenen Richtungen. Jede von ihnen formalisiert ganz bestimmte Grundkonzepte, auch wiederum ähnlich wie in Programmiersprachen. Genauso wie man sich für normale Anwendungen eine möglichst gut geeignete Programmiersprache aussucht, muß man für konkrete deduktive Anwendungen eine **adäquate Logik** auswählen. Für einfache Puzzles kann die Aussagenlogik ausreichen, während für kompliziertere Anwendungen und Beispiele (wise-man puzzle) eine Multi-Modallogik gut geeignet ist. Eng mit der Auswahl der Logik verknüpft ist die Frage der Formulierung eines Problems, d.h. “wie formuliere ich das Problem in der jeweiligen Logik am besten?”. Z. B. hätte man das wise-man puzzle auch rein in Prädikatenlogik formulieren können. Offensichtlich ist diese Formulierung schwieriger. Damit stellt sich auch die Frage; “ist das Problem überhaupt richtig formuliert?”, also nach der “Korrektheit” der Formulierung.

Die elementarste Logik ist die *Aussagenlogik*. Sie formalisiert die Grundkonzepte *wahr* und *falsch* und führt darauf die Booleschen Verknüpfungen $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \dots$ ein. In Aussagenlogik kann man z.B. “es regnet \Rightarrow Straße wird-nass” hinschreiben, aber nicht über irgendwelche Mengen quantifizieren. Das Problem, herauszufinden, ob eine Aussage aus einer anderen folgt ist in Aussagenlogik entscheidbar.

Das gilt jedoch nicht mehr in der nächstkomplizierten Logik, der *Gleichungslogik*. Neben Quantifizierungen über Mengen führt Gleichungslogik das Konzept

der Funktion ein. Man kann dann z.B. hinschreiben:

$$\forall x, y : x + 0 = x \wedge x + s(y) = s(x + y)$$

Die Formel beschreibt die Addition auf natürlichen Zahlen wobei $s(x)$ als $x + 1$ zu interpretieren ist. Man muß aber beachten, daß s eine syntaktische Repräsentation der $+1$ -Funktion ist, und nicht die Funktion selbst. Betrachtet man die tatsächlichen Zahlen als Abkürzungen für entsprechend viele Verschachtelungen der Funktion s , dann kann man mit Hilfe dieser Axiome schon richtig rechnen. Zum Beispiel ergibt sich:

$$3 + 2 = s(s(s(0))) + s(s(0)) = s(3 + s(0)) = s(s(3 + 0)) = s(s(3)) = 5$$

Gleichungslogik ist aber wiederum nur ein Spezialfall von *Prädikatenlogik*, wo neben der ganz speziellen Relation der Gleichheit ganz allgemeine Relationen als neues Konzept hinzukommen. Hier kann man dann beispielsweise formulieren:

$$\forall x, y : Katze(x) \wedge Vogel(y) \Rightarrow Mag(x, y) Katze(Garfield) \wedge Vogel(Tweety)$$

“Katze“ und “Vogel“ bezeichnen einstellige Relationen, die auf ein Objekt zu treffen können oder nicht. “Mag“ bezeichnet eine zweistellige Relation. In der Prädikatenlogik gibt es Relationen mit beliebiger Stelligkeit.

Eine weitere Logik ist zum Beispiel die *Modallogik*, die man zur (automatischen) Lösung des Wise Men Puzzles benutzen kann. Das Grundkonzept, das bei den Modallogiken hinzukommt, ist das der Zustände – manchmal auch Welten genannt – und der Zustandsübergänge. Wie beim Wise Men Puzzle macht es offensichtlich Sinn, nicht nur über eine Welt zu reden, in der eine Aussage einen bestimmten Wahrheitswert hat, sondern es kann notwendig sein, gleichzeitig über viele verschiedenen hypothetischen Welten und deren Beziehungen zueinander zu reden. Genau das ist mit Modaloperatoren in sehr eleganter Weise möglich. Beim wise-man-puzzle kann man sich vorstellen, dass jeder ein eigene Menge von gültigen Aussagen haben kann.

Von Logikern und Philosophen wurden noch eine Unzahl anderer Logiken und deren Beziehungen zueinander untersucht. Für eine konkrete Anwendung ist die Situation daher wie bei der Wahl einer geeigneten Programmiersprache. Die beste ist die, die das, was in der Anwendung vorkommt möglichst einfach und elegant zu formulieren erlaubt, und für die es einen möglichst guten Compiler oder Interpretierer gibt. Die Formalismen, die man braucht, um eine Logik zu beschreiben und zu untersuchen sind zunächst eine Beschreibung der Syntax, d.h. der Grammatik der Sprache und eine Beschreibung der Bedeutung der syntaktischen Elemente, d.h. der Semantik. Der Semantikformalismus, der uns hier interessiert, geht zurück auf Alfred Tarski und wird deshalb auch Tarski-Semantik genannt. Die Idee ist, eine Abbildung der syntaktischen Elemente auf mathematische Objekte anzugeben, und diese Abbildung zu benutzen, um dann Terme auf Objekte einer bestimmten Menge, der sogenannten *Trägermenge* oder auch *Universum* genannt, abzubilden, und Formeln zu wahr oder falsch zu evaluieren. Eine solche Abbildung heißt auch *Interpretation*. Typischerweise bildet

man in diesem Formalismus Konstanten- und Variablensymbole auf Elemente der Trägermenge, Funktionssymbole auf Funktionen und Prädikatensymbole auf Relationen ab. Eine solche Abbildung, die eine Formel wahr macht heißt *Modell* der Formel. Zum Beispiel kann man für die Formel

$$Katze(Garfield) \wedge Vogel(Tweety)$$

als Trägermenge tatsächlich die Menge aller Katzen und Vögel nehmen. Das Prädikatensymbol “Katze“ wird auf die Relation *Katze* abgebildet, d.h. $Katze(x)$ trifft zu wenn x wirklich eine Katze ist. Entsprechend machen wir das mit dem Symbol “Vogel“. Wenn jetzt die Konstantensymbole “Garfield“ und “Tweety“ tatsächlich auf eine Katze bzw. Vogel abgebildet werden, dann ist die Formel wahr und wir haben damit ein Modell der Formel. Wenn wir stattdessen das Symbol “Garfield“ auf einen Vogel abbilden, dann ist die Formel falsch und die Interpretation ist kein Modell. Genausogut können wir auch als Trägermenge die natürlichen Zahlen nehmen und das Prädikatensymbol “Katze“ auf die Relation *gerade* und das Prädikatensymbol “Vogel“ auf die Relation *ungerade* abbilden. Wenn jetzt z.B. “Garfield“ auf die 0 und “Tweety“ auf die 1 abgebildet wird haben wir wieder ein Modell der Formel. Man bezeichnet eine Formel als allgemeingültig oder als *Tautologie* wenn sie unter *allen* Interpretationen wahr ist. Typische Tautologien sind “ $P \vee \neg P$ “, aber auch Formeln wie “Euklidische Geometrieaxiome \Rightarrow Satz des Pythagoras“. Formeln, die unter keiner Interpretation wahr sind heißen *unerfüllbar* oder *widersprüchlich*. Typische Widersprüche sind “ $P \wedge \neg P$ “, aber auch Formeln wie “ \neg (Euklidische Geometrieaxiome \Rightarrow Satz des Pythagoras)“. Formeln, die weder allgemeingültig noch widersprüchlich sind, die man also je nach Interpretation wahr oder falsch machen kann, heißen *erfüllbar*. Die obige Formel $Katze(Garfield) \wedge Vogel(Tweety)$ ist von diesem Typ. Die Tarski Semantik erlaubt die Definition eines semantischen Folgebegriffs: $F \models G$ (G folgt aus F) falls G in allen Modellen von F gilt. Das heißt, wie immer auch die Symbole, die in F vorkommen interpretiert werden, falls F unter dieser Interpretation wahr ist, muß auch G unter dieser Interpretation wahr sein. Dieser semantische Folgebegriff definiert präzise, was “Folgerung“ heißen soll. Da die Anzahl der möglichen Interpretationen meist unendlich ist, gibt sie aber keinen Hinweis, wie man die Folgerungsbeziehung für zwei konkrete Formeln F und G auch tatsächlich überprüfen kann (das ist Sache der zweiten Stufe eines Deduktionssystems, des Kalküls). Spezielle Fragestellungen, die man auf der Ebene der Logiken noch untersucht, sind z.B. die Beziehung zwischen semantischer Folgerung und syntaktischer Implikation, d.h. die Frage, ob das sogenannte *Deduktionstheorem* gilt:

$$F \models G \text{ gdw. } F \Rightarrow G \text{ allgemeingültig.}$$

In Prädikatenlogik gilt es, in anderen Logiken nicht unbedingt. Eine andere Fragestellung ist mehr anwendungsorientiert, nämlich die Frage nach der Existenz von Übersetzern, die Formeln einer Logik in eine andere übersetzen. Analog wie bei der Entwicklung einer Programmiersprache ein Compiler die Ausführung von Programmen erheblich beschleunigen kann, kann ein Übersetzer zwischen Logiken den Test der Folgerungsbeziehung erheblich erleichtern. Beispiele für solche

Übersetzer sind die Transformation von epistemischer Logik in Prädikatenlogik, die bei der Lösung des Wise Men Puzzles benutzt werden kann. Ein weiteres Beispiel ist die Transformation in Klauselnormalform. Klauselnormalform ist eine echte Unterklasse der Prädikatenlogik, so daß man diese Transformation auch als Übersetzung in eine andere Logik sehen kann.

Kalkül Der in der Logik definierte semantische Folgerungsbegriff hilft meist nicht, für gegebene A und B algorithmisch zu bestimmen, ob B wirklich aus A folgt. Dies ist Aufgabe des *Kalküls*, der zweiten Schicht eines Deduktionssystems. Ein Kalkül definiert syntaktische *Ableitungen* als Manipulationen auf den (syntaktisch gegebenen) Formeln. Damit kann aus einer Formel A durch reine Symbolmanipulation eine Formel B gewonnen werden, wobei die Bedeutung der in A und B vorkommenden Symbole überhaupt keine Rolle spielt. Ein syntaktisch aus A abgeleitetes B soll aber trotzdem semantisch folgen und umgekehrt. Ein korrekter Kalkül stellt daher nur solche Ableitungsoperationen zur Verfügung, die garantieren, daß alles syntaktisch Ableitbare auch semantisch folgt. Wenn umgekehrt alles, was semantisch folgt, auch syntaktisch ableitbar ist, ist der Kalkül vollständig. Es gibt auch Kalküle, die statt auf Formeln auf anderen Datenstrukturen operieren, z.B. i) Folgen von Formeln (Sequenzenkalkül), ii) Mengen von Formeln, iii) auf einem Graph, der u.a. mit Formeln markiert ist.

Nach dem Unvollständigkeitssatz von Kurt Gödel [Göd31] sind vollständige Kalküle ab einer gewissen Ausdrucksstärke der Logiken jedoch nicht möglich. Dazu gehört die sogenannte Prädikatenlogik zweiter Stufe, in der neben Quantifizierungen über Mengen auch Quantifizierungen über Funktionen über den Mengen erlaubt sind. Hierzu zählt auch die Logik, die die Theorie der natürlichen Zahlen beschreibt, d.h. alle Formeln die genau für die natürlichen Zahlen gelten. In diesen Logiken gibt es Aussagen, die über den semantischen Folgerungsbegriff aus anderen folgen, was aber mit keinem Kalkül, der nur mit Symbolmanipulation arbeitet, nachgewiesen werden kann.

Meist werden Ableitungsregeln folgendermaßen geschrieben: $\frac{F_1, \dots, F_n}{F}$. Das soll bedeuten, wenn F_1, \dots, F_n schon abgeleitete Formeln sind, dann ist es auch erlaubt, die Formel F abzuleiten. Eine typische Ableitungsregel dieser Art ist die *Instantiierungsregel*:

$$\frac{\forall x : F[x]}{F[t/x]}$$

Dabei ist $F[x]$ eine Formel, in der die Variable x vorkommt und $F[t/x]$ ist eine Variante von F , bei der alle Vorkommnisse von x durch den Term t ersetzt worden sind (t ist beliebig). Die Regel besagt, daß eine Aussage, die für alles gilt ($\forall x$) auch für jedes spezielle Objekt gilt. Zweimal angewendet läßt sich damit zum Beispiel aus

$$\forall x, y : Katze(x) \wedge Vogel(y) \Rightarrow Mag(x, y)$$

ableiten:

$$Katze(Garfield) \wedge Vogel(Tweety) \Rightarrow Mag(Garfield, Tweety)$$

Eine weitere Regel dieser Art ist die *Modus Ponens* Regel, die schon auf die griechischen Philosophen zurückgeht:

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

Sie drückt folgendes aus: wenn es gilt, daß aus der Aussage A die Aussage B folgt und wenn weiterhin bekannt ist, daß die Aussage A tatsächlich wahr ist, dann darf man annehmen, daß auch die Aussage B wahr ist.

Beide Regeln, die Instantiierungsregel als auch die Modus Ponens Regel eignen sich als reine Zeichenkettenmanipulationen und lassen sich daher problemlos auf einem Rechner ausführen.

Typische Fragestellungen auf der Kalkülebene sind folgende:

Korrektheit eines gegebenen Kalküls: Folgt alles abgeleitete auch semantisch?

Vollständigkeit eines gegebenen Kalküls: Läßt sich alles semantisch folgerbare auch ableiten?

Für die Praxis hat sich gezeigt, daß die Vollständigkeit so wie oben definiert an den Kalkül leicht angepasst werden muß. Was man z.B. bei Resolutionsbeweisen wirklich braucht ist die *Widerlegungsvollständigkeit*: Wenn aus einer Aussage F eine Aussage G semantisch folgt, ist dann ein Widerlegungsbeweis für $F \wedge \neg G$ möglich? Resolution ist z.B. ein Kalkül, der nicht vollständig, aber widerlegungsvollständig ist. Beispielsweise folgt aus A die Aussage $A \vee B$. Sie ist aber nicht mit Resolution ableitbar. Jedoch ist $A \wedge \neg(A \vee B) (= A \wedge \neg A \wedge \neg B)$ mit Resolution widerlegbar. Ein weiteres Problem im Bereich der Kalküle ist die Frage nach der *Effizienz* eines Kalküls. Dabei betrachtet man meist zwei Effizienzkriterien, i) die Verzweigungsrate im Suchraum und die ii) Länge der Beweise. Die Regeln eines Kalküls sind im allgemeinen an vielen verschiedenen Stellen einer Formelmengen anwendbar. Nicht alles, was damit abgeleitet wird, ist jedoch für den gesuchten Beweis brauchbar. Daher definieren die Regeln eines Kalküls einen Suchraum, der durch irgendein Suchverfahren abgesucht werden muß. Je größer die Verzweigungsrate, d.h. je mehr Stellen es in der aktuellen Formelmengen gibt, auf die die Kalkülregeln anwendbar sind, desto aufwendiger ist meist die Suche. Die Verzweigungsrate ist jedoch kein generelles Kriterium. Ist die Verzweigungsrate niedrig, liegt aber dafür der gesuchte Beweis sehr tief im Suchraum, dann ist auch nicht viel gewonnen. Was man braucht ist einerseits eine möglichst kleine Verzweigungsrate und andererseits Kalkülregeln, die einen Beweis mit möglichst wenig Schritten finden, und diese Schritte sollen mit möglichst wenig Aufwand berechenbar sein. Die Kunst des Kalkülentwerfens besteht darin, solche Kalküle zu entwickeln, die einen guten Kompromiss zwischen allen drei Faktoren bilden.

Die rein syntaktisch arbeitenden Kalküle wie Resolution, die nichts von der Bedeutung der Symbole, die sie manipulieren wissen, sind zwar im Prinzip ausreichend. Für viele häufig vorkommende Konstrukte kennt man jedoch Algorithmen und Kalküle, die spezielle Probleme erheblich schneller und besser lösen als die universell anwendbaren Kalküle. Niemand würde zum Beispiel auf die Idee kommen, die Gleichung $3 + 4 = x$ mit einem universellen Kalkül wie Resolution zu lösen. Das rechnet man einfach aus. Daher besteht ein weiterer Zweig der Kalküentwicklung darin, Algorithmen für wichtige Spezialfälle zu integrieren, bzw. erst überhaupt zu entwickeln. Als ein Rahmenkonzept dafür hat sich die von Mark Stickel vorgeschlagene *Theorieresolution* [Sti86] als sehr nützlich gezeigt. Die Idee dabei ist, einen Resolutionsschritt nicht mehr über die syntaktische Komplementarität – gleiches Prädikatensymbol und Argumente, verschiedenes Vorzeichen – sondern über semantische Widersprüchlichkeit der Resolutionsliterals zu steuern. Diese semantische Widersprüchlichkeit kann von einem speziellen Algorithmus, der über die Bedeutung der vorkommenden Symbole etwas weiß, getestet werden. Beispielsweise kann man damit folgern:

$$\frac{a < b \vee P \quad a > b \vee Q}{P \vee Q}$$

eben weil $a < b$ und $a > b$ unter der üblichen Bedeutung von $<$ und $>$ widersprüchlich sind. Ein Teil des Skripts ist solchen Spezialverfahren gewidmet.

Logische Zustandsübergangssysteme Die dritte Schicht eines Deduktionssystems, die Schicht der logischen Zustandsübergangssysteme, bestimmt die Darstellung von Formeln oder Formelmengen, deren Beziehungen zueinander, sowie von den jeweiligen Zuständen der Ableitungsketten. Ein logisches Zustandsübergangssystem besteht aus einer Menge S von Zuständen und einer binären Relation \rightarrow , der Übergangsrelation. Jeder Zustand ist dabei die Repräsentation einer Formelmenge, im einfachsten Fall lediglich die Menge selbst. In ausgefeilteren Systemen enthalten die Zustände aber noch mehr Komponenten, Information über die Geschichte der Ableitung, Repräsentationen der von dem aktuellen Zustand aus machbaren Ableitungen zusammen mit strategischer und heuristischer Information über den Nutzen des jeweiligen Schritts usw. Die Übergangsrelation $S \rightarrow S'$ für Zustände S und S' ergibt sich aus dem Kalkül zunächst mal ganz einfach folgendermaßen: Wenn $\frac{F_1, \dots, F_n}{F}$ eine Kalkülregel ist und F_1, \dots, F_n im Zustand S repräsentiert ist dann ist S' eine Repräsentation für $\mathcal{F} \cup \{F\}$, wobei \mathcal{F} die in S repräsentierte Formelmenge ist. Auf dieser Ebene werden oft zusätzliche Operationen eingeführt, etwa das Löschen von redundanten Aussagen, so daß im Kalkül noch mögliche Ableitungen nun nicht mehr möglich – und hoffentlich auch nicht mehr nötig – sind. Die Übergangsrelation \rightarrow enthält daher nicht nur Ableitungsschritte, sondern auch Reduktionsschritte, die die Formelmenge von unnützem Ballast befreien sollen.

Für jedes logische Zustandsübergangssystem sind drei Eigenschaften von Interesse: Korrektheit, Vollständigkeit und Konfluenz. Durch neue Regeln, mit

denen Formeln oder Teile davon gelöscht werden, wird die Korrektheit und Vollständigkeit des zugrundeliegenden Kalküls beeinflusst. Für jede solche Regel muß man neu beweisen, daß man damit keine falschen Beweise erzeugt (Korrektheit), und daß man mögliche Beweise auch finden kann (Vollständigkeit). Die letzte Eigenschaft der Konfluenz ist für die Implementierung eines Deduktionssystems wichtig. Konfluenz bedeutet, daß wenn man von einem Zustand S aus zwei Nachfolgezustände S_1 und S_2 erreichen kann, dann gibt es für S_1 und S_2 einen gemeinsamen Nachfolgezustand S' . Ist ein System konfluent, dann kann man es sich bei der Suche leisten, zunächst mal in die falsche Richtung zu suchen. Wenn vom Ausgangszustand (S) überhaupt ein Weg zu einem Beweis (S') existiert, dann existiert er auch von jedem weiteren Zustand (S_1 oder S_2) aus. Das heißt, bei der Suche gibt es keine Sackgassen; man braucht nie mehr zu früheren Zuständen zurückzukehren (kein Backtracking). Das entspricht auch der Intuition für Beweissuchen: Eine Aussage, die man einmal abgeleitet hat, mag zwar nutzlos für den aktuellen Beweis sein, es sollte aber nicht nötig sein, deren Ableitung selbst ungeschehen zu machen. Vollständigkeit des Deduktionsprozesses bedeutet, daß man unabhängig von den ausgeführten Transformationen vom aktuellen Zustand aus den Beweis noch finden kann. Hat man sehr starke Reduktions- oder Steuerungsregeln, so kann das Zustandsübergangssystem des Deduktionssystems die Eigenschaft der Konfluenz verlieren. In diesem Fall bedeutet Vollständigkeit des Deduktionssystems, daß man nach der Ausführung gewisser Transformationen den Beweis nur noch dadurch findet, daß man zurücksetzt in einen vorangegangenen Zustand und eine andere Alternative wählt.

Steuerung Die letzte Schicht eines Deduktionssystems, die *Steuerung*, enthält schließlich die Strategien und Heuristiken, mit denen unter den möglichen Ableitungsschritten die jeweils sinnvollen ausgewählt werden. Hier steckt die eigentliche "Intelligenz" des Systems. Die Idee dabei ist, "gute" Schritte zu bevorzugen und "schlechte" Schritte zu vermeiden. Man unterscheidet Restriktionsstrategien und Ordnungsstrategien.

Restriktionsstrategien schränken den Suchraum weiter ein, in dem sie bestimmte Ableitungen einfach generell verbieten. Zum Beispiel verbietet die "Set of Support"-Strategie bei Widerlegungsbeweisen Ableitungen zwischen Axiomen untereinander. Die Idee dabei ist, daß bei Ableitungen zwischen Axiomen allein garantiert kein Widerspruch zu finden ist. Da Restriktionsstrategien bestimmte Ableitungen prinzipiell verbieten, muß man jeweils nachweisen, daß im verbleibenden Suchraum immer noch ein Beweis zu finden ist (Vollständigkeit).

Ordnungsstrategien sortieren die möglichen Ableitungen nach bestimmten Kriterien, z.B. könnte man Ableitungen, die kleine Formeln erzeugen, bevorzugen. Der Zweck der Ordnungsstrategien ist, die Suche selbst geschickt zu organisieren und natürlich die Suche zu terminieren, d.h. wenn es einen Beweis gibt, dessen Auffindung mit endlich vielen Schritten zu garantieren (Terminierung).

Verwendung, Dialogverhalten Ein wichtiger praktischer Aspekt ist die Art der Verwendung des Systems, bzw. die Art der Unterstützung die ein Deduktionssystem bietet:

Vollautomatisch Das Deduktionssystem hat als Eingabe die Axiome und die nach zuweisende Schlußfolgerungen, und als Ausgabe nur Erfolg/Mißerfolg und eine Begründung (Beweis). Dies ist die Idealvorstellung eines Automatischen Deduktionssystems. Pragmatische System beschränken sich auf eine (einfache) Auswahl von Eingaben, z.B. einfache Verifikationsbedingungen. Im allgemeinen haben die Automatischen Deduktionssystems eine Menge von Parametern, die vor dem Lauf eingestellt werden müssen. So daß z.B. die Arbeit mit dem System Otter darin bestehen kann, in vielen Läufen, die richtigen Parameter bzw. Heuristiken einzustellen, so daß es am Ende doch eher interaktiv wirkt.

Es gibt Varianten dieser Systeme, die während der Suche interaktive Hilfestellung erlauben.

Halbautomatisch Bestimmte Systeme für Logiken höherer Ordnung (Typenlogik) führen einen Dialog mit dem Benutzer. Einfache Folgerungen werden automatisch durchgeführt, für andere wird von Benutzer erwartet, die Zwischenziele (Lemmas) zu formulieren und Beweistaktiken zu programmieren. Der geführte Beweis ist im Erfolgsfall korrekt, allerdings kann ein solcher Dialog einige Wochen dauern. Z.B. ist mit dieser Methode der Nachweis der Korrektheit eines Tokenizers von T. Nipkow (TU München) durchgeführt worden

Beweisprüfung Diesen Systemen muß die komplette Axiomatisierung und der Beweis vorgelegt werden. Dies kann selbst für einfache Dinge sehr mühsam sein, da der Detaillierungsgrad sehr hoch gewählt ist: (Jeder (formale) Schritt des Beweises muß formal korrekt vorliegen).

4 Prädikatenlogik (PL_1) und Resolution

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 die Elemente einer Trägermenge. 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. [EFT86].

Aus praktischer Sicht gibt es viele Zusammenhänge, die sich in anderen Logiken wesentlich eleganter und einfacher formulieren lassen als in PL_n . Kurt Gödel hat gezeigt, daß es im Gegensatz zu PL_1 für PL_n , $n \geq 2$, keinen vollständigen Kalkül mehr geben kann [Göd31]. D.h. es gibt Aussagen in PL_2 , die in allen Interpretationen gültig sind, was aber mit keinem durch Symbolmanipulation arbeitenden Verfahren mehr nachgewiesen werden kann. Das heißt nicht, daß man für PL_2 keine Deduktionssysteme entwickeln kann – sie können eben nur nicht alle gültigen Theoreme beweisen; aber das können heutige Deduktionssysteme für PL_1 auf heutigen Computern auch nur theoretisch. Deduktionssysteme für PL_2 sind aber um einiges komplizierter als solche für PL_1 [And81, RB79]. Wir werden in der Vorlesung auch Kalküle für Logiken höherer Ordnung betrachten, falls die Zeit ausreicht.

4.1 Syntax der Prädikatenlogik erster Stufe

PL_1 ist zunächst mal eine formale Sprache, deren Syntax sich durch eine entsprechende Grammatik angeben läßt. Im Gegensatz zu vielen Programmiersprachen z.B. ist die Grammatik von PL_1 aber extrem einfach. 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. 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. Die logischen Junktoren und Quantoren sind fest und zählen daher nicht zu der Signatur. 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*. Der Unterschied zwischen

Termen und Formeln ist in erster Linie semantischer Natur. Terme bezeichnen Objekte einer Trägermenge und Formeln bezeichnen Wahrheitswerte. Die formale Definition ist:

Definition 4.1. (*Syntax von PL_1*)

Signatur : $\Sigma = (\mathcal{F}, \mathcal{P})$, 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 zugeordnet Stelligkeit : $\text{arity}(f) \geq 0$. Funktionssymbole mit der Stelligkeit 0 bezeichnet man auch als Konstantensymbole. $\{f \in \mathcal{F} \mid \text{arity}(f) = 0\} = \text{Konstantensymbole}$. 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 $f \in \mathcal{F}, \text{arity}(f) = n, t_1, \dots, t_n \in T(\Sigma, V)$ dann $f(t_1, \dots, t_n) \in T(\Sigma, V)$. Hierbei ist $f(t_1, \dots, t_n)$ zu lesen als Zeichenkette bzw. als ein Baum.

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

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

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 4.2. Signatur $\Sigma := (\{a, b, f, g\}, \{P, Q, R\})$ mit $\text{arity}(a) = \text{arity}(b) = \text{arity}(P) = 0, \text{arity}(f) = \text{arity}(Q) = 1, \text{arity}(g) = \text{arity}(R) = 2$.

$$V = \{x, y, z, \dots\}$$

$T_{\Sigma} =$	$F_{\Sigma} =$
$f(a), f(b), f(x), \dots$	$\{P, Q(a), Q(b), Q(x), \dots, R(a, a), \dots R(a, b), \dots$
$g(a, a), g(a, b), g(a, f(a)), \dots$	$\neg P, \neg Q(a), \dots$
$f(f(a)), f(f(b)), \dots f(g(a, a)), \dots$	$P \wedge Q(a), P \wedge \neg Q(a), \dots$
$g(f(f(a)), a), \dots$	$P \vee Q(a), P \vee \neg Q(a), \dots$
\dots	$P \Rightarrow Q(a), P \Leftrightarrow \neg Q(a), \dots$
	$\forall x : Q(x), \dots$
	$\exists x : R(x, y) \Rightarrow Q(x), \dots$

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 4.3) werden sie dann auch konsequenterweise wieder eliminiert. Nichtsdestotrotz erleichtern sie die Lesbarkeit von Formeln beträchtlich und sind daher mit eingeführt worden.

Definition 4.3. Konventionen:

- Da 0-stellige Funktionssymbole als Konstantensymbole dienen³, 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 $\exists x : P(x)$ 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 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

³ Indem man Konstantensymbole nicht extra ausweist, spart man sich in vielen Fallunterscheidungen eben diesen speziellen Fall.

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

Folgende Definition (der freien Variablen) demonstriert eine Anwendung des rekursiven Definitionsschemas für Terme und Formeln. Eine Variable wird als frei bezeichnet wenn sie sich nicht im Bindungsbereich (Skopus) eines Quantors befindet.

Definition 4.4. (Freie Variablen für Terme und Formeln) Die Operation $FV(\cdot)$ sammelt alle nicht durch Quantoren gebundenen Variablensymbole in Termen und Formeln auf.

Definition von FV für Terme:

$$FV(t) = \begin{cases} t & \text{falls } t \text{ eine Variable ist (Basisfall)} \\ FV(t_1) \cup \dots \cup FV(t_n) & \text{falls } t = f(t_1, \dots, t_n) \text{ (Rekursionsfall)} \end{cases}$$

Definition von FV für Formeln: Basisfall: $H = P(t_1, \dots, t_n)$ ist ein Atom, dann: $FV(H) = FV(t_1) \cup \dots \cup FV(t_n)$ (FV ist für Terme schon definiert.)
Rekursionsfälle: Fallunterscheidung nach der Struktur von H :

$$\begin{aligned} \text{Fall: } H = \neg F & \quad \text{dann } FV(H) := FV(F) \\ \text{Fall: } H = F \vee G & \quad \text{dann } FV(H) = FV(F) \cup FV(G) \\ \text{Fall: } H = F \wedge G & \quad \text{dann } FV(H) = FV(F) \cup FV(G) \\ \text{Fall: } H = F \Rightarrow G & \quad \text{dann } FV(H) = FV(F) \cup FV(G) \\ \text{Fall: } H = F \Leftrightarrow G & \quad \text{dann } FV(H) = FV(F) \cup FV(G) \\ \text{Fall: } H = \forall x : F & \quad \text{dann } FV(H) = FV(F) \setminus \{x\} \\ \text{Fall: } H = \exists x : F & \quad \text{dann } FV(H) = FV(F) \setminus \{x\} \end{aligned}$$

Beispiel 4.5. Unten sind x, y, z Variablensymbole

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

Zum Schluß des Abschnitts über die Syntax von PL_1 werden noch einige Sprechweisen und Begriffe, die z.T. auch schon benutzt wurden, eingeführt. Da sie sehr häufig gebraucht werden, muß man sie sich unbedingt einprägen.

Definition 4.6. Einige übliche Sprechweisen:

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

Beispiel 4.7. Für eine geschlossene Formel: $\forall x : \exists y : P(x, y)$. Nicht geschlossen ist: $\exists y : P(x, y)$, da $FV(\exists y : P(x, y)) = \{y\}$.

4.2 Semantik von PL_1 (nach Tarski)

Durch die formale Definition der Syntax von PL_1 ist man jetzt zwar in der Lage, Aussagen als Formeln hinzuschreiben und zu überprüfen ob das was man hingeschrieben hat auch syntaktisch in Ordnung (wohlgeformt) ist. Man kann jetzt auch Operationen zur Manipulation von Termen und Formeln definieren. Was diese Formeln, die ja bisher nur reine Zeichenketten sind, und die Operationen, die nur Zeichenketten manipulieren, aber bedeuten sollen ist bisher nur intuitiv angedeutet worden. Um aber eine Zeichenkettenmanipulation als korrekte Ableitungsregel zu interpretieren braucht man einen Formalismus, der Termen und Formeln Bedeutung zuordnet, so daß man damit eine semantische Folgerungsbeziehung definieren kann. Dies ist eine Verallgemeinerung der entsprechenden Konzepte in der Aussagenlogik.

Es gibt auch eine Analogie zu Programmiersprachen. Die reine Syntaxdefinition in einer Backus-Naur Form zum Beispiel hilft zu entscheiden, ob ein gegebenes Programm syntaktisch korrekt ist. Um aber einen Interpretierer oder Compiler für die Sprache zu schreiben und insbesondere, um dessen Korrektheit nachzuweisen, muß man sagen, was die Konstrukte, die in der Sprache auftreten, bedeuten sollen, d.h. man braucht eine formale Semantik. Dafür gibt es verschiedene Möglichkeiten. Die abstrakteste ist wohl die denotationale Semantik, bei der die Programmiersprachenkonstrukte im wesentlichen auf Mengenoperationen abgebildet werden. Diese Operationen sind dann einfach und übersichtlich zu erklären. Hiermit hat man eine Möglichkeit, die Ausführung von Programmkonstruktionen zu definieren bzw. zu überprüfen.

Um die Semantik einer Logik zu definieren gibt es ebenfalls verschiedene Möglichkeiten.

Eine operationale Methode der Definition einer Logik ist ein sogenannter Hilbertkalkül. Dabei gibt man zunächst einen Satz von Formeln oder Formelschemata an, die man apriori als gültige Aussagen (Axiome) ansehen will (z.B.

$P \Rightarrow P$). Weiterhin braucht man einen Satz von (syntaktischen) Ableitungsregeln, die aus den Axiomen neue Formeln generieren, welche man dann ebenfalls als gültige Formeln annimmt. Man muß nur aufpassen, daß diese Ableitungsregeln nicht gleichzeitig eine Formel und deren Negation generieren (Korrektheit). Ein solcher Hilbertkalkül charakterisiert eine Logik zwar genau, ist aber i.a. zu indirekt. Eine Charakterisierung durch einen Hilbertkalkül eignet sich nicht, um andere, für die Automatisierung besser geeignete Kalküle, zu entwickeln.

Eine sehr natürliche Möglichkeit (die “denotationale Semantik“ von PL_1), die Semantik einer Logik und insbesondere von PL_1 anzugeben ist von Alfred Tarski entwickelt worden [Tar53]. Die Grundidee dabei ist, eine Abbildung der syntaktischen Objekte auf mathematische Objekte (Mengen, Funktionen, Relationen etc.) anzugeben, so daß man unter dieser Abbildung Formeln zu Wahrheitswerten evaluieren kann. Gesucht sind daher:

- geeignete mathematische Objekte
- eine geeignete Abbildungsvorschrift, wobei die Abbildung in drei Schritten definiert wird:
 1. Abbildung der Signatur (für PL_1 auf Funktionen und Relationen)
 2. Abbildung der Terme auf Elemente einer Grundmenge Trägermenge, Universum)
 3. Abbildung der Formeln auf Wahrheitswerte.

Für PL_1 wird diese Abbildung dann so aufgebaut, daß man zunächst eine Menge definiert, in die die Terme abgebildet werden sollen; Trägermenge, Domain oder Universum genannt. Funktionssymbole werden auf Funktionen über dieser Menge und Prädikatensymbole auf Relationen über dieser Menge abgebildet. Damit ist die Grundstruktur festgelegt. Die Abbildung von Termen und Formeln ergibt sich jetzt ganz natürlich aus ihrer syntaktischen Struktur. Ein Term $f(a, b)$ zum Beispiel bezeichnet gerade eine Funktionsanwendung: Die dem Symbol f zugeordnete Funktion wird auf die den Konstantensymbolen a und b zugeordneten Werte angewendet. Dem Term selbst wird gerade das Ergebnis dieser Funktionsanwendung zugeordnet. Wenn z.B. f die arithmetische Funktion $+$ zugeordnet wird und a und b die Zahlen 3 und 4, dann wird damit automatisch dem Term $f(a, b)$ der Wert 7 zugeordnet. Mit Hilfe der Abbildung von Prädikatensymbolen auf Relationen lassen sich Atome auf Wahrheitswerte abbilden. Wenn z.B. das Prädikatensymbol P auf die arithmetische $<$ -Relation abgebildet werden und wie oben die Konstantensymbole a und b auf die Zahlen 3 und 4, dann ist $P(a, b)$ wahr, da ja $3 < 4$ wahr ist. Wenn man dagegen a und b umgekehrt auf 4 und 3 abbildet, dann ist $P(a, b)$ falsch. Ausgehend von dieser Grundvorschrift für Atome lassen sich dann zusammengesetzte Formeln aus den Werten für ihre Komponenten und der Bedeutung des obersten logischen Symbols auswerten. Wenn z.B. die Formel F zu falsch ausgewertet wird, dann wird $\neg F$ zu wahr ausgewertet usw.

Man beachte, daß die syntaktischen Objekte, wie z.B. der Term $f(a, b)$, nichts weiter sind als Zeichenketten. In einem Computer werden sie als Bitfolgen repräsentiert, die für den Computer keinerlei Bedeutung haben. Im Gegensatz dazu ist mit $+$ wirklich die arithmetische Additionsfunktion gemeint. Wenn man

auf Papier einen Unterschied machen will zwischen der Darstellung von f als Funktionssymbol und z.B. $+$ als Funktion, dann muß man zunächst Konventionen zur Schreibweise einführen, um Datenstruktureobjekte und mathematische Dinge auseinander zu halten. Es sind aber völlig unterschiedliche Dinge damit gemeint. In rein mathematischen Texten sind mit den Zeichen, die auf dem Papier innerhalb von Formeln erscheinen im allgemeinen die semantischen Objekte, d.h. die Funktionen usw. gemeint. In logischen Texten wie auch in diesem Skript sind es dagegen die syntaktischen Objekte, um die es primär geht.

Die Idee für die Semantik von PL_1 ist offensichtlich sehr einfach und natürlich. Der Formalismus ist umfangreich aber einfach und gibt ganz genau die oben geschilderten Prinzipien wieder.

Für eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$, die nach Definition 4.1 aus Funktions- und Prädikatensymbolen besteht, definiert man zunächst Σ -Algebren⁴ bestehend aus einer Trägermenge und einer geeigneten Anzahl von Funktionen geeigneter Stelligkeit. Für jedes in \mathcal{F} vorkommende Funktionssymbol mit Stelligkeit n braucht man genau eine n -stellige Funktion. Um auch die Prädikatensymbole zu interpretieren, werden die Σ -Algebren zu Σ -Strukturen erweitert, die für jedes in Σ vorkommende Prädikatensymbol eine passende Relation enthalten. (Den Unterschied zwischen Σ -Algebren und Σ -Strukturen macht man deshalb, weil Σ -Algebren zur Interpretation von Termen ausreichen, und Σ -Algebren für Gleichheitslogiken und für die Unifikation ausreichen.) Schematisch sieht die Abbildung der Signatur dann so aus:

$$\Sigma\text{-Struktur: } \begin{cases} \Sigma\text{-Algebra : } \{ \text{Funktionssymbole} \rightarrow \text{Funktionen} \\ \text{Prädikatensymbole} \rightarrow \text{Relationen} \end{cases}$$

Definition 4.8. Σ -Algebra, Σ -Struktur)

Sei $\Sigma = (\mathcal{F}, \mathcal{P})$ eine Signatur. $A = (D_A, F_A)$ heißt eine Σ -Algebra gdw.

- D_A ist eine nichtleere Menge (die Trägermenge von A)
- F_A enthält für jedes Funktionssymbol in \mathcal{F} eine passende totale Funktion, d.h. $F_A = \{f_A \in (D_A)^n \rightarrow D_A \mid f \in \mathcal{F}, \text{arity}(f) = n\}$.

$S = (D_S, F_S, \mathcal{P}_S)$ heißt eine Σ -Struktur gdw.

- (D_S, F_S) ist eine Σ -Algebra und
- \mathcal{P}_S enthält für jedes Prädikatensymbol in \mathcal{P} mit Stelligkeit ≥ 1 eine passende Relation, d.h. $\mathcal{P}_S = \{P_S \subseteq (D_S)^n \mid P \in \mathcal{P}, \text{arity}(P) = n\}$.
- Jedes Prädikatensymbol der Stelligkeit 0 wird auf 0 oder 1 abgebildet. Die konstanten Formel werden abgebildet wie folgt: **true** auf 1 und **false** auf 0.

Die zugeordneten Funktionen, Prädikate, Wahrheitswerte zu einem Symbol R werden i.a. mit einem Index S gekennzeichnet. Z.B. $f \rightarrow f_S$.

Für eine Σ -Algebra oder Σ -Struktur S soll in Zukunft mit D_S deren Trägermenge bezeichnet werden.

⁴ Das Σ in Σ -Algebra bezieht sich auf die Signatur Σ : Für jede Signatur Σ gibt es eine Klasse von Σ -Algebren

Beispiel 4.9. Σ -Algebren und Σ -Strukturen

Sei $\Sigma = (\{o, s\}, \{L\})$ mit $\text{arity}(o) = 0, \text{arity}(s) = 1, \text{arity}(L) = 2$.

Σ -Struktur $S_1 = (\mathbb{N}, \{0, \text{inc}\}, \{<\})$ wobei (\mathbb{N} = natürliche Zahlen, $\text{inc}(x) := x + 1$)

Σ -Struktur $S_2 = (\{Mo, Di, Mi, Do, Fr, Sa\}, \{Mo, \text{morgen}\}, \{vor\})$ wobei $\text{morgen}(Mo) = Di, \text{morgen}(Di) = Mi, \dots$ und Mo vor Di, Di vor Mi, \dots, Sa vor So , aber nicht Mo vor Mi usw. (d.h. vor ist nicht transitiv.)

Die Trägermenge in S_1 sind die natürlichen Zahlen, dem Konstantensymbol o entspricht die Zahl 0, dem Funktionssymbol s entspricht die Funktion inc und dem Prädikatensymbol L entspricht die $<$ -Relation. Die Trägermenge in S_2 sind die Wochentage, dem Konstantensymbol o entspricht der Montag, dem Funktionssymbol s entspricht die Funktion morgen und dem Prädikatensymbol L entspricht die vor -Relation.

Wenn man sagt, für die Signatur Σ hat man die Σ -Algebra \mathcal{A} , dann ist ganz genau festgelegt, wie die Funktions- und Prädikatensymbole von Σ in \mathcal{A} interpretiert werden. Die Interpretation der Variablensymbole ist jedoch vorläufig noch offen. Auch ist noch nicht gesagt, wie die Abbildung der Terme formal definiert wird. Man will ja nicht irgendeine Abbildung, sondern eine, die zur Struktur der Terme paßt. Die eleganteste Formulierung dieser Abbildung, die auch für viele spätere Zwecke die richtige Begriffsbildung zur Verfügung stellt, geht von der Beobachtung aus, daß man für eine Signatur Σ aus der Menge der Terme $T(\Sigma, V)$ selbst zusammen mit einer Menge von "Termkonstruktorfunktionen" eine Σ -Algebra erhält. Diese Termkonstruktorfunktionen nehmen n Terme t_1, \dots, t_n und bauen daraus einen neuen Term $f(t_1, \dots, t_n)$ auf. Diese *Termalgebra* ist das geeignete Objekt, um eine Abbildung von Termen auf eine Σ -Algebra sauber zu definieren.

Definition 4.10. (*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 $\{f_\Sigma \mid f \in \mathcal{F}, \text{arity}(f) = n, f_\Sigma(t_1, \dots, t_n) := f(t_1, \dots, t_n)\}$. Dann nennt man $(T(\Sigma, V), F_T)$ die Termalgebra über der Signatur Σ .

Beachte, daß bei $f_\Sigma(t_1, \dots, t_n)$ die Klammern die n Argumente einklammern, auf die die Funktion f_Σ angewendet wird, während $f(t_1, \dots, t_n)$ ein Datenstrukturobjekt ist, d.h. ein Term in $T(\Sigma, V)$.

Aussage 4.11. $(T(\Sigma, V), F_T)$ ist eine Σ -Algebra.

Beweis. Da mindestens ein Konstantensymbol in der Signatur vorhanden ist, ist die Trägermenge $T(\Sigma, V)$ nicht leer. Für jede Funktion $f_\Sigma \in F_T$ gilt offensichtlich, daß die Stelligkeit paßt und daß f_Σ Terme auf Terme abbildet. Damit sind die Bedingungen der Definition 4.8 erfüllt. \square

Termmengen lassen sich somit als Σ -Algebren interpretieren. Worauf es dabei ankommt ist, daß die Abbildung von Elementen der Trägermenge und den Funktionen und Relationen in bestimmter Weise verträglich sind.

Was bisher noch fehlt, ist die Interpretation von Variablensymbolen.

Definition 4.12. Interpretation Gegeben sei eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und eine Menge von Variablen V . Eine Interpretation $I = (S, I_V)$ besteht aus einer Σ -Struktur \mathcal{S} und einer Variablenbelegung I_V . Die Variablenbelegung ist eine Abbildung $I_V : V \rightarrow D_S$, die jedem Variablensymbol einen Wert in der Trägermenge von S zuordnet.

Diese Interpretation wird verträglich erweitert auf Terme⁵:

$$I_h(t) = \begin{cases} I_V(t) & \text{falls } t \text{ eine Variable ist} \\ f_S(I_h(t_1), \dots, I_h(t_n)) & \text{falls } t = f(t_1, \dots, t_n) \end{cases}$$

Im Folgenden schreiben wir der Einfachheit halber I statt I_h .

Eine Interpretation wie sie in 4.12 definiert wurde, enthält alle Informationen, um den in einer Signatur vorkommenden Symbolen eine Bedeutung zuzuordnen. Die Belegung von Funktions- und Prädikatsymbolen wurde getrennt von der Belegung von Variablensymbolen definiert, weil die Belegung von Variablensymbolen mit Werten aus der Trägermenge einer Σ -Algebra erst geschieht, wenn quantifizierte Formeln zu Wahrheitswerten ausgewertet werden, wobei die Belegung der Variablen noch variiert wird. Für eine Formel " $\forall x : P(x)$ " zum Beispiel wird die Belegung des Prädikatsymbols " P " durch eine Σ -Struktur fixiert, während das Symbol x mit allen Werten der Trägermenge der Σ -Struktur belegt werden muß, um herauszufinden, ob $P(x)$ für alle x gilt.

Nachdem jetzt geklärt ist, wie man Terme evaluiert, wird im nächsten Schritt festgelegt, wie man Formeln auf Wahrheitswerte abbildet. Für atomare Formeln, d.h. Formeln der Art $P(t_1, \dots, t_n)$ funktioniert das einfach so, daß man zunächst die Terme t_1, \dots, t_n auf Elemente $I_h(t_1), \dots, I_h(t_n)$ der Trägermenge der Σ -Struktur abbildet und dann prüft, ob diese Elemente in der dem Prädikatsymbol zugeordneten Relation liegen. Wenn ja, ist das Atom wahr, wenn nicht, dann ist das Atom falsch. Für die zusammengesetzten Formeln wird die Abbildung auf Wahrheitswerte entsprechend der intuitiven Bedeutung der logischen Verknüpfungen und Quantoren definiert. (Dies ist die Stelle wo die Bedeutung der logischen Symbole präzisiert wird.)

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

Definition 4.13. Auswertung von Formeln: Sei $I = (S, I_V)$ eine Interpretation. Basisfälle:

$$\begin{aligned} \text{Fall: } H = P(t_1, \dots, t_n) & \quad \text{falls } (I(t_1), \dots, I(t_n)) \in P_S^6, \text{ dann } I(H) := 1. \\ & \quad \text{falls } (I(t_1), \dots, I(t_n)) \notin P_S, \text{ dann } I(H) := 0. \\ \text{Fall } H = P & \quad I(P) := P_S. \end{aligned}$$

Rekursionsfälle:

⁵ Man kann das auch als Homomorphismus von Algebren und Strukturen definieren.

⁶ P_S ist die in \mathcal{S} dem Symbol P zugeordnete Relation

Fall: $H = \mathbf{false}$ dann $I(H) = 0$
 Fall: $H = \mathbf{true}$ 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 4.14. (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 (Tautologie, Satz) 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.

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 4.15.

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 wählen wir $\{0, 1\}$, als Interpretation für P ebenfalls die Menge $\{0, 1\}$. Dann ergibt eine Interpretation I bzgl. dieser Struktur $S_2: I(\forall x.P(x))$ gdw $0 \in P_{S_2}$ und $1 \in P_{S_1}$. D.h. $I(\forall x.P(x)) = 1$.
2. Als Menge wählen wir $\{0, 1\}$, als Interpretation für P die Menge $\{0\}$. Dann ergibt eine Interpretation I bzgl. dieser Struktur $S_2: I(\forall x.P(x))$ gdw $0 \in P_{S_2}$ und $1 \in P_{S_1}$. D.h. $I(\forall x.P(x)) = 0$.

Als weiteren, einfachen Testfall untersuchen wir Klauseln.

Beispiel 4.16. Wann ist die Klausel $\{P(s), \neg P(t)\}$ eine Tautologie?

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 es keine Tautologie. Um dies nachzuweisen, muß man eine Interpretation finden, die diese Klausel falsch macht.

Zuerst definieren wir eine Trägermenge und eine Σ -Algebra. Man startet mit einer Menge A_0 , die mindestens so viele Elemente enthält, wie die Terme s, t Konstanten und Variablen enthalten. Also $A_0 := \{a_1, \dots, a_n, c_{x_1}, \dots, c_{x_m}\}$, 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 (die Menge), 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. es kann keine Tautologie sein.

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

Übungsaufgabe 4.17. 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 4.18. (*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 4.19. Man muß sich folgendes klar machen: Bezüglich einer gegebenen Σ -Algebra ist eine gegebene geschlossene Formel F entweder wahr oder falsch. Das heißt aber noch nicht, daß entweder F oder $\neg F$ eine Tautologie ist, denn dieser Begriff ist definiert über **alle** Σ -Strukturen.

Bemerkung 4.20. 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 4.21. 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.

Satz 4.22. *Deduktionstheorem Für alle Formeln F und G gilt: $F \models G$ gdw. $F \Rightarrow G$ ist allgemeingültig (Tautologie).*

Beweis. “ \Rightarrow “ Es gelte $F \models G$. Sei I eine beliebige Interpretation. Wenn F in I wahr ist, dann ist auch G wahr, nach Annahme. Damit ist aber auch die Formel $F \Rightarrow G$ in I wahr. Wenn F in I falsch ist, dann ist die Formel $F \Rightarrow G$ in I wahr.

Also gilt $F \Rightarrow G$ in allen Interpretationen I , d.h. $F \Rightarrow G$ ist allgemeingültig.

“ \Leftarrow “ Annahme ist jetzt: $F \Rightarrow G$ ist allgemeingültig. Sei I eine beliebige Interpretation. Wenn I die Formel F erfüllt, dann gilt das auch für G , da $F \Rightarrow G$ ist allgemeingültig ist.

Da die Interpretation beliebig gewählt war, gilt somit $F \models G$ □

Bemerkung 4.23. Will man wissen, ob eine Formel F aus einer Menge von Axiomen A_1, \dots, A_n folgt, so kann man dies zunächst auf die äquivalente Frage zurückführen, ob F aus der Konjunktion der Axiome $A_1 \wedge \dots \wedge A_n$ folgt und dann auf die äquivalente Frage, ob die Implikation $(A_1 \wedge \dots \wedge A_n) \Rightarrow F$ eine Tautologie ist.

Das Deduktionstheorem gilt in anderen Logiken i.a. nicht mehr. 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.

Berechenbarkeitseigenschaften der Prädikatenlogik Es gilt die Unentscheidbarkeit der Prädikatenlogik:

Satz 4.24. *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.

Satz 4.25. *Die Menge der Sätze der Prädikatenlogik ist rekursiv aufzählbar.*

Die Begründung ist analog.

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 (aus Info-3+4), daß die Eigenschaft “Tautologie“ für QBF entscheidbar ist (PSPACE-vollständig).

4.3 Normalformen von PL_1 -Formeln

Ziel dieses Abschnitts ist es, einen Algorithmus zu entwickeln, der beliebige Formeln in eine Klauselnormalform (conjunctive normal form, CNF), d.h. eine Kon-

junktion (\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.

Satz 4.26. Elementare Rechenregeln

$$\begin{array}{lll}
(F \Leftrightarrow G) & \Leftrightarrow (F \Rightarrow G) \wedge (G \Rightarrow F) & (\text{erlaubt Elimination von } \Leftrightarrow) \\
(F \Rightarrow G) & \Leftrightarrow (\neg F \vee G) & (\text{erlaubt Elimination von } \Rightarrow) \\
\neg \neg F & \Leftrightarrow F & \\
\neg(F \wedge G) & \Leftrightarrow \neg F \vee \neg G & \\
\neg(F \vee G) & \Leftrightarrow \neg F \wedge \neg G & \\
\neg \forall x : F & \Leftrightarrow \exists x : \neg F & \\
\neg \exists x : F & \Leftrightarrow \forall x : \neg F & \\
(\forall x : F) \wedge G & \Leftrightarrow \forall x : (F \wedge G) & \text{falls } x \text{ nicht frei in } G \\
(\forall x : F) \vee G & \Leftrightarrow \forall x : (F \vee G) & \text{falls } x \text{ nicht frei in } G \\
(\exists x : F) \wedge G & \Leftrightarrow \exists x : (F \wedge G) & \text{falls } x \text{ nicht frei in } G \\
(\exists x : F) \vee G & \Leftrightarrow \exists x : (F \vee G) & \text{falls } x \text{ nicht frei in } G \\
\forall x : F \wedge \forall x : G & \Leftrightarrow \forall x : (F \wedge G) & \\
\exists x : F \vee \exists x : G & \Leftrightarrow \exists x : (F \vee G) & \\
F \vee (G \wedge H) & \Leftrightarrow (F \vee G) \wedge (F \vee H) & (\text{Distributivität}) \\
F \wedge (G \vee H) & \Leftrightarrow (F \wedge G) \vee (F \wedge H) & (\text{Distributivität})
\end{array}$$

Beweis. Wir beweisen beispielhaft den Fall: $\neg \forall x : F \Leftrightarrow \exists x : \neg F$

“ \Rightarrow “ Sei I eine Interpretation, die $\neg \forall x : F$ erfüllt.

d.h. $\forall x : F$ gilt nicht in I (4.13).

\leadsto D.h.: es gilt nicht: für alle a aus der Trägermenge von I : $I[a/x]$ erfüllt F .

\leadsto Somit gibt es ein a aus der Trägermenge von I , so daß F nicht in $I[a/x]$ gilt.

\leadsto Es gibt ein a aus der Trägermenge von I , so daß $I[a/x]$ die Formel $\neg F$ erfüllt (4.13, Fall \neg)

$\leadsto I$ erfüllt damit $\exists x : \neg F$. (4.13, Fall \exists)

“ \Leftarrow “ Sei I eine Interpretation, die $\exists x : \neg F$ erfüllt. Die obige Folgerungskette kann auch rückwärts durchlaufen werden. \square

Bemerkung 4.27. Mithilfe der obigen Tautologien kann man die sogenannte Pränexform und auch die Negations-normalform 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. $\exists x : P(x) \rightarrow P(a)$ Ersetze in $\forall x_1 \dots x_n : \exists y : P(x_1, \dots, x_n, y)$ das y durch eine Funktion von x_1, \dots, x_n , d.h. $\forall x_1 \dots x_n : \exists y : P(x_1, \dots, x_n, y) \rightarrow \forall x_1 \dots x_n : P(x_1, \dots, x_n, f(x_1, \dots, x_n))$.

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

Satz 4.28. Skolemisierung

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

Beweis. Wir zeigen die Erfüllbarkeitsäquivalenz. Die Unerfüllbarkeitsäquivalenz ergibt sich dann automatisch.

“ \Rightarrow “ Sei $I_1 = (A_1, I_V)$ eine Σ_1 -Interpretation mit $I_1 \models F$, wobei Σ_1 die Signatur für \mathcal{F} ist. Für $\Sigma_2 = \Sigma_1$ plus zusätzliches $n + 1$ -stelliges Funktionssymbol f konstruieren wir ein Σ_2 -Modell I_2 für F' folgendermaßen: $I_2 = (A_2, I_V)$ mit $A_2 := A_1$ plus eine neue $n + 1$ -stellige Funktion f_{A_2} , wobei f_{A_2} wie folgt arbeitet: für alle a_1, \dots, a_n aus der Trägermenge von A_2 : $f_{A_2}(a_1, \dots, a_n) := c$, so daß

$$I_1[a_1/x_1, \dots, a_n/x_n, c/y] \models G[x_1, \dots, x_n, y]$$

. Da I_1 ein Modell für F ist, existiert so ein c immer.

$$\begin{aligned} &\leadsto I_2[a_1/x_1, \dots, a_n/x_n] \models G[x_1, \dots, x_n, f(x_1, \dots, x_n)] \\ &\leadsto I_2 \models \forall x_1, \dots, x_n : G[x_1, \dots, x_n, f(x_1, \dots, x_n)] \\ &\leadsto F' \text{ ist erfüllbar.} \end{aligned}$$

Die “ \Leftarrow “ Richtung ist trivial. □

Beispiel 4.29. Skolemisierung

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

Beispiel 4.30. Skolemisierung erhält i.a. nicht die Allgemeingültigkeit (Falsifizierbarkeit):

$$\begin{aligned} &\forall x : P(x) \vee \neg \forall x : P(x) \text{ ist eine Tautologie} \\ &\forall x : P(x) \vee \exists x : \neg P(x) \text{ ist äquivalent zu} \end{aligned}$$

$\forall x : P(x) \vee \neg P(a)$ 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 allquantifizierten 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.

Satz 4.31. 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, \dots, x_n . D.h. F ist von der Form $F[\exists y : G'[x_1, \dots, x_n, y]]$.

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

Definition 4.32. Transformation in Klauselnormalform unter Erhaltung der Unerfüllbarkeit.

Folgende Prozedur wandelt jede prädikatenlogische Formel in Klauselform (CNF) um:

1. Elimination von \Leftrightarrow und \Rightarrow : $F \Leftrightarrow G \rightarrow F \Rightarrow G \wedge G \Rightarrow F$ (Lemma 4.26) 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

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

4. Alle gebundenen Variablen sind systematisch umzubenennen, um Namenskonflikte aufzulösen.
5. Existenzquantoren werden durch Skolemisierung eliminiert
6. Allquantoren löschen (alle Variablen werden als allquantifiziert angenommen).
7. Distributivität (und Assoziativität, Kommutativität) iterativ anwenden, um \wedge nach außen zu schieben ("Ausmultiplikation"). $F \vee (G \wedge H) \rightarrow (F \vee G) \wedge (F \vee H)$ (Das duale Distributivgesetz würde eine disjunktive Normalform ergeben.)

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

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

oder in (Multi-)Mengenschreibweise:

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

Beispiel 4.33. Der CNF-Algorithmus (4.32) ist im schlechtesten Fall exponentiell, d.h. die Anzahl der Literale in der Klauselform wächst exponentiell mit der Schachtelungstiefe der Ausgangsformel (siehe 2.4).

Analog zu 2.4 und unter den gleichen Bedingungen gibt es einen verbesserten Algorithmus, der nur lineares Anwachsen der Formel zur Folge hat.

Beispiel 4.34.

- 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 4.35. verschiedene Typen von Normalformen:

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

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

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

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

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

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

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

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

5 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. Er arbeitet in erster Linie auf Klauseln.

5.1 Grundresolution: Resolution ohne Unifikation

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

Definition 5.1. *Resolution im Fall direkt komplementärer Resolutionsliterals.*

$$\begin{array}{l} \text{Elternklausel 1: } L, K_1, \dots, K_m \\ \text{Elternklausel 2: } \neg L, N_1, \dots, N_n \\ \hline \text{Resolvente: } K_1, \dots, K_m, N_1, \dots, N_n \end{array}$$

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 5.2. siehe Beispiel 4.34

- 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.

Aussage 5.3. *Die Grund-Resolution ist korrekt:*

$$\begin{array}{l} C_1 := L, K_1, \dots, K_m \\ C_2 := \neg L, N_1, \dots, N_n \\ \hline R = K_1, \dots, K_m, N_1, \dots, N_n \end{array}$$

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. \square

Bemerkung 5.4. 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.

Satz 5.5. *(Widerlegungsvollständigkeit der Grundresolution) Jede endliche unerfüllbare Grundklauselmenge läßt sich durch Resolution widerlegen.*

Beweis. siehe Beweis von Satz 2.31 \square

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 5.6. *(Substitution)*

Eine Substitution σ ist eine Abbildung, die Terme auf Terme abbildet und die nur endlich viele Variablen verändert. Zusätzlich gilt: $\sigma(f(t_1, \dots, t_n)) = f(\sigma(t_1), \dots, \sigma(t_n))$. 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:

$$\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$$

Mit Hilfe der rekursiven Definition:

$$\sigma(t) := \begin{cases} t_i & \text{falls } t \equiv x_i \text{ und } \sigma(x_i) = t_i \\ f(\sigma(t_1), \dots, \sigma(t_n)) & \text{falls } t \equiv f(t_1, \dots, t_n) \end{cases}$$

lassen sich solche Mengen von Variable - Term Paaren korrekt als (mit Datenstruktur verträgliche) Abbildung auf beliebigen Termen definieren. Entsprechend kann man die Anwendung von Substitutionen auf Literale und Klauseln definieren.

Beispiel 5.7.

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

Definition 5.8. Für eine Substitution $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ heißt

$$\begin{aligned}
\text{dom}(\sigma) &:= \{x_1, \dots, x_n\} & \text{die Domain} \\
\text{cod}(\sigma) &:= \{t_1, \dots, t_n\} & \text{die Codomain}
\end{aligned}$$

Substitutionen σ mit $\sigma(\sigma(t)) = \sigma(t)$ für alle Terme t heißen idempotent.

Die Komposition $\sigma\tau$ von Substitutionen σ und τ ist definiert als: $\sigma\tau(t) := \sigma(\tau(t))$.

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 Instantiierungsrelation $\leq [W]$ 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.

Bemerkung 5.9. Die Relationen $\leq [W]$ ist reflexiv und transitiv, aber nicht antisymmetrisch. D.h. es ist eine Präordnung.

Die Relation $\approx [W]$ ist eine Äquivalenzrelation.

Eine Substitution ist genau dann idempotent wenn die Variablen der Domain nicht in der Codomain vorkommen.

Beispiel 5.10.

Komposition:

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

Instantiierungsrelation:

- $\{x \mapsto y\} \leq \{x \mapsto a\}[x]$
- $\{x \mapsto y\} \leq \{x \mapsto a\}[x, y]$ 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)

Für die Äquivalenz von Substitutionen gilt: $\sigma \approx \tau[W]$ gdw. $\sigma = \rho\tau[W]$ für eine Variablenpermutation ρ . Eine Variablenpermutation ρ ist eine Substitution, die Variablen permutiert, d.h. $\text{Dom}(\rho) = \text{Cod}(\rho)$ und ρ bildet Variablen auf Variablen ab. (Beweis Übungsaufgabe)

Definition 5.11. (*Resolution mit Unifikation*)

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

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

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

$$\frac{\text{rasiert}(x, x), \text{rasiert}(\text{Friseur}, x) \quad \sigma = \{x \mapsto \text{Friseur}, y \mapsto \text{Friseur}\} \quad \neg \text{rasiert}(\text{Friseur}, y), \neg \text{rasiert}(y, y)}{\text{rasiert}(\text{Friseur}, \text{Friseur}), \neg \text{rasiert}(\text{Friseur}, \text{Friseur})}$$

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" instantiiert, und dann die Literale verschmolzen.

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

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

Definition 5.13. (*Faktorisierung*)

$$\begin{array}{ll} \text{Elternklausel: } L, L', K_1, \dots, K_m & \sigma(L) = \sigma(L') \\ \text{Faktor: } & \frac{\sigma(L, K_1, \dots, K_m)}{\sigma(L, L', K_1, \dots, K_m)} \end{array}$$

Damit besteht der Resolutionskalkül jetzt aus Resolution und Faktorisierung.

Definition 5.14. *Der Resolutionskalkül transformiert Klauselmengen S wie folgt:*

1. $S \rightarrow S \cup \{R\}$, wobei R eine Resolvente von zwei (nicht notwendig verschiedenen) Klauseln aus S ist.
2. $S \rightarrow S \cup \{F\}$, 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 variablen-disjunkt sind.

Beispiel 5.15. 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 |

5.2 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 5.16. Unifikatoren und allgemeinste Unifikatoren.

$$\frac{P(x), Q(x)}{\frac{\neg P(y), R(y)}{Q(a), R(a)} \quad \sigma = \{x \mapsto a, y \mapsto a\}} \quad \sigma \text{ ist ein Unifikator}$$

$$\frac{P(x), Q(x)}{\frac{\neg P(y), R(y)}{Q(y), R(y)} \quad \sigma = \{x \mapsto y\}} \quad \sigma \text{ ist ein allgemeinster Unifikator}$$

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 instantiiert. Optimal ist es dann, wenn alle Unifikatoren durch weitere Einsetzung in den allgemeinsten Unifikator erzeugt werden können.

Definition 5.17. *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

$$\begin{aligned} \sigma \text{ ein Unifikator ist} & \quad \text{(Korrektheit)} \\ \text{für alle Unifikatoren } \tau \text{ gilt } \sigma \leq \tau[W] & \quad \text{(Vollständigkeit)} \end{aligned}$$

Der allgemeinste Unifikator ist i.a. nur eindeutig bis auf $\equiv [W]$:

Beispiel 5.18. Es gibt (mindestens) zwei allgemeinste Unifikatoren für x und y : $\{x \mapsto y\}$, $\{y \mapsto x\}$, die sich aber nur durch Variablenumbenennung unterscheiden.

Die Beispiele legen die Vermutung nahe, daß es bis auf Variablenumbenennung immer einen allgemeinsten Unifikator gibt. Wir geben den Unifikationsalgorithmus in Form eines Regelpaketes an, das auf Mengen von Gleichungen operiert.

Definition 5.19. *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:

$$\begin{array}{ll}
 \frac{f(s_1, \dots, s_n) \stackrel{?}{=} f(t_1, \dots, t_n), \Gamma}{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n, \Gamma} & (Dekomposition) \\
 \frac{x \stackrel{?}{=} x, \Gamma}{\Gamma} & (Tautologie) \\
 \frac{\Gamma}{x \stackrel{?}{=} t, \Gamma} & x \in FV(\Gamma), x \notin FV(t) \quad (Anwendung) \\
 \frac{x \stackrel{?}{=} t, \{x \mapsto t\} \Gamma}{t \stackrel{?}{=} x, \Gamma} & \\
 \frac{t \stackrel{?}{=} x, \Gamma}{x \stackrel{?}{=} t, \Gamma} & t \notin V \quad (Orientierung)
 \end{array}$$

Abbruchbedingungen:

$$\begin{array}{ll}
 \frac{f(\dots) \stackrel{?}{=} g(\dots), \Gamma}{Fail} & \text{wenn } f \neq g \quad (Clash) \\
 \frac{x \stackrel{?}{=} t, \Gamma}{Fail} & \text{wenn } x \in FV(t) \text{ vorkommt (occurs check Fehler)} \\
 & \text{und } t \neq x
 \end{array}$$

Steuerung:

Starte mit $\Gamma = \Gamma_0$, und transformiere Γ solange durch (nichtdeterministische) 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 $\{x_1 \stackrel{?}{=} t_1, \dots, x_k \stackrel{?}{=} t_k\}$, wobei keine der Variablen x_i in einem t_j vorkommt; d.h. sie ist in gelöster Form. Das Resultat ist dann $\{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$. (siehe Lemma 5.22).

Beispiel 5.20. Unifikation von zwei Termen durch Anwendung der obigen Regeln:

$$\begin{aligned}
& \{k(f(x, g(a, y)), g(x, h(y))) \stackrel{?}{=} k(f(h(y), g(y, a)), g(z, z))\} \\
\rightarrow & \{f(x, g(a, y)) \stackrel{?}{=} f(h(y), g(y, a)), g(x, h(y)) \stackrel{?}{=} g(z, z)\} & (\text{Dekomposition}) \\
\rightarrow & x \stackrel{?}{=} h(y), g(a, y) \stackrel{?}{=} g(y, a), g(x, h(y)) = g(z, z) & (\text{Dekomposition}) \\
\rightarrow & x \stackrel{?}{=} h(y), a \stackrel{?}{=} y, y \stackrel{?}{=} a, g(x, h(y)) \stackrel{?}{=} g(z, z) & (\text{Dekomposition}) \\
\rightarrow & x \stackrel{?}{=} h(y), y \stackrel{?}{=} a, g(x, h(y)) \stackrel{?}{=} g(z, z) & (\text{Orientierung}) \\
\rightarrow & x \stackrel{?}{=} h(a), y \stackrel{?}{=} a, g(x, h(a)) \stackrel{?}{=} g(z, z) & (\text{Anwendung, } y) \\
\rightarrow & x \stackrel{?}{=} h(a), y \stackrel{?}{=} a, x \stackrel{?}{=} z, h(a) \stackrel{?}{=} z & (\text{Dekomposition}) \\
\rightarrow & x \stackrel{?}{=} h(a), y \stackrel{?}{=} a, x \stackrel{?}{=} z, z \stackrel{?}{=} h(a) & (\text{Orientierung}) \\
\rightarrow & x \stackrel{?}{=} h(a), y \stackrel{?}{=} a, z \stackrel{?}{=} h(a) & (\text{Anwendung, } z)
\end{aligned}$$

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

Satz 5.21. *Der Unifikationsalgorithmus U1 terminiert, ist korrekt und vollständig.*

Beweis. Seien s und t die zu unifizierenden Terme. Sei $W := FV(s, t)$.

Korrektheit: Wir müssen zeigen, daß die berechnete Substitution auch wirklich ein Unifikator für die beiden Eingabeterme ist. Die Idee dazu ist folgende: Man zeige, daß für jede Transformationsregel gilt: wenn die Gleichungsmenge Γ' nach einer Regelanwendung unifizierbar ist, dann ist auch die entsprechende Menge vor der Regelanwendung unifizierbar. Per Induktion nach der Anzahl der Regelanwendungen ergibt sich dann: Wenn die Lösungsgleichungsmenge unifizierbar ist, dann sind auch die Ausgangsterme unifizierbar. Man muß jetzt jede Unifikationsregel daraufhin untersuchen.

Wir machen das am Beispiel der Dekompositionsregel:

Angenommen, $\Gamma' = \{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n\} \cup \Gamma$ ist unifizierbar, d.h. es existiert eine Substitution $\tau \in U(\{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n\} \cup \Gamma)$. Dann gilt selbstverständlich $\tau f(s_1, \dots, s_n) \stackrel{?}{=} \tau f(t_1, \dots, t_n)$, d.h. das Gleichungssystem vorher ist unifizierbar. Für die anderen Regeln ist die Argumentation analog.

Terminierung Um zu zeigen, daß der Unifikationsalgorithmus immer terminiert, müssen wir ein (fundiertes) Maß für Gleichungssysteme finden, das nach jeder Regelanwendung kleiner wird. Ein Maß ist das Tupel (Anzahl der ungelösten Variablen, Anzahl der Symbole, Anzahl der falsch orientierten Gleichungen) mit lexikographischer Ordnung. (Eine Variable x heißt gelöst

wenn sie nur noch in der Gleichung $x \stackrel{?}{=} t$, aber x nicht in t vorkommt.). Dieses Maß wird nach jeder Regelanwendung kleiner: Wenn die Dekomposition oder Tautologieregel feuert, wird die Anzahl der Symbole kleiner, wenn die Anwendungsregel feuert, wird die Anzahl der ungelösten Variablen kleiner, und die Orientierungsregel verkleinert die dritte Komponente und läßt die anderen fest.

Vollständigkeit: Wir zeigen, daß alle Regeln die Menge der Unifikatoren nicht verändert. Da die Korrektheit schon gezeigt ist, d.h. die Menge vergrößert sich nicht, müssen wir jetzt nur noch zeigen, daß die Menge der Unifikatoren sich nicht verkleinert. Sei τ eine Substitution mit $\tau \in U(\Gamma)$ und sei $\Gamma \rightarrow \Gamma'$ transformiert worden. Wenn die entsprechende Regel die Dekomposition, Tautologie oder die Orientierung war, dann sieht man leicht ein, daß $\tau \in U(\Gamma')$. Betrachtung wir die Regel (Anwendung). Dann gilt $\tau x = \tau t$. Hieraus folgt, daß $\tau = \tau\{x \mapsto t\}$ ist, denn für x gilt: $\tau(x) = \tau(t)$ und $\tau\{x \mapsto t\}(x) = \tau(t)$. Für andere Variablen $y \neq x$ gilt offenbar $\tau\{x \mapsto t\}(y) = \tau(y)$. Folglich gilt auch hier $\tau \in U(\Gamma')$. □

Lemma 5.22. *Der Unifikationsalgorithmus U1 terminiert entweder mit einem Unifikator in gelöster Form, oder er bricht ab mit "nicht unifizierbar".*

Beweis. nach 5.21 terminiert U1 immer. Es gibt folgende Möglichkeiten: entweder bricht er ab mit "nicht unifizierbar" oder es ist keine Regel mehr anwendbar. Wenn keine Regel mehr anwendbar ist, dann gibt es nur noch Gleichungen $x \stackrel{?}{=} t$, Dekomposition ist nicht mehr anwendbar, die Gleichungen sind richtig orientiert und x kommt nur einmal in Γ vor, insbesondere nicht in t . □

Lemma 5.23. *Sei $\Gamma = \{x_1 \stackrel{?}{=} t_1, \dots, x_k \stackrel{?}{=} t_k\}$ ein Gleichungssystem in gelöster Form. Dann ist $\sigma := \{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$ ein allgemeinsten Unifikator.*

Beweis. Offenbar ist σ ein Unifikator. Zum Nachweis der Allgemeinstheit sei τ ein Unifikator von Γ . Dann gilt $\tau\sigma = \tau$: $\tau\sigma x_i = \tau t_i = \tau x_i$ für $i = 1, \dots, k$ und für eine Variable $x \notin \{x_1, \dots, x_k\}$ gilt $\tau\sigma x = \tau x$. Also ist σ allgemeiner als τ . □

Satz 5.24. *Für jedes unifizierbare Gleichungssystem Γ gibt es bis auf Umbenennung von Variablen genau einen allgemeinsten Unifikator und U1 berechnet einen solchen.*

5.3 Komplexität des Unifikationsalgorithmus

Aussage 5.25. *Der Unifikationsalgorithmus U1 hat exponentielle Laufzeit in der Anzahl der Symbole der zu unifizierenden Terme:*

Beispiel 5.26. Zu unifizieren sei:

$$f(x_2, x_3, x_4, \dots, x_n) \stackrel{?}{=} f(g(x_1, x_1), g(x_2, x_2), g(x_3, x_3), \dots, g(x_{n-1}, x_{n-1}))$$

Der Unifikationsalgorithmus arbeitet dann folgendermaßen:

$$\begin{aligned}
& f(x_2, x_3, x_4, \dots, x_n) \stackrel{?}{=} f(g(x_1, x_1), g(x_2, x_2), g(x_3, x_3), \dots, g(x_n \Leftrightarrow 1, x_n \Leftrightarrow 1)) \\
\rightarrow x_2 & \stackrel{?}{=} g(x_1, x_1), x_3 \stackrel{?}{=} g(x_2, x_2), x_4 \stackrel{?}{=} g(x_3, x_3), \dots, x_n \stackrel{?}{=} g(x_{n-1}, x_{n-1}) & \text{(Dekomposition)} \\
\rightarrow x_2 & \stackrel{?}{=} g(x_1, x_1), x_3 \stackrel{?}{=} g(g(x_1, x_1), g(x_1, x_1)), \dots, x_n \stackrel{?}{=} g(x_{n-1}, x_{n-1}) & \text{(Anwendung, } x_2) \\
\rightarrow x_2 & \stackrel{?}{=} g(x_1, x_1), x_3 \stackrel{?}{=} g(g(x_1, x_1), g(x_1, x_1)), \\
& x_4 \stackrel{?}{=} g(g(g(x_1, x_1), g(x_1, x_1)), g(g(x_1, x_1), g(x_1, x_1))), \dots, x_n \stackrel{?}{=} g(x_{n-1}, x_{n-1}) & \text{(Anwendung, } x_3)
\end{aligned}$$

Da die Terme exponentiell anwachsen, benötigt man in diesem Beispiel exponentiell viel Platz.

Eine Verbesserung ist die Verwendung von Sharing, d.h. gerichteten Graphen statt Bäumen). Danach wird der Platzbedarf stark verbessert (linear), allerdings benötigt der (naive) occurs-check exponentiellen Aufwand. Auch diesen kann man verbessern durch Verwendung von optimierten Graph-algorithmen, so daß man nach Optimierung einen im schlimmsten Fall quadratischen Algorithmus erhält.

Es gibt jedoch Unifikationsalgorithmen, die durch geschickte Repräsentation der Terme eine lineare Komplexität haben [PM68]. Allerdings sind für praktische Zwecke die Varianten des Algorithmus von Martelli und Montanari geeigneter [MM82].

5.4 Der allgemeine Resolutionskalkül

Wir können nun zusammenfassend sagen, wie der Resolutionskalkül auf Klauselmengen arbeitet:

Definition 5.27. *Der Resolutionskalkül ist ein Kalkül, der Klauselmengen in Klauselmengen transformiert mit den folgenden beiden Regeln:*

1. *Sei C eine Menge von Klauseln. Sei R eine Resolvente von zwei (nicht notwendig verschiedenen) Klauseln in CS , wobei der zugehörige Unifikator ein allgemeinsten ist. Dann bilde $C' := C \cup \{R\}$*
2. *Sei C eine Menge von Klauseln. Sei F ein Faktor einer der Klauseln in CS , wobei der zugehörige Unifikator ein allgemeinsten ist. Dann bilde $C' := C \cup \{F\}$*

5.5 Schritte zum Vollständigkeitsbeweis der allgemeinen Resolution

Definition 5.28. *Herbrand Interpretation: Sei C eine Klauselmenge über der Signatur Σ und V . Eine Herbrand Interpretation ist eine Interpretation $(I_V, T(\Sigma, \emptyset), P)$ wobei*

- $T(\Sigma, \emptyset)$ die Grundtermalgebra ist (ohne Variablensymbole) (auch Herbrand Universum genannt) und
- I_V eine Variablenbelegung in $T(\Sigma, \emptyset)$, d.h. eine Abbildung auf Grundterme
- P für jedes Prädikatsymbol eine Relation auf Grundtermen festlegt.

(Für jedes Prädikatsymbol kann man die Relation dadurch festlegen, daß man eine Menge von Atomen angibt, die als wahr in der Interpretation angenommen werden sollen.)

Bemerkung: Für eine feste Klauselmenge unterscheiden sich die Herbrand-Interpretationen nur durch die Interpretation der Variablen und Prädikatsymbole.

Satz 5.29. (Herbrand Theorem) Für eine unerfüllbare Menge C von Klauseln gibt es eine endliche unerfüllbare Menge von Grundinstanzen

Beweis. 1.) Die Menge C_{gr} aller Grundinstanzen von C ist unerfüllbar. Wir nehmen an, daß die Menge C_{gr} aller Grundinstanzen von C erfüllbar ist, von einer Interpretation I . Wir konstruieren eine Herbrandinterpretation I_H für C_{gr} wie folgt: In jeder Grundklausel wird mindestens ein Literal von I wahr gemacht. All diese Literale zusammen ergeben dann genau eine (Herbrand) Interpretation der Prädikatsymbole, die in C vorkommen.

Um zum Widerspruch zu kommen, müssen wir jetzt zeigen, daß diese Herbrandinterpretation auch die ursprüngliche Klauselmenge C erfüllt. Das heißt, für jede Klausel D und für jede Belegung der Variablen x_1, \dots, x_n in der Klausel mit Grundtermen muß gelten, daß $I_H[t_1/x_1, \dots, t_n/x_n] \models D$. Es ist jedoch $\{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}C$ gerade eine der Grundklauseln, die von I_H erfüllt wird. Da bei Herbrand-Interpretationen Variablenbelegungen dasselbe sind wie Substitutionen, gilt: $I_H[t_1/x_1, \dots, t_n/x_n] \models C$. Also wird die Klauselmenge C von I_H erfüllt, und das ist der gesuchte Widerspruch.

2.) Es gibt eine endliche unerfüllbare Teilmenge von C_{gr} :

Annahme: jede endliche Teilmenge von C_{gr} erfüllbar ist. Wir konstruieren damit eine Interpretation von C_{gr} . Sei A_1, A_2, \dots eine Aufzählung der Atome einer Herbrandbasis (alle Grundatome). Sei $\mathcal{B} := \bigcup \mathcal{B}_n$ wobei $\mathcal{B}_n := \{B \mid B : \{A_1, A_2, \dots, A_n\} \rightarrow \{0, 1\}\}$. Das ist die Menge aller möglichen (Teil-) Belegungen von Atomen. Dies ergibt einen Baum mit den Belegungen als Knoten; von B nach B' geht genau dann eine gerichtete Kante, wenn $B \in \mathcal{B}_n$, $B' \in \mathcal{B}_{n+1}$ und B und B' auf $\{A_1, A_2, \dots, A_n\}$ übereinstimmen. Ein Knoten $B \in \mathcal{B}_n$ in diesem Baum heißt *abgeschlossen*, wenn es eine Klausel in C_{gr} gibt, so daß alle in C_{gr} vorkommenden Atome in $\{A_1, A_2, \dots, A_n\}$ sind und B diese Klausel zu falsch evaluiert. Offenbar ist auch jeder Nachfolgeknoten eines abgeschlossenen Knotens wieder abgeschlossen. Für jedes n gibt es einen nicht abgeschlossenen Knoten in \mathcal{B}_n , denn anderenfalls gibt es für jeden Knoten in \mathcal{B}_n eine Klausel in C_{gr} , die zu falsch evaluiert wird. Dies sind endlich viele Klauseln, deren Konjunktion könnte dann von keiner Interpretation erfüllt werden, was ein Widerspruch zur obigen Annahme ist. Der Teilbaum der nicht abgeschlossenen Knoten ist ein unendlicher Baum mit endlicher Verzweigungsrate. Nach König's Lemma gibt

es einen unendlichen Pfad in diesem Baum. Dieser unendliche Pfad stellt gerade eine Interpretation von C_{gr} dar. Dies ist ein Widerspruch zur Annahme, daß C_{gr} unerfüllbar ist. Damit haben wir gezeigt, daß es eine *endliche* unerfüllbare Teilmenge von C_{gr} gibt. \square

Das hier nicht bewiesene Kompaktheitstheorem für PL_1 sagt aus, daß jede unerfüllbare unendliche Menge von Aussagen eine endliche, unerfüllbare Teilmenge hat. Unter Ausnutzung dieses Satzes kann man im zweiten Teil des obigen Beweises sofort schließen, daß es eine unerfüllbare endliche Teilmenge von C_{gr} geben muß.

Das Herbrand Theorem können wir jetzt für folgende Argumentation benutzen: Für eine unerfüllbare Klauselmenge gibt es offensichtlich eine endliche Menge von Grundinstanzen, so daß diese Grundklauselmenge unerfüllbar ist. Diese Grundklauselmenge können wir durch Resolution ohne Unifikation widerlegen, das sagte gerade das Theorem 5.5. Was wir jetzt noch brauchen ist die Möglichkeit, diese Widerlegung auf der variablenfreien Ebene zu einer Widerlegung auf der Ebene der Originalklauseln zu "liften", d.h. wir müssen zeigen, daß es für jeden Grundresolutionsschritt eine entsprechende allgemeine Resolution gibt, so daß die Grundresolvente eine Instanz der allgemeinen Resolvente ist. Das leistet das sogenannte Lifting Lemma

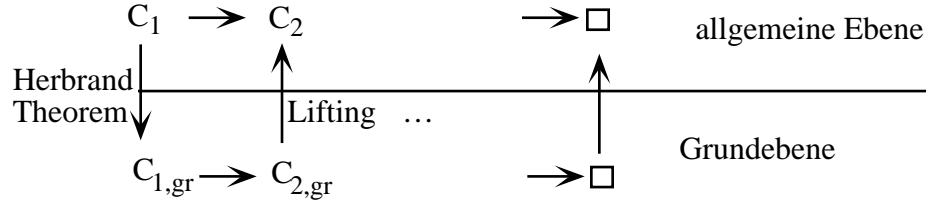
Lemma 5.30. Lifting Lemma

Seien C und D zwei beliebige Klauseln ohne gemeinsame Variablen, C_{gr} und D_{gr} zwei Grundinstanzen davon. Für jede Resolvente R_{gr} zwischen C_{gr} und D_{gr} gibt es eine entsprechende Resolvente R von Faktoren (siehe Definition. 5.13) von C und D , so daß R_{gr} eine Grundinstanz von R ist.

Beweis. siehe einschlägige Literatur. \square

Satz 5.31. *Vollständigkeit der Resolution und Faktorisierung mit allgemeinsten Unifikatoren: Für jede unerfüllbare Klauselmenge gibt es einen Widerlegungsbe-
weis mittels Resolution und Faktorisierung.*

Beweis. Wir sammeln einfach alle bisherigen Ergebnisse auf: Nach dem Herbrand Theorem (5.29) gibt es eine endliche unerfüllbare Grundklauselmenge, für die man nach Theorem 5.5 eine Resolutionswiderlegung findet. Durch Induktion nach der Anzahl der Grundresolutionen bekommt man jetzt mittels des Lifting Lemmas, daß es für jede dieser Grundresolutionen eine entsprechende Sequenz von Faktorisierungen und Resolution mit allgemeinsten Unifikatoren gibt, so daß die Resolvente allgemeiner ist als die Grundresolvente. Für die allerletzte Grundresolvente, die leere Klausel, ist die einzige allgemeinere Klausel ebenfalls die leere Klausel. Daher wird auf der allgemeineren Ebene ebenfalls die leere Klausel produziert. \square



5.6 Ein Beispiel für Resolutionsbeweise

Wir formulieren in PL_1 Aussagen über eine andere Logik, nennen wir sie MINI. Dies ist eine Implementierung einer Variante des Hilbertkalküls für Aussagenlogik. Die Syntax dieser Logik MINI lasse nur nullstellige Prädikatsymbole und einen Implikationsjunktork IMP zu. Wir betrachten einen Kalkül für MINI mit zwei logischen Axiomen

$$X \text{ IMP } (Y \text{ IMP } X)$$

$$(X \text{ IMP } (Y \text{ IMP } Z)) \text{ IMP } ((X \text{ IMP } Y) \text{ IMP } (X \text{ IMP } Z))$$

und der einzigen Schlußregel Modus Ponens

$$\frac{X, \quad (X \text{ IMP } Y)}{Y}$$

D.h. der Kalkül arbeitet auf Mengen von Formeln, und fügt entsprechend der Schlußregel Modus ponens neue Formeln hinzu. Allerdings ist diese Schlußregel mit Instanziierung verbunden, so daß diese korrekter lauten sollte:

$$\frac{X_1, \quad (X_2 \text{ IMP } Y)}{\sigma(Y)} \quad \sigma \text{ ist allgemeinsten Unifikator von } X_1, X_2$$

Dabei stehen X_1, X_2, Y, Z jeweils für beliebige MINI-Formeln. Wir wollen zeigen, daß im MINI-Kalkül die Formel $(X \text{ IMP } X)$ für beliebige MINI-Formeln X ableitbar ist.

Um die Aussagen über MINI in PL_1 zu formulieren, benutzen wir ein PL_1 -Prädikatsymbol "ableitbar" und kodieren den Junktork als ein PL_1 -Funktionssymbol "imp" (in Präfixschreibweise). Dadurch werden MINI-Formeln als PL_1 -Terme dargestellt. Der MINI-Kalkül läßt sich dann folgendermaßen durch PL_1 -Formeln beschreiben:

$$F1: \forall x, y : \text{ ableitbar}(\text{imp}(x, \text{imp}(y, x)))$$

$$F2: \forall x, y, z : \text{ ableitbar}(\text{imp}(\text{imp}(x, \text{imp}(y, z)), \text{imp}(\text{imp}(x, y), \text{imp}(x, z))))$$

$$F3: \forall x, y : \text{ ableitbar}(x) \wedge \text{ ableitbar}(\text{imp}(x, y)) \Rightarrow \text{ ableitbar}(y)$$

Wir wollen zeigen, daß daraus folgende Formel folgt:

$$B : \forall x : \text{ableitbar}(\text{imp}(x, x))$$

Die Klauselform von $F1 \wedge F2 \wedge F3 \wedge \neg B$ ist eine Menge von vier Klauseln, wobei die Variable in $\neg B$ durch eine nullstellige Skolemfunktion, also eine Skolemkonstante c ersetzt ist:

$$C1 = \text{ableitbar}(\text{imp}(x_1, \text{imp}(y_1, x_1)))$$

$$C2 = \text{ableitbar}(\text{imp}(\text{imp}(x_2, \text{imp}(y_2, z_2)), \text{imp}(\text{imp}(x_2, y_2), \text{imp}(x_2, z_2))))$$

$$C3 = \neg \text{ableitbar}(x_3), \neg \text{ableitbar}(\text{imp}(x_3, y_3)), \text{ableitbar}(y_3)$$

$$B = \neg \text{ableitbar}(\text{imp}(c, c))$$

Im folgenden bezeichne C, n das n -te Literal einer Klausel C , das Kürzel $C, n + D, m \vdash R$ steht für eine Anwendung der Resolutionsregel mit den entsprechenden Resolutionsliteralen der Elternklauseln C und D und der Resolvente R . Die Variablen jeder Resolvente werden systematisch umbenannt, um Namenskonflikte zu vermeiden. Wir führen folgende Resolutionsableitung durch:

$$\begin{aligned}
& C1 + C3, 1\{x_3 \mapsto \text{imp}(x_1, \text{imp}(y_1, x_1))\} \\
& \quad \vdash R1' = \neg \text{ableitbar}(\text{imp}(\text{imp}(x_1, \text{imp}(y_1, x_1)), y_3)), \text{ableitbar}(y_3) \\
& \quad \rightarrow R1 = \neg \text{ableitbar}(\text{imp}(\text{imp}(x_4, \text{imp}(y_4, x_4)), z_4)), \text{ableitbar}(z_4) \\
& C2 + R1, 1\{z_4 \mapsto \text{imp}(\text{imp}(x_4, y_4), \text{imp}(x_4, x_4)), x_2 \mapsto x_4, y_2 \mapsto y_4, z_2 \mapsto x_4\} \\
& \quad \vdash R2' = \text{ableitbar}(\text{imp}(\text{imp}(x_4, y_4), \text{imp}(x_4, x_4))) \\
& \quad \rightarrow R2 = \text{ableitbar}(\text{imp}(\text{imp}(x_5, y_5), \text{imp}(x_5, x_5))) \\
& R2 + C3, 2\{x_3 \mapsto \text{imp}(x_5, y_5), y_3 \mapsto \text{imp}(x_5, x_5)\} \\
& \quad \vdash R3' = \neg \text{ableitbar}(\text{imp}(x_5, y_5)), \text{ableitbar}(\text{imp}(x_5, x_5)) \\
& \quad \rightarrow R3 = \neg \text{ableitbar}(\text{imp}(x_6, y_6)), \text{ableitbar}(\text{imp}(x_6, x_6)) \\
& C1 + R3, 1\{x_6 \mapsto x_1, y_6 \mapsto \text{imp}(y_1, x_1)\} \\
& \quad \vdash R4' = \text{ableitbar}(\text{imp}(x_1, x_1)) \\
& \quad \rightarrow R4 = \text{ableitbar}(\text{imp}(x_7, x_7)) \\
& B + R4 \quad \vdash \quad R5 = \square
\end{aligned}$$

6 Löschregeln: Subsumption, 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 enthalten, z.B. $\{P, \neg P, \dots\}$. 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 6.1. 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 6.2. 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} .

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:

Satz 6.3. *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 6.4. Betrachte die Klauselmenge

$$C1 : P(a)$$

$$C2 : P(b)$$

$$C3 : \neg Q(b)$$

$$C4 : \neg P(x), Q(x)$$

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 Subsumption

Definition 6.5. 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, wäre diese extra Schritte überflüssig.

Die entsprechende Reduktionsregel ist:

Definition 6.6. 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 6.7.

- 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 Subsumptionslöschregel 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 Klauseln 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 Subsumption 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 $\{\neg P(x), P(f(x))\}$. 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. In der Praxis macht das keine Schwierigkeiten, da man den Subsumptionstest auch unvollständig ausführen kann, 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 Subsumption zu einem Resolutionsbeweiser hinzufügen kann ohne daß die Widerlegungsvollständigkeit verlorengeht, zeigen wir zunächst ein Lemma.

Lemma 6.8. *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 variabelendisjunkt 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 R von 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 $D_R = \{L_1, \dots, L_m\} \cup D'_R$ 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.

□

Satz 6.9. *Der Resolutionskalkül zusammen mit der Löschung subsumierter Klauseln ist widerlegungsvollständig.*

Beweis. Induktion nach der Länge einer Resolutionsableitung mit Lemma 6.8 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 6.10. *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 4.16).

Die entsprechende Reduktionsregel ist:

Definition 6.11. 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.

Satz 6.12. *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 6.8 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:

Satz 6.13. *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 Subsumptionsregel 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 Subsumption 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 Subsumptionslöschung kann man dann sehen als Ersetzen der ursprünglichen Klausel durch den Faktor. Diese Operation wird auch *Subsumptionsresolution* genannt. Die Prozedur von Davis und Putnam (siehe Abschnitt 2.6) zum Entscheiden der Unerfüllbarkeit von aussagenlogische Klauselmengen kann man jetzt leicht aus Resolution, Subsumptionsregel, Isolationsregel und Fallunterscheidung zusammenbauen.

7 Einschränkung der Resolution

Der Resolutionskalkül hat durch die Verwendung allgemeinsten Unifikatoren den Vorteil, daß die Verzweigungsrate im Suchraum endlich ist, d.h. für jede Klauselmengemenge gibt es nur endlich viele, und i.a. nicht allzu viele Möglichkeiten, Resolventen abzuleiten. Trotzdem gibt es noch nicht “den super-Resolutionsbeweiser, der alle machbaren Verifikations- und mathematischen Probleme löst. Einer Grund ist, daß immer noch sehr viele offensichtliche Redundanzen im Kalkül stecken, die die Suche erschweren.

7.1 Set-of-Support

Die am meisten benutzte und oft sehr wirkungsvolle Restriktionsstrategie ist *Set-of-Support*. Dazu teilt man die Klauselmengemenge auf in die Klauseln, die aus den Voraussetzungen entstanden sind, und die Klauseln, die von der negierten Behauptung stammen. Man geht davon aus, daß die Voraussetzungen nicht selbst bereits widersprüchlich sind und deshalb ein Widerspruch nur unter Beteiligung der Behauptung erzielt werden kann. Die Restriktion verbietet deshalb die Erzeugung von Resolventen zwischen zwei Voraussetzungs-klauseln.

Wenn im Beispiel $\{P, Q\}, \{\neg P, Q\}, \{P, \neg Q\}, \{\neg P, \neg Q\}$ die Klausel $\{P, Q\}$ als Set-of-Support definiert ist, dann darf man nicht innerhalb der restlichen Klauselmengemenge resolvieren.

Im allgemeinen hat man die Axiome A_1, \dots, A_n und einen Satz der Form $B_1 \wedge \dots \wedge B_n \Rightarrow F$. Wenn B_i, F Literale sind, dann ist in der Klauselform nur $\neg F$ im set-of-support. Die Resolution mit SOS kann also nur damit anfangen zu schließen. Die Einschränkung ist am Anfang einer Beweissuche mittels Resolution sehr wirksam. Allerdings vergrößert sich die SOS-Menge nach einigen Resolutionsschritten, so daß die Wirkung nachläßt.

Etwas allgemeiner kann man die Klauselmengemenge, gegeben eine Struktur S in die Menge der Klauseln aufteilen, die von S erfüllt werden, und das Komplement. Ein Widerspruch kann nicht entstehen, wenn zwischen in S gültigen Klauseln resoliert wird.

7.2 UR-Resolution

Bei der UR-Resolution (englisch: unit resulting resolution) wird jeweils eine Klausel mit $n + 1$ Literalen, der sogenannte “Nukleus“, simultan mit n unären Klauseln resoliert, so daß eine neue unäre Klausel entsteht. Seien beispielsweise folgende Klauseln gegeben:

$$C1 \{ \neg P(x, y), \neg P(y, z), P(x, z) \}$$

$$C2 \{ P(a, b) \}$$

$$C3 \{ P(b, c) \}$$

Damit sind u.a. folgende Resolutionsschritte möglich:

$$C1, 1 + C2 \vdash R1 = \{\neg P(b, z), P(a, z)\}$$

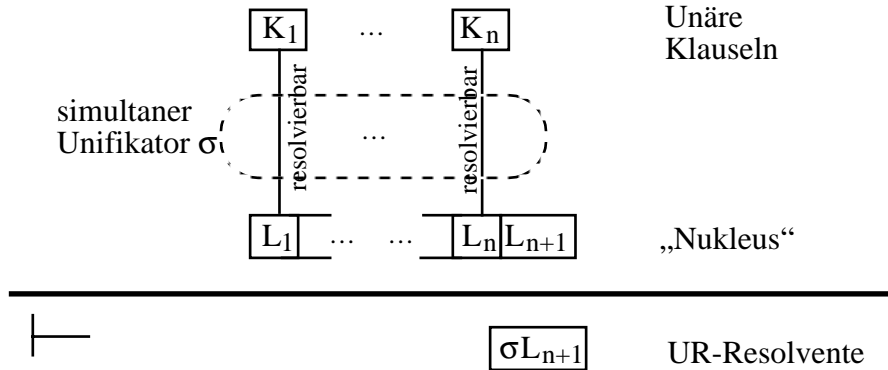
$$R1, 1 + C3 \vdash R2 = \{P(a, c)\}$$

Die zweite Resolvente entsteht aber auch mit einer anderen Ableitung, die sich nur durch eine unwesentliche Vertauschung der Reihenfolge der Schritte von der ersten unterscheidet:

$$C1, 2 + C3 \vdash \{R1' = \neg P(x, b), P(x, c)\}$$

$$R1, 1 + C2 \vdash R2 = \{P(a, c)\}$$

Die UR-Resolution faßt die beiden Schritte so zusammen, daß R2 unmittelbar abgeleitet wird und die Reihenfolge der Schritte keine Rolle mehr spielt. Das allgemeine Schema für UR-Resolution sieht in graphischer Darstellung folgendermaßen aus:



Diese Darstellung verdeutlicht, daß die unären Klauseln in der Ausgangssituation gleichberechtigt sind, und daß daher die Reihenfolge, in der sie bearbeitet werden, keine Auswirkung auf das endgültige Resultat haben sollte. Der gemeinsame Unifikator wird berechnet, indem man durch Aneinanderhängen der Termlisten der unären Klauseln und der Termlisten der Partnerlitterale im Nukleus zwei große Termlisten bildet und diese ganz normal unifiziert. Im Transitivitätsbeispiel von oben sind die beiden Termlisten (x, y, y, z) und (a, b, b, c) . Deren Unifikator ist $\{x \mapsto a, y \mapsto b, z \mapsto c\}$. Es gibt allerdings auch effizientere Methoden.

Die Wirkung der UR-Resolution ist nicht nur, daß von der Reihenfolge der n unären Resolutionsschritte abstrahiert wird. Obendrein werden die als Zwischenergebnisse anfallenden $n - 1$ Klauseln gar nicht erzeugt, im obigen Beispiel würden also $R1$ oder $R1'$ nicht in die Klauselmenge eingefügt. Da es dafür

zunächst keinen Grund gibt, jedenfalls nicht mit irgendeiner gängigen Reduktionsregel, entspricht die UR-Resolution genau genommen sogar einer ganz neuen Schlußregel, für die wieder dieselben Eigenschaften gezeigt werden müssen wie für die Resolutionsregel.

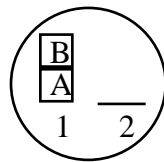
Die UR-Resolution ist korrekt, da sie immer das gleiche Ergebnis liefert wie eine entsprechende Folge von normalen Resolutionen. Sie ist nicht widerlegungsvollständig wie die folgende unerfüllbare Klauselmenge zeigt:

$$\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, \neg Q\}, \{\neg P, Q\}\}$$

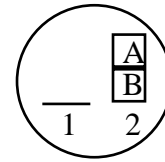
Diese Klauselmenge enthält überhaupt keine unären Klauseln und kann damit nicht mit UR-Resolution widerlegt werden. Für eine eingeschränkte Klasse von Klauseln ist sie jedoch widerlegungsvollständig, und zwar für die sogenannten unär widerlegbare Klauselmengen. Diese Klauselmengen kann man widerlegen, indem man die Resolution dahingehend einschränkt, daß man fordert, daß eine der beteiligten Elternklauseln immer eine unäre Klausel ist, d.h. aus einem Literal besteht. Es gibt (bis jetzt) keine geeignete syntaktische Charakterisierung dieser Eigenschaft. Eine wichtige Unterklasse der unär widerlegbaren Klauselmengen sind Mengen von *Hornklauseln*, die in dieser Vorlesung noch behandelt werden.

Beispiel 7.1. UR-Resolution zur Aktionsplanung Typische “Blockswelt“:

initial
state



goal
state



Gesucht ist eine Aktionsfolge, die den initialen Zustand in den Zielzustand transformiert.

Axiomatisierung des Problems:

Es wird verwendet:

- $at(z, x, l)$: im Zustand z ist Block x auf Position l .
- $on(z, x, y)$: im Zustand z ist Block x auf Block y .
- $cl(z, x)$: im Zustand z ist Block x frei.
- $M(x, l)$: Block x geht nach Position l .
- $n(x, y)$: Zustand nach Aktion y vom Zustand x aus.
- Konstanten: S : Startzustand, A, B : Bausteine.

Allgemeine Regel:

$$C1 : 1 \neq 2$$

Ausgangszustand:

$$C2 : at(S, A, 1)$$

$$C3 : at(S, B, 1)$$

$$C4 : cl(S, B)$$

$$C5 : on(S, B, A)$$

Axiomatisierung der Aktion $M(x, l)$:

$$C6 : \forall s, x, l : (cl(s, x) \Rightarrow at(n(s, M(x, l)), x, l))$$

$$C7 : \forall s, x, y, l_1, l_2 : (on(s, x, y) \wedge at(s, x, l_1) \wedge l_1 \neq l_2 \Rightarrow cl(n(s, M(x, l_2)), y))$$

$$C8 : \forall s, x, y, l : (at(s, x, l) \Rightarrow at(n(s, M(y, l)), x, l))$$

Ziel:

$$\exists z (at(z, A, 2) \wedge at(z, B, 2))$$

Klauselform:

$$C1 : 1 \neq 2$$

$$C2 : at(S, A, 1)$$

$$C3 : at(S, B, 1)$$

$$C4 : cl(S, B)$$

$$C5 : on(S, B, A)$$

$$C6 : \neg cl(s, x), at(n(s, M(x, l)), x, l)$$

$$C7 : \neg on(s, x, y), \neg at(s, x, l1), l1 = l2, cl(n(s, M(x, l)), y)$$

$$C8 : \neg at(s, x, l), at(n(s, M(y, l)), x, l)$$

$$Z : \neg at(z, A, 2), \neg at(z, B, 2)$$

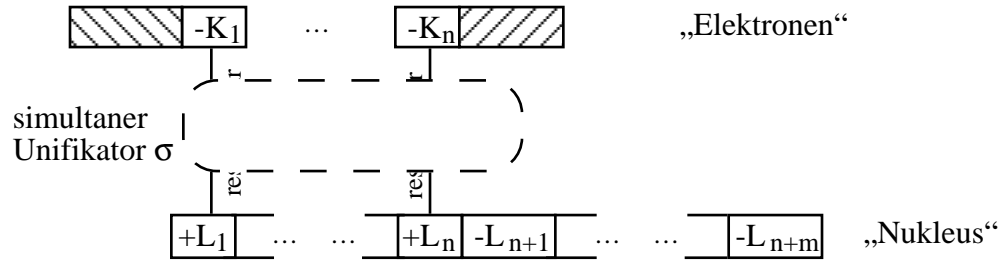
UR-Resolutionsfolge:

$$\begin{array}{lll}
C6 + C4 & \vdash R1 : at(n(S, M(B, l)), B, l) & \{s \mapsto S, x \mapsto B\} \\
C7 + C5, C2, C1 & \vdash R2 : cl(n(S, M(B, 2)), A) & \{s \mapsto S, x \mapsto B, y \mapsto A, l_1 \mapsto 1, l_2 \mapsto 2\} \\
C6 + R2 & \vdash R3 : at(n(n(S, M(B, 2)), M(A, l)), A, l) & \{s \mapsto n(S, M(B, 2)), x \mapsto B\} \\
C8 + R1 & \vdash R4 : at(n(n(S, M(B, l)), M(y, l)), B, l) & \{s \mapsto n(S, M(B, l)), x \mapsto B\} \\
Z + R3, R4 & \vdash R5 : \square & \{l \mapsto 2, z \mapsto n(n(S, M(B, 2)), M(A, 2))\}
\end{array}$$

Die gesuchte Aktionsfolge ist also $M(B, 2)M(A, 2)$.

7.3 Hyperresolution

Die Hyperresolution kann man als Verallgemeinerung der UR-Resolution ansehen. Sie wurde ebenfalls von John Alan Robinson entworfen und wird durch folgendes Schema beschrieben:



\vdash $\boxed{\sigma}$... $\boxed{\sigma}$ $\boxed{\sigma - L_{n+1}}$... $\boxed{\sigma - L_{n+m}}$ Hyperresolvente

Als „Nukleus“ dient eine Klausel, die mindestens ein positives Literal enthält. Solche Klauseln gibt es in unerfüllbaren Klauselmengen immer. Für jedes positive Literal des Nukleus benötigt man ein sogenanntes „Elektron“, eine Klausel mit nur negativen Literalen. Rein negative Klauseln kommen in unerfüllbaren Klauselmengen ebenfalls immer vor. Der Nukleus wird wieder simultan mit allen Elektronen resolviert, wodurch eine rein negative Klausel entsteht, die wiederum als Elektron im nächsten Hyperresolutionsschritt dienen kann. Die rein negativen Klauseln übernehmen hier also die gleiche Rolle wie die unären Klauseln in der UR-Resolution. Dual zu dieser sogenannten negativen Hyperresolution definiert man die positive Hyperresolution, bei der die Vorzeichen der Literale in Nukleus und Elektronen gerade vertauscht sind. Da im Normalfall eine negierte Behauptung nur negative Literale enthält und damit als Elektron für negative Hyperresolution verwendbar ist, eignet sich diese für Rückwärtsschließen von der Behauptung in Richtung Voraussetzungen, während positive Hyperresolution gerade umgekehrt von den Voraussetzungen in Richtung auf die Behauptung arbeiten kann. Beide Varianten der Hyperresolution (mit eingebauter Faktorisierung) sind widerlegungsvollständig für beliebige Klauselmengen.

7.4 Input-Resolution und Unäre Resolution

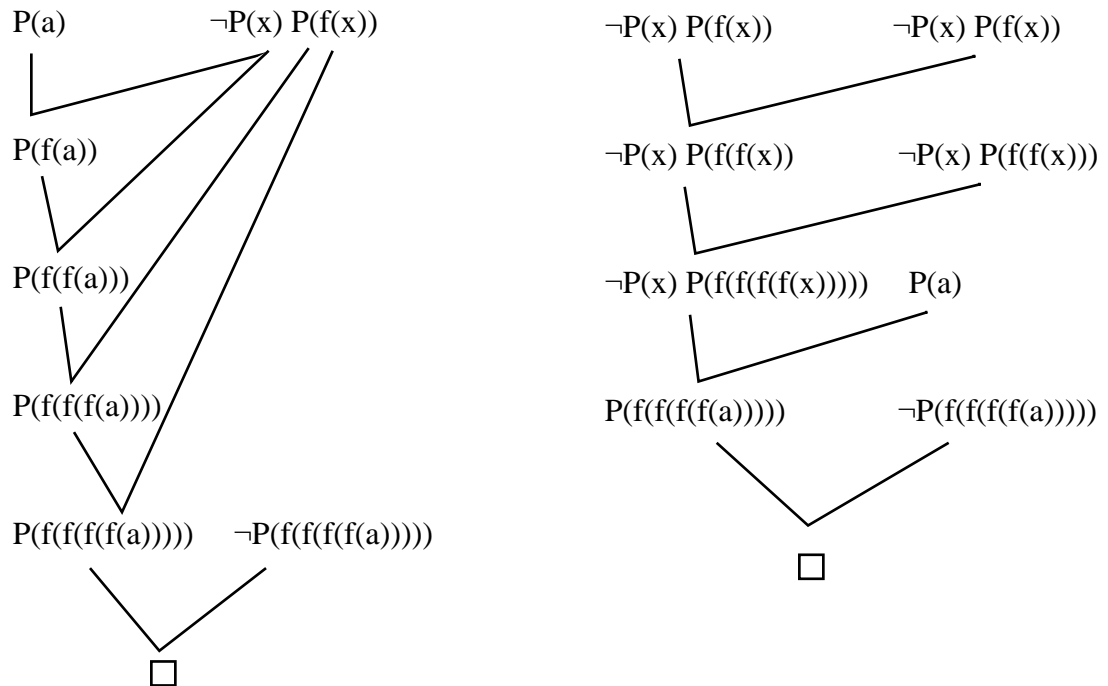
unäre Resolution (englisch *unit resolution*) verbietet die Erzeugung von Resolventen zwischen zwei Elternklauseln, wenn beide mehr als ein Literal enthalten. Positiv formuliert muß also jede Resolvente mindestens eine unäre Elternklausel besitzen. Faktorisierung ist ebenfalls erlaubt. Diese Restriktion schränkt die möglichen Nachfolgezustände einer Klauselmenge stark ein, führt außerdem stets zu Resolventen mit weniger Literalen als sie die längere Elternklausel besitzt. Sie ist die Basisvariante zur UR-Resolution.

Beispiel 7.2. (Unäre Resolution)

Wir widerlegen die Klauselmenge $\{\{P(a)\}, \{\neg P(x), P(f(x))\}, \{\neg P(f(f(f(f(a))))\}\},$ einmal mit unärer Resolution und dann ohne eine feste Strategie. Zur Darstellung benutzen wir einen sogenannten Widerlegungsbaum, bei dem jeder Knoten genau die beiden Elternklauseln als Vorgänger hat. Die Wurzel (diesmal wirklich unten) ist die leere Klausel.

Widerlegungsbäume

mit unärer Resolution mit allgemeiner Resolution



Wie man sieht, ist die unäre Widerlegung länger als die uneingeschränkte. Man kann das Beispiel leicht erweitern, so daß man sieht, daß die unäre Widerlegung exponentiell länger ist. Das bedeutet aber nicht, daß der Suchraum auch exponentiell größer ist, im Gegenteil: meist ist der Suchraum erheblich viel kleiner.

Die unäre Resolution ist im allgemeinen nicht widerlegungsvollständig, das heißt, die leere Klausel ist mit dieser Strategie nicht von allen unerfüllbaren Klauselmengen aus ableitbar. Immerhin ist die Widerlegungsvollständigkeit für eine wichtige Klasse von Klauselmengen gewährleistet, die die Klasse der Hornklauselmengen umfaßt und die man mangels einer syntaktischen Charakterisierung die *unär widerlegbare* Klasse nennt.

7.5 Eingabe-Resolution

Für die Klasse der unär widerlegbaren Klauselmengen ist auch eine andere wichtige Restriktionsstrategie widerlegungsvollständig, die *Eingaberresolution* (englisch *input resolution*). Diese verbietet die Erzeugung von Resolventen, deren

Elternklauseln beide Resolventen sind. Positiv formuliert muß also jede Resolvente wenigstens eine Elternklausel aus der initialen Klauselmenge besitzen. Der wesentliche Vorteil dieser Restriktion besteht daran, daß von allen möglichen Resolutionsschritten ein Resolutionsliteral a priori bekannt ist. Faktorisierung ist ebenfalls erlaubt. Insbesondere treten damit nur Unifikationen zwischen jeweils einem beliebigen und einem a priori bekannten Term auf, so daß man für jeden dieser bekannten Terme aus den initialen Klauseln einen speziellen Unifikationsalgorithmus "kompilieren" kann. Dieser ist in der Regel wesentlich effizienter als einer, der zwei beliebige Terme unifizieren können muß.

Es gilt: Eine Klauselmenge ist genau dann mit unärer Resolution widerlegbar, wenn sie mit Inputresolution widerlegbar ist. In diesem Sinne sind beide Strategien gleichwertig.

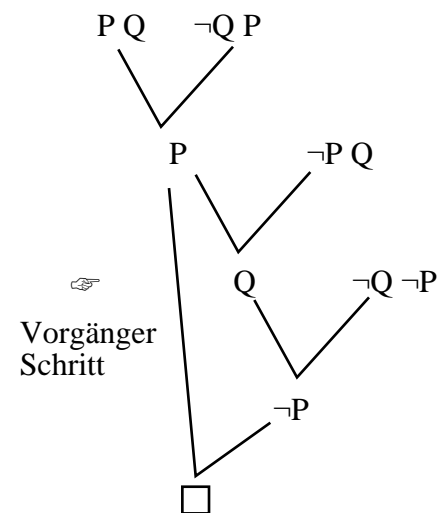
7.6 Linear Resolution

Es gibt eine Reihe von Restriktionsstrategien, die die Grundidee der Eingaberesolution möglichst weitgehend erhalten wollen, aber widerlegungsvollständig für beliebige Klauselmengen sind. Die meisten basieren auf der *linearen Resolution*. Man arbeitet immer mit einer aktuellen Klausel, der Zentralklausel. Resolution mit Eingabe-Klauseln ist erlaubt, ebenso Faktorisierung, wobei diese auch implizit sein kann. Außerdem werden Resolutionsschritte zwischen zwei Resolventen in den Fällen zugelassen, in denen eine ein "Vorgänger" der anderen ist.

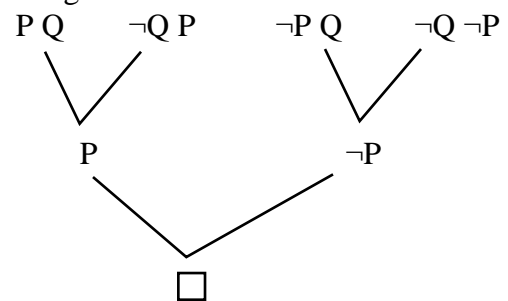
Beispiel 7.3. Lineare Resolution

Wir widerlegen die Klauselmenge $\{P, Q\}, \{\neg P, Q\}, \{P, \neg Q\}, \{\neg P, \neg Q\}$, einmal mit linearer Resolution und einmal mit allgemeiner Resolution:

lineare Resolution



allgemeine Resolution



Beispiel 7.4. Folgendes Beispiel zeigt, daß man Faktorisierung braucht. Die Klauselmengse sei $\{\{P(x), P(y)\}, \{\neg P(x), \neg P(y)\}\}$. Lineare Resolution ohne Faktorisierung ergibt immer eine Zentralklausel der Länge zwei, wobei auch keine automatische Verschmelzung auftritt.

Durch Verschärfungen dieser Bedingung sowie durch Beschränkungen für die als Resolutionslitterale zulässigen Litterale einer Klausel ergeben sich verschiedene Varianten der linearen Resolution, zum Beispiel die sogenannte *SL-Resolution*.

7.7 SL-Resolution

SL-Resolution ist eine Restriktionsstrategie, die i.a. gekoppelt ist mit einer Ordnungsstrategie [?]. Die Bedeutung der SL-Resolution liegt darin, daß sie eine prozedurale “goal/subgoal“-Sichtweise der Resolution erlaubt. SL-Resolution ist die Strategie, die der logischen Programmiersprache **PROLOG** zugrundeliegt. Im folgenden wird die SL-Resolution für Horn Klauseln behandelt. Im Falle der Hornklauselmengen ist die Faktorisierung nicht notwendig.

7.8 SL-Resolution für Horn Klauseln

Angenommen, wir haben eine Menge von initialen Horn Klauseln und eine “Zielklausel“, d.h. ein negiertes Theorem ohne Kopfliteral. Beginnend mit der Zielklausel als “Zentralklausel“ wählt SL-Resolution in jedem Schritt die letzte Resolvente als neue Zentralklausel und bestimmt die zulässigen Resolutionsschritte wie folgt: Wenn die aktuelle Zentralklausel $C = C_1 \vee C_2$ ist, wobei C_2 der Block von Litteralen ist, der von der letzten “Seitenklausel“ abstammt, wird aus C_2 ein Literal L ausgewählt und dafür eine Resolvente mit einer passenden Klausel aus der initialen Klauselmengse generiert. Daher resolviert SL-Resolution nur zwischen der letzten Resolvente und den Eingabeklauseln. Darüberhinaus werden diejenigen Litterale zuerst wegresolviert, die als letzte in die Klausel hinein kamen. Die Auswahl des Litterals in der Zentralklausel durch eine *Selektionsfunktion* ist deterministisch. Dem liegt die Erkenntnis zugrunde, daß die Reihenfolge, in der man Litterale einer Klausel “wegresolviert“ unerheblich ist – wenn es in einer Reihenfolge nicht geht, geht es auch in keiner anderen. Die Auswahl des Resolutionspartners ist dagegen nichtdeterministisch: eine erfolgreiche Resolutionskette ist ausreichend.

Beispiel 7.5. SL-Resolution mit Horn Klauseln

Prolog Notation:	Klausel Notation:
C1: $A(x, y) \Leftarrow P(x, y)$	$A(x, y) \vee \neg P(x, y)$
C2: $A(x, z) \Leftarrow P(x, y) \wedge A(y, z)$	$A(x, z) \vee \neg P(x, y) \vee \neg A(y, z)$
C3: $P(a, b)$	
C4: $P(b, c)$	
C5: $P(c, d)$	

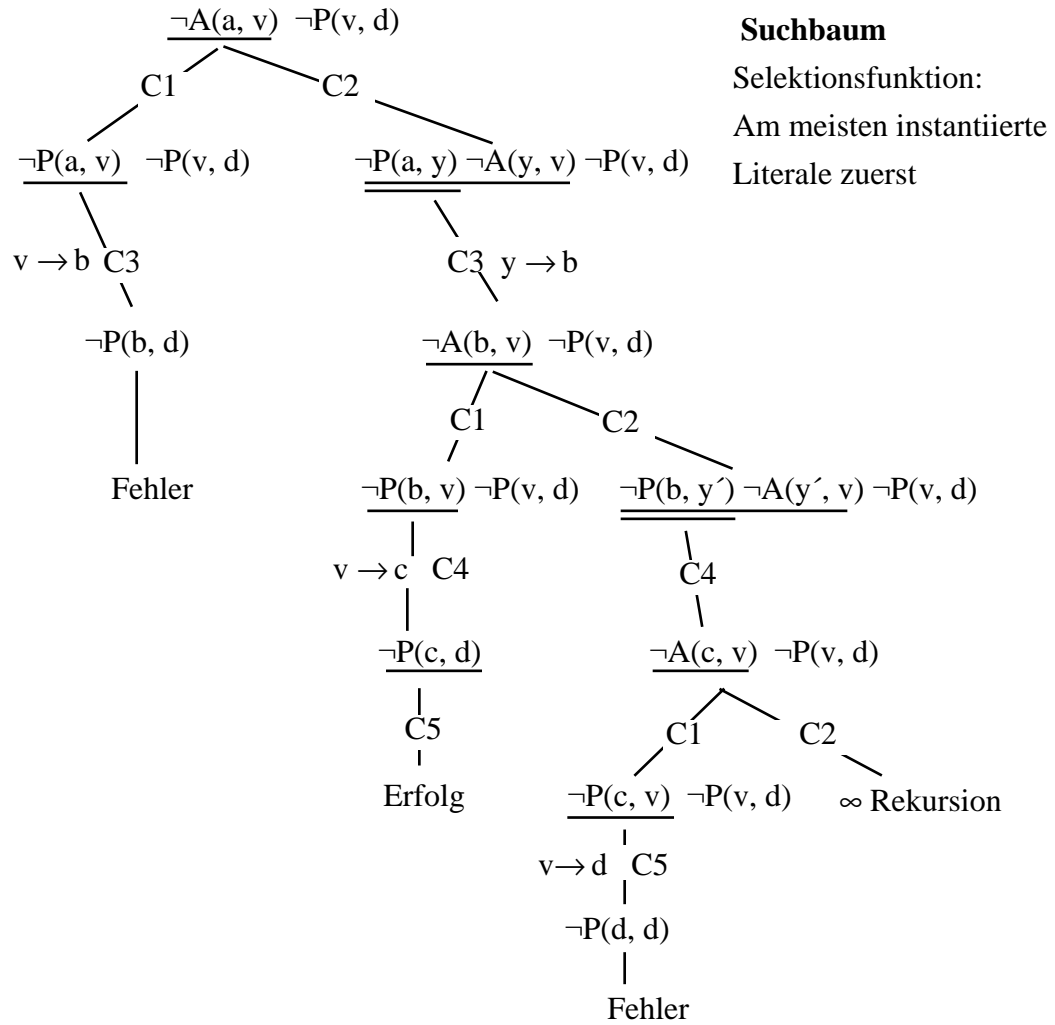
Theorem: $\exists v : A(a, v) \wedge P(v, d)?$

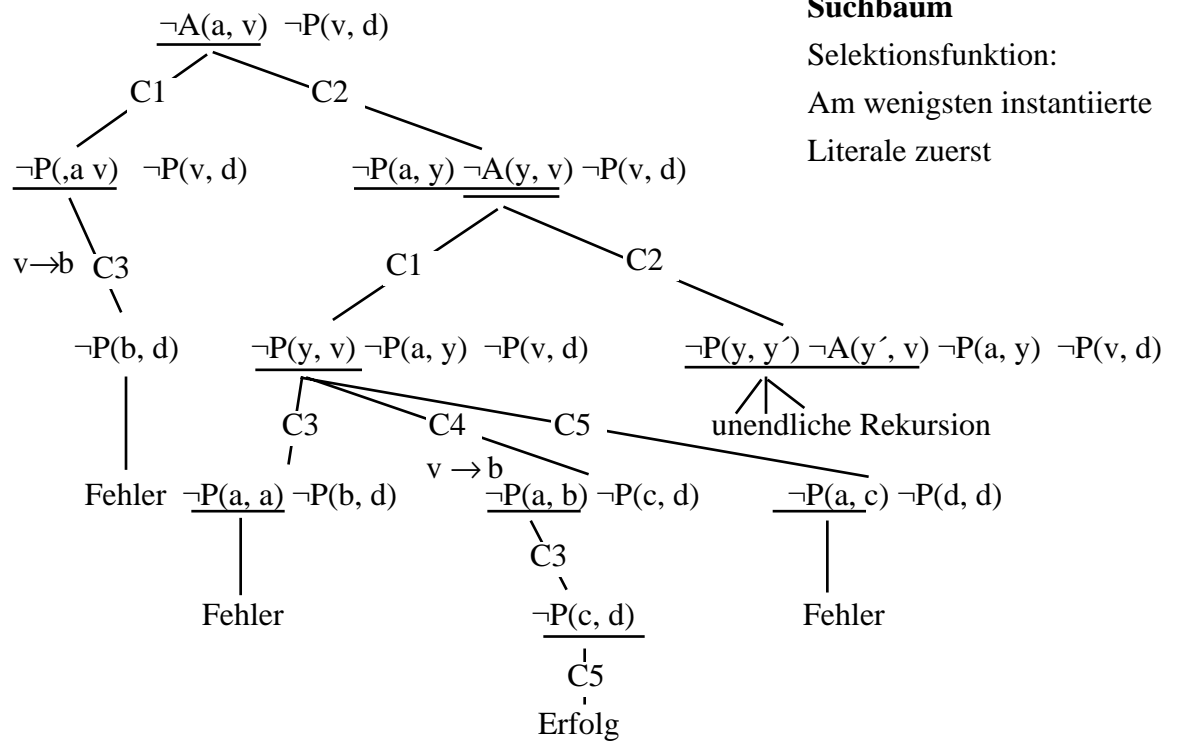
Negiertes Theorem: $\neg A(a, v) \vee \neg P(v, d)$

In den beiden folgenden Bildern wird der Suchraum für dieses Beispiel und für zwei verschiedene Selektionsfunktionen dargestellt. Die Selektionsfunktionen sind:

1. Das am stärksten instantiierte Literal zuerst
2. Das am wenigsten instantiierte Literal zuerst.

Die zuletzt eingeführten Literale, die zuerst wegresolviert werden, sind unterstrichen. Da SL-Resolution einen normalen Suchbaum generiert, können die normalen Suchalgorithmen angewendet werden [?]. Zum Beispiel Tiefensuche, wie sie in **Prolog** angewendet wird, hat den Vorteil, daß nur ein Ast zu einer Zeit entwickelt wird. Backtracking zu früheren Zuständen kann leicht durch Stacks implementiert werden. Tiefensuche ist jedoch nicht vollständig wenn der Suchraum unendliche Äste enthält. Bei Breitensuche und bei heuristischer Suche muß man dagegen alle offenen Knoten abspeichern, was sehr aufwendig sein kann.





8 Grundlagen der logischen Programmierung: deklarative Programmierung

Logisches Programmieren (z.B. Prolog) hat als theoretische Basis die Prädikatenlogik erster Stufe, und als Ausführung der Programme eine Variante der Resolution.

Programme in logischen Programmierens verwenden spezielle Klauseln, sogenannte Horn-Klauseln.

Definition 8.1.

- Eine Hornklausel ist eine Klausel mit maximal einem positiven Literal⁷.
- Eine definite Klausel ist eine Klausel mit genau einem positiven Literal. D.h. die Klausel ist von der Form: $\neg A \vee B_1 \vee \dots \vee B_m$ oder in der in der logischen Programmierung verwendeten Notation:

$$A \Leftarrow B_1, \dots, B_m \quad \text{bzw.} \quad A :- B_1, \dots, B_m$$

Man nennt A den Kopf, $\{B_1, \dots, B_m\}$ den Rumpf der Klausel.

- Eine Klausel ohne positive Literale nennt man definites Ziel (Anfrage, Query, goal). Notation:

$$\Leftarrow B_1, \dots, B_n$$

Die B_i werden Unterziele (Subgoals) genannt.

- Eine Unit-Klausel (oder Fakt) ist eine definite Klausel mit leerem Rumpf. Notation: $A \Leftarrow$.
- 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.

Ein definites Ziel kann man in PL_1 so hinschreiben:

$$\forall x_1, \dots, x_n : \neg B_1 \vee \dots \vee \neg B_n$$

Dies ist äquivalent zu:

$$\neg \exists x_1, \dots, x_n : B_1 \wedge \dots \wedge B_n$$

D.h. daß man eine Anfrage als eine existentiell quantifizierte Aussage sehen kann:

$$\exists x_1, \dots, x_n : B_1 \wedge \dots \wedge B_n$$

deren Negation zum Programm hinzugefügt wird. Die Herleitung besteht in der Erzeugung eines Widerspruchs (der leeren Klausel \square). Die Herleitung des Widerspruchs kann bei Hornklauseln immer so gemacht werden, daß man die existierenden Objekte erzeugt.

⁷ Ein positives Literal ist ein Atom; ein negatives Literal ist ein negiertes Atom

8.1 SLD-Resolution

SLD-Resolution ist die prozedurale Semantik von definiten Programmen. D.h. SLD-Resolution definiert die Ausführung von logischen Programmen:

S: selection function
L: lineare Resolution
D: definite clauses

Zunächst erklären wir die SLD-Resolution in ihrer allgemeinen (nicht-deterministischen) Form:

Definition 8.2. Sei $G = \leftarrow A_1, \dots, A_m, \dots, A_k$ ein definites Ziel und $C = A \leftarrow B_1, \dots, B_q$ eine definite Klausel. Dann kann man aus G und C ein neues Ziel G' wie folgt mit Resolution herleiten:

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

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

Definition 8.3. Sei P ein definites Programm und G ein definites Ziel. Eine SLD-Ableitung von $P \cup \{G\}$ ist eine Folge

$$G \rightarrow_{C_1, m_1} G_1 \rightarrow_{C_2, m_2} G_2 \dots$$

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 8.4. 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 Einschränkung von $\theta_n \circ \dots \circ \theta_1$ auf die Variablen von G erhält, wobei $\theta_1 \dots \theta_n$ die allgemeinsten Unifikatoren (in dieser Reihenfolge) zu einer SLD-Widerlegung von G sind.

Satz 8.5. Soundness der SLD-Resolution

Sei P ein definites Programm und G ein definites Ziel. Dann ist jede berechnete Antwort θ auch korrekt. D.h. $P \models \theta(\neg G)$

8.2 Vollständigkeit der SLD-Resolution

Wir werden verschiedene Vollständigkeitsaussagen für die SLD-Resolution formulieren.

Satz 8.6. (*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.

Satz 8.7. (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 $x \in FV(G)$: $\gamma\sigma(x) = \theta(x)$.

8.3 Strategien zur Berechnung von Antworten

Definition 8.8. Der folgende Algorithmus berechnet alle Antworten mit einer breadth-first Strategie:

1. Gegeben ein Ziel $\Leftarrow A_1, \dots, A_n$:
 - (a) Probiere alle Möglichkeiten aus, ein Unterziel A aus A_1, \dots, 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 , Instanzieren 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.

Aussage 8.9. 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: die Suchstrategie, die irgendein Atom auswählt, dann alle Klauseln durchprobiert, usw. ist vollständig bzgl der Antworten.

D.h.man kann 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. Dies betrifft nicht die *Reihenfolge, in der Klauseln ausgewählt werden*, diese ist nach wie vor nichtdeterministisch.

8.4 Implementierung logischer Programmiersprachen: Prolog

In Implementierung wird i.a. die Suche durch verschiedene Vereinfachungen und Festlegungen deterministisch gemacht:

- Die Anfrage wird als geordnete Liste 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.

Weitere Veränderungen gegenüber der theoretischen Fundierung sind:

- Der occurs-check wird i.a. nicht durchgeführt.
- Es gibt extra Operatoren zum Beeinflussen des Backtracking (*Cut*), der u.a. bewirken kann daßfür ein Unterziel statt vielen Lösungen maximal eine berechnet wird.
- Assert/Retract: Damit können Programmklauseln zum Programm hinzugefügt bzw. gelöscht werden.
- 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, daßdas jeweilige Argument nur als Eingabe bzw. nur als Ausgabe verwendet wird. z.B.
- 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.

In Implementierungen Folgende Tabelle vergleicht die theoretischen Eigenschaften mit denen implementierter logischer Programmiersprachen.

Pures Prolog	nichtpures Prolog	nichtpur, ohne occurs-check
Definite Programme	Cut, Negation, Klauselreihenfolge fest	
SLD: ist vollständig	SLD: unvollständig	SLD: i.a. nicht korrekt

Gibt man eine Reihenfolge der Klauseln vor, wie in einer Prolog-Implementierung, dann ist die SLD-Resolution nicht mehr vollständig: Manche Lösungen werden gefunden und aufgezählt, aber:

- es kann eine Schleife auftreten ohne eine weitere Antwort, obwohl es noch welche gibt.
- Es werden unendlich viele Antworten aufgezählt, aber nicht alle korrekten Antworten werden abgedeckt.

Es folgt auch, daß die Reihenfolge der Literale im Rumpf keinen Einfluß auf die Vollständigkeit hat.

Ein Beispiel für die Unvollständigkeit der festgelegten Reihenfolge der Klauseln:

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

Die symmetrische-transitive Hülle kann nicht berechnet werden, da das Programm in Schleifen gerät. Aber eine SLD-Widerlegung existiert, da diese nicht-deterministisch vorgehen darf, und insbesondere sich eine Programmklausel aussuchen darf.

8.5 SLD-Bäume

Aufgrund der obigen Vollständigkeitsaussage können wir SLD-Bäume definieren, die den ganzen Suchraum repräsentieren für ein Ziel, gegeben ein definites Programm.

Definition 8.10. *Gegeben ein definites Programm P , und ein definites Ziel G : Ein SLD-Baum für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:*

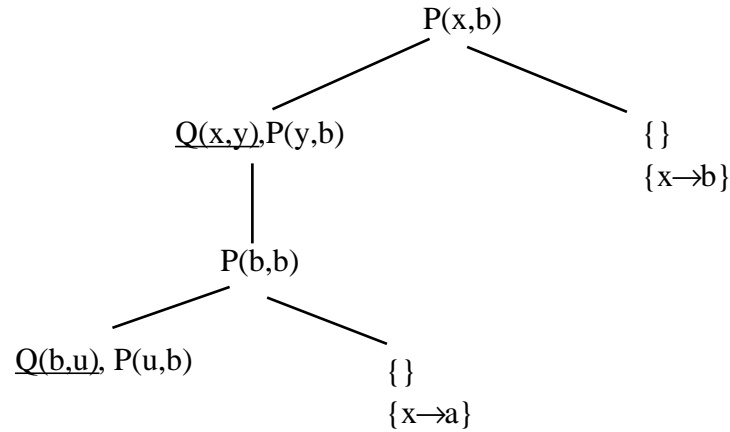
1. *Jeder Knoten ist markiert mit einem definiten Ziel*
2. *die Wurzel ist markiert mit G*
3. *In jedem Knoten mit nichtleerem Ziel wird ein Atom A des Ziels ausgewählt. die Söhne dieses Knoten sind dann die möglichen Ziele nach genau einem SLD-Resolutionsschritt mit einer Regel in P .*
4. *Knoten, die 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.

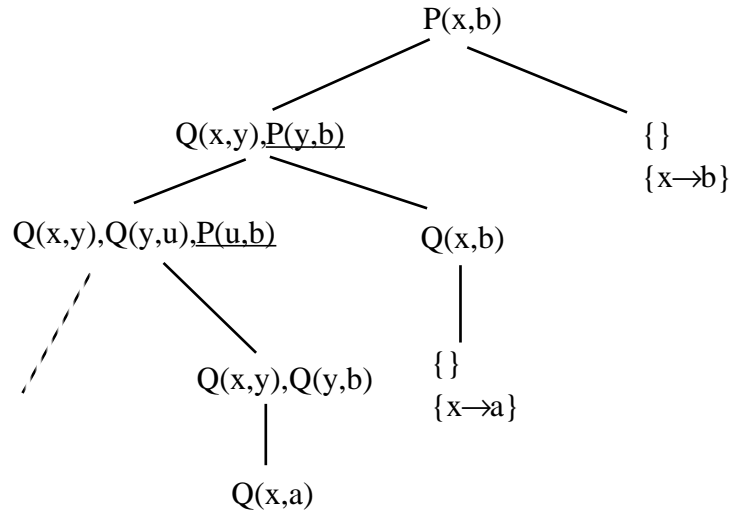
Äste entsprechen SLD-Ableitungen. Erfolgreiche Widerlegungen sind Erfolgspfade, unendliche SLD-Ableitungen entsprechen unendliche Pfaden. Fehlschläge entsprechen Pfaden zu Blättern, die mit einem nichtleeren Ziel markiert sind.

Beispiel 8.11. Programm:

```
P(X,Z) :- Q(X,Y), P(Y,Z).
P(X,X).
Q(a,b).
```



Nimmt man eine andere Auswahl, dann kann der Baum unendlich werden:



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

8.6 Beispielprogramme und Ausführung

Programm 1:

```
likes (maria, essen).
likes(maria,wein).
likes(peter, maria).
likes(peter,wein).
```

Einfache Anfragen:

```
>    ?- likes(peter,maria)
      yes
```

Konjunktive Anfragen (mit logischem und verknüpfte)

```
>    ?- likes(peter,maria), likes(maria,peter)
      no
```

Zuerst wird das erste Unterziel bearbeitet: Antwort: ja, Dann zweite: nein.
Konjunktive Anfragen mit Instanziierung.

```
>    ?- likes(peter, X), likes(maria,X)
```

Abarbeitung: 1. X = maria, likes(maria,maria): no

2. X = wein: likes(maria,wein). yes

X = wein

Backtracking ist notwendig (Zurücksetzen) zum Finden weiterer Lösungen.

Programm 2:

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

vorfahr(X,Y) :- vater(X,Y).
vorfahr(X,Y) :- mutter(X,Y).
vorfahr(X,Z) :- vorfahr(X,Y),vorfahr(Y,Z).
```

Anfragen:

```
?- vorfahr(elisabeth,maria)
```

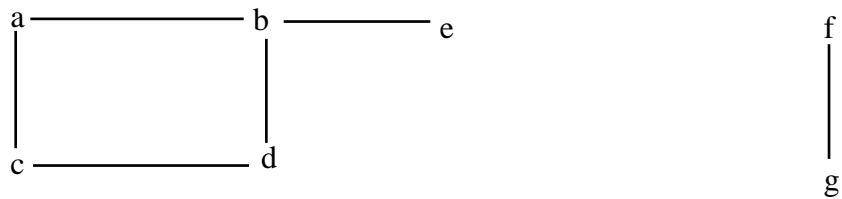
Abarbeitung :

```

1. vorfahr(elisabeth,maria)
  1.1 vater(elisabeth,maria). nein
  1.2 mutter(elisabeth,maria): nein
  1.3 vorfahr(elisabeth, X0), vorfahr(X0,maria)
    1.3.1 vater(elisabeth, X0), vorfahr(X0,maria): nein
    1.3.2 mutter(elisabeth, X0), vorfahr(X0,maria): X0 = peter
      1.3.2 vorfahr(peter,maria):
        1.3.2.1 vater(peter,maria): ja
    usw.

```

Beispiel 8.12. ungerichteter Graph.



```

kante(a,b).
kante(a,c).
kante(c,d).
kante(d,b).
kante(b,e).
kante(f,g).

```

```

verbunden(X,Y) :- kante(X,Y).
verbunden(X,Y) :- kante(Y,X).
verbunden(X,Z) := verbunden(X,Y),verbunden(Y,Z).

```

?- verbunden(c,e)

```

    verbunden(c,Y1), verbunden(Y1,e)
...
    verbunden(c,a), verbunden(a,e)

    verbunden(a,Y2), verbunden(Y2,e)

    verbunden(a,b), verbunden(b,e)
yes.

```

Übungsaufgabe 8.13. was passiert bei folgender Eingabe?

?- verbunden(e,f)

8.7 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 beginnen, oder mit einem Unterstrich : `_`. Der Unterstrich alleine `_` ist der Name einer anonymen Variablen (wildcard, joker, Leerstelle)

zusammengesetzte Terme (komplexe Terme) haben die Form

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

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))`.

8.8 Darstellung arithmetischer Ausdrücke:

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

Ausdrücke der Form `a * b + c` sind 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 in Prolog bekannt. Der Zahlbereich ist abhängig von der Implementierung.

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.

arithmetische Vergleiche:

`=`, `/=`, `<`, `>`, `=<`, `>=`, `=\=`, `:=`

sind möglich zwischen instantiierten Werten.

D.h.: wenn `expl op exp2` ausgewertet wird, muss `expl` und `exp2` zu einer Zahl auswertbar sein. Beachte:

`3 + 4 = 7` ergibt: no

`3 + 4 := 7` ergibt yes

D.h., es gibt einen Unterschied zwischen Term und Wert.

Will man einer Variablen einen Wert zuweisen, so benötigt man das “is” Prädikat:

$X \text{ is } 3 + 5$ danach: $X \mapsto 8$

$X = 3 + 5$ danach $X \mapsto “3 + 5”$

Relationale Addition und Multiplikation kann man verwenden bzw. definieren:

`times(X,Y,Z)`
`plus(X,Y,Z)`

Diese Prädikate kann man vorw—ärts und rückwärts verwenden.

?- `plus(X,Y,5)`
 $X = 1, Y = 4$
 $X = 2, Y = 3$
 \dots

?- `plus(2,3,X)`

$X = 5$

8.9 Datenstrukturen und Unifikation in Prolog

Intuition: Die Bäume werden gleichgemacht durch Instantiierung der Variablen.

$$k(X, a) = k(b, Y) \text{ ergibt } X = b, Y = a$$

Auch Unterstrukturen können Wert von Variablen sein:

$$k(X, h(a, Z)) = k(b, Y) \text{ ergibt } X = b, Y = h(a, Z)$$

Etwas schwierigere Fälle sind:

$$k(X, h(a, Z), X) = k(h(Y, b), U, U)$$

ergibt zunächst:

$$X = h(Y, b), U = h(a, Z), X = U$$

Endergebnis:

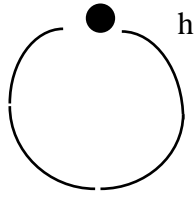
$$X = h(a, b), U = h(a, b), Y = a, Z = b$$

Behandlung des **occurs-check**:

$$k(X, X) = k(Y, h(Y))$$

erfordert Unifikation von $X = h(X)$.

Mit unendlichen Bäume als Termen ist die Unifikation möglich: $X = h(X)$ kann gleichgemacht werden durch: $X = h(h(h(\dots)))$. Dieses Objekt kann als gerichteter Graph dargestellt werden:



Prolog-Implementierungen vermeiden den occurs-check. Teilweise wird er durch statische Analyse eliminiert. Bzw. wird ganz abgeschaltet. In diesem Falle nimmt man in Kauf, daß das Programm bzgl der PL_1 -Semantik inkorrekt wird.

8.10 Listen:

Diese können in logischen Programmiersprachen direkt dargestellt werden. Man benutzt im allgemeinen eine abkürzende Syntax:

$[1, 2, 3]$ entspricht `cons(1, cons(2, cons(3, NIL)))`

`[]` entspricht `NIL`

Beispiel für den Element-Test:

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

Terminiert `member` in allen möglichen Situationen?

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

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.

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


Diese Funktion terminiert ebenfalls für Listen ohne Variablen aber nicht für `islist(X)`.

Eine Umstellung der Klauseln ergibt:

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

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

Antwort: $X = []$, falls die Eingabe eine Liste ist. Antwort ist dann ja.

```
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.
```

Die Anfrage

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

terminiert in manchen Implementierungen nicht: Wenn `is` auch rückwärts funktioniert, dann ergibt diese Anfrage als Antwort $X = [_1, _2]$.

Beispiel sortierte Listen von Zahlen:

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

Programmierung von `append`: in Prolog:

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

8.11 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 $[?, ?, ?]$. Dies ist auch historisch gesehen die erste Anwendung gewesen. Prolog verwendet als Suchstrategie Tiefensuche mit Backtracking. Dies kann direkt für einen rekursiv absteigenden Parser

verwendet werden, wobei man nicht nur kontextfreie Grammatiken verwenden kann, sondern auch erweiterte Grammatiken, z.B. für (schriftliche) Eingabe in natürlicher Sprache. Diese haben als Erweiterung, daß bestimmte Attribute der Terminale testbar sind, z.B. Geschlecht, Fall, Zeit, usw. und die Akzeptanz einer Phrase steuern können.

Kontextfreie Grammatiken für einfache englische Sprache

Es folgt eine Tabelle der *syntaktischen Kategorien*:

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	

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 \rightarrow NP VP

NP \rightarrow Det N OptRel

NP \rightarrow PN

Optrel $\rightarrow \varepsilon$

Optrel \rightarrow that VP

VP \rightarrow TV NP

VP \rightarrow IV

Lexikoneinträge:

PN \rightarrow terry

PN \rightarrow shrdlu

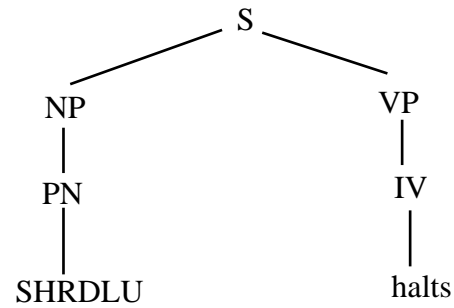
Det \rightarrow a

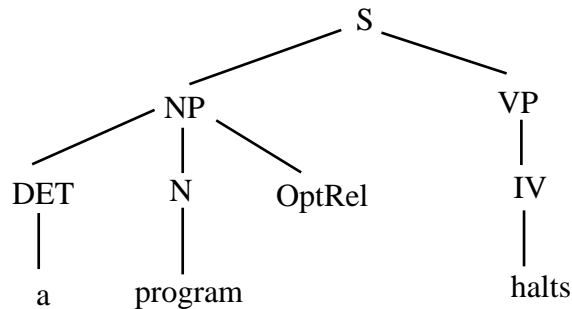
N \rightarrow program

IV \rightarrow halts

TV \rightarrow writes

Beispiel 8.14.





Der Anfang des Prologprogramms, das eine als Liste von Worten gegebenen Satz parst (erkennt), sieht so aus:

```

istsatz(Ein)      :- s(Ein, []).
s(Ein, Rest)      :- np(Ein, Rest1), vp(Rest1, Rest).
np(Ein, Rest)     :- det(Ein, Rest1), n(Rest1, Rest2), optrel(Rest2, Rest).
np(Ein, Rest)     :- pn(Ein, Rest).
optrel(Ein, Rest) :- Ein = Rest.
optrel(Ein, Rest) :- terminal(that, Ein, Rest1), vp(Rest1, Rest).
...
terminal(Wort, Ein, Rest1) :- Ein = [Wort | Rest1],
%% oder:      terminal(Wort, [Wort | Rest1], Rest1) .

```

Man erkennt, daß die Grammatik eins-zu-eins übersetzt wird, wobei nur die Behandlung der Eingabe- und Restliste hinzukommt. Diese Methode nennt man auch Verwendung von Differenzlisten, wobei man manchmal auch das Paar $(Ein, Rest)$ als Datentyp "Differenzliste" beschreibt.

Dieser Parser antwortet nur mit ja/nein bzw. terminiert nicht.

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.

8.12 Definite Clause Grammars

Definite Clause Grammars (DCG's) sind eine 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:

```

s(Number)  --> 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].

```

Übersetzt man in der gleichen Weise wie oben nach Prolog, so ergibt sich:

```

s(P0,P, Number)    :- 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).

```

Benutzung von Differenzlisten

Verwendet man das Minuszeichen als Konstruktor für Differenzlisten, so ergibt sich:

```

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 Definition für `terminal` ist:

```
terminal(X, [X|P] - 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 Attribute⁸ 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 Verbmarlierungen im Lexikon. Auch die Zeit oder (Aktiv/Passiv) kann man hinzufügen.

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 steckt, 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 (semantics) können dort enthalten sein.

Beispiel 8.15.

```
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, ...
```

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 ---> []
```

⁸ Attribute werden in der Computerlinguistik auch features genannt; Die zugehörigen Grammatiken Unifikationsgrammatiken

```
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(kein_relativesatz,...).
optrel(relativ_satz(TR_iv),...) :- iv(TR_iv,...)
```

Hier sollte z.B. der Satz “Shrdlu halts“ als Ausgabe den Syntaxbaum $s(np(pn(shrdlu)), vp(iv(halts)))$ erzeugen.

DCG's als formales System

DCG's 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 attributierte 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 Erweiterung der CFGs. Man kann aber mit DCG's über kontextfrei hinausgehen:

DCG-Grammatik für die nicht-kontextfreie formale Sprache $\{a^n b^n c^n d^n \mid n \in \mathbb{N}\}$

```
S          --->  A(x)B(x)C(x)D(x)
A(s(x))    --->  a A(x)
A(0)        --->  a
B(s(x))    --->  b B(x)
B(0)        --->  b
C(s(x))    --->  c C(x)
C(0)        --->  c
D(s(x))    --->  d D(x)
D(0)        --->  d
```


Bemerkungen zur Verarbeitung natürlicher Sprache

Eine natürliche Sprache ist i.a. so aufgebaut, daß Mehrdeutigkeiten vom Sprecher (Schreiber) vermieden werden können. D.h. daß es für einen Satz möglichst nur einen Parsebaum gibt. Bei mehr als einem Parse für einen Satz wird der gemeinte Inhalt nicht eindeutig erkannt. Z.B.

“Er sah das Mädchen mit dem Fernglas“

D.h. nicht, daß die Grammatik eindeutig sein muß; die Bildung mehrdeutige Sätze sind möglich. Allerdings werden bei der Kommunikation alle “Tricks“ verwendet um Aussagen eindeutig zu machen: (Inhaltliche, Kontext der Aussage, vorhergehende Unterhaltung bzw. Text; bei gesprochener Sprache: Betonung, bei Sichtkontakt: Gesten usw.)

Es gibt Sprachen (Latein; auch Deutsch zählt dazu), Englisch eher nicht), in denen man Satzteile (Phrasen) teilweise umsortieren kann, ohne die Bedeutung des Satzes zu ändern. Dies liegt oft an den Endungen, die die Rolle der Konstituenten deutlich machen:

the man bites the dog

the dog bites the man

Der Hund beißt den Mann

den Mann beißt der Hund

9 Tableau Kalkül für PL_1

Den aussagenlogischen Tableaunkalkül kann man auf PL1 erweitern (siehe [Fit90]). Zu den α (konjunktiven) und β -Formeln (disjunktiven) kommen noch die Quantorformeltypen γ und δ . Bei den α - und β -Formeltypen brauchte man eine Definition der direkten Unterformeln α_1, α_2 (bzw. β_1, β_2). Bei quantifizierten Formeln benötigt man Instanzen der Formeln.

All-quantor		Ex-quantor	
γ	$\gamma(t)$	δ	$\delta(t)$
$\forall x : \Phi$	$\Phi[t/x]$	$\exists x : \Phi$	$\Phi[t/x]$
$\neg \exists x : \Phi$	$\neg \Phi[t/x]$	$\neg \forall x : \Phi$	$\neg \Phi[t/x]$

9.1 Tableau mit Grundtermen

Zunächst eine Variante des Tableau-kalküls, der mit Grundtermen arbeitet.

Im Laufe eines Beweises ist es bei dieser Variante notwendig, neue Konstanten einzuführen. Diese werden auch *Parameter* genannt.

Die Tableau-Erweiterungsregeln, die für die Prädikatenlogik erster Stufe notwendig werden, sind:

$$\frac{\gamma}{\gamma(t)} \qquad \frac{\delta}{\delta(t)}$$

t ist geschlossener Term t ist ein neuer Parameter

der Parameter enthalten kann

Die Eingabe und die Schliessungsregeln sind wie beim aussagenlogischen Tableau. D.h. ein Tableau ist geschlossen, wenn jeder Pfad geschlossen ist, und dies wiederum heisst, daß in jedem Pfad ein Paar direkt komplementärer Literale vorkommt

Beispiel 9.1. Im folgenden ein Tableaubeweis der Formel

$$(\forall x : (P(x) \vee Q(x)) \Rightarrow (\exists x : P(x) \vee \forall x : Q(x)))$$

Auch hier wird, wie beim aussagenlogischen Tableau, mit der Negation der Formel gestartet.

$$\begin{array}{ll}
\neg(\forall x : (P(x) \vee Q(x)) \Rightarrow (\exists x : P(x) \vee \forall x : Q(x))) & \\
\forall x : (P(x) \vee Q(x)) & \\
\neg(\exists x : P(x) \vee \forall x : Q(x)) & \\
\neg(\exists x : P(x)) & \\
\neg(\forall x : Q(x)) & \\
\neg Q(c) & \delta\text{-Expansion} \\
\neg P(c) & \gamma\text{-Expansion mit } t = c \\
P(c) \vee Q(c) & \gamma\text{-Expansion mit } t = c \\
P(c) \mid Q(c) & \beta\text{-Expansion}
\end{array}$$

Diese Beweisprozedur terminiert i.a. nicht, im Gegensatz zum aussagenlogischen Tableau, da man für die γ -Formeln beliebige Terme einsetzen kann, und es auch nicht ausreicht, sich auf eine endliche Menge zu beschränken.

In dieser Form entspricht der Tableaurekalkül der Resolution ohne Unifikation, aber mit Raten von Klauselinstanzen. Beim Tableaurekalkül bleibt die Struktur der Formeln erhalten, so daß man ohne Normalform und (zunächst) ohne Skolemisierung auskommt.

Aussage 9.2. *Das Tableau-Verfahren für PL1 ist widerlegungsvollständig.*

Im Falle, daß es nicht terminiert, wird in einem Pfad ein Modell generiert, sofern Fairnessbedingungen erfüllt sind: Jede Formel muß in jedem Pfad und mit jeder Regel (insbesondere jede γ -Instanz) expandiert werden.

Beweis. siehe [Fit90] □

Die Eigenschaft, daß im Falle der Erfüllbarkeit ein Modell erzeugt wird, ist ein Vorteil gegenüber resolutionsbasierten Methoden. Allerdings ist dies eher theoretisch zu sehen, denn i.a. ist der Pfad dann unendlich lang.

Eine einfache Heuristik zum Expandieren des Tableaus ist: zuerst α -Regeln, dann β -Regeln, dann δ -Regeln und dann erst die γ -Regeln. Es ist auch erlaubt, Mehrfachexpansion von Formeln auf dem gleichen Pfad zu verhindern.

Es ist möglicherweise ausreichend, die δ -Regel pro Formel und Pfad nur einmal zu verwenden. Allerdings gibt es dann immer noch eine Quelle der Nichtterminierung: Jede γ -Formel muß im Prinzip mit jedem Parameter einmal instanziiert werden. Dies kann darauf hinauslaufen, daß man immer mehr Parameter erzeugen muß: Nach Erzeugung eines neuen Parameters d muss eine Formel oben im Tableau unten neu angehängt werden; Diese ergibt nach einigen Schritten eine neue δ -Formel, die einen neuen Parameter erzeugt, usw.

9.2 Tableau mit freien Variablen

Um den ausufernden Nichtdeterminismus der γ -Regel zu verhindern, kann man Unifikation ausnutzen, muß dazu aber die Datenstruktur des Tableaus leicht verändern, und freie Variablen zulassen.

Gleichzeitig werden statt Parametern Skolemterme bei der Expansion verwendet:

$$\frac{\gamma}{\gamma(x)}$$

$$\frac{\delta}{\delta(f(x_1, \dots, x_n))}$$

x ist neue freie Variable

f neue Skolemfunktion und

$$x_1, \dots, x_n$$

sind alle bisher benutzten freien Variablen

Definition 9.3. (*Instantiierungsregel*) Ein Tableau darf verändert werden, indem man eine Substitution σ auf das ganze Tableau anwendet. Hierbei muß die Substitution frei für das Tableau sein. D.h. es werden nur die freien Variablen ersetzt, und die eingesetzten Terme dürfen keine Variablen enthalten, die in den Bindungsbereich eines Quantors geraten (von einem Quantor eingefangen werden).

Die Schlußregel ist auch hier, daß nach der Instantiierung alle Pfade ein Paar komplementärer Literale enthalten müssen.

Z.B. ist die Substitution $\{x \mapsto f(y)\}$ nicht frei für die Formel $\forall y : P(x, f(x), a, z)$, da das y in $f(y)$ nach der Einsetzung gebunden wäre.

Die Instantiierungsregel und die Schlußregel kann man ersetzen durch eine mit Unifikation kombinierte Schlußregel:

Definition 9.4. Das Tableau ist geschlossen, wenn es eine Substitution σ gibt (frei für das Tableau), die das Tableau nach Instantiierung im herkömmlichen Sinne schließt.

D.h. statt einer Instantiierung macht man einen Unifikationsversuch, der einen Unifikator berechnet, der nach Anwendung alle Pfade gleichzeitig schließt. Dafür muß der Unifikationsalgorithmus geeignete Kandidatenlitterale verwenden, und diese unifizieren.

D.h. hier ist doch noch ein Nichtdeterminismus verborgen, da man auf sehr viele Weisen solche Kandidaten auswählen kann.

Beispiel 9.5. Im folgenden ein Tableaubeweis mit freien Variablen für die gleiche Formel wie in Beispiel 9.1

$$(\forall x : (P(x) \vee Q(x)) \Rightarrow (\exists x : P(x) \vee \forall x : Q(x)))$$

$$\begin{array}{ll}
\neg(\forall x : (P(x) \vee Q(x)) \Rightarrow (\exists x : P(x) \vee \forall x : Q(x))) & \\
\forall x : (P(x) \vee Q(x)) & \\
\neg(\exists x : P(x) \vee \forall x : Q(x)) & \\
\neg(\exists x : P(x)) & \\
\neg(\forall x : Q(x)) & \\
\neg Q(a) & \delta\text{-Expansion} \\
\neg P(y_1) & \gamma\text{-Expansion mit } y_1 \\
P(y_2) \vee Q(y_2) & \gamma\text{-Expansion mit } y_2 \\
P(y_2) \quad | \quad Q(y_2) & \beta\text{-Expansion}
\end{array}$$

Geschlossen mit $\sigma = \{y_1 \mapsto a, y_2 \mapsto a\}$

Folgendes Beispiel zeigt, wie ein Modell erzeugt wird:

Beispiel 9.6. Betrachte die Formel $(\exists x : P(x)) \Rightarrow (\forall x : P(x))$. Diese ist keine Satz. Versucht man einen Tableaubeweis, so erhält man ein Gegenbeispiel (bzw. ein Modell für die Negation).

$$\begin{array}{l}
\neg(\exists x : P(x)) \Rightarrow (\forall x : P(x)) \\
\exists x : P(x) \\
\neg(\forall x : P(x)) \\
P(a) \\
\neg P(b)
\end{array}$$

Das erzeugte Modell hat zwei Elemente: a, b . Die Struktur ist $\{P(a), \neg P(b)\}$, und in dieser Struktur ist die obige Formel falsch.

Beispiel 9.7. Beweis der Formel

$$(\exists w : \forall x : R(x, w, f(x, w))) \Rightarrow (\exists w : \forall x : \exists y : R(x, w, y))$$

- (1) $\neg((\exists w : \forall x : R(x, w, f(x, w))) \Rightarrow (\exists w : \forall x : \exists y : R(x, w, y)))$
 - (2) $\exists w : \forall x : R(x, w, f(x, w))$
 - (3) $\neg(\exists w : \forall x : \exists y : R(x, w, y))$
 - (4) $\forall x : R(x, a, f(x, a))$ aus (2) mit δ
 - (5) $\neg(\forall x : \exists y : R(x, v_1, y))$ aus (3) mit γ
 - (6) $\neg(\exists y : R(g(v_1), v_1, y))$ aus (3) mit δ
 - (7) $R(v_2, a, f(v_2, a))$ aus (4) mit γ
 - (8) $\neg(R(g(v_1), v_1, v_3))$ aus (6) mit γ
- geschlossen mit Unifikator $\sigma = \{v_1 \mapsto a, v_2 \mapsto g(a), v_3 \mapsto f(g(a), a)\}$

Die Frage nach der Terminierung und der Anzahl der notwendigen Kopien wird durch folgendes Beispiel illustriert:

Beispiel 9.8. Widerlege die Formelmenge $\{\{P(a) \vee P(b)\}, \{\forall x : \neg P(x)\}\}$.

- (1) $P(a) \vee P(b)$
- (2) $\forall x : \neg P(x)$
- (3) $\neg P(y)$
- (4) $P(a) \quad | \quad P(b)$

Dieses Tableau läßt sich nicht schließen, da $a \neq b$. Erst mit einer weiteren γ -Expansion der Formel $\forall x : \neg P(x)$ auf dem rechten Pfad ist das Tableau geschlossen.

Eine Umstellung der Expansionen zeigt aber, daß man in diesem Fall mit einer Expansion pro Pfad auskommt.

Vermutlich gilt somit, daß eine γ -Formel, die keine inneren Disjunktionen enthält, nur einmal pro Pfad expandiert werden muß.

Beispiel 9.9. Widerlege die Formelmenge

$$\{\{P(a)\}, \{\neg P(f(f(a)))\}, \{\forall x : P(x) \Rightarrow P(f(x))\}\}$$

- (1) $P(a)$
- (2) $\neg P(f(f(a)))$
- (3) $\forall x : P(x) \Rightarrow P(f(x))$
- (4) $P(y_1) \Rightarrow P(f(y_1))$ aus (3)
- (5) $\neg P(y_1) \quad | \quad P(f(y_1))$
- (6) $P(y_2) \Rightarrow P(f(y_2))$ aus (3)
- (6) $\neg P(y_2) \quad | \quad P(f(y_2))$

Das Tableau ist geschlossen mit $\{y_1 \mapsto a, y_2 \mapsto f(a)\}$.

Man sieht, daß es nicht möglich ist, das Tableau vorher zu schließen bzw. mit nur einer Kopie der Formel $\forall x : P(x) \Rightarrow P(f(x))$ pro Pfad. Eine Erweiterung dieses Beispiels zeigt auch, daß man die Anzahl der notwendigen γ -Expansionen pro Pfad im Prinzip nicht begrenzen kann. Aufgrund der Unentscheidbarkeit kann man durch keinen Trick die Tableau-methode in ein Entscheidungsverfahren verwandeln. D.h. es gibt immer eine Regel, die für das Nichtterminieren verantwortlich ist.

Die Anzahl der δ -Expansionen kann man auf eine pro Formel und Pfad beschränken. Beschränkt man die Anzahl der erlaubten Kopien pro Formel und Pfad willkürlich von vorneherein durch die sogenannte “Q-Tiefe“, dann terminiert das Verfahren.

Die Terminierung kann man wie im aussagenlogischen Fall begründen, wobei man das verwendete Maß leicht variieren muß:

- Basis ist das Paar (Anzahl der Quantoren, Größe der Formel).
- Pro Pfad die Multimenge der obigen Paare, wobei man für jede Formel mit Quantoren das Paar sofort in die Multimenge einfügt, wie man die Formel noch expandieren kann: (Q-Tiefe – wie oft γ -expandiert). Für α, β, δ -Formeln ist diese Zahl jeweils 1.
- Multimenge der Maße aller Pfade.

Unter Beschränkung der Q-Tiefe ist das Verfahren nicht mehr vollständig. Allerdings wird es wieder vollständig, wenn man das Verfahren mehrfach ablaufen läßt mit jeweils grösserer Q-Tiefe. Dies ergibt dann ein Semientscheidungsverfahren.

10 Gleichheitsbehandlung

Das bekannte Leibnizsche Prinzip besagt, daß zwei Dinge gleich sind, wenn sie bezüglich aller ihrer Eigenschaften gleich sind. In jedem beliebigen Kontext kann daher “Gleiches durch Gleiches” ersetzt werden. Diese grundlegende Eigenschaft wird so häufig ausgenutzt, daß sie wohl jedem geläufig ist, der schon mit Gleichungen hantiert hat.

Traditionell wird die Gleichheit mit dem binären Symbol “=” bezeichnet.

In einer Logik höherer Ordnung kann man Gleichheit leicht definieren und erklären durch:

$$\forall x, y : x = y \Leftrightarrow \forall P : P(x) \Leftrightarrow P(y)$$

Allerdings ist das nicht direkt in PL_1 übersetzbar.

Beispiel 10.1. Der Beweis des folgenden Theorems zeigt, welche typischen Schwierigkeiten bei Gleichheitsbeweisen auftreten. Zu zeigen sei: Eine Gruppe mit $\forall x : x^2 = 1$ ist kommutativ. Die Axiome einer Gruppe mit einem zweistelligen Verknüpfungssymbol \bullet und einer einstelligen Funktion i (Inverses) sind:

$$\forall x, y, z : (x \bullet y) \bullet z = x \bullet (y \bullet z) \quad (\text{Assoziativität})$$

$$\forall x : 1 \bullet x = x \quad (\text{Links-Eins})$$

$$\forall x : x \bullet 1 = x \quad (\text{Rechts-Eins})$$

$$\forall x : i(x) \bullet x = 1 \quad (\text{Links-Inverses})$$

$$\forall x : x \bullet i(x) = 1 \quad (\text{Rechts-Inverses})$$

Als zusätzliche Voraussetzung soll gelten:

$$\forall x : x \bullet x = 1 \quad (\text{Voraussetzung})$$

Die Behauptung ist nun:

$$\forall x, y : x \bullet y = y \bullet x$$

Ein Beweis der Kommutativität wird zum Beispiel durch folgende Gleichungskette geführt:

$$\begin{aligned}
(1) \quad x \bullet y &= (1 \bullet x) \bullet y && \text{(Links-Eins)} \\
(2) \quad &= ((y \bullet y) \bullet x) \bullet y && \text{(Voraussetzung)} \\
(3) \quad &= ((y \bullet y) \bullet x) \bullet (y \bullet 1) && \text{(Rechts-Eins)} \\
(4) \quad &= ((y \bullet y) \bullet x) \bullet (y \bullet (x \bullet x)) && \text{(Voraussetzung)} \\
(5) \quad &= (y \bullet ((y \bullet x) \bullet (y \bullet x))) \bullet x && \text{(mehrmals Assoziativität)} \\
(6) \quad &= (y \bullet 1) \bullet x && \text{(Voraussetzung)} \\
(7) \quad &= y \bullet x && \text{(Rechts-Eins)}
\end{aligned}$$

Der Beweistrick besteht darin, an den richtigen Stellen mit dem Einselement zu erweitern, das Einselement durch Anwendung der Voraussetzung geschickt darzustellen, umzuklammern und wieder zusammenzufassen. Welche der Umformungen in jedem einzelnen Schritt durchgeführt werden müssen, um zum Ziel zu kommen, ist zunächst jedoch überhaupt nicht zu erkennen. Zum Beispiel wären nach Schritt (3) auch die Operationen

$$\begin{aligned}
(3) \quad ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= (1 \bullet x) \bullet (y \bullet 1) && \text{(Voraussetzung)} \\
\text{oder } ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= ((y \bullet y) \bullet x) \bullet (y \bullet (1 \bullet 1)) && \text{(Links- oder Rechts-Eins)} \\
\text{oder } ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= ((y \bullet y) \bullet x) \bullet ((y \bullet 1) \bullet 1) && \text{(Rechts-Eins)} \\
\text{oder } ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= ((y \bullet y) \bullet x) \bullet ((1 \bullet y) \bullet 1) && \text{(Links-Eins)} \\
\text{oder } ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= (y \bullet (y \bullet x)) \bullet (y \bullet 1) && \text{(Assoziativität)} \\
\text{oder } ((y \bullet y) \bullet x) \bullet (y \bullet 1) &= ((y \bullet y) \bullet x) \bullet (y \bullet (i(w) \bullet w)) && \text{(Links-Inverses)}
\end{aligned}$$

sowie noch etliche andere ausführbar. Von jedem dieser nutzlosen Schritte aus gibt es wiederum eine Vielzahl weiterer alternativer Umformungen, so daß durch reines Ausprobieren aller Möglichkeiten ca. 10^{11} Versuche notwendig sind, um endlich zur anderen Seite der Gleichung zu kommen.

10.1 Formalisierung der Gleichheit innerhalb der Prädikatenlogik

In PL_1 kann das Gleichheitsprädikat $=$ nur dann durch eine endliche Axiomenmenge formalisiert werden, wenn man die Signatur (Funktionssymbole, Prädikaten-symbole) kennt und wenn diese endlich ist. Eine unabhängige Axiomatisierung ist erst in Logik einer höheren Stufe möglich. In PL_1 kann man stattdessen

die benötigte Axiomenmenge durch Instantiierung eines “Axiomenschemas” erzeugen:

Definition 10.2. *Axiomenschema für die Gleichheit:*

$$\forall x : \quad x = x \quad (\text{Reflexivität})$$

$$\forall x, y : \quad x = y \Rightarrow y = x \quad (\text{Symmetrie})$$

$$\forall x, y, z : \quad x = y \wedge y = z \Rightarrow x = z \quad (\text{Transitivität})$$

Für jedes Argument von jedem in der Formelmenge vorkommenden

Funktionssymbol f wird ein Substitutionsaxiom folgender Form benötigt:

$$\forall x_1, \dots, x_n, y : x_i = y \Rightarrow f(x_1, \dots, x_i, \dots, x_n) = f(x_1, \dots, y, \dots, x_n)$$

Für jedes Argument von jedem in der Formelmenge vorkommenden

Prädikatensymbol P wird ebenfalls ein weiteres Substitutionsaxiom benötigt:

$$\forall x_1, \dots, x_n, y : x_i = y \wedge P(x_1, \dots, x_i, \dots, x_n) \Rightarrow P(x_1, \dots, y, \dots, x_n)$$

Eine naive Methode, um Gleichheitsbeweise im Rahmen der Prädikatenlogik zu automatisieren, wäre also, zu den gegebenen Formeln nach dem obigen Schema die nötigen Gleichheitsaxiome hinzuzufügen und dann mit Resolution einen Beweis zu suchen. Wenn man das für das Gruppenbeispiel von oben durchführt und den Beweis im Resolutionskalkül nachrechnet, wird man feststellen, daß erstens die einzelnen Umformungsschritte sehr umständlich durch mehrere Resolutionsschritte nachgebildet werden müssen, und daß zweitens der Suchraum durch neue Ableitungsmöglichkeiten mit den Gleichheitsaxiomen noch weiter anwächst, in diesem Fall bis auf ca. 10^{21} Ableitungen bei reiner Breitensuche. Diese Vorgehensweise ist also nicht praktikabel.

Da die Formalisierung der Gleichheit mit den obigen Axiomen nicht zu brauchbaren Ergebnissen führt, hat man schon früh versucht, diese Relation unmittelbar in die Logik einzubauen. In den jeweiligen Kalkülen können dann spezielle Ableitungsoperationen, die deren Bedeutung direkt ausnutzen, formuliert werden.

Der *modelltheoretische* Einbau in die Logik geschieht, indem man von allen möglichen Interpretationen für Formeln nur noch diejenigen zuläßt, bei denen das Symbol $=$ die Identität in Σ -Strukturen abgebildet wird. Die heißen dann E -Interpretationen (E von equality) oder Gleichheitsinterpretationen. Äquivalent ist, daß die Gleichheit eine Kongruenz auf der Menge der Elemente des Modells ist.

Mit Hilfe der E -Interpretationen lassen sich jetzt die anderen semantischen Begriffe wie E -Modelle, E -Folgerung, E -erfüllbar und E -unerfüllbar in der üblichen Weise definieren.

10.2 Paramodulation

Eine natürliche, in der Mathematik häufig verwendete Regel zur Behandlung der Gleichheit beruht auf dem Prinzip der Untertermersetzung: In beliebigem Kontext kann Gleiches durch Gleiches ersetzt werden. Ähnlich wie bei Resolution ist auch hierbei in vielen Fällen die Anwendung einer Instantiierungsregel nötig, um die Voraussetzungen für die Anwendbarkeit der Untertermersetzung herzustellen, wobei Unifikation willkürliche Instantiierungen überflüssig macht und stattdessen ein zielgerichtetes Instantiieren auf der allgemeinsten Ebene ermöglicht. So wie der Modus Ponens unter Verwendung des Unifikationsprinzips zur Resolution verallgemeinert wurde, haben G. Robinson und L. Wos auch die Gleichheitersetzungsregel zur Paramodulationsregel verallgemeinert.

Formal wird die Paramodulationsregel wie folgt definiert:

Definition 10.3. (Paramodulation)

$$\begin{array}{l} C_1 := \{L_1[s], L_2, \dots, L_n\} \\ C_2 := \{l = r, K_2, \dots, K_m\} \\ \hline P := \sigma\{L_1[r], L_2, \dots, L_n, K_2, \dots, K_m\} \end{array}$$

Hierbei enthält $L_1[s]$ den Unterterm s , s und l sind unifizierbar mit dem allgemeinsten Unifikator σ (d.h. $\sigma s \equiv \sigma l$) und P ist Paramodulant der Klauseln C_1 und C_2 , wobei $L_1[r/s]$ aus L_1 durch Ersetzen des Terms s durch den Term r entsteht.

Paramodulation könnte man als bedingte Ersetzung bezeichnen, wobei die restlichen Literale der Klauseln wie Bedingungen wirken und in der Paramodulante die Bedingungen aufgesammelt werden. Die einfachste Variante ist die Paramodulation ohne Bedingungen. Dies wird auch *Termersetzung* genannt:

$$\begin{array}{l} C_1 := \{L[s]\} \\ C_2 := \{l = r\} \\ \hline P := \sigma\{L[r]\} \end{array}$$

wobei σ allgemeinsten Unifikator von s und l .

Beispiel 10.4. Für einen Paramodulationsschritt.

$$\begin{aligned}
C_1 &:= \{P(c, h(f(a, y), b))\} \\
C_2 &:= \{f(x, e) = g(x), Q(x)\} \\
P &:= \{P(c, h(g(a), b)), Q(a)\}
\end{aligned}$$

$P = \{P(c, h(g(a), b)), Q(a)\}$ ist Paramodulant der beiden Klauseln. Die Terme $f(a, y)$ und $f(x, e)$ sind unifizierbar mit $\{x \mapsto a, y \mapsto e\}$. Diese Substitution muß auf die neue Klausel angewendet werden.

Paramodulation ist in zweifacher Hinsicht allgemeiner als die oben erwähnte Regel “Gleiches durch Gleiches zu ersetzen“:

1. Paramodulation ist nicht nur mit unbedingten, sondern auch mit bedingten Gleichungen anwendbar, das heißt die Gleichungs-Klausel kann noch weitere Literale enthalten.
2. Die beiden Terme, welche eine Ersetzung ermöglichen sollen, müssen nicht gleich, sondern lediglich unifizierbar sein, was bedeutet, daß es Instanzen der beteiligten Klauseln geben muß, so daß die entsprechenden Terme gleich sind.

10.3 Eigenschaften der Paramodulation

Die Paramodulationsregel ist korrekt: falls S eine E -erfüllbare (bzw. E -unerfüllbare) Menge von Klauseln und C Paramodulant von zwei Klauseln aus S ist, dann ist auch $S \cup \{C\}$ E -erfüllbar (bzw. E -unerfüllbar).

Erweitert man den Resolutionskalkül um die Paramodulationsregel und erweitert die Eingabeklauseln um die Klausel (das Axiom) $\{x = x\}$, dann erhält man den RP-Kalkül für die Prädikatenlogik erster Stufe mit Gleichheit.

Dieser ist widerlegungsvollständig: für jede E -unerfüllbare Menge von Klauseln existiert eine Ableitung der leeren Klausel unter Anwendung der Regeln und Axiome des Kalküls.

Das Reflexivitätsaxiom ist nötig, da andernfalls die leere Klausel nicht aus der E -unerfüllbaren, einelementigen Klauselmenge $\{\neg a = a\}$ ableitbar wäre.

Verglichen mit der expliziten Verwendung der Gleichheitsaxiome ist der Suchraum für die Ableitung der leeren Klausel durch Einführung der Paramodulation deutlich kleiner. Viele der sinnlosen Resolutionen mit und zwischen den Gleichheitsaxiomen sind nicht mehr möglich. Trotzdem sind die entstehenden Suchräume ohne geschickte heuristische Steuerung der Paramodulation noch viel zu groß, da auch diese Regel fast überall in der Klauselmenge angewendet werden kann. Für das obige Gruppenbeispiel sind nach Alan Bundy bei reiner Breiten-suche etwa 10^{11} Paramodulationsschritte versuchsweise nötig, um den Beweis des Satzes zu finden.

Beispiel 10.5. für Kombination von Resolution und Paramodulation.
 Zu zeigen ist: Das Einselement einer Untergruppe ist dasselbe wie das Einselement der Gruppe selbst.

Axiomatisierung:

$$\forall X, Y : X \subseteq Y \Rightarrow (\forall x : x \in X \Rightarrow x \in Y) \quad (\text{Teilmengenbeziehung})$$

$$\forall X : \text{Gruppe}(X) \Rightarrow (\forall x : x \in X \Rightarrow x * \text{inv}(x) = \text{id}(X)) \quad (\text{Def. Einselement})$$

$$\forall X : \text{Gruppe}(X) \Rightarrow \text{id}(X) \in X$$

$$\forall X, Y : \text{Gruppe}(X) \wedge \text{Gruppe}(Y) \wedge X \subseteq Y \Rightarrow \text{id}(X) = \text{id}(Y) \quad (\text{Theorem})$$

Klauselform:

$$C1 : \neg X \subseteq Y, \neg x \in X, x \in Y$$

$$C2 : \neg \text{Gruppe}(X), \neg x \in X, x * \text{inv}(x) = \text{id}(X)$$

$$C3 : \neg \text{Gruppe}(X), \text{id}(X) \in X$$

$$T1. \text{Gruppe}(A)$$

$$T2 : \text{Gruppe}(B)$$

$$T3 : A \subseteq B$$

$$T4 : \text{id}(A) \neq \text{id}(B)$$

Resolutions- und Paramodulationswiderlegung:

$$\begin{aligned}
C1, 1 + T3 &\vdash R1 : \neg x \in A, x \in B \\
C2, 1 + T1 &\vdash R2 : \neg x \in A, x * inv(x) = id(A) \\
C3, 1 + T1 &\vdash R3 : id(A) \in A \\
R2, 1 + R3 &\vdash R4 : id(A) * inv(id(A)) = id(A) \\
C2, 1 + T2 &\vdash R5 : \neg x \in B, x * inv(x) = id(B) \\
R1, 2 + R5, 1 &\vdash R6 : \neg x \in A, x * inv(x) = id(B) \\
R6, 1 + R3 &\vdash R7 : id(A) * inv(id(A)) = id(B) \\
R7, 1r + T4 &\vdash R8 : id(A) * inv(id(A)) \neq id(A) \text{ (Paramodulation)} \\
R4 + R8 &\vdash R9 : \square
\end{aligned}$$

11 Verwendung von Theorien: Semantische Ansätze: Theorieresolution

Die Theorieresolution ist ein Schema, um Information über die Bedeutung von Prädikaten- und Funktionssymbolen unmittelbar im (Resolutions-) Kalkül auszunutzen, indem man anstelle von Axiomen für diese Symbole maßgeschneiderte Schlußregeln verwendet. Sie wurde von Mark Stickel am SRI allgemein formuliert. Viele Spezialfälle davon, u.a. Paramodulation, waren jedoch schon vorher unter anderen Namen bekannt.

Z.B. ist es dadurch möglich, direkt mit arithmetischen Bedingungen umzugehen, die man ohne diese Technik nur direkt (und umständlich) in Prädikatenlogik kodieren kann.

Zur Motivation der Vorgehensweise erinnern wir uns an die Begründung für die Korrektheit der einfachen Resolutionsregel:

$$\begin{array}{lcl} \text{Klausel1:} & L, K_1, \dots, K_n & \\ \text{Klausel2:} & \neg L, M_1, \dots, M_m & \\ \hline \text{Resolvente:} & K_1, \dots, K_n, M_1, \dots, M_m & \end{array}$$

Das entscheidende Argument dafür, daß die Resolvente aus den Elternklauseln folgt, war: wenn eine Interpretation das Literal L erfüllt, dann falsifiziert sie $\neg L$. Wesentlich ist also, daß keine Interpretation sowohl L als auch $\neg L$ erfüllen kann. Diese Eigenschaft haben zwei Literale dann, wenn sie die rein syntaktische Bedingung erfüllen, komplementär zu sein, also übereinstimmende Prädikaten-symbole und Termlisten, aber verschiedene Vorzeichen zu besitzen.

In vielen Fällen kann man den syntaktischen Komplementaritätsbegriff verallgemeinern, indem man ausnutzt, daß nicht völlig beliebige, sondern nur bestimmte Klassen von Interpretationen in Frage kommen. Zum Beispiel könnte eine Formelmenge geeignete Axiome für das Prädikatensymbol $<$ enthalten, so daß nur solche Interpretationen Modelle sein können, die dem Symbol $<$ eine strikte Ordnungsrelation auf dem Universum zuordnen. Aufgrund der Eigenschaften solcher Relationen kann keine derartige Interpretation sowohl $a < b$ als auch $b < a$ erfüllen. In dem genannten Kontext sind diese beiden Literale zwar nicht syntaktisch, aber sozusagen semantisch komplementär, und auch folgender Ableitungsschritt ist korrekt:

$$\begin{array}{lcl} \text{Klausel1:} & a < b, K & \\ \text{Klausel2:} & b < a, M & \\ \hline \text{Resolvente:} & K, M & \end{array}$$

Als weitere Verallgemeinerung können wir sogar die Beschränkung auf zwei Elternklauseln aufgeben. Keine Interpretation der genannten Klasse kann sowohl $a < b$ als auch $b < c$ als auch $c < a$ erfüllen. Damit kann man analog zur Begründung für die einfache Resolution, nur mit mehr Fallunterscheidungen, auch folgenden Schritt als korrekt nachweisen:

Klausel1: $a < b, K$

Klausel2: $b < c, M$

Klausel3: $c < a, N$

Resolvente: K, M

Die Idee ist also, von dem Spezialfall zweier syntaktisch komplementärer Resolutionsliterals überzugehen zu einer beliebigen Menge von Literalen, so daß keine Interpretation einer gegebenen Klasse alle diese Resolutionsliterals erfüllen kann. Die "Klasse von Interpretationen" wird durch den Theoriebegriff präzisiert.

Einer erfüllbaren Menge \mathcal{A} von Formeln läßt sich in PL_1 eindeutig die Klasse \mathcal{M} ihrer Modelle zuordnen, also von Interpretationen, die die Formeln wahr machen. Dieser Klasse von Interpretationen entspricht wiederum eindeutig eine (unendliche) maximale Menge \mathcal{T} von Formeln, die von allen Interpretationen in \mathcal{M} erfüllt werden. Die Formelmengung \mathcal{T} ist maximal in dem Sinn, daß jede weitere Formel die Modellklasse \mathcal{M} einschränkt, also von wenigstens einem Modell für \mathcal{A} falsifiziert wird. Per definitionem ist \mathcal{T} gerade die Menge der Folgerungen aus \mathcal{A} . In dieser Sichtweise enthalten \mathcal{M} und \mathcal{T} dieselbe Information, und beide werden oft als Theorie bezeichnet. Da verschiedene Formelmengen dieselben Modelle haben können, stellt ein gegebenes \mathcal{A} nur eine Alternative dar, die Theorie zu definieren. Man nennt \mathcal{A} auch eine Präsentation oder Axiomatisierung der Theorie.

Definition 11.1. (Theorien)

Für eine Formelmengung \mathcal{A} ist: $Theorie(\mathcal{A}) = Menge\ aller\ Modelle\ von\ \mathcal{A} \sim Menge\ aller\ Formeln,\ die\ in\ allen\ Modellen\ von\ \mathcal{A}\ gelten.$

Sei \mathcal{T} eine gegebene Theorie:

\mathcal{T} -erfüllbar:: Eine Formel F ist \mathcal{T} -erfüllbar, falls es ein Modell von \mathcal{T} gibt, das F erfüllt.

\mathcal{T} -unerfüllbar:: Eine Formel F ist \mathcal{T} -unerfüllbar, falls es kein Modell von \mathcal{T} gibt, das F erfüllt.

\mathcal{T} -allgemeingültig: Eine Formel F ist \mathcal{T} -allgemeingültig: falls F in allen Modellen von \mathcal{T} gilt.

\mathcal{T} -Folgerung $F \models_{\mathcal{T}} G$ Falls G in allen \mathcal{T} -Modellen von F gilt.

11 Verwendung von Theorien: Semantische Ansätze: Theorieresolution

Die Theorieresolution ist ein Schema, um Information über die Bedeutung von Prädikaten- und Funktionssymbolen unmittelbar im (Resolutions-) Kalkül auszunutzen, indem man anstelle von Axiomen für diese Symbole maßgeschneiderte Schlußregeln verwendet. Sie wurde von Mark Stickel am SRI allgemein formuliert. Viele Spezialfälle davon, u.a. Paramodulation, waren jedoch schon vorher unter anderen Namen bekannt.

Z.B. ist es dadurch möglich, direkt mit arithmetischen Bedingungen umzugehen, die man ohne diese Technik nur direkt (und umständlich) in Prädikatenlogik kodieren kann.

Zur Motivation der Vorgehensweise erinnern wir uns an die Begründung für die Korrektheit der einfachen Resolutionsregel:

$$\begin{array}{ll} \text{Klausel1:} & L, K_1, \dots, K_n \\ \\ \text{Klausel2:} & \neg L, M_1, \dots, M_m \\ \hline \text{Resolvente:} & K_1, \dots, K_n, M_1, \dots, M_m \end{array}$$

Das entscheidende Argument dafür, daß die Resolvente aus den Elternklauseln folgt, war: wenn eine Interpretation das Literal L erfüllt, dann falsifiziert sie $\neg L$. Wesentlich ist also, daß keine Interpretation sowohl L als auch $\neg L$ erfüllen kann. Diese Eigenschaft haben zwei Literale dann, wenn sie die rein syntaktische Bedingung erfüllen, komplementär zu sein, also übereinstimmende Prädikaten-symbole und Termlisten, aber verschiedene Vorzeichen zu besitzen.

In vielen Fällen kann man den syntaktischen Komplementaritätsbegriff verallgemeinern, indem man ausnutzt, daß nicht völlig beliebige, sondern nur bestimmte Klassen von Interpretationen in Frage kommen. Zum Beispiel könnte eine Formelmenge geeignete Axiome für das Prädikatensymbol $<$ enthalten, so daß nur solche Interpretationen Modelle sein können, die dem Symbol $<$ eine strikte Ordnungsrelation auf dem Universum zuordnen. Aufgrund der Eigenschaften solcher Relationen kann keine derartige Interpretation sowohl $a < b$ als auch $b < a$ erfüllen. In dem genannten Kontext sind diese beiden Literale zwar nicht syntaktisch, aber sozusagen semantisch komplementär, und auch folgender Ableitungsschritt ist korrekt:

$$\begin{array}{ll} \text{Klausel1:} & a < b, K \\ \\ \text{Klausel2:} & b < a, M \\ \hline \text{Resolvente:} & K, M \end{array}$$

Als weitere Verallgemeinerung können wir sogar die Beschränkung auf zwei Elternklauseln aufgeben. Keine Interpretation der genannten Klasse kann sowohl $a < b$ als auch $b < c$ als auch $c < a$ erfüllen. Damit kann man analog zur Begründung für die einfache Resolution, nur mit mehr Fallunterscheidungen, auch folgenden Schritt als korrekt nachweisen:

Klausel1: $a < b, K$

Klausel2: $b < c, M$

Klausel3: $c < a, N$

Resolvente: K, M

Die Idee ist also, von dem Spezialfall zweier syntaktisch komplementärer Resolutionsliterals überzugehen zu einer beliebigen Menge von Literalen, so daß keine Interpretation einer gegebenen Klasse alle diese Resolutionsliterals erfüllen kann. Die "Klasse von Interpretationen" wird durch den Theoriebegriff präzisiert.

Einer erfüllbaren Menge \mathcal{A} von Formeln läßt sich in PL_1 eindeutig die Klasse \mathcal{M} ihrer Modelle zuordnen, also von Interpretationen, die die Formeln wahr machen. Dieser Klasse von Interpretationen entspricht wiederum eindeutig eine (unendliche) maximale Menge \mathcal{T} von Formeln, die von allen Interpretationen in \mathcal{M} erfüllt werden. Die Formelmengemenge \mathcal{T} ist maximal in dem Sinn, daß jede weitere Formel die Modellklasse \mathcal{M} einschränkt, also von wenigstens einem Modell für \mathcal{A} falsifiziert wird. Per definitionem ist \mathcal{T} gerade die Menge der Folgerungen aus \mathcal{A} . In dieser Sichtweise enthalten \mathcal{M} und \mathcal{T} dieselbe Information, und beide werden oft als Theorie bezeichnet. Da verschiedene Formelmengen dieselben Modelle haben können, stellt ein gegebenes \mathcal{A} nur eine Alternative dar, die Theorie zu definieren. Man nennt \mathcal{A} auch eine Präsentation oder Axiomatisierung der Theorie.

Definition 11.1. (Theorien)

Für eine Formelmengemenge \mathcal{A} ist: $Theorie(\mathcal{A}) = Menge\ aller\ Modelle\ von\ \mathcal{A} \sim Menge\ aller\ Formeln,\ die\ in\ allen\ Modellen\ von\ \mathcal{A}\ gelten.$

Sei \mathcal{T} eine gegebene Theorie:

\mathcal{T} -erfüllbar:: Eine Formel F ist \mathcal{T} -erfüllbar, falls es ein Modell von \mathcal{T} gibt, das F erfüllt.

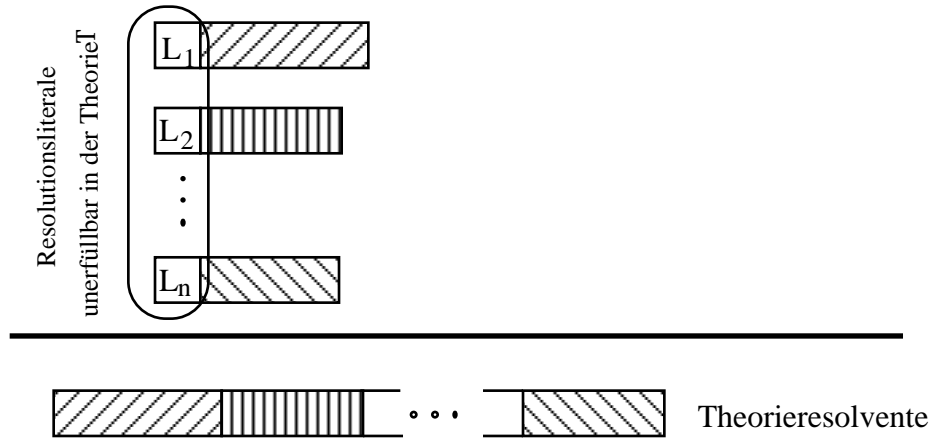
\mathcal{T} -unerfüllbar:: Eine Formel F ist \mathcal{T} -unerfüllbar, falls es kein Modell von \mathcal{T} gibt, das F erfüllt.

\mathcal{T} -allgemeingültig: Eine Formel F ist \mathcal{T} -allgemeingültig, falls F in allen Modellen von \mathcal{T} gilt.

\mathcal{T} -Folgerung $F \models_{\mathcal{T}} G$ Falls G in allen \mathcal{T} -Modellen von F gilt.

Sei beispielsweise \mathcal{A} die Menge $\{\forall x, y : P(x, y) \Rightarrow P(y, x)\}$, die nur aus dem Symmetrieaxiom für P besteht. Die Theorie \mathcal{T} ergibt sich dann aus allen Interpretationen, die dem Prädikatsymbol P eine symmetrische Relation auf dem Universum zuordnen. Die Formel $P(a, b) \wedge \neg P(b, a)$ ist erfüllbar, aber nicht \mathcal{T} -erfüllbar. $P(b, a)$ ist eine \mathcal{T} -Folgerung von $P(a, b)$.

Definition 11.2. (Totale Theorieresolution ohne Unifikation) *Das aussagenlogische Schema für die totale Theorieresolution sieht nun folgendermaßen aus: Seien eine Theorie \mathcal{T} und Klauseln C_1, \dots, C_n gegeben, und jede Klausel enthalte ein Literal L_i , so daß die Konjunktion all dieser Literale \mathcal{T} -unerfüllbar ist. Die Vereinigung der n Klauseln abzüglich dieser Resolutionsliterale L_i ergibt eine \mathcal{T} -Resolvente. Diese ist eine \mathcal{T} -Folgerung von $C_1 \wedge \dots \wedge C_n$. Graphisch:*



Analog zur einfachen Resolution muß die Konjunktion der L_i im prädikatenlogischen Fall nicht unmittelbar \mathcal{T} -widersprüchlich sein. Man benötigt eine Substitution σ , einen \mathcal{T} -Unifikator, so daß die Formel $\sigma L_1 \wedge \dots \wedge \sigma L_n$ \mathcal{T} -widersprüchlich ist. Die \mathcal{T} -Resolvente muß dann mit σ instantiiert werden. Allerdings ist ein allgemeinsten \mathcal{T} -Unifikator für eine Menge von Ausdrücken nicht mehr bis auf Variablenumbenennung eindeutig bestimmt. Abhängig von \mathcal{T} kann es einen, endlich viele oder unendlich viele voneinander unabhängige allgemeinste \mathcal{T} -Unifikatoren geben. Zwei Substitutionen werden dabei als unabhängig bezeichnet, wenn sich nicht eine durch Instantiierung von Variablen aus der anderen ergibt. In böartigen Fällen existieren überhaupt keine allgemeinsten \mathcal{T} -Unifikatoren, sondern nur nicht-allgemeinste.

Eine Theorie mit endlich vielen \mathcal{T} -Unifikatoren ist die oben erwähnte Theorie der Symmetrie von P , die maximal zwei allgemeinste Unifikatoren erzeugt. Für

Klausel 1: $P(a, b), Q$

Klausel 2: $\neg P(x, y), R(x)$

besitzen die beiden ersten Literale die \mathcal{T} -Unifikatoren $\sigma_1 = \{x \mapsto a, y \mapsto b\}$ und $\sigma_2 = \{x \mapsto b, y \mapsto a\}$, so daß die zwei unabhängigen \mathcal{T} -Resolventen $\{Q, R(a)\}$ und $\{Q, R(b)\}$ abgeleitet werden können.

Das Konzept der Theorieresolution erlaubt, häufig vorkommende Standardinterpretationen von Symbolen wesentlich natürlicher und effizienter als mit der üblichen Axiomatisierung und normaler Resolution zu handhaben. Die Information über die spezielle Theorie steckt dabei in erster Linie im Unifikationsalgorithmus, der allerdings für jede Theorie neu entwickelt werden muß. Um die Widerlegungsvollständigkeit der Theorieresolutionsverfahren sicherzustellen, muß der verwendete Theorieunifikationsalgorithmus *alle* allgemeinsten Unifikatoren erzeugen bzw. im unendlichen Fall zumindest aufzählen können.

Der für die Realisierung der Theorieresolution erforderliche Unifikationsalgorithmus ist für manche Theorien zu aufwendig oder überhaupt nicht bekannt. Dies gilt insbesondere, wenn die Theorie eigentlich aus mehreren Untertheorien besteht, die aber nicht unabhängig voneinander sind. Als Beispiel betrachten wir die Theorie \mathcal{T}_1 der Interpretationen, die dem Prädikatensymbol \leq (antisymmetrische) Ordnungsrelationen zuordnen, sowie die Theorie \mathcal{T}_2 der Interpretationen, die dem Prädikatensymbol $=$ die Gleichheitsrelation zuordnen. Die Kombination dieser beiden Theorien macht die Konjunktion der Literale $a \leq b, b \leq a, P(a), \neg P(b)$ unerfüllbar, so daß sie ohne weiteres als Resolutionslitterale für eine Theorieresolution in Frage kommen. Ein Algorithmus, der das erkennen kann, müßte jedoch genau für die Kombination dieser beiden Theorien konzipiert sein. Sobald eine dritte hinzukommt, ist er nicht mehr anwendbar. Deshalb versucht man, Algorithmen für einzelne Theorien zu entwickeln und einen allgemeineren Mechanismus verfügbar zu machen, der den Austausch zwischen beliebigen Theorien besorgt.

Betrachten wir die Kombination der genannten Theorien und folgende Klauseln:

Klausel1: $a \leq b, K$

Klausel2: $b \leq a, L$

Klausel3: $P(a), M$

Klausel4: $\neg P(b), N$

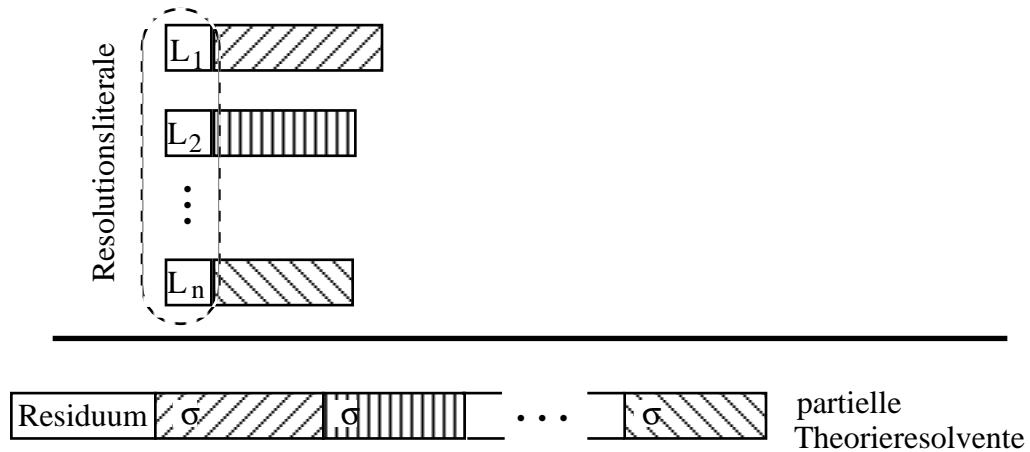
Daraus müßte eigentlich die Resolvente $\{K, L, M, N\}$ ableitbar sein. Dies kann man durch einen verallgemeinerten \mathcal{T}_1 -Schritt und einen anschließenden \mathcal{T}_2 -Schritt erreichen. Wenn eine Interpretation in der Theorie \mathcal{T}_1 sowohl $a \leq b$ als auch $b \leq a$ erfüllt, dann muß sie wegen der Antisymmetrie auch das Literal $a = b$ erfüllen. Man sieht leicht, daß deshalb die Klausel $\{a = b, K, L\}$ eine \mathcal{T}_1 -Folgerung aus Klausel 1 und Klausel 2 ist. Die Literale $a = b, P(a), \neg P(b)$ können dann von dem Algorithmus für \mathcal{T}_2 als Resolutionslitterale für eine Gleichheitstheorieresolution zwischen der neuen Klausel sowie Klausel 3 und Klausel 4 erkannt werden, womit die gewünschte Klausel $\{K, L, M, N\}$ entsteht.

Der erste Schritt geht über die Theorieresolution im bisherigen Sinn hinaus, da die Konjunktion der Resolutionsliterals allein noch nicht unerfüllbar in der Theorie ist, und da ein neues Literal abgeleitet und in die Resolvente aufgenommen wurde. Dieses sogenannte *Residuum* hat die Eigenschaft, daß es in der Theorie aus den Resolutionsliterals folgt. Sein Prädikatsymbol braucht nicht einmal in den Elternklauseln vorzukommen. Ist ein Residuum beteiligt, spricht man von *partieller Theorieresolution*, andernfalls von *totaler Theorieresolution*.

Als Residuum kann man auch eine Disjunktion von mehreren Literalen zulassen. Das leere Residuum steht dann wie die leere Klausel für “falsch“ und folgt daher nur dann aus den Resolutionsliterals, wenn deren Konjunktion \mathcal{T} -unerfüllbar ist. Dieser Spezialfall entspricht der totalen Theorieresolution.

Definition 11.3. (Allgemeine (partielle) Theorieresolution) Seien eine Theorie \mathcal{T} und Klauseln C_1, \dots, C_n gegeben, und jede Klausel enthalte ein Literal L_i , so daß für eine Substitution σ gilt: $\sigma L_1 \wedge \dots \wedge \sigma L_n \models_{\mathcal{T}} \text{Residuum}$, dann ist $\sigma((C_1 \setminus L_1) \cup \dots \cup (C_n \setminus L_n)) \cup \text{Residuum}$ eine allgemeine (partielle) Theorieresolvente.

Schematisch:



Als Bedingung für die Widerlegungsvollständigkeit der partiellen Theorieresolution ergibt sich, daß der Unifikationsalgorithmus nicht nur alle allgemeinsten Unifikatoren, sondern auch alle “allgemeinsten Residuen“ erzeugen können muß. Für das Suchverhalten des Verfahrens ist das sehr problematisch. Auch hier muß man heuristisch sich gute (kurze, kleine) Residuen aussuchen.

Paramodulation ist ein typisches Beispiel für partielle Theorieresolution. Die Theorie ist die Gleichheit, d.h. die Theorie, die sich aus dem Axiomensystem der Gleichheit (Definition 10.2) ergibt. Das paramodulierte Literal ist das Residuum.

12 Unifikation unter Gleichheitstheorien

Wie bereits in dem Kapitel über die Gleichheitsbehandlung ausgeführt wurde, bereiten Formeln, in denen die Gleichheit vorkommt, ziemliche Schwierigkeiten beim automatischen Beweisen, trotz Paramodulationsregel. Besonders unangenehm verhalten sich dabei zum Beispiel Formeln, die die Kommutativität gewisser Funktionssymbole definieren, d.h. wenn etwa $\forall x, y : g(x, y) = g(y, x)$ gilt. Diese Kommutativitätsformel führt unter Umständen zu einem ständigen Vertauschen der Argumente von Termen mit dem kommutativen Funktionssymbol. Daher wurde schon relativ früh versucht, solche Gleichungsformeln aus den Formelmengen zu entfernen und durch entsprechend veränderte Schlußregeln zu ersetzen. G. Plotkin schlug vor, die Resolutionsregel dahingehend zu ändern, daß die Unifikation durch eine die herausgenommenen Gleichungsformeln berücksichtigende Unifikation ersetzt wird. Er gab auch die wesentlichen Kriterien an, unter denen das geschehen kann, ohne daß die Vollständigkeit des Kalküls verloren geht. Vorausgesetzt, das Gleichheitsprädikat kommt nur in unären Klauseln vor, d.h. die Klauselmenge enthält endlich viele Klauseln $l_1 = r_1, \dots, l_n = r_n$, und es tritt sonst in keiner weiteren Klausel auf, dann darf die Unifikation durch eine sogenannte *Theorieunifikation* ersetzt werden, die die Gleichungsaxiome $l_1 = r_1, \dots, l_n = r_n$ berücksichtigt.

Der Resolutionskalkül RF wird dann folgendermaßen abgeändert:

statt auf $C \cup E$

wird auf C Resolution und Faktorisierung mit Unifikation bzgl. E verwendet.

Voraussetzung: C enthält keine Gleichheitssymbole, E besteht nur aus Unit-Klauseln, die Gleichungen sind. Der E -Unifikationsalgorithmus liefert i.a. eine Lösungsmenge von Unifikatoren, die alle auszuprobieren sind.

Es gilt: Wenn der E -Unifikationsalgorithmus eine vollständige Menge von Unifikatoren berechnet, dann ist der Kalkül widerlegungsvollständig.

Hierbei muß die Lösungsmenge (endlich oder unendlich) und rekursiv aufzählbar sein.

Theorieunifikationsalgorithmen

Beispiel 12.1. Die beiden Terme $g(x, y)$ und $g(a, b)$ haben den üblichen Unifikator $\theta_1 = \{x \mapsto a, y \mapsto b\}$. Unter der Annahme, daß g kommutativ ist, ergibt sich aber noch ein zweiter Unifikator, nämlich $\theta_2 = \{x \mapsto b, y \mapsto a\}$. Offensichtlich sind beide Unifikatoren unabhängig voneinander, d.h. es gibt in diesem Fall mehr als einen allgemeinsten Unifikator. Die Gleichung $k(f(x, g(a, y)), g(x, h(y))) \stackrel{?}{=} k(f(h(y), g(y, a)), g(z, z))$ hat eine Lösung $\sigma_1 = \{x \mapsto h(a), y \mapsto a, z \mapsto h(a)\}$, die z.B. mit dem Unifikationsalgorithmus U1 (Definition 5.19) berechnet werden kann. Unter der Annahme, daß g kommutativ ist, ist $\sigma_2 = \{x \mapsto h(y), z \mapsto h(y)\}$ ebenfalls ein Unifikator, der in diesem Fall offensichtlich sogar allgemeiner ist als der oben berechnete. Man erhält den obigen Unifikator durch weitere Instantiierung, indem man y durch a substituiert. Benutzt man σ_2 als Unifikator (und zusätzlich $y' \mapsto y$ dann kann man zum Beispiel folgende (totale) Theorieresolution durchführen:

$$\begin{array}{c}
P(y, k(f(x, g(a, y)), g(x, h(y)))) , Q(x, y) \\
\\
\neg P(y', k(f(h(y'), g(y', a)), g(z, z))) R(y', z) \\
\hline
Q(h(y), y), R(y, h(y))
\end{array}$$

Die Theorie, die dabei benutzt wurde ist die Kommutativität von g , d.h. $\forall x, y : g(x, y) = g(y, x)$.

Wir definieren jetzt einen Unifikationsalgorithmus für kommutativ deklarierte Funktionssymbole. Er ist eine leichte Abwandlung von U1.

Definition 12.2. *Unifikationsalgorithmus U_C : Eingabe:*

1. *zwei Terme oder Atome s und t ,*
2. *sowie eine Liste F_C von zweistelligen Funktionssymbolen, die als kommutativ deklariert sind.*

Ausgabe: "nicht unifizierbar" oder eine vollständige Menge von Unifikatoren.

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

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

Unifikationsregeln:

$$\frac{f(s_1, \dots, s_n) \stackrel{?}{=} f(t_1, \dots, t_n), \Gamma}{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n, \Gamma} \quad (\text{Dekomposition})$$

$$\frac{g(s_1, s_2) \stackrel{?}{=} g(t_1, t_2), \Gamma}{s_1 \stackrel{?}{=} t_2, s_2 \stackrel{?}{=} t_1, \Gamma} \quad (\text{Vertauschung})$$

falls g in F_C enthalten ist, d.h. g ist kommutativ.

$$\frac{x \stackrel{?}{=} x, \Gamma}{\Gamma} \quad (\text{Tautologie})$$

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

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

Abbruchbedingungen:

$$\frac{f(\dots) \stackrel{?}{=} g(\dots), \Gamma}{Fail} \quad \text{wenn } f \neq g \quad (\text{Clash})$$

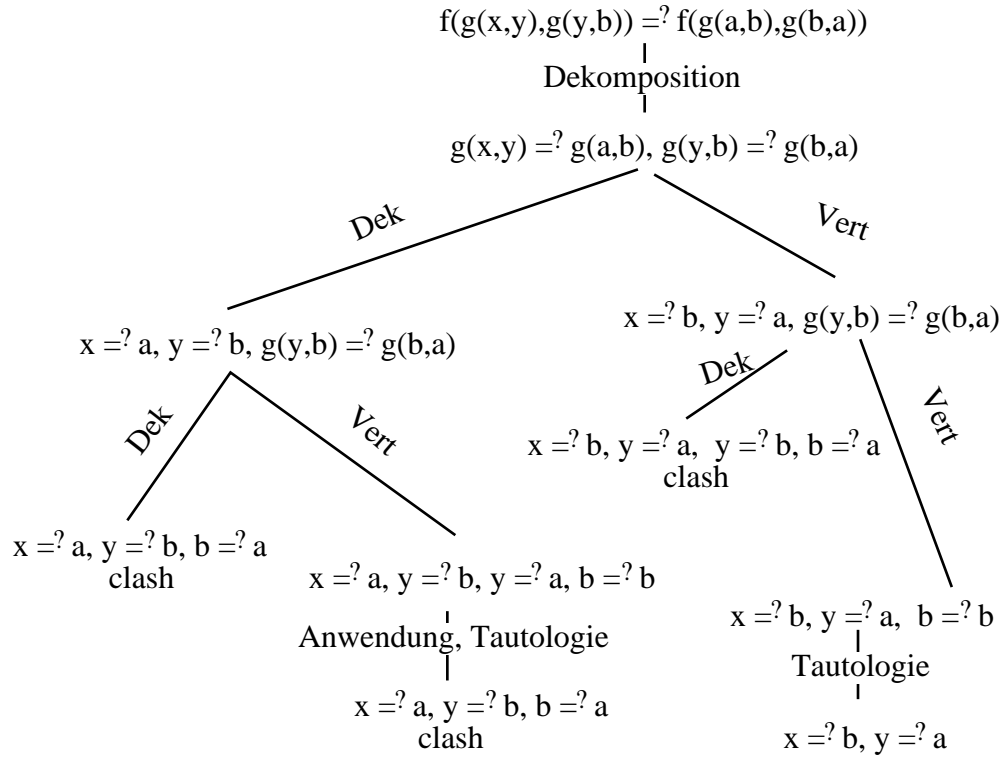
$$\frac{x \stackrel{?}{=} t, \Gamma}{Fail} \quad \text{wenn } x \in FV(t) \text{ vorkommt (occurs check Fehler)}$$

Steuerung: Erzeuge einen Baum folgendermaßen: Der Wurzelknoten ist $\Gamma := \Gamma_0$. Für jeden Knoten, der mit einem Gleichungssystem Γ markiert ist: Wähle eine Gleichung $s \stackrel{?}{=} t$ in Γ aus, auf die mindestens eine Regel anwendbar ist und erzeuge die Nachfolgeknoten (maximal zwei) dadurch, daß alle anwendbaren Regeln auf $s \stackrel{?}{=} t$ angewendet werden. Auf diese Weise werden so lange neue Äste im Baum erzeugt, bis entweder eine Abbruchbedingung erfüllt ist oder keine Regel mehr anwendbar ist. Die Gleichungsmenge in den Blattknoten, auf die keine Abbruchbedingung zutrifft hat die Form $\{x_1 \stackrel{?}{=} t_1, \dots, x_k \stackrel{?}{=} t_k\}$. Die daraus generierten Substitutionen $\{x_1 \mapsto t_1, \dots, x_k \mapsto t_k\}$ werden aufgesammelt und als Ergebnis zurückgeliefert.

Beispiel 12.3. $g(x, y) \stackrel{?}{=} g(a, b)$ kann auf zwei Weisen dekomponiert werden, wenn $g \in F_C$ angenommen wird. Dies ergibt sofort zwei Systeme von Gleichungen:

- $x \stackrel{?}{=} a, y \stackrel{?}{=} b$.
- $x \stackrel{?}{=} b, y \stackrel{?}{=} b$.

Ein weiteres Beispiel ist folgender Berechnungsbaum:



Es gibt also nur eine Lösung in diesem Fall: $\{x \mapsto b, y \mapsto a\}$.

Wie man jetzt leicht nachvollziehen kann, würde der Algorithmus U_C für das Beispiel aus 12.1 $k(f(x, g(a, y)), g(x, h(y))) \stackrel{?}{=} k(f(h(y), g(y, a)), g(z, z))$ beide Lösungen $\sigma_1 = \{x \mapsto h(a), y \mapsto a, z \mapsto h(a)\}$ und $\sigma_2 = \{x \mapsto h(y), z \mapsto h(y)\}$ ausrechnen, wobei σ_2 allgemeiner ist als σ_1 . Das heißt der Algorithmus ist *nicht minimal*, er berechnet eventuell mehr Lösungen als notwendig. Da nur endlich viele Lösungen berechnet werden, kann man aber die überflüssigen im nachhinein ermitteln (mittels Unifikation der Termpaare $\sigma_1(x_i)$ und $\sigma_2(x_i)$ unter Kommutativität, wobei man auf einer Seite die Variablen als konstant ansieht. Danach kann man die nicht allgemeinsten Unifikatoren eliminieren. Für Kommutativität gibt es also im allgemeinen mehr als einen allgemeinsten Unifikator, aber immer noch endlich viele, wie man sich leicht überlegen kann und wie wir auch unten zeigen werden.

Es kann aber noch unangenehmer sein, wenn etwa eine Funktion f assoziativ ist, wenn also $\forall x, y, z : f(x, f(y, z)) = f(f(x, y), z)$ gilt. In diesem Fall hat etwa das Gleichungssystem $\{f(x, a) \stackrel{?}{=} f(a, x)\}$ unendlich viele unabhängige Lösungen:

$$\{x \mapsto f(a, a)\}, \{x \mapsto f(a, f(a, a))\}, \{x \mapsto f(a, f(a, f(a, a)))\}, \dots$$

Alle Lösungen sind im Sinne der Assoziativität von f gleich zu einer dieser Lösungen, d.h. der für x substituierte Term ist bis auf Umklammerung einer der hier angegebenen. Allerdings erspart man sich auch in diesem unangenehmen Fall gegenüber der Resolution (bzw. Paramodulation) ohne Theorieunifikation etwa das ständige Umklammern von Termen mit assoziativem Funktionssymbol.

Das folgende Beispiel zeigt, daß es neben den Fällen, daß es maximal einen, endlich viele oder unendlich viele allgemeinste Unifikatoren gibt, auch noch den Fall gibt, daß es für eine Theorie gar keine allgemeinsten Unifikatoren mehr gibt:

Beispiel 12.4. Es ist möglich, daß es in speziellen Fällen keine allgemeinsten Unifikatoren gibt.

Sei $L := \{\text{append}(x, \text{nil}) = x, \text{first}(\text{append}(x, y)) = \text{first}(x), \text{first}(\text{nil}) = \text{nil}\}$ die Menge der Axiome einer Theorie (eine Theorie für), wobei es eine Konstante **nil**, ein einstelliges Symbol **first** und ein zweistelliges Funktionssymbol **append** gibt. Dann hat das L -Unifikationsproblem $\{\text{first}(x) \stackrel{?}{=} \text{nil}\}$ die vollständige Menge von Unifikatoren:

$$cU_L(\text{first}(x) \stackrel{?}{=} \text{nil}) = \{\sigma_n \mid n \geq 0\}$$

mit

$$\begin{aligned}
\sigma_0 &= \{x \mapsto \mathbf{nil}\} \\
\sigma_1 &= \{x \mapsto \mathbf{append}(\mathbf{nil}, x_1)\} \\
\sigma_2 &= \{x \mapsto \mathbf{append}(\mathbf{append}(\mathbf{nil}, x_1), x_2)\} \\
&\dots \\
\sigma_n &= \{x \mapsto \mathbf{append}(\mathbf{append}(\dots(\mathbf{append}(\mathbf{nil}, x_1), x_2), \dots, x_n)\} \\
&\dots
\end{aligned}$$

Aber jeder Unifikator σ_n ist allgemeiner als der Vorgänger σ_{n-1} : Man substituiere x_n durch \mathbf{nil} und wende das erste Axiom aus L an, dann erhält man den Vorgänger. Man hat hier also eine Kette von immer allgemeiner werdenden Unifikatoren. Diese ist auch vollständig, aber kein Unifikator alleine ist bereits vollständig, also gibt es keine allgemeinsten.

Da die Verhältnisse offensichtlich ziemlich kompliziert sind, ist es nötig, Theorie-unifikation etwas formaler zu untersuchen und zu definieren. Wir haben jetzt den Fall, daß zwei Terme, z.B. $g(a, b)$ und $g(b, a)$ zwar syntaktisch verschieden sind, aber unter der Annahme der Kommutativität von g das gleiche bedeuten. D.h. es gibt eine Relation zwischen $g(a, b)$ und $g(b, a)$, die wir mit $g(a, b) =_C g(b, a)$ (C für Commutativity) bezeichnen wollen. (Für die beiden Terme $f(g(a, b))$ und $f(g(b, a))$ soll dann auch gelten $f(g(a, b)) =_C f(g(b, a))$, d.h. die Relation $=_C$ zieht sich hoch auf zusammengesetzte Terme.) Ähnliches gilt für Substitutionen: $\{x \mapsto g(a, b)\}$ und $\{x \mapsto g(b, a)\}$ sehen zwar auch verschieden aus, durch die Kommutativität von g bedeuten sie aber das gleiche, d.h. $\{x \mapsto g(a, b)\} =_C \{x \mapsto g(b, a)\}$.

Die formalen Definitionen dafür sind:

Definition 12.5. Eine Menge von Axiomen $E := \{l_1 = r_1, \dots, l_n = r_n\}$ von unären Klauseln mit dem Gleichheitsprädikat (Gleichheitsaxiome) induziert eine Gleichheitstheorie auf der Termalgebra $T(\Sigma, V)$ über einer Signatur Σ , die mindestens die Symbole der Gleichheitsaxiome in E enthält. Diese Gleichheitstheorie ist die kleinste Äquivalenzrelation $=_E$ auf $T(\Sigma, V)$ die alle Termpaare (l_i, r_i) für $l_i = r_i$ aus E enthält, und abgeschlossen ist gegenüber der Termbildung und der Substitution von Variablen:

1. Wenn $s_1 =_E t_1, \dots, s_n =_E t_n$ und f ist ein n -stelliges Funktionssymbol, dann $f(s_1, \dots, s_n) =_E f(t_1, \dots, t_n)$
2. Wenn $s =_E t$ und σ ist eine Substitution, dann $\sigma s =_E \sigma t$.

Eine solche Relation heißt auch Kongruenzrelation.

Zwei Terme sind genau dann in der Relation $=_E$, wenn man ihre Gleichheit aus den Axiomen E ableiten kann mit den folgenden Regeln:

1. $\vdash t = t$
2. $s = t \vdash t = s$
3. $s = r, r = t \vdash s = t$
4. $s_1 = t_1, \dots, s_n = t_n \vdash f(s_1, \dots, s_n) =_E f(t_1, \dots, t_n)$, wobei f ein n -stelliges Funktionssymbol.
5. $s = t \vdash \sigma t = \sigma s$ für eine beliebige Substitution σ .

Die Regeln sind so zu verstehen: Man startet mit E , den Axiomen, und erweitert diese Menge von Gleichungen unter Benutzung der Regeln, d.h. $E \subseteq A_1 \subseteq A_2 \subseteq \dots$ wobei jeweils eine Gleichung hinzugefügt wurde.

Der Satz von Birkhoff macht eine Aussage über die Vollständigkeit des Ableitungssystems. Hierbei ist \models als \models_E zu lesen.

Satz 12.6. Satz von Birkhoff: Für jede endliche Menge E von Gleichungen und zwei Terme s, t gilt:

$$E \models s = t \Leftrightarrow E \vdash s = t$$

Beweis. (siehe Literatur)

Beispiel 12.7. Sei $C := \{g(x, y) = g(y, x)\}$ die Theorie der Kommutativität. C -gleiche Terme sind dann:

$$g(a, b) =_C g(b, a) \text{ oder } f(x, g(a, b), z) =_C f(x, g(b, a), z).$$

Die Gleichheit von zwei Termen kann mit effizienteren Methoden überprüft werden. Die Idee ist, daß zwei Terme s und t gleich sind unter einer Theorie $TH(E)$, wenn sich s mittels Termersetzung in t überführen läßt, wobei die Axiome in E mitverwendet werden dürfen. Eine Darstellung der Termersetzung findet sich in [BN98, Ave95].

Definition 12.8. (Termersetzungskalkül, Rewrite-Kalkül)

Gegeben eine Menge E von Gleichungen (Axiomatisierung einer Theorie). Dann sei für einen Term s :

$$s \rightarrow_E s[\pi \rightarrow \sigma(r)]$$

genau dann wenn eine Gleichung $l = r$ oder $r = l$ in E existiert, eine Position π und eine Substitution σ mit $\sigma(l) = s|_\pi$. Die Termersetzungsrelation \Leftrightarrow_E ist der reflexive, transitive Abschluß von \rightarrow_E .

Als Beispiel betrachten wir den Term $f(x, g(a, b), z)$, wobei g ein kommutatives Symbol ist. Das Axiom der Kommutativität ist $g(x, y) = g(y, x)$. Mit $\sigma = \{x \mapsto a, y \mapsto b\}$ gilt: $\sigma g(x, y) = g(a, b)$. Mit einem Ersetzungsschritt kann der Term $f(x, g(a, b), z)$ in den Term $f(x, g(b, a), z)$ überführt werden. Die Position könnte man hier als [2] notieren. Also gilt: $f(x, g(a, b), z) \Leftrightarrow_E f(x, g(b, a), z)$. Der Termersetzungskalkül ist vollständig, d.h., er kann (unter einer Menge von Axiomen E) alle gleichen Term ineinander überführen:

Satz 12.9. Korrektheit und Vollständigkeit des Termersetzungs-kalküls:

1. *Der Termersetzungskalkül ist korrekt.*
2. *Ist E eine Menge von Axiomen und sind s und t zwei Terme mit $s =_E t$, dann gilt $s \Leftrightarrow_E t$.*

Beweis. Korrektheit (Übungsaufgabe).

Vollständigkeit: Zunächst muß man zeigen, daß eine leichte Abwandlung des Birkhoff-kalküls noch vollständig ist: Die Regel 5 wird ersetzt durch

$$s = t \vdash \sigma(s) = \sigma(t)$$

für eine beliebige Substitution σ und falls $s = t$ ein Axiom ist.

Induktion nach der Anzahl der Schritte einer Herleitung von $s =_E t$ mittels des obigen Kalküls. Hierbei betrachten wir nur die Anzahl der Schritte 1) bis 4).

Induktionsbasis: Länge = 0. Dann ist $s = t$ ein Axiom oder Instanz eines Axioms. Aus der Definition 12.8 folgt dann $s \rightarrow_E t$, insbesondere $s \Leftrightarrow_E t$.

Induktionsschritt: Hier muß man alle Fälle des Kalküls durchgehen und die Ersetzungsschritte jeweils zu einem ganzen Reduktionsbeweis zusammensetzen.

□

Definition 12.10. (E -Unifikationsprobleme) Ein Unifikationsproblem unter der Gleichheitstheorie E – ein E -Unifikationsproblem – ist ein Gleichungssystem $\Gamma = \{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n\}_E$. Eine Lösung von Γ ist eine Substitution σ mit $\sigma s_i =_E \sigma t_i$ (für alle i mit $1 \leq i \leq n$). Sie heißt E -Unifikator von Γ . Die Menge der E -Unifikatoren bezeichnen wir mit $U_E(\Gamma)$ oder auch $U_E(s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n)$.

Im allgemeinen sind die Mengen $U_E(\Gamma)$ unendlich und man möchte sie daher durch eine möglichst kleine repräsentative Teilmenge ersetzen. Bei der üblichen syntaktischen Unifikation – sie entspricht der Unifikation bezüglich der Theorie mit der leeren Axiomatisierung – gelingt dies sogar mit einelementigen Teilmengen, wie Robinson gezeigt hat (siehe auch 5.16).

Definition 12.11. Minimale und vollständige Mengen von E -Unifikatoren – auch Mengen allgemeinsten E -Unifikatoren (auch Lösungsbasis genannt) – $\mu U_E(\Gamma)$, sind Mengen von Unifikatoren, die den folgenden Bedingungen genügen:

(1) $\mu U_E(\Gamma) \subseteq U_E(\Gamma)$ (Korrektheit)

(2) Für alle $\delta \in U_E(\Gamma)$ existiert ein $\sigma \in \mu U_E(\Gamma)$

mit $\delta x =_E \lambda(\sigma x)$ (für alle $x \in V(\Gamma)$) (Vollständigkeit)

(3) Für alle $\sigma, \tau \in U_E(\Gamma)$ mit $\tau x =_E \lambda(\sigma x)$ (für alle $x \in V(\Gamma)$)

ist $\sigma = \tau$ (Minimalität)

Das heißt: (1) alle allgemeinsten E -Unifikatoren lösen Γ , (2) jeder beliebige E -Unifikator ist Instanz eines allgemeinsten E -Unifikators, und (3) verschiedene allgemeinste E -Unifikatoren sind keine Instanzen voneinander. Unifikatormengen $cU_E(\Gamma)$ mit (1) und (2) heißen auch vollständige Mengen von E -Unifikatoren. $U_E(\Gamma)$ selbst ist trivialerweise eine vollständige Lösungsmenge.

Für das Beispiel aus 12.1, $\Gamma = \{k(f(x, g(a, y)), g(x, h(y))) \stackrel{?}{=} k(f(h(y), g(y, a)), g(z, z))\}$ berechnet unser Unifikationsalgorithmus U_C die zwei Lösungen $\{x \mapsto h(a), y \mapsto a, z \mapsto h(a)\}, \{x \mapsto h(y), z \mapsto h(y)\}$, die zwar eine vollständige Menge bilden, aber nicht minimal ist. Die minimale Menge ist $\{x \mapsto h(y), z \mapsto h(y)\}$.

Mit den neuen Begriffen können wir jetzt Korrektheit, Terminierung und Vollständigkeit des Unifikationsalgorithmus U_C für Kommutativität beweisen.

Aussage 12.12. *Der Algorithmus U_C (Definition 12.2) ist korrekt, terminiert und ist vollständig.*

Beweis. Der Beweis geht jetzt ganz analog zum Beweis für den Algorithmus U_1 (Theorem 5.21). Es muß anstelle der syntaktischen Gleichheit die Kongruenzrelation $=_C$ benutzt werden. Wie im Beweis zur Vollständigkeit von U_1 zeigt man, daß für die Regeln Tautologie, Anwendung, Orientierung und Dekomposition für nicht C -Symbole keine Lösungen verlorengehen. Im Falle der Dekomposition und Vertauschung für C -Symbole gilt: $U_C(g(s_1, s_2) \stackrel{?}{=} g(t_1, t_2), \Gamma)$ und $U_C(s_1 \stackrel{?}{=} t_1, s_2 \stackrel{?}{=} t_2, \Gamma) \cup U_C(s_1 \stackrel{?}{=} t_2, s_2 \stackrel{?}{=} t_1, \Gamma)$ sind dieselben Mengen. Die Terminierung zeigt man analog zu der Terminierung des Algorithmus U_1 . Daß durch den Abbruch bei occurs-check keine Lösungen verlorengehen, folgt daraus, daß für Terme s und t mit $s =_C t$ gilt, daß s und t die gleiche Tiefe haben müssen, was mittels des Ersetzungskalküls in 12.8 bewiesen werden kann. \square

Da der Algorithmus vollständig ist, terminiert und der aufgebaute Baum nur eine endliche Verzweigungsrate hat, bedeutet das, daß es im Fall der Kommutativität immer maximal endlich viele allgemeinste Unifikatoren gibt.

Jetzt können wir allgemeine Theorieresolution unter Gleichheit definieren.

Definition 12.13. (Theorieresolution unter Gleichheit)

Vorausgesetzt man hat ein Verfahren, das für jedes Gleichungssystem die Lösungsmenge $U_E(\Gamma)$ – oder besser noch eine möglichst kleine repräsentative

Teilmenge $\mu U_E(\Gamma)$ - berechnet, dann kann man die Resolutionsregel wie folgt abwandeln:

$$\text{Klausel1:} \quad P(s_1, \dots, s_n), K_1, \dots, K_m$$

$$\text{Klausel2:} \quad \frac{\neg P(t_1, \dots, t_n), L_1, \dots, L_k \quad \sigma \in U_E(s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n)}{\quad}$$

$$E\text{-Resolvente: } \sigma K_1, \dots, \sigma K_m, \sigma L_1, \dots, \sigma L_k$$

12.1 Eigenschaften von Lösungsmengen

Die folgenden Theoreme zeigen, daß wir hier die “richtige” Form der Repräsentation gewählt haben. Die Eindeutigkeit garantiert, daß verschiedene minimale und vollständige Mengen von E -Unifikatoren eines Gleichungssystems Γ äquivalent im Sinne der folgenden durch wechselseitige Instanzenbildung induzierten E -Äquivalenz von Substitutionen sind: $\sigma =_E \tau \Leftrightarrow \exists \lambda, \mu : \sigma x =_E \lambda(\tau x)$ und $\tau x =_E \mu(\sigma x)$ für alle $x \in V(\Gamma)$, d.h. wenn jede eine Instanz der anderen ist. Die Aussage zur Repräsentation zeigt, daß die allgemeinsten Unifikatoren genau die Menge der Unifikatoren repräsentieren: Eine Substitution löst Γ genau dann, wenn sie Instanz eines allgemeinsten Unifikators ist. Die untenstehende Aussage zur Vererbung schließlich besagt, daß man die Lösungsbasen schrittweise berechnen kann: Um ein Gleichungssystem zu lösen, kann man die allgemeinsten Unifikatoren einer der Gleichungen berechnen, diese auf die anderen Gleichungen anwenden und die resultierenden Restsysteme analog lösen.

Aussage 12.14. Eindeutigkeit

1. Die Lösungsbasen eines Gleichungssystems Γ haben gleiche Kardinalität und unterscheiden sich höchstens um E -Äquivalenz ihrer Elemente.
2. Ersetzt man Elemente einer Lösungsbasis von Γ durch E -äquivalente Substitutionen, so erhält man wieder eine Lösungsbasis.

Aussage 12.15. Repräsentation:

1. Für jede vollständige Menge $cU_E(\Gamma)$ von E -Unifikatoren gilt: δ löst Γ genau dann, wenn $\delta x =_E \lambda(\sigma x)$ für alle $x \in V(\Gamma)$ mit $\sigma \in cU_E(\Gamma)$ und einer Substitution λ .
2. Die Lösungsbasen sind die kleinsten Mengen mit dieser Eigenschaft, d.h. wenn man eine Substitution aus einer Lösungsbasis entfernt, ist sie nicht mehr vollständig.

Aussage 12.16. Vererbung Sei $cU_E(\Gamma_1)$ eine vollständige Menge von E -Unifikatoren von Γ_1 und seien $cU_E(\sigma\Gamma_2)$ vollständige Mengen von E -Unifikatoren für Γ_2 für alle Lösungen $\sigma \in cU_E(\Gamma_1)$.

Dann ist die Menge $\{\tau\sigma \mid \tau \in cU_E(\sigma\Gamma_2), \sigma \in cU_E(\Gamma_1)\}$ eine vollständige Lösungsmenge für das System aus beiden Gleichungen $\Gamma_1 \cup \Gamma_2$

Wie wir bereits gesehen haben, ist es möglich, daß die Menge der allgemeinsten Unifikatoren unendlich groß ist (siehe das Assoziativitätsbeispiel von oben). Es kann sogar sein, daß überhaupt keine Menge allgemeinsten E -Unifikatoren existiert (Beispiel 12.4). D.h. die Forderung nach gleichzeitiger Vollständigkeit und Minimalität können unvereinbar sein.

12.2 Die Unifikationshierarchie

Eine der wichtigsten Aufgaben in der Unifikationstheorie ist es daher, für spezielle Theorien oder gar ganze Klassen von Theorien Existenztheoreme zu finden: Für die Theorie E oder die Klasse \mathcal{K} von Theorien existiert für jedes Unifikationsproblem stets eine Lösungsbasis. Theorien E , für die gewisse Gleichungssysteme keine Lösungsbasis besitzen, heißen auch vom *Unifikationstyp 0*. Man ist darüberhinaus daran interessiert, die Theorien, die nicht Typ 0 sind, zu klassifizieren: Wenn einige Unifikationsprobleme unendliche Lösungsbasen haben, ist die Theorie E vom Typ ∞ oder infinitär. Wenn die Lösungsbasen aller Gleichungssysteme endlich sind, dann ist E vom Typ ω oder *finitär*. Der beste Fall ist, wenn alle Lösungsbasen einelementig sind; dann ist E vom Typ 1 oder *unitär*. Die letzteren sind insbesondere in der Praxis von Bedeutung, da auch bei einer finitären Theorie die Anzahl der allgemeinsten Unifikatoren beliebig groß werden kann. Eine weitere Klassifikation kann man nach der Komplexität des Entscheidungsproblems durchführen: D.h. in welche Klasse fällt die Frage "Ist Γ bzgl. E unifizierbar?" Die Hauptaufgabe in der "praktischen" Unifikationstheorie ist es nun, für spezielle Theorien Unifikationsalgorithmen zu entwickeln. Das sind Algorithmen (oder Verfahren) die als Eingabe ein E -Unifikationsproblem erhalten und eine - möglichst minimale - Menge von E -Unifikatoren bezüglich der speziellen Theorie E zurückliefern.

Als Mindestvoraussetzung für eine Theorie stellt sich damit folgendes

Problem 1: (Unifizierbarkeitsproblem)

Ist die Unifizierbarkeit eines E -Unifikationsproblems in der gegebenen Theorie entscheidbar?

Daß dies im allgemeinen nicht der Fall ist, zeigt die bekannte Unentscheidbarkeit von Hilbert's zehntem Problem: Gibt es ein Verfahren das die Lösbarkeit einer Polynomgleichung über den ganzen Zahlen (Diophantische Gleichung) entscheidet? Antwort: nein. Für Implementierungen ist natürlich dann auch die Frage nach der Anzahl der zurückgelieferten Unifikatoren relevant. Ob überhaupt ein Algorithmus existieren kann oder nur ein Aufzählungsverfahren, folgt unmittelbar aus der Lage der zu untersuchenden Theorie in der Unifikationshierarchie.

Problem 2: (Hierarchieproblem)

Welchen Unifikationstyp hat die gegebene Theorie?

Schließlich stellt sich die letzte Frage fast von allein: Existiert überhaupt ein Algorithmus, der minimale Mengen von Unifikatoren berechnet bzw. aufzählt?

Hierbei wollen wir noch eine kleine Unterscheidung zwischen finitären und infinitären Theorien treffen. Ein E -Unifikationsalgorithmus heißt *typkonform*, wenn er eine Menge Ψ von E -Unifikatoren berechnet - oder aufzählt - mit:

- Ψ ist immer eine vollständige Menge von E -Unifikatoren
- der Algorithmus terminiert (mit endlichem Ψ), falls eine endliche, vollständige Lösungsmenge existiert
- Ψ ist eine Lösungsbasis, falls keine endliche, vollständige Menge existiert.

Mit anderen Worten: Ein typkonformer Algorithmus berechnet immer eine endliche, vollständige Lösungsmenge, falls eine solche existiert, oder er zählt eine unendliche, aber minimale auf. Natürlich darf die Theorie dann nicht vom Typ 0 sein. Insbesondere berechnen typkonforme Algorithmen für finitäre Theorien immer endliche, vollständige Mengen. Die Aufzählung einer unendlichen, vollständigen Menge wäre nicht sehr sinnvoll, da die gesamte Lösungsmenge bereits vollständig und aufzählbar ist. Ideal sind natürlich minimale Algorithmen, die immer minimale, vollständige Lösungsmengen berechnen oder aufzählen. Damit ergibt sich

Problem 3: (typkonformer Algorithmus)
 Gibt es für eine gegebene Theorie –die nicht vom Typ 0 ist – einen typkonformen beziehungsweise einen minimalen Algorithmus?

12.3 Einige Resultate für spezielle Theorien

Wir geben eine tabellarische Übersicht über einige Theorien, die bereits näher untersucht wurden. Es sind dies im wesentlichen die durch folgende Axiome definierten Theorien:

$\emptyset(f)$	$:= \emptyset$	(leere Theorie; freie Funktion)
$A(f)$	$:= \{f(x, f(y, z)) = f(f(x, y), z)\}$	(Assoziativität)
$C(f)$	$:= \{f(x, y) = f(y, x)\}$	(Kommutativität)
$I(f)$	$:= \{f(x, x) = x\}$	(Idempotenz)
$Dl(f, g)$	$:= \{f(g(x, y), z) = g(f(x, z), f(y, z))\}$	(Links-Distributivität)
$Dr(f, g)$	$:= \{f(x, g(y, z)) = g(f(x, y), f(x, z))\}$	(Rechts-Distributivität)
$D(f, g)$	$:= Dl(f, g) \cup Dr(f, g)$	(Distributivität)
$Inv_l(f, i, e)$	$:= \{f(i(x), x) = x\}$	(Links-Inverse)
$Inv_r(f, i, e)$	$:= \{f(x, i(x)) = x\}$	(Rechts-Inverse)
$Inv(f, i, e)$	$:= Inv_l(f, i, e) \cup Inv_r(f, i, e)$	(Inverse)
$Nl(f, e)$	$:= \{f(e, x) = x\}$	(Links-Neutrales)
$Nr(f, e)$	$:= \{f(x, e) = x\}$	(Rechts-Neutrales)
$N(f, e)$	$:= Nl(f, e) \cup Nr(f, e)$	(Neutrales)
$Iv(f)$	$:= \{f(f(x)) = x\}$	(Involution)
$E(h, f)$	$:= \{h(f(x_1, \dots, x_n)) = f(h(x_1), \dots, h(x_n))\}$	(Endomorphismus)
$AE(h, f)$	$:= \{h(f(x_1, \dots, x_n)) = f(h(x_n), \dots, h(x_1))\}$	(Anti-Endomorphismus)

Durch Kombinationen aus diesen Axiomen lassen sich bereits viele wichtige mathematische Theorien aufbauen: Monoide $(A \cup N)$, Gruppen $(A \cup N \cup Inv)$ und andere.

Spezielle Kombinationen, die in der Unifikationstheorie untersucht wurden, sind zum Beispiel

$$AC(f) := A(f) \cup C(f)$$

$$AI(f) := A(f) \cup I(f)$$

$$ACI(f) := A(f) \cup C(f) \cup I(f)$$

Folgende Tabelle spiegelt einige Eigenschaften dieser Theorien wider:

Theorie	entscheidbar	Typ	Algorithmus
\emptyset	ja	unitär	minimal
$A(f)$	ja	infinitär	typkonform
$C(f)$	ja	finitär	typkonform
$I(f)$	ja	finitär	typkonform
$AC(f)$	ja	finitär	minimal
$AI(f)$	ja	0	?
$D(f, g)$	ja	infinitär	typkonform*
$D(f, g) \cup A(f)$	nein	infinitär	typkonform*
$D(f, g) \cup AC(f)$	nein	infinitär	typkonform*
$E(f, g)$	ja	unitär	minimal
$H(f, g_1, g_2)$	ja	unitär	minimal
Minus-Theorien	ja	(in)finitär**	typkonform
Abelsche Gruppen	ja	finitär	minimal
Boolesche Ringe	ja	unitär**	minimal

12.4 Theorieresolution mit “compilierten Theorien“

Die Idee der compilierten Theorien ist, bestimmte Klauseln aus der Klauselmengen herauszunehmen und durch eine spezielle Theorieresolutionsregel zu ersetzen.

Beispiel 12.17. Die Symmetrieklausel: $\neg P(x, y), P(y, x)$ erlaubt, die beiden unären Klauseln $P(a, b)$ und $\neg P(b, a)$ durch zwei aufeinanderfolgende Resolutionsschritte zu widerlegen. Viel einfacher ist es jedoch, einen Unifikationsalgorithmus zu benutzen, der die Symmetrie direkt ausnutzt und dann mit einem Schritt die Widerlegung findet.

Definition 12.18. *Compilierte Theorieresolution.* Sei $C = \{L_1, \dots, L_n\}$ eine Klausel mit zwei Literalen, die nicht selbstresolvierend ist, d.h. nicht mit einer Kopie von sich selbst resoltiert. Dann läßt sich daraus folgende Theorieresolutionsregel erzeugen:

$$\begin{array}{ll}
 K_1 \vee R_1 & L_i \text{ und } K_i \text{ haben unterschiedliches Vorzeichen.} \\
 \dots & \sigma \text{ ist simultaner Unifikator für } |L_i|^1 \stackrel{?}{=} |K_i|, i = 1, \dots, n, \text{ d.h.} \\
 \dots & |\sigma L_1| = |\sigma K_1| \\
 \dots & |\sigma L_n| = |\sigma K_n| \\
 \hline
 K_n \vee R_n & \sigma' \text{ ist } \sigma \text{ eingeschränkt auf die Variablen in } K_1, \dots, K_n \\
 \hline
 \sigma'(R_1 \vee \dots \vee R_n) & (\text{Theorieresolvente in der Theorie von } C)
 \end{array}$$

Beispiel 12.19.

1. Sei $C = \{\neg P(x, y), P(y, x)\}$ (Die Selbstresolvente² ist eine Tautologie und kann daher ignoriert werden.)

$$\begin{array}{ll}
 P(a, z), Q(z) & \sigma = \{x \mapsto a, y \mapsto b, z \mapsto b\} \\
 \neg P(b, a) & \sigma' = \{z \mapsto b\} \\
 \hline
 Q(b)
 \end{array}$$

2. Sei $C = \{\neg \text{links}(x, y), \text{rechts}(y, x)\}$

¹ $|L|$ bedeutet L ohne Vorzeichen, d.h. den atomaren Anteil von L .

² Das Ergebnis der Resolution einer Klausel mit sich selbst (bzw. einer Kopie)

$$\begin{array}{lcl}
links(a, z), Q(z) & \sigma = \{x \mapsto a, y \mapsto b, z \mapsto b\} & \\
\neg rechts(b, a) & \sigma' = \{z \mapsto b\} & \\
\hline
Q(b) & &
\end{array}$$

3. Sei $C = \{\neg \text{Vater}(x, y), \neg \text{Vater}(y, z), \text{Grossvater}(x, z)\}$

$$\begin{array}{lcl}
\text{Vater}(Uli, Karl), S & \sigma = \{x \mapsto Uli, y \mapsto Karl, z \mapsto Willi, u \mapsto Willi\} & \\
\text{Vater}(Karl, Willi), R & \sigma' = \{u \mapsto Willi\} & \\
\neg \text{Grossvater}(Uli, u), Q(u) & & \\
\hline
S, R, Q(Willi) & &
\end{array}$$

Die Korrektheit des Verfahrens ergibt sich unmittelbar aus der Tatsache, daß ein solcher Theorieresolutionsschritt sich in eine Folge von n einzelnen Resolutionsschritten zerlegen läßt, die wiederum korrekt sind.

Vorteile des Verfahrens: Größere Schritte, zielgerichteteres Vorgehen.

Nachteile: Eventuell größere Verzweigungsrate im Suchraum und damit eventuell mehr Suche.

Das folgende Beispiel zeigt, was passieren kann wenn man die Forderung, daß die Klausel C nicht selbstresolvierend ist, fallen läßt.

Beispiel 12.20. Gegenbeispiel:

$C = \{\neg P(x), P(f(x))\}$ Die Klauseln $P(a)$ und $\neg P(f(f(a)))$ sind zusammen mit C unerfüllbar und auch widerlegbar mit normaler Resolution, jedoch nicht mit der kompilierten Theorieresolution: (Die Selbstresolventen sind: $\{\neg P(x), P(f(f(x)))\}$, $\{\neg P(x), P(f(f(f(x))))\}$, \dots

Mit der ersten Selbstresolvente würde es funktionieren. Dann funktionieren aber andere Beispiele nicht.)

Es bleibt noch das Problem, wie man mehrere Klauseln gleichzeitig kompiliert. Dafür gilt:

Satz 12.21. *Gegeben eine endliche konsistente Klauselmengende D , die alle ihre Resolventen und Faktoren enthält. Dann kann man alle diese Klauseln einzeln in eine Theorieresolutionsregel übersetzen und erhält damit einen korrekten und vollständigen Kalkül.*

Beweis. Der Beweis geht ganz analog zum Vollständigkeitsbeweis für Resolution: Man zeigt zuerst die Vollständigkeit auf der (variablenfreien) Grundebene und

“liftet“ dann die Schritte. Der einzige wesentliche Unterschied der sich ergibt, ist der Basisfall in der “literal excess number“ Induktion (vgl. 5.5). In diesem Fall ist die Menge der unären Grund Klauseln C_{gr} nicht mehr notwendigerweise allein dadurch widersprüchlich weil sie zwei komplementäre Klauseln enthält, sondern auch weil sie mit der Menge D zusammen widersprüchlich ist. Da D vervollständigt ist bzgl. Resolventen und Faktorenbildung, genügt jedoch eine Klausel A aus D , um mit der passenden Anzahl von unären Klauseln aus C_{gr} zur leeren Klausel zu resolvieren. Diese Folge von Resolutionen läßt sich jetzt als ein Theorieresolutionsschritt modulo A interpretieren. Der Rest des Beweises geht ganz analog zur normalen Resolution. \square

Beispiel 12.22. Gegeben seien die Klauseln $\{P(x), Q(x)\}$ und $\{\neg Q(y), R(y)\}$. Eine daraus vervollständigte Menge ist: $\{\{P(x), Q(x)\}, \{\neg Q(y), R(y)\}, \{\neg P(z), R(z)\}\}$.

Jetzt kann man damit die folgende Klauselmengewe widerlegen:

$$\neg P(a), Q(b), P(c)$$

$$\neg Q(a)$$

$$\neg R(b)$$

$$\neg R(c)$$

12.5 Resolution mit bedingten Klauseln (Constraints)

Die Idee der Theorieresolution kann man verallgemeinern zur Resolution mit eingeschränkten Quantoren oder auch Constraint-Resolution (siehe [Bür91]). Wenn die Hintergrundtheorie eine Gleichungstheorie ist, dann ist es damit möglich trotz einer großen oder unendlichen Anzahl von Unifikatoren die Verzweigungsrate des Suchraumes klein zu halten, indem man andere Darstellungen für die Menge der Unifikatoren wählt. Eine solche ist zum Beispiel das Gleichungssystem selbst, das ursprünglich zu lösen war.

Statt Klauseln nimmt man bedingte Klauseln bestehend aus einem Paar (\mathcal{C}, C) , wobei \mathcal{C} eine *Bedingung (constraint)* ist und C eine normale Klausel. Wir wollen nur den einfachen Fall betrachten, daß die Prädikate der Theorie in dem Literalteil C der Klauseln nicht vorkommen. Den Constraintteil kann man sich als eine PL_1 -Formel vorstellen. Im Fall der Gleichungstheorien wäre zum Beispiel ein Gleichungssystem $\{x \stackrel{?}{=} f(y), g(x) \stackrel{?}{=} g(y)\}$ eine solche Bedingung. Die logische Bedeutung einer verallgemeinerten Klausel (\mathcal{C}, C) ist einfach $\forall(V)(\mathcal{C} \Rightarrow C)$ ³, wobei V die Menge aller in (\mathcal{C}, C) freien Variablen ist. Klauseln, deren Constraint-Anteil unerfüllbar ist, d.h. $\forall(V)\neg\mathcal{C}$ gilt, sind Tautologien. Um

³ $\forall(V)$ soll die Quantifikation über alle Variablen in V bezeichnen

diese Klauseln sicher zu erkennen, ist ein Entscheidungsverfahren für die Konsistenz von Bedingungen erforderlich. Die leere Klausel ist (\mathcal{C}, \emptyset) , wobei \mathcal{C} ein konsistenter Constraint sein muß.

Ein (einfacher) *bedingter Resolutionsschritt* sieht dann wie folgt aus: $(\mathcal{C}_1, \{L_1\} \cup C_1)$ und $(\mathcal{C}_2, \{\neg L_2\} \cup C_2)$ haben als Resolvente: $(\mathcal{C}_1 \wedge \mathcal{C}_2 \wedge \{L_1 \stackrel{?}{=} L_2\}, C_1 \cup C_2)$. Ein verallgemeinerter Resolutionsschritt schließt die Faktorisierung mit ein, indem statt L_1 und L_2 mehrere Literale gleichzeitig an der Resolution teilnehmen können.

Es sieht so aus, als sei Unifikation und Variableneinsetzung unnötig geworden. Dies ist nicht der Fall. Man hat aber etwas gewonnen: um einen Resolutionsschritt auszuführen, braucht man nur zu testen, ob die zugehörige Bedingung konsistent ist.

Die bedingte Resolutionsmethode ist am einfachsten und anschaulichsten für sogenannte *definite Theorien erster Ordnung*, das sind Theorien, die mit einer endlichen Hornklauselmengemenge (siehe Kapitel. 8) definiert werden können. In diesen gilt, daß die Bedingungen Lösungen haben, die als (evtl. unendliche) Menge von Substitutionen beschrieben werden können. Eine etwas allgemeinere Formulierung ist, daß die Theorie ein *generisches Modell* \mathcal{G} hat, d.h. für jede Bedingung \mathcal{C} gilt $\mathcal{G} \models \exists \mathcal{C}$ gdw. für alle Modelle $\mathcal{M} : \mathcal{M} \models \exists \mathcal{C}$. Für solche Theorien ist die normale bedingte Resolution vollständig:

Satz 12.23. *Für Theorien erster Ordnung mit generischem Modell kann allgemeine bedingte Resolution aus jeder unerfüllbaren bedingten Klauselmengemenge die leere Klausel ableiten.*

Eine Theorie, die nicht definit ist, ist z.B. die durch $Pa \vee Pb$ definierte Theorie. Den Begriff einer Lösung für die Bedingung $\{P(x)\}$ läßt sich nicht mehr so formulieren wie man das für Unifikatoren gewöhnt ist. Was man noch sagen kann ist: $\{P(x)\}$ ist äquivalent zu $\{x \stackrel{?}{=} a \vee x \stackrel{?}{=} b\}$. Dies kann man als “indefinite Lösung” ansehen. Die Konsistenzbedingung für Bedingungen in indefiniten Theorien ist nur, daß die Bedingung in einem Modell konsistent ist. Für solche Theorien gilt der folgende Vollständigkeitssatz:

Satz 12.24. *Für eine unerfüllbare Menge von bedingten Klauseln lassen sich mittels bedingter Resolution endlich viele leere Klauseln $(\mathcal{C}_1, \emptyset), \dots, (\mathcal{C}_n, \emptyset)$ ableiten, so daß die Konjunktion $(\mathcal{C}_1 \wedge \dots \wedge (\mathcal{C}_n \text{ unerfüllbar ist, d.h. man hat einen Widerspruch.}$*

Für ein Deduktionssystem für bedingte Resolution bedeutet das, daß nicht nur eine leere Klausel zu suchen ist, sondern daß viele leere Klauseln abzuleiten und zu sammeln sind und die Konjunktion der Constraints auf Erfüllbarkeit zu testen. Bisher hat der allgemeine Fall der bedingten Resolution für indefinite Theorien keine praktische Bedeutung erlangt. Implementierungen gibt es jedenfalls noch keine.

Beispiel 12.25. Sei f ein assoziatives Funktionssymbol mit dem Axiom $f(x, f(y, z)) = f(f(x, y), z)$ und sei folgende Klauselmengemenge gegeben:

$$C1 (\emptyset, Qf(a, b))$$

$$C2 (\emptyset, \neg Qx, Pf(x, y))$$

$$C3 (\emptyset, \neg P(f(a, f(b, z))))$$

Bedingte Resolution auf $C2, 2$ und $C3$ ergibt:

$$R1 \quad (\{f(x, y) \stackrel{?}{=} f(a, f(b, z))\}, \neg Q(x)) \quad \text{Bedingung ist erfüllbar}$$

$$R1 + C1 (\{f(x, y) \stackrel{?}{=} f(a, f(b, z)), x \stackrel{?}{=} f(a, b)\}, \emptyset) \quad \text{Bedingung ist erfüllbar mit}$$

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

Hätte man assoziative Unifikation und Resolution verwendet, dann hätte man eine Menge der allgemeinsten Unifikatoren von $f(x, y) \stackrel{?}{=} f(a, f(b, z))$ betrachten müssen. Dies sind unter anderem: $\{x \mapsto a, y \mapsto f(b, z)\}, \{x \mapsto f(a, b), y \mapsto z\}, \dots$

Mittels bedingter Resolution lassen sich auch ganzzahlige Arithmetik oder ähnlich bekannte Theorien in einem Deduktionssystem verwenden. Dies sind dann evtl. noch definite Theorien, aber sie sind nicht unbedingt erster Ordnung, denn z.B. die Menge der natürlichen Zahlen läßt sich nicht mit einer Theorie erster Ordnung charakterisieren. Bei der bedingten Resolution tritt dann das Problem auf, daß die Konsistenz von Bedingungen möglicherweise nicht mehr entscheidbar ist.

13 Erweiterungen der Prädikatenlogik

In diesem Kapitel werden Erweiterungen der Prädikatenlogik vorgestellt, die es erlauben, einerseits bestimmte Sachverhalte einfacher, bzw. überhaupt auszudrücken und andererseits den Deduktionskalkül anzupassen.

13.1 Mehrsortige Logiken

Es gibt syntaktische Varianten von PL_1 , die zwar nicht die prinzipielle Ausdrucksfähigkeit erhöhen, aber eine kompaktere Darstellung von Formeln erlauben ([Wal87, SS89, Wei95, Wei96]). Damit ist in den einzelnen Unterausdrücken mehr Information konzentriert, die dann von Kalkülen ausgenutzt werden kann, um sinnlose Aussagen zu erkennen und zu vermeiden. Die wichtigsten derartigen Varianten sind die *mehrsortigen Logiken*. Die Grundidee dabei ist, nicht einfach ein strukturloses Universum zu betrachten, sondern die Objekte in Klassen einzuteilen. Aussagen werden nur für Objekte bestimmter Klassen formuliert und sind für andere Objekte gar nicht definiert. Beispielsweise könnte man Aussagen über Menschen und über natürliche Zahlen in einer mehrsortigen Logik so formulieren, daß die Behauptung, Sokrates und Napoleon seien teilerfremd, nicht einfach wahr oder falsch, sondern sinnlos wäre.

Dazu stellen mehrsortige Logiken zusätzlich zu den genannten Datenobjekten noch eine Menge von *Sorten* zur Verfügung. Die Sorten können einfach eine unstrukturierte Menge bilden (flache Sortenstruktur) oder durch eine partielle Ordnung \sqsubseteq , die *Untersortenbeziehung*, geordnet sein (hierarchische Sortenstruktur). Man würde etwa die Menge der Sortensymbole $\{Integer, Menschen\}$ flach und die Menge $\{Integer, Real, Complex\}$ hierarchisch strukturieren. Die zulässige Struktur der Untersortenbeziehung, z.B. linear, baumförmig oder Halbverband hat Einfluß auf die möglichen Einsetzungen für Variablensymbole und beeinflußt daher die Konstruktion von Kalkülen für die jeweilige spezielle Logik.

Die Signatur der Logik wird nun folgendermaßen erweitert: jedem Konstanten- und Variablensymbol ist eine Sorte zugeordnet, Funktionssymbolen werden Argument-/ Ergebnissorten- Beziehungen mit der entsprechenden Stelligkeit zugeordnet, Prädikatensymbolen die zulässigen Argumentsorten. Beispielsweise spezifiziert

$$Geburtsjahr : Mensch \rightarrow Integer$$

eine Funktion, die nur Argumente der Sorte *Mensch* zuläßt und dann ein Ergebnis der Sorte *Integer* liefert. Mit

$$+ : (Real \times Real \rightarrow Real$$

$$Real \times Integer \rightarrow Real$$

$$Integer \times Real \rightarrow Real$$

$$Integer \times Integer \rightarrow Integer$$

charakterisiert man mehrere mögliche Beziehungen der Argumentsorten zu den Ergebnissorten der Funktion $+$. *Teilerfremd* : *Integer* \times *Integer* besagt, daß das Prädikatsymbol *Teilerfremd* nur Terme vom Typ *Integer* als Argument akzeptiert.

Die Sorte eines zusammengesetzten Terms wird nun aus der Sortendeklaration für das oberste Funktionssymbol und eventuell aus den Sorten der Unterterme berechnet. Ein korrekter Term *Geburtsjahr*(...) hat zum Beispiel immer die Sorte *Integer*, während die Sorte eines Terms $+(\dots, \dots)$ von den Sorten der Argumente abhängt. Ein Term wie *Geburtsjahr*($+(\dots, \dots)$) ist nicht *sortenrecht* und daher verboten. Damit wird rein syntaktisch die Formulierung vieler unsinniger Aussagen verhindert.

Daß mehrsortige Logiken nicht wirklich ausdrucksstärker sind als unsortierte, zeigt sich darin, daß die gesamte Information über die Sorten in einstellige Prädikate kodiert werden kann: Die Untersortenbeziehungen $S \sqsubseteq P$ werden als Implikationen ausgedrückt: $\forall x : S(x) \Rightarrow P(x)$. Quantifizierungen über sortierte Variable $\forall x : S : \mathcal{F}$ werden in die Form $\forall x : S(x) \Rightarrow \mathcal{F}$ übersetzt, und die Argument/Ergebnissorten-Beziehungen der Funktionen $f : S_1 \times \dots \times S_n \rightarrow S$ lassen sich durch Formeln $\forall x_1 \dots x_n : S_1(x_1) \wedge \dots \wedge S_n(x_n) \Rightarrow S(f(x_1, \dots, x_n))$ beschreiben. Semantisch unsinnige Terme wie der oben erwähnte Term *Geburtsjahr*($+(\dots, \dots)$) sind in dieser Form jedoch nicht ausgeschlossen, deshalb setzten sich mehrsortige Logiken in der Praxis immer mehr durch.

Beispiel 13.1. Ein Beispiel, welches zeigt, wie per Resolution in unsortierter Prädikatenlogik ziemlicher Unsinn abgeleitet werden kann:

Axiome:

Jeder Mensch, der Bananen ißt, lebt gesund. (Ob das stimmt oder nicht steht hier nicht zur Diskussion)

Judy ist ein Affe und ißt Bananen

Tom ist ein Mensch und ißt Bananen.

Kodierung in PL_1 :

$$\forall x : \text{Mensch}(x) \wedge \text{isst}(x, \text{Bananen}) \Rightarrow \text{lebtgesund}(x)$$

$$\text{Affe}(\text{Judy})$$

$$\text{isst}(\text{Judy}, \text{Bananen})$$

$$\text{Mensch}(\text{Tom})$$

$$\text{isst}(\text{Tom}, \text{Bananen})$$

Klauselform:

$C1 : \neg \text{Mensch}(x), \neg \text{isst}(x, \text{Bananen}), \text{lebtgesund}(x)$

$C2 : \text{Affe}(\text{Judy})$

$C3 : \text{isst}(\text{Judy}, \text{Bananen})$

$C4 : \text{Mensch}(\text{Tom})$

$C5 : \text{isst}(\text{Tom}, \text{Bananen})$

Jetzt ist z.B. eine Resolution mit $C1, 2$ und $C3$ möglich. Die Resolvente ist: $\neg \text{Mensch}(\text{Judy}), \text{lebtgesund}(\text{Judy})$.

Um damit weiterzuarbeiten, müßte man einen Resolutionspartner für $\neg \text{Mensch}(\text{Judy})$ finden, d.h. eine Klausel mit einem Literal $\text{Mensch}(\text{Judy})$, was offensichtlich in der intendierten Bedeutung nicht möglich ist. D.h. die Resolvente ist nutzlos. Um diese Resolution zu verhindern, muß man die Variablen typisieren (mit Sorteninformation versehen), so daß eine Einsetzung $x \mapsto \text{Judy}$, wobei x die Sorte **Mensch** und **Judy** die Sorte **Affe** hat, verboten werden kann.

Jetzt zu den formalen Definitionen. Wir definieren eine einfache Variante von Sortenlogik erster Ordnung: OSPL (*Order Sorted Predicate Logic*).

13.2 Syntax von OSPL

Wir halten uns an das Schema wie es auch in Kapitel 4.1 zur Definition von PL_1 benutzt wurde. D.h. wir sagen zunächst mal, was die Signatur, d.h. die verwendeten Symbole sind. Im Unterschied zu unsortierten Signaturen nehmen wir die Variablen diesmal mit in die (sortierte) Signatur auf.

Definition 13.2. Signaturen mit Sorten

Eine sortierte Signatur S besteht aus den Komponenten $(V, F, P, S, \mathcal{S}, SD, FD, PD)$. V, F, P sind die Variablensymbole, Funktionsymbole und Prädikatensymbole, so wie in Definition 4.1. Die weiteren Komponenten sind:

- S ist eine Menge von Sortensymbolen
- $\mathcal{S} : V \rightarrow S$ ist eine Funktion, die jedem Variablensymbol eine Sorte zuordnet. Meist schreibt man jedoch z.B. $x : A$ anstelle von $\mathcal{S}(x) = A$ um zu sagen, daß die Variable x die Sorte A hat.
- SD ist eine Menge von Untersortendeklarationen der Form $R \sqsubseteq S$,
- FD ist eine Menge von Sortendeklarationen für Funktionen. Eine Sortendeklaration für Funktionen ist einfach ein Tupel $t : S$ welches besagt, daß alle Instanzen des Terms t von der Sorte S sind, wobei nur Terme der Form $f(x_1, \dots, x_n)$ mit verschiedenen x_i zugelassen sind. Oft werden Deklarationen $f(x_1, \dots, x_n) : S$ als $f : S_1 \times \dots \times S_n \rightarrow S$ geschrieben, wobei S_i die Sorte der Variable x_i ist.

- PD ist eine Menge von Sortendeklarationen für Prädikatensymbole. Sie haben die Form $P : S_1 \times \dots \times S_n$, wobei die S_i Sorten sind.

Wir nehmen an, daß das Gleichheitsprädikat immer vorhanden ist und daß die Deklarationen $= : R \times S$ für alle Sorten R, S in PD sind, d.h. Gleichheit ist für alle Sorten definiert. Für eine Signatur Σ , sei \sqsubseteq_Σ die partielle Ordnung auf S , die durch die reflexive und transitive Hülle der Untersortendeklarationen gegeben ist.

Beispiel 13.3. Beispiel für eine sortierte Signatur:

Variablensymbole: $\{x, y, z, u, v, w \dots\}$

Funktionssymbole: $\{+, *, inv, o, Fido, Tweety, Vater\},$

$arity(+)=arity(*)=2, arity(inv)=1, arity(Fido)=arity(o)=0$

$arity(Tweety)=0, arity(Vater)=1$

Prädikatensymbole: $\{<, gerade, brav, =\}$

Sortensymbole: $\{komplex, reell, nat, gaussch, Hund, Tier\}$

Sortenfunktion: $\mathcal{S} : \mathcal{S}(x) = komplex, \mathcal{S}(y) = reell, \mathcal{S}(z) = nat, \mathcal{S}(u) = Hund \dots$

Untersorten: $reell \sqsubseteq komplex, gaussch \sqsubseteq komplex, nat \sqsubseteq gaussch, nat \sqsubseteq reell, Hund \sqsubseteq Tier$

Funktionen: $o : nat$

$+: komplex \times komplex \rightarrow komplex$

$reell \times reell \rightarrow reell$

$nat \times nat \rightarrow nat$

$* : komplex \times komplex \rightarrow komplex$

$reell \times reell \rightarrow reell$

$nat \times nat \rightarrow nat$

$inv : komplex \times komplex \rightarrow komplex, \dots$

$Fido : Hund; Tweety : Tier; Vater : Hund \rightarrow Hund \dots$

Prädikate: $gerade : nat; < : komplex \times komplex; brav : Hund \dots$

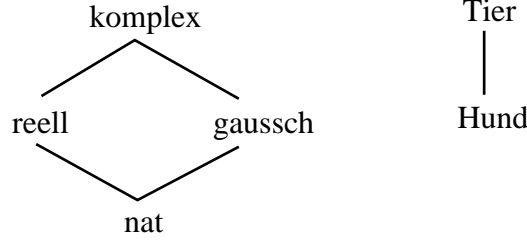
Aus den Untersortendeklarationen ergibt sich z.B.:

$komplex \sqsubseteq_{\Sigma} komplex, \dots$ (reflexive Hülle)

$reell \sqsubseteq_{\Sigma} komplex, \dots$ (war sowieso schon der Fall)

$nat \sqsubseteq_{\Sigma} komplex, \dots$ (transitive Hülle)

Meist stellt man das auch graphisch dar:



In der so definierten Signatur steckt Information, die man sonst explizit mit Formeln ausdrücken müßte. Z.B. ist die Deklaration $reell \sqsubseteq komplex$ ausdrückbar in unsortiertem PL_1 mit $\forall x : reell(x) \Rightarrow komplex(x)$.

Als nächstes müssen wir die Terme neu definieren. Dabei sollen Terme der Art $inv(Fido)$ oder $Vater(*(o, o))$ als “nicht wohlsortiert” ausgesondert werden. Das macht man, indem man positiv die Menge der wohlsortierten Terme definiert. Die Idee dafür ist, von den Deklarationen für Terme auszugehen, die sind allemal wohlsortiert, und daraus durch wohlsortierte Instantiierung weitere wohlsortierte Terme zu generieren.

Definition 13.4. Wohlsortierte Terme Die Menge der wohlsortierten Terme $T_{\Sigma, A}$ der Sorte A in der Signatur $\Sigma = (V, F, P, S, \mathcal{S}, SD, FD, PD)$ wird durch folgende drei Regeln rekursiv konstruiert:

- $x \in T_{\Sigma, A}$, falls $\mathcal{S}(x) \sqsubseteq_{\Sigma} A$.
- $f(x_1, \dots, x_n) \in T_{\Sigma, A}$, falls mit $x_i : S_i$ und $f : S_1 \times \dots \times S_n \rightarrow A \in FD$ und $R \sqsubseteq_{\Sigma} A$
- $t[r/x] \in T_{\Sigma, A}$ falls $t \in T_{\Sigma, A}$, $r \in T_{\Sigma, R}$ und $x \in V, x : S$ so daß $R \sqsubseteq_{\Sigma} S$.⁴

Die Menge T_{Σ} aller wohlsortierten Terme über Σ (Σ -Terme) erhält man als Vereinigung:

$$T_{\Sigma} = \bigcup \{T_{\Sigma, A} \mid A \in S\}$$

Die Sorte $S(t)$ eines Terms t ist die Sorte A mit kleinster Menge (als Mengeneinklusion) $T_{\Sigma, A}$, welches t enthält. Wenn diese Sorte nicht eindeutig ist, liefert $S(t)$ die Menge aller Sorten A , so daß $t \in T_{\Sigma, A}$.

Manchmal schreibt man auch $t : A$, um auszudrücken, daß A die Sorte von t ist.

⁴ $t[r/x]$ bedeutet, r für x in t einsetzen.

Beispiel 13.5. In der Signatur von Beispiel 13.3 ist:

$$\begin{aligned}
T_{\Sigma, Hund} &= \{Fido, Vater(Fido), Vater(Vater(Fido)), \dots \\
&\quad u, Vater(u), Vater(Vater(u)), \dots\} \\
T_{\Sigma, Tier} &= T_{\Sigma, Hund} \cup \{Tweety\} \\
T_{\Sigma, nat} &= \{o, +(o, o), *(o, o), +(o, *(o, o)), \dots, \\
&\quad z, +(z, z), +(o, z), \dots\} \\
T_{\Sigma, reel} &= T_{\Sigma, nat} \cup \{y, +(y, y), \dots\} \\
S(Vater(Fido)) &= Hund
\end{aligned}$$

Im nächsten Schritt werden die zulässigen Formeln definiert. Der Unterschied zu PL_1 ist im wesentlichen, daß die Sortendeklarationen für Prädikate benutzt werden, um nicht wohlsortierte Atome wie z.B. $brav(+(o, o))$ im obigen Beispiel auszuschließen.

Definition 13.6. Wohlsortierte Formeln

Ein Atom $P(t_1, \dots, t_n)$ ist wohlsortiert, falls $t_i \in T_{\Sigma, S_i}$ für $i = 1, \dots, n$ und $P : S_1 \times \dots \times S_n$ ist in Σ . Eine wohlsortierte Formel ist eine Formel, die mit wohlsortierten Atomen und den logischen Junktoren und Quantoren $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \forall, \exists$ wie üblich gebildet wird. Wir schreiben $\forall x : A : F$ und $\exists x : A : F$, um anzuzeigen, daß A die Sorte der Variablen x ist.

Beispiel 13.7. In der Signatur von 13.3 sind die Formeln $< (o, o), \forall x : komplex : < (x, x)$ wohlsortiert, während $< (Fido, Tweety)$, $brav(o) \neg x : komplex : brav(x)$ nicht wohlsortiert sind.

13.3 Semantik von OSPL

In Abschnitt 4.2 haben wir die Semantik von PL_1 über Σ -Algebren und Σ -Homomorphismen definiert. Mit den Erweiterungen von OSPL gegenüber PL_1 muß man auch die Definitionen von Σ -Algebren und Σ -Homomorphismen entsprechend anpassen. Insbesondere benötigen wir in der Semantik Gegenstücke zu den Sortensymbolen, Untersortendeklarationen und Deklarationen für Terme und Prädikatensymbolen.

Wir verzichten hier auf weitergehende Formalisierung und geben nur die Idee an: die Sortensymbole als Teilmengen der Trägermenge zu interpretieren, die Untersortenbeziehungen als Teilmengenbeziehungen und die Deklarationen im wesentlichen als Einschränkungen über Definitions- und Wertebereiche von Funktionen aufzufassen. Das heißt, daß Interpretationen und Modelle der Sortensignatur angepaßte Bedingungen erfüllen müssen. Eine wichtige Einschränkung

ist, daß die Einsetzungen für Variablen deren Sorte entsprechen muß. D.h. wenn $x : A$, dann darf nur ein Element d aus dem Domain für x eingesetzt werden, das zur Menge D_A gehört. Dies betrifft zum Beispiel die Variablen die durch Quantoren gebunden sind. Die Definitionen der Relation der Erfüllbarkeit die gleiche wie im unsortierten Fall.

13.4 Relativierung

Das erste was sich Logiker fragen, wenn sie mit einer neuen Logik konfrontiert sind, ist, ob diese neue Logik wirklich ausdrucksstärker ist als schon vorhandene, d.h. ob man darin Sachverhalte ausdrücken kann, die man in anderen Logiken, insbesondere einfacher Prädikatenlogik, prinzipiell nicht ausdrücken kann.

Wenn der Verdacht besteht, daß das nicht der Fall ist, kann man diese Vermutung dadurch bestätigen, daß man eine korrekte und vollständige Methode angibt, Formeln dieser Logik in eine andere Logik zu übersetzen (zu *relativieren*). Korrekt und vollständig heißt in diesem Zusammenhang, daß eine erfüllbare Formel der Ausgangslogik in eine erfüllbare Formel der Ziellogik übersetzt wird (Korrektheit) und umgekehrt, wann immer die übersetzte Formel erfüllbar ist, ist auch die Ausgangsformel erfüllbar (Vollständigkeit). Solche Übersetzer bezeichnet man auch manchmal als konservative Transformationen.

Hat man einen Übersetzer dieser Art definiert, dann kann man im Prinzip auf einen Kalkül in der neuen Logik verzichten. Man übersetzt in die Ziellogik und beweist die übersetzte Formel. Dann weist man, daß die ursprüngliche Formel ebenfalls gültig ist. Ob diese Vorgehensweise zweckmäßig ist, hängt von praktischen Gesichtspunkten ab - wie effizient die Kalküle sind, ob Implementierungen von Beweisern zur Verfügung stehen, usw. Für OSPL geben wir jetzt eine Übersetzung in unsortierte Prädikatenlogik an. Sie zeigt einerseits daß OSPL nicht wirklich ausdrucksstärker ist als PL_1 , und erlaubt andererseits die Kalküle zu vergleichen. (Für OSPL wird ein Kalkül noch nachgeliefert.) Die Übersetzung in PL_1 wird bewußt ausführlich betrachtet, weil der Mechanismus prototypisch für weitere Übersetzungsverfahren ist, die Formeln nichtklassischer Logiken in Prädikatenlogik übersetzen. In diesen Fällen benutzt man dann die Übersetzung, um mit Resolution auf den übersetzten Formeln zu operieren.

Definition 13.8. Übersetzer: $OSPL \rightarrow PL_1$)

Gegeben eine OSPL-Spezifikation (Σ, \mathcal{F}) , d.h. eine OSPL Signatur Σ zusammen mit einer Menge \mathcal{F} von geschlossenen Formeln über Σ , dann besteht ein Übersetzer, der diese OSPL-Spezifikation in eine PL_1 Spezifikation (Σ', \mathcal{F}') überträgt, aus drei Teilen:

Ψ_{sig} übersetzt die OSPL-Signatur in eine PL_1 -Signatur,

$\Psi_{sig, \mathcal{F}}$ generiert aus der OSPL-Signatur PL_1 -Formeln und

Ψ_{Σ} übersetzt die Formeln.

Sie sind folgendermaßen definiert:

Ψ_{sig} : Sei $\Sigma = (V, \mathcal{F}, \mathcal{P}, S, \mathcal{S}, SD, FD, PD)$ eine OSPL Signatur. Dann ist $\Psi_{sig}(\Sigma) = (\mathcal{F}, \mathcal{P} \cup \{A' \mid A \in S\})$. Es ist $\text{arity}(A') = 1$. (d.h. aus jedem Sortensymbol wird ein einstelliges Prädikatsymbol gemacht.)

$\Psi_{sig, F}$:

$$\Psi_{sig, F}(\Sigma) = \{\forall x : A(x) \Rightarrow B(x) \mid A \sqsubseteq B \in SD\}$$

$$\cup \{\forall x_1, \dots, x_n (A_1(x_1) \wedge \dots \wedge A_n(x_n)) \Rightarrow A(f(x_1, \dots, x_n)) \mid f(x_1, \dots, x_n) : A \in FD\}$$

Die Deklarationen für Prädikatsymbole werden nicht übersetzt.

Ψ_Σ : Die Übersetzung von Formeln wird rekursiv über die Formelstruktur definiert:

Basisfall:

$$\Psi_\Sigma(L) = L \text{ falls } L \text{ ein Atom ist.}$$

Rekursionsfälle:

$$\Psi_\Sigma(\neg F) = \neg \Psi_\Sigma(F)$$

$$\Psi_\Sigma(F @ G) = \Psi_\Sigma(F) @ \Psi_\Sigma(G), @ \in \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$$

$$\Psi_\Sigma(\forall x : A : F) = \forall x : A(x) \Rightarrow \Psi_\Sigma(F)$$

$$\Psi_\Sigma(\exists x : A : F) = \exists x : A(x) \wedge \Psi_\Sigma(F)$$

Insgesamt ergibt sich also: $\Psi(\Sigma, \mathcal{F}) = (\Psi_{sig}(\Sigma), \Psi_{sig, F}(\Sigma) \cup \{\Psi_\Sigma(F) \mid F \in \mathcal{F}\})$.

Beispiel 13.9. Für eine Übersetzung von OSPL in PL_1
Wir nehmen im wesentlichen wieder die Signatur aus 13.3:

Variablensymbole: $\{x, y, z, u, v, w \dots\}$

Funktionssymbole: $\{+, *, inv, o, Fido, Tweety, Vater\},$

$$arity(+) = arity(*) = 2, arity(inv) = 1, arity(Fido) = arity(o) = 0$$

$$arity(Tweety) = 0, arity(Vater) = 1$$

Prädikatensymbole: $\{<, gerade, brav, =\}$

Sortensymbole: $\{komplex, reell, nat, gaussch, Hund, Tier\}$

Sortenfunktion: $\mathcal{S}: \mathcal{S}(x) = komplex, \mathcal{S}(y) = reell, \mathcal{S}(z) = nat, \mathcal{S}(u) = Hund \dots$

Untersorten: $reell \sqsubseteq komplex, gaussch \sqsubseteq komplex, nat \sqsubseteq gaussch, nat \sqsubseteq reell, Hund \sqsubseteq Tier$

Funktionen: $o : nat$

$$+ : komplex \times komplex \rightarrow komplex$$

$$reell \times reell \rightarrow reell$$

$$nat \times nat \rightarrow nat$$

$$* : komplex \times komplex \rightarrow komplex$$

$$reell \times reell \rightarrow reell$$

$$nat \times nat \rightarrow nat$$

$$inv : komplex \times komplex \rightarrow komplex, \dots$$

$$Fido : Hund; Tweety : Tier; Vater : Hund \rightarrow Hund \dots$$

Prädikate: $gerade : nat; < : komplex \times komplex; brav : Hund \dots$

Als rein formale Übung übersetzen wir folgende Formeln:

$$\mathcal{F} = \{\forall x : komplex \exists z : nat : (* (x, x) = z \wedge brav(Fido) \Rightarrow Alter(Fido) = z)\}$$

Es ergibt sich: $\Psi_{sig}(\Sigma) = (\{x, y, z, u, v, w \dots\}, \{+, *, inv, o, Fido, Alter\}, \{<, gerade, brav, =, komplex, reell, nat, gaussch, Hund, Tier\})$.

Die Stelligkeiten übertragen sich: $arity(komplex) = arity(reell) = arity(gaussch) = arity(Hund) = arity(Tier) = 1$.

$$\begin{aligned}
\Psi_{sig}(\Sigma) = & \{\forall x : rell(x) \Rightarrow komplex(x), \\
& \forall x : gaussch(x) \Rightarrow komplex(x), \\
& \forall x : nat(x) \Rightarrow gaussch(x), \\
& \forall x : nat(x) \Rightarrow rell(x), \\
& \forall x : Hund(x) \Rightarrow Tier(x), \\
& nat(o), \\
& \forall x, y : komplex(x) \wedge komplex(y) \Rightarrow komplex(+ (x, y)), \\
& \forall x, y : rell(x) \wedge rell(y) \Rightarrow rell(+ (x, y)), \\
& \forall x, y : nat(x) \wedge nat(y) \Rightarrow nat(+ (x, y)), \\
& \forall x, y : komplex(x) \wedge komplex(y) \Rightarrow komplex(* (x, y)), \\
& \forall x, y : rell(x) \wedge rell(y) \Rightarrow rell(* (x, y)), \\
& \forall x, y : nat(x) \wedge nat(y) \Rightarrow nat(* (x, y)), \\
& \forall x, y : komplex(x) \Rightarrow komplex(* (x, inv(x))), \\
& Hund(Fido), \\
& \forall x : Hund(x) \Rightarrow nat(Alter(x))\}.
\end{aligned}$$

Die Formel wird folgendermaßen übersetzt:

$$\begin{aligned}
& \Psi_{\Sigma}(\forall x : komplex : \exists z : nat(* (x, x) = z \wedge brav(Fido) \Rightarrow Alter(Fido) = z)) \\
& = \forall x : komplex(x) \Rightarrow \Psi_{\Sigma}(\exists z : nat(* (x, x) = z \wedge brav(Fido) \Rightarrow Alter(Fido) = z)) \\
& = \forall x : komplex(x) \Rightarrow \exists z : nat(z) \wedge \Psi_{\Sigma}(* (x, x) = z \wedge brav(Fido) \Rightarrow Alter(Fido) = z) \\
& = \forall x : komplex(x) \Rightarrow \exists z : nat(z) \wedge (* (x, x) = z \wedge brav(Fido) \Rightarrow Alter(Fido) = z)
\end{aligned}$$

In diesem Beispiel sind also aus einer OSPL-Formel 16 PL_1 -Formeln geworden. Das deutet darauf hin, daß es besser sein könnte, mit der OSPL- Formel zu arbeiten.

Um die Korrektheit und Vollständigkeit der Übersetzung zu beweisen, müssen wir zeigen, daß jede erfüllbare Spezifikation in eine erfüllbare Spezifikation übersetzt wird (und umgekehrt).

Satz 13.10. (Korrektheit und Vollständigkeit der Übersetzung von OSPL in PL_1)

Sei $\mathcal{S} = (\Sigma, \mathcal{F})$ eine OSPL-Spezifikation und $\mathcal{S}' = (\Sigma', \mathcal{F}')$ die übersetzte PL_1 Spezifikation.

Dann ist \mathcal{S} erfüllbar gdw. \mathcal{S}' erfüllbar ist.

Den Beweis bietet keine echte Schwierigkeit. Er ist etwas länglich, da viele Fälle und Details beachtet werden müssen, und beide Richtungen gezeigt werden müssen.

Die Korrektheit sichert, daß wenn eine übersetzte Spezifikation widerlegt worden ist, auch die Originalformel unerfüllbar ist. D.h. Beweis durch Übersetzen und Widerlegen der negierten Behauptung ist ein korrektes Verfahren. Um nachzuweisen, daß die Methode auch immer funktioniert müssen wir noch die Vollständigkeit nachweisen, d.h. zeigen, daß Modelle für übersetzte Formeln auch immer die Existenz von Modellen für die Originalformeln sicherstellen.

Konsequenzen aus den Vollständigkeits- und Korrektheitsbeweisen für die Übersetzung in PL_1 sind:

- OSPL ist als Logik äquivalent zu PL_1 . Daher kann man bestimmte Resultate wie Kompaktheit direkt übertragen.
- Formeln in OSPL sind i.a. erheblich kürzer und intuitiver als die entsprechenden Formeln in PL_1 .

Daher kann man viele Sachverhalte in OSPL wesentlich besser ausdrücken. Außerdem ist zu erwarten, daß der Kalkül in OSPL effizienter ist.

13.5 Klauselnormalform für OSPL

Auf dem Weg zu einem Resolutionskalkül ist der erste Schritt die Herstellung der Klauselform. Bis auf eine Ausnahme funktioniert das wie in PL_1 . Die Ausnahme ist die Skolemisierung.

Definition 13.11. Skolemisierung in OSPL

Eine OSPL Formel $\forall x_1 : A_1 \dots \forall x_n : A_n \exists x : A : F$ wird folgendermaßen skolemisiert:

$$\forall x_1 : A_1 \dots \forall x_n : A_n \exists x : AF \rightarrow \forall x_1 : A_1 \dots \forall x_n : A_n F[x \rightarrow f(x_1, \dots, x_n)]$$

und folgende Funktionsdeklaration wird der Signatur hinzugefügt:

$$f : A_1 \times \dots \times A_n \rightarrow A$$

Der Beweis für die Korrektheit der Skolemisierung ist genau wie in PL_1 (Theorem 4.28). Die Korrektheit der Funktionsdeklaration ist offensichtlich. Da die Sorte des neuen Terms $f(x_1, \dots, x_n)$ gleich der Sorte der Variablen x ist, garantiert die Funktionsdeklaration, daß die neue Formel $F[x \rightarrow f(x_1, \dots, x_n)]$ auch wohlsortiert ist.

13.6 Unifikation

Zunächst wird ein allgemeiner Unifikationsalgorithmus für OSPL vorgestellt und dann werden Optimierungen für speziellere und für eingeschränkte Sortenstrukturen besprochen.

Wir geben einen Algorithmus an, der für vernünftige sortierte Signaturen korrekt und vollständig ist:

Definition 13.12. gutartige sortierte Signatur

Sei f_{sort} die Relation auf den Sorten, definiert als: $f_{sort}(S_1, \dots, S_n) = S$ gdw. $S_1 \times \dots \times S_n \rightarrow S$ eine Funktionsdeklaration ist. Folgendes sei erfüllt:

- \sqsubseteq ist eine partielle Ordnung auf den Sorten
- die kleinste Sorte eines Terms ist eindeutig bestimmt.
- f_{sort} ist eine (partielle) Funktion.
- Wenn f_{sort} auf (S_1, \dots, S_n) definiert ist und $(S'_1, \dots, S'_n) \sqsubseteq (S_1, \dots, S_n)$, dann ist f_{sort} auch auf (S'_1, \dots, S'_n) definiert.
- f_{sort} ist monoton auf den Sorten.

Dann ist die sortierte Signatur gutartig.

Definition 13.13. Unifikationsalgorithmus U_S für gutartige Signaturen:

Eingabe: zwei Terme oder Atome s_0 und t_0 :

Ausgabe: “nicht unifizierbar“ oder eine Aufzählung der allgemeinsten Unifikatoren:

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

Initialzustand: $\Gamma_0 = \{s_0 \stackrel{?}{=} t_0\}$.

Unifikationsregeln:

$\frac{f(s_1, \dots, s_n) \stackrel{?}{=} f(t_1, \dots, t_n), \Gamma}{s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n, \Gamma}$		(Dekomposition)
$\frac{x \stackrel{?}{=} f(s_1, \dots, s_n), x \stackrel{?}{=} f(t_1, \dots, t_n), \Gamma}{x \stackrel{?}{=} f(t_1, \dots, t_n), s_1 \stackrel{?}{=} t_1, \dots, s_n \stackrel{?}{=} t_n, \Gamma}$		(Dekomposition)
$\frac{x \stackrel{?}{=} x, \Gamma}{\Gamma}$		(Tautologie)
$\frac{x \stackrel{?}{=} y, \Gamma}{x \stackrel{?}{=} y, \{x \mapsto y\} \Gamma}$	wenn $S(y) \sqsubseteq S(x)$	(Ersetzung)
$\frac{t \stackrel{?}{=} x, \Gamma}{x \stackrel{?}{=} t, \Gamma}$	$t \notin V$	(Orientierung)

$\frac{x \stackrel{?}{=} y, \Gamma}{y \stackrel{?}{=} x, \Gamma}$	$\text{falls } S(x) \sqsubseteq S(y)$	(Variablenorientierung)
$\frac{x \stackrel{?}{=} y, \Gamma}{x \stackrel{?}{=} z, y \stackrel{?}{=} z, \Gamma}$	$z \text{ neu mit } S(z) \sqsubseteq S(x), S(z) \sqsubseteq S(y)$	(Variablenabschwächung)
$\frac{x \stackrel{?}{=} f(t_1, \dots, t_n), \Gamma}{x \stackrel{?}{=} f(x_1, \dots, x_n), \Gamma}$	$x_i \text{ neu mit } S(x_i) = S(t_i)$	(Auffalten)
$\frac{x \stackrel{?}{=} f(x_1, \dots, x_n), \Gamma}{x \stackrel{?}{=} f(y_1, \dots, y_n), x_1 \stackrel{?}{=} y_1, \dots, x_n \stackrel{?}{=} y_n, \Gamma}$	$\text{falls } S(f(x_1, \dots, x_n)) \not\sqsubseteq S(x). y_i \text{ neu mit } S(y_i) \sqsubseteq S(t_i)$	(Termabschwächung)

Abbruchbedingungen:

$\frac{f(\dots) \stackrel{?}{=} g(\dots), \Gamma}{\text{Fail}}$	wenn $f \neq g$	(Clash)
$\frac{x_1 \stackrel{?}{=} f(\cdot, x_2, \cdot), \dots, x_n \stackrel{?}{=} f(\cdot, x_1, \cdot)}{\text{Fail}}$	d.h. Zyklus gefunden:	(occurs check)
$(x \stackrel{?}{=} y) \in \Gamma$	Variablenabschwächung nicht anwendbar	
$(x \stackrel{?}{=} f(x_1, \dots, x_n)) \in \Gamma$	Termabschwächung nicht anwendbar	

Die Verzweigungsregeln sind: Termabschwächung und Variablenabschwächung. Die Steuerung ist nicht-deterministisch. Start ist mit Γ_0 . Am günstigsten ist es, mit hoher Priorität die Simplifikationsregeln zu verwenden: Dekomposition, Tautologie, Ersetzung, Orientierung, Auffalten. Die Suche wird nicht-deterministisch

in Breitensuche durchgeführt. Wenn eine Gleichungen $\{x_1 \stackrel{?}{=} t_1, \dots, x_k \stackrel{?}{=} t_k\}$ erzeugt wird, auf die keine Regel mehr anwendbar ist und keine Abbruchbedingung zutrifft, wird die Substitution $\{x_i \mapsto t_i\}$ als Unifikator ausgegeben.

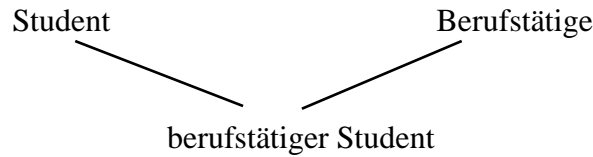
Bemerkung 13.14. Die Anwendungsregel kann optimiert werden indem man die Beschränkung auf wohlsortierte Substitutionen aufgibt. Die Vorgehensweise ist dann, zunächst Unifikation ohne Beachtung der Sorten durchzuführen, und danach erst die Sorten anzupassen. Es können dabei zwischendurch nicht wohlsortierte Terme auftreten. Das wird wieder ausgeglichen, da eine Gleichung $x \stackrel{?}{=} t$, die für die Anwendungsregel benutzt wird, nur dann gelöst ist, wenn die Sorte von t kleiner oder gleich der Sorte von x ist. Das kann durch die Abschwächungsregeln erreicht werden.

Beispiel 13.15. Wir geben zunächst Beispiele, die die einzelnen Regeln illustrieren.

1. Sorten: $N \sqsubseteq R$

$$\begin{aligned} & x : N \stackrel{?}{=} y : R \\ & \rightarrow y : R \stackrel{?}{=} x : N \text{ (Variablenorientierung)} \\ & \rightarrow \{y \mapsto x\} \end{aligned}$$

2. Sortenstruktur:



$$\begin{aligned} & x : \text{Student} \stackrel{?}{=} y : \text{Berufstätig} \\ & \rightarrow x \stackrel{?}{=} z, y \stackrel{?}{=} z : \text{berufstätiger Student} \text{ (Variablenabschwächung)} \\ & \rightarrow \{x \mapsto z, y \mapsto z\} \end{aligned}$$

4. Sorten: $\text{nat} \sqsubseteq \text{reell}$

Funktionsdeklarationen:

$$\text{plus} : \text{reell} \times \text{reell} \rightarrow \text{reell}$$

$$\text{nat} \times \text{nat} \rightarrow \text{nat}$$

$$x : nat \stackrel{?}{=} plus(u : reell, v : reell)$$

$$\rightarrow x : nat \stackrel{?}{=} plus(u_1, v_1), u \stackrel{?}{=} u_1 : nat, v \stackrel{?}{=} v_1 : nat \text{ (Termabschwächung)}$$

Bei den hier betrachteten gutartigen Sortenstrukturen haben alle Unifikationsprobleme nur endlich viele allgemeinste Unifikatoren.

Jetzt, wo wir wissen wie die Unifikation funktioniert können wir auch Resolutionsbeweise führen. Der Unterschied in der Performanz wird verdeutlichen an einem Beispiel, das man einmal unsortiert und einmal sortiert rechnet. Es handelt sich um das als *Schuberts Steamroller* bekannte Beispiel.

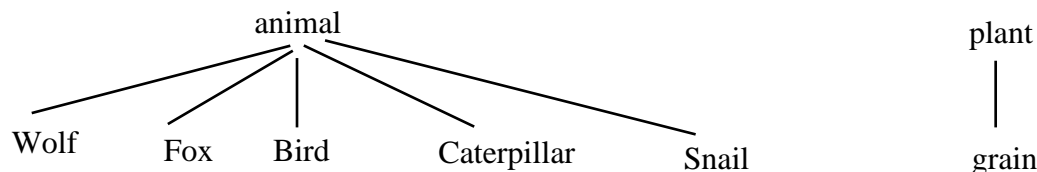
Beispiel 13.16. Schuberts Steamroller

Zunächst die unsortierte Variante:

```
set(ur_res).
assign(max_mem,1500).
assign(max_seconds,1800).
set(free_all_mem).

formula_list(axioms).
(all x (Wolf(x) -> animal(x))).
(all x (Fox(x) -> animal(x))).
(all x (Bird(x) -> animal(x))).
(all x (Caterpillar(x) -> animal(x))).
(all x (Snail(x) -> animal(x))).
(all x (Grain(x) -> plant(x))).
(all x (animal(x) -> ((all y (plant(y) -> eats(x,y))) |
(all z ((animal(z) & Smaller(z,x) & (exists u (plant(u) & eats(z,u))))
-> eats(x,z))))))).
(all x all y ((Caterpillar(x) & Bird(y)) -> Smaller(x,y))).
(all x all y ((Snail(x) & Bird(y)) -> Smaller(x,y))).
(all x all y ((Bird(x) & Fox(y)) -> Smaller(x,y))).
(all x all y ((Fox(x) & Wolf(y)) -> Smaller(x,y))).
(all x all y ((Bird(x) & Caterpillar(y)) -> eats(x,y))).
(all x (Caterpillar(x) -> (exists y (plant(y) & eats(x,y))))).
(all x (Snail(x) -> (exists y (plant(y) & eats(x,y))))).
(all x all y ((Wolf(x) & Fox(y)) -> -eats(x,y))).
(all x all y ((Wolf(x) & Grain(y)) -> -eats(x,y))).
(all x all y ((Bird(x) & Snail(y)) -> -eats(x,y))).
-(exists x exists y (animal(x) & animal(y) & eats(x,y) &
(all z (Grain(z) -> eats(y,z))))).
end_of_list.
```

Der Beweis dazu benötigt 70 Resolutions+ Faktorisierungs-schritte
 Eine Version mit Sorten hat folgende Sortenstruktur:



Mit einem Trick wurde der OTTER Beweiser dazu gebracht auch mit Sorten umzugehen. Der Beweis benötigt nur noch 18 Schritte ist und ist sehr viel schneller.

13.7 Optimierungen der Unifikation in OSPL

Die Unifikation in nicht gutartigen Sortenstrukturen ist unentscheidbar und infinitär. Aber auch bei gutartigen kann die Komplexität hoch sein und die Unifikation umständlich. Weitere Einschränkungen, die zu besseren Algorithmen führen, sind:

Man läßt nur Halbverbände als partielle Ordnung der Sortenhierarchie zu. Halbverbände bedeutet in unserem Zusammenhang, daß es für je zwei Sorten maximal eine größte gemeinsame Untersorte gibt. D.h. zwei Sorten haben entweder einen glb oder keine gemeinsame Untersorte.

In diesem Fall kann man die Variablenabschwächung eindeutig durchführen.

$$x : R \stackrel{?}{=} y : S \rightarrow x \stackrel{?}{=} z : T, y \stackrel{?}{=} z : T$$

wobei $T = glb(R, S)$ ist. Wenn der glb nicht existiert, kann man abbrechen.

Die Unifikation kann unitär (d.h. mgu ist eindeutig) werden, wenn in der Signatur gilt, daß zusätzlich die Termabschwächung immer auf eindeutige Weise ausgeführt werden kann

14 Im WWW verfügbare Deduktionssysteme

Einige Emailadressen von Beweissystemen: bzw. weitere Verweise:

- Otter: <http://www-unix.mcs.anl.gov/AR/otter/>
- EQP: <http://www-unix.mcs.anl.gov/AR/EQP/>
- SPASS: <http://www.mpi-sb.mpg.de/units/ag2/projects/SPASS/>
- TPTP: <http://www.cs.jcu.edu.au/tptp/> <http://wwwjessen.informatik.tu-muenchen.de/tptp/>

Literatur

- [And81] P. Andrews. Theorem proving via general matings. *J. ACM*, 28(2):193–214, 1981.
- [And86] Peter B. Andrews. *An Introduction to mathematical logic and type theory: to truth through proof*. Academic Press, 1986.
- [Ave95] Jürgen Avenhaus. *Reduktionssysteme*. Springer-Verlag, 1995.
- [BB87] K.-H. Bläsius and H.-J. Bürckert. *Deduktionssysteme - Automatisierung des logischen Denkens*. Oldenbourg Verlag, 1987.
- [Bib83] W. Bibel. *Automated Theorem Proving*. Vieweg, Braunschweig, 1983.
- [Bib92] W. Bibel. *Deduktion, Automatisierung der Logik*. Oldenbourg, 1992.
- [BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- [Bra90] Ivan Bratko. *Prolog – Programming for Artificial Intelligence*. Addison Wesley, 1990.
- [BS94] Franz Baader and Jörg H. Siekmann. Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2. Oxford Science Publications, 1994.
- [BS98] Wolfgang Bibel and Peter H. Schmitt, editors. *Automated Deduction - A Basis for Applications*, volume I – III. Kluwer Academic Publishers, 1998.
- [BS99] Franz Baader and Wayne Snyder. Unification theory. In *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. to appear.
- [Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, London, 1983.
- [Bür91] H.-J. Bürckert. *A Resolution Principle for a Logic with Restricterd Quantifiers*. Number 568 in Lecture Notes of Artificial Intelligence. Springer-Verlag, 1991.
- [CL73] Chin-Liang Chang and Richard Char-Tung Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York, 1973.
- [Duf91] David A. Duffy. *Principles of Automated Theorem Proving*. John Wiley & Sons, 1991.
- [Ede92] Elmar Eder. *Relative Complexities of First order Calculi*. Vieweg, Braunschweig, 1992.
- [EFT86] H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Wissenschaftliche Buchgesellschaft Darmstadt, 1986.
- [Fit90] Melvin Fitting. *First order logic and automated theorem proving*. Springer-Verlag, 1990.
- [FLTZ93] C. Fermüller, A. Leitsch, T. Tammet, and N. Zamov. *Resolution methods for the decision problem*, volume 679 of *LNCS*. Springer-Verlag, 1993.
- [Gal86] Jean H. Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper & Row Publishers, New York, 1986.
- [GLSDS91] Annie Gal, Guy Lapalme, Patrick Saint-Dizier, and Harold Somers. *Prolog for Natural Language Processing*. John Wiley & sons, 1991.
- [GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers, 1987.
- [Göd31] K Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Mh. Math. Phys.*, 38:173–198, 1931.
- [Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

- [HK89] Dieter Hofbauer and Ralf-Detlef Kutsche. *Grundlagen des maschinellen Beweisens*. Vieweg, 1989.
- [ki-98] Künstliche Intelligenz: Themenheft Deduktion und Anwendung, 1998. in german.
- [KK71] R. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.
- [Kow79] R. Kowalski. *Logic for Problem Solving*. North Holland, Amsterdam, 1979.
- [Lov78] Donald W. Loveland. *Automated Theorem Proving: A Logical Basis*, volume 6 of *Fundamental Studies in Computer Science*. North-Holland, New York, 1978.
- [MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM transaction on Programming Languages and Systems*, 4(2):258–282, 1982.
- [Nil80] N. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.
- [Pau94] Larry C. Paulson. *Isabelle: a Generic Theorem prover*. Springer-Verlag, 1994.
- [PM68] M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Science*, 16(2):158–167, 1968.
- [PS87] Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. CSLI, 1987.
- [RB79] J S. Moore R. Boyer. *A Computational Logic*, volume 29. Academic Press, London, 1979.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
- [SA94] Rolf Socher-Ambrosius. *Deduktionssysteme*. BI-Wissenschaftsverlag, 1994. in german.
- [Smu71] Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, 1971.
- [SS89] M. Schmidt-Schauß. *Computational Aspects of an order-sorted logic with term declarations*, volume 395 of *LNCS*. Springer-Verlag, Dover, 1989.
- [Sti86] Mark E. Stickel. A prolog technology theorem prover: implementation by an extended prolog compiler. In *Proc. of 8th Int. Conf. on Automated Deduction*, number 230 in *Lecture Notes in Computer Science*, pages 573–587. Springer-Verlag, 1986.
- [Tar53] A. Tarski. *Der Wahrheitsbegriff in den formalisierten Sprachen*. Studia Philosophica I. 1953.
- [Tha88] André Thayse, editor. *From Standard Logic to Logic Programming*. John Wiley & Sons, 1988.
- [Wal87] C. Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation*. Pitman & Kaufman, 1987.
- [Wei95] Christoph Weidenbach. First-order tableaux with sorts. *Journal of the Interest Group in Pure and Applied Logics, IGPL*, 3(6):887–906, 1995.
- [Wei96] Christoph Weidenbach. Unification in sort theories and its applications. *Annals of Mathematics and Artificial Intelligence*, 1996.
- [WOLB84] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning – Introduction and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1984.