# Einführung in die Methoden der Künstlichen Intelligenz

Sommersemester 2014

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

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

Stand: 27. Mai 2023

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Themen und Literatur

Für die Veranstaltung sind die folgenden Inhalte geplant (kann sich noch leicht ändern).

- Einführung: Ziele der künstliche Intelligenz, intelligente Agenten.
- Suchverfahren: Uninformierte und informierte Suche, Suche in Spielbäumen.
- Evolutionäre (oder auch genetische) Algorithmen.
- Wissensrepräsentation und Deduktion in der Aussagenlogik.
- Prädikatenlogik
- Qualitatives zeitliches Schließen am Beispiel von Allens Intervalllogik
- Konzeptbeschreibungsschreibungssprachen
- Maschinelles Lernen

Wir werden einige Algorithmen erörtern, wobei wir neben (imperativem) Pseudo-Code auch desöfteren eine entsprechende Implementierung in der funktionalen Programmiersprache Haskell angeben. Die Haskellprogramme dienen eher zum Verständnis und sind daher selbst kein Prüfungsstoff.

### 1.1.1 Wesentliche Lehrbücher für diese Veranstaltung

D. Poole, and A. Mackworth, *Artificial Intelligence: Foundations of Computational Agents*. Cambridge University Press, 2010. auch online unter http://artint.info/ frei verfügbar

S.J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach.* Third Edition, Prentice Hall, 2010.

W. Ertel *Grundkurs Künstliche Intelligenz: Eine praxisorientierte Einführung*. Vieweg-Teubner Verlag, 2009. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://www.springerlink.com/content/t13128/

D. Poole, A. Mackworth, and R. Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, 1998.

Franz Baader, Deborah McGuiness, Daniele Nardi, and Peter Patel-Schneider. *The description logic handbook*. Cambridge university press, 2002. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://site.ebrary.com/lib/frankfurtm/docDetail.action?docID=10069975

Christoph Beierle and Gabriele Kern-Isberner. *Methoden wissensbasierter Systeme*. 4. Auflage, Vieweg, 2008. Aus dem Netz der Universität als E-Book kostenlos verfügbar unter http://www.springerlink.com/content/h67586/

P. Winston. *Artificial Intelligence*. Addison Wesley, 1992.

McDermott Charniak. *Introduction to Artificial Intelligence*. Addison Wesley, 1985.

Genesereth and Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 1988.

M. Ginsberg. *Essentials of artificial intelligence*. Morgan Kaufmann, 1993.

Wolfgang Bibel, Steffen Hölldobler, and Torsten Schaub. *Wissensrepräsentation und Inferenz. Eine grundlegende Einführung*. Vieweg, 1993.

Joachim Herzberg. *Planen: Einführung in die Planerstellungsmethoden der Künstlichen Intelligenz*. Reihe Informatik. BI Wissenschaftsverlag, 1989.

Michael M. Richter. *Prinzipien der Künstlichen intelligenz*. Teubner Verlag, 1992.

### 1.1.2 Weitere Lehr- und Hanbücher

A. Barr, P.R. Cohen, and E.A. Feigenbaum. *The Handbook of Artificial Intelligence*. Addison Wesley, 1981-1989. 4 Bände.

E. Davies. *Representation of Commonsense Knowledge*. Morgan Kaufmann, 1990.

H.L. Dreyfus. *What Computers Can't Do*. Harper Colophon Books, 1979.

Günter Görz, editor. *Einführung in die künstliche Intelligenz*. Addison Wesley, 1993.

Peter Jackson. *Introduction to expert systems, 3rd ed.* Addison-Wesley, 1999.

George F. Luger and William A. Stubblefield . *Artificial intelligence: structures and strategies for complex problem solving. 3rd ed*. Addison-Wesley, 1998.

Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs*. Springer-Verlag, 1994.

Todd C. Moody. *Philosophy and Artificial Intelligence*. Prentice Hall, 1993.

Frank Puppe. *Problemlösungsmethoden in Expertensystemen*. Springer-Verlag, 1990.

Frank Puppe. *Einführung in Expertensysteme*. Springer-Verlag, 1991.

Peter Schefe. *Künstliche Intelligenz, Überblick und Grundlagen*. Reihe Informatik. BI Wissenschaftsverlag, 1991.

S. Shapiro. *Encyclopedia of Artificial Intelligence*. John Wiley, 1989-1992. 2 Bände.

Herbert Stoyan. *Programmiermethoden der Künstlichen Intelligenz*. Springer-Verlag, 1988.

J. Retti u.a. *Artificial Intelligence – Eine Einführung*. Teubner Verlag, 1986.

Vidysagar. *A theory of learning and generalization*. Springer-Verlag.

Ingo Wegener, editor. *Highlights aus der Informatik*. Springer, 1996.

Reinhard Wilhelm, editor. *Informatics – 10 years back – 10 years ahead*. Lecture Notes in Computer Science. Springer-Verlag, 2000.

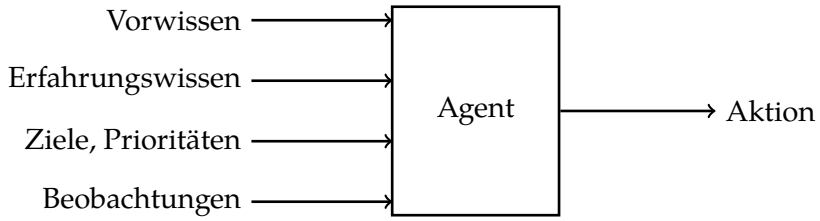
## 1.2 Was ist Künstliche Intelligenz?

Ziel der Künstlichen Intelligenz ist die Herstellung eines *intelligenten Agenten*. Dies kann man auch als die Herstellung eines möglichst guten (autonomen, lernenden, intelligenten, automatischen) Informationssystems ansehen. Einige Beispiele für solche Systeme sind:

- Taschenrechner, die analog zum Menschen Rechenaufgaben lösen können (dabei aber anders Rechnen als der Mensch).
- Schachspielende Computer wie z.B. Deep Blue, Deep Thought und Deep Fritz
- Sprachübersetzer wie z.B. GoogleTranslate, Babelfish, etc.
- wissensbasierte Systeme in vielen Varianten
- Texterkennungssysteme in Kombinationen mit Hintergrundwissen wie z.B. das Watson-Programm[1], welches bei der amerikanischen TV-Show „Jeopardy“ gegen frühere Champions antrat und gewann.
- Roboter, z.B. Haushaltsrobotor wie Staubsaugerroboter, Industrieroboter, etc.
- intelligente Informationssysteme

[1] www.ibm.com/de/watson

Allgemein ist ein intelligenter Agent, der im Kern einen Computer hat und mit der physikalischen Umwelt agiert, ein *Roboter*. Sind die Umgebungen nicht-physikalisch, so wird dies meist durch spezielle Namen gekennzeichnet: Ein Software-Robotor (softbot) hat eine Wissensbasis und gibt Antworten und Ratschläge, aber agiert nicht mit der physikalischen Umwelt. Web-Roboter (webbot) interagieren mit dem WWW, z.B. zum Durchsuchen und Erstellen von Suchdatenbanken im WWW. Ein Chat-Roboter (chatbot) agiert in einem Chat. Ein Agent hat als Eingaben und Zwischenzustände zum einen Vorwissen über die Umgebung (z.B. eine Karte), aber auch erlerntes Wissen, Testfälle etc. Die vom Agent zu erreichenden Ziele sind üblicherweise mit Prioritäten und Wichtigkeiten versehen. Allgemein kann ein Agent Beobachtungen über die Umgebung und auch Beobachtungen über sich selbst machen (und daraus Schlüsse ziehen, beispielsweise Lernen). Aufgrund dieser „Eingaben“ berechnet der Agent als als Ausgabe seine nächste *Aktion*.




Bevor wir auf die genauere Modellierung von Agenten eingehen, wollen wir erörtern, was man unter Künstlicher Intelligenz versteht, d.h. wann ein System (Computer) über künstliche Intelligenz verfügt und welche Eigenschaften eine Methode als KI-Methode auszeichnet.

Allerdings gibt es keine eindeutige allgemeine Definition der künstlichen Intelligenz. Wir werden daher verschiedene Ansätze und Kennzeichen für künstliche Intelligenz erläutern. Gemäß (Russell & Norvig, 2010) kann man die verschiedenen Ansätze entsprechend der zwei Dimensionen:

- menschliches Verhalten gegenüber rationalem Verhalten
- Denken gegenüber Handeln

in vier Klassen von Ansätzen unterteilen: Menschliches Denken, rationales Denken, menschliches Handeln und rationales Handeln. Während „menschlich“ bedeutet, dass das Ziel ist, den Menschen nachzuahmen, meint „rational“ ein vernünftiges Vorgehen, das optimal (bezüglich einer Kostenabschätzung o.ä.) ist. Wir erläutern im Folgenden genauer, was KI entsprechend dieser Klassen meint und als Ziel hat.

### 1.2.1 Menschliches Handeln

Verwendet man menschliches Handeln als Ziel künstlicher Intelligenz, so steht im Vordergrund, Systeme oder auch Roboter zu erschaffen, die analog zu Menschen handeln.

Ein Ziel dabei ist es, Methoden und Verfahren zu entwickeln, die Computer dazu bringen, Dinge zu tun, die momentan nur der Mensch kann, oder in denen der Mensch noch den Computern überlegen ist. Eine Methode, um „nachzuweisen“, ob der Computer entsprechend intelligent handelt, besteht darin, seine Handlungen mit den Handlungen des Menschen zu vergleichen (wie z.B. der Turing-Test, siehe unten).

### 1.2.2 Menschliches Denken

Dieser Ansatz verfolgt das Ziel, dass ein Computer „wie ein Mensch denkt“. Dabei muss zunächst erforscht werden, wie der Mensch selbst denkt, bevor entsprechende Modelle auf Computer übertragen werden können. Dieses Erforschen kann beispielsweise durch psychologische Experimente oder Hirntomografie erfolgen. Ist eine Theorie aufgestellt, so kann diese in ein System umgesetzt werden. Wenn die Ein- und Ausgaben dieses Systems dem menschlichen Verhalten entsprechen, kann man davon ausgehen, dass dieses System auch im Mensch eingesetzt werden könnte und daher dem menschlichen Denken entspricht. Dieser Ansatz ist jedoch nicht alleinig in der Informatik angesiedelt, sondern entspricht eher dem interdisziplinären Gebiet der Kognitionswissenschaft, die informatische Modelle mit experimentellen psychologischen Versuchen kombiniert, um menschliches Verhalten durch Modelle zu erfassen und nachzuahmen. Beispiele für dieses Gebiet sind Fragen danach, wie der Mensch etwas erkennt (z.B. wie erkennen wir Gesichter), oder Verständnis des Spracherwerbs des Menschen. Ein System ist *kognitiv adäquat*, wenn es strukturell und methodisch wie ein Mensch arbeitet und entsprechende Leistungen erzielt. Z.B. ist ein Taschenrechner nicht kognitiv adäquat, da er anders addiert als der Mensch, die Methodik stimmt also nicht überein. Daher ist dieser Ansatz der *kognitiven Simulation* eher vom Gebiet der *künstlichen Intelligenz* abzugrenzen (also als verschieden anzusehen), da er sehr auf exakte Nachahmung des menschlichen Denkens zielt. Wir werden daher diesen Ansatz nicht weiter verfolgen, sondern ihn der Kognitionswissenschaft überlassen.

### 1.2.3 Rationales Denken

Rationales Denken versucht Denken durch Axiome und *korrekte* Schlussregeln zu formalisieren. Meistens wird hierzu eine Logik verwendet. D.h. der logische Ansatz (ein großes Teilgebiet der Künstlichen Intelligenz) ist es, aufgrund einer Logik ein *Deduktionssystem* zu implementieren, das sich intelligent verhält. Eine Hürde bei diesem Ansatz ist es, das entsprechende Problem bzw. das Wissen in einer Logik zu formalisieren.

### 1.2.4 Rationales Handeln

Dies ist der Agenten-Ansatz. Ein Agent ist ein System, dass agiert, d.h. auf seine Umgebung (die Eingaben) eine Reaktion (die Ausgaben) durchführt. Dabei erwartet man im Allgemeinen, dass der Agent autonom operiert, d.h. auf die Umgebung für längere

Zeiträume reagiert, sich an Änderungen anpassen kann und ein Ziel verfolgt (und dieses eventuell den Gegebenheiten anpasst). Ein rationaler Agent ist ein Agent, der sein Ergebnis maximiert, d.h. stets das bestmögliche Ergebnis erzielt. Dieser Ansatz ist sehr allgemein und kann auch teilweise die anderen Ansätze miteinbeziehen. So ist es durchaus denkbar, dass ein rationaler Agent eine Logik mitsamt Schlussregeln verwendet, um rationale Entscheidungen zu treffen. Die Wirkung des rationalen Agentes kann menschlich sein, wenn dies eben rational ist. Es ist jedoch keine Anforderung an den rationalen Agent, menschliches Verhalten abzubilden. Dies ist meist ein Vorteil, da man sich eben nicht an einem gegebenen Modell (dem Mensch) orientiert, sondern auch davon unabhängige Methoden und Verfahren verwenden kann. Ein weiterer Vorteil ist, dass Rationalität klar mathematisch definiert werden kann und man daher philosophische Fragen, was intelligentes Verhalten oder Denken ist, quasi umgehen kann.

### 1.2.5 Ausrichtungen und Ausrichtung der Vorlesung

Die Kognitionswissenschaft versucht das intelligente Handeln und Denken des Menschen zu analysieren, zu modellieren und durch informatische Systeme nachzuahmen. Dafür werden insbesondere das Sprachverstehen, Bildverstehen (und auch Videoverstehen) untersucht und durch Lernverfahren werden entsprechende Systeme trainiert und verbessert. Die *ingenieurmäßig ausgerichtete KI* hingegen stellt Methoden, Techniken und Werkzeuge zum Lösen komplexer Anwendungsprobleme bereit, die teilweise kognitive Fähigkeiten erfordern. Als Beispiele sind hier Deduktionstechniken, KI-Programmiersprachen, neuronale Netze als Mustererkenner, Lernalgorithmen, wissensbasierte Systeme, Expertensysteme zu nennen.

In der Vorlesung werden wir die ingenieurmäßige Ausrichtung verfolgen und uns mit direkt programmierbaren KI-Methoden beschäftigen, die insbesondere Logik in verschiedenen Varianten verwenden.

Einige *Gebiete der Künstlichen Intelligenz* sind

- Programmierung strategischer Spiele (Schach, ...)
- Automatisches/Interaktives Problemlösen und Beweisen
- Natürlichsprachliche Systeme (Computerlinguistik)
- Bildverarbeitung
- Robotik
- (medizinische) Expertensysteme
- Maschinelles Lernen

## 1.3 Philosophische Aspekte

Wir betrachten kurz einige philosophische Aspekte der künstlichen Intelligenz, indem wir zunächst die philosophischen Richtungen Materialismus, Behaviorismus und Funktionalismus erötern.

**Materialismus** Dem Materialismus liegt der Gedanke zu grunde, dass es nichts gibt außer Materie. Diese bewirkt auch den Geist bzw. die Gedanken eines Menschen. Die Implikation daraus ist, dass alles was einen Menschen ausmacht prinzipiell mithilfe naturwissenschaftlicher Methoden (Chemie, Physik, Biologie, ...) analysierbar ist. Als Konsequenz (insbesondere für die künstliche Intelligenz) ergibt sich, dass prinzipiell auch alles konstruierbar ist, insbesondere ist bei diesem Ansatz im Prinzip auch der denkende, intelligente Mensch konstruierbar.

**Behaviorismus** Der Behaviorismus geht davon aus, dass nur das Beobachtbare Gegenstand der Wissenschaft ist. D.h. nur verifizierbare Fragen und Probleme werden als sinnvoll angesehen. Glauben, Ideen, Wissen sind nicht direkt, sondern nur indirekt beobachtbar. Bewußtsein, Ideen, Glaube, Furcht, usw. sind Umschreibungen für bestimmte Verhaltensmuster. Ein Problem dieser Sichtweise ist, dass man zum Verifizieren evtl. unendlich viele Beobachtungen braucht. Zum Falsifizieren kann hingegen eine genügen. Äquivalenz von Systemen (z.B. Mensch und Roboter) wird angenommen, wenn sie gleiches Ein-/Ausgabe-Verhalten zeigen.

**Funktionalismus** Der Funktionalismus geht davon aus, dass geistige Zustände (Ideen, Glauben, Furcht, ...) interne (funktionale) Zustände eines komplexen System sind. Einzig die Funktion definiert die Semantik eines Systems. D.h. der Zustand $S_1$ des Systems $A$ is funktional äquivalent zu Zustand $S_2$ des Systems B, gdw. A und B bei gleicher Eingabe die gleiche Ausgabe liefern und in funktional äquivalente Zustände übergehen. Z.B. sind das Röhren-Radio und das Transistor-Radio funktional äquivalent, da sie dieselbe Funktion implementieren. Im Unterschied dazu sind Radio und CD-Player nicht äquivalent. Dieser Ansatz läuft darauf hinaus, dass der Mensch im Prinzip ein endlicher Automat mit Ein/Ausgabe ist, wenn auch mit sehr vielen Zuständen.

### 1.3.1 Die starke und die schwache KI-Hypothese

Im Rahmen der Künstlichen Intelligenz unterscheidet die Philosophie die schwache und die starke KI-Hypothese. Die *schwache KI-Hypothese* ist die Behauptung, dass Maschinen agieren können, *alsob* sie intelligent wären.

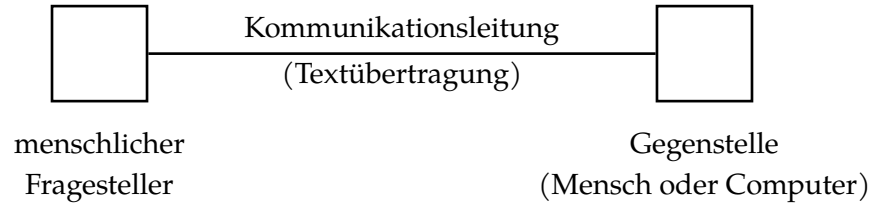
Die *starke KI-Hypothese* geht im Gegensatz dazu davon aus, dass Maschinen wirklich denken können, d.h. sie simulieren nicht nur das Denken.

Die schwache KI-Hypothese wird von den meisten als gegeben hingenommen. Die starke KI-Hypothese steht in der KI-Forschung im Allgemeinen nicht im Vordergrund, sondern eher die pragmatische Sichtweise, dass es irrelevant ist, ob das „Denken“ nur anscheinlich oder tatsächlich passiert – hauptsache das System funktioniert.

Trotzallem sollte man sich als KI-Forscher über die ethisch-moralischen Konsequenzen seiner Forschung bewusst sein.

### 1.3.2 Der Turing-Test

Ein Test, der die starke KI-Hypothese nachweisen könnte, könnte der folgende von Alan Turing vorgeschlagene Test sein. Der sogenannte *Turingtest* besteht darin, dass ein menschlicher Fragesteller schriftliche Fragen an den Computer (die Gegenstelle) stellt und dieser darauf antwortet. Der Test ist bestanden, wenn der Fragesteller nicht erkennen kann, ob die Antworten von einem Computer oder von einem Menschen stammen, d.h. die Gegenstelle sollte wechselweise mal durch den Computer mal durch einen Mensch ersetzt werden.




Ein Problem des Turingtests ist, dass dieser nicht objektiv ist, da er von den Fähigkeiten des menschlichen Fragestellers abhängt, z.B. vom Wissen über aktuelle Fähigkeiten eines Computers. Die Aussagekraft kann durch wiederholen des Experiments mit mehreren Fragestellern erhöht werden.

Der Test ist absichtlich so ausgelegt, dass es keine physische Interaktion zwischen Mensch und Computer gibt (der Mensch sieht den Computer nicht und er stellt die Fragen schriftlich), da diese zusätzlichen Anforderungen nicht die Intelligenz ausmachen.

Hingegen verwendet der *totale Turingtest* zusätzlich ein Videosignal, so dass der Computer anhand dieses Signals auch den menschlichen Befrager analysisieren kann (z.B. durch Gestenerkennung), und umgekehrt der Befrager feststellen kann, ob der Befragte über wahrnehumngsorientierte Fähigkeiten verfügt. Zusätzlich kann der Befrager physisische Objekte an den Befragten übergeben. Der Computer muss daher über Fähigkeiten zur Objekterkennung und Robotik (Bewegen des Objekts) etc. verfügen.

Ein Wettbewerb, der sich am Turingtest orientiert, ist der seit 1991 vergebene Loebner-Preis[2]. Dabei werden Chat-Bots getestet und der von einer Jury am menschenähnlichsten ausgewählte Chat-Bot wird zum jährlichen Sieger gekührt. In jeder Runde des Tests kommuniziert ein Jury-Mitglied simultan mit einem Chat-Bot und einem Menschen ausschließlich über Textnachrichten in begrenzter Zeit. Im Anschluss entscheidet der Juror

[2] http://www.loebner.net/Prizef/loebner-prize.html

wer Mensch, wer Maschine ist. Es gibt zudem noch einmalige Preise für den ersten Chat-Bot, der es schafft die Jury überzeugend zu täuschen, ein Mensch zu sein, und schließlich den mit 100.000 USD dotierten Preis, für den ersten Chat-Bot, der dies auch bezüglich audio-visueller Kommunikation schafft. Beide einmalige Preise wurden bisher nicht vergeben.

Das Interessante an den Turing-ähnlichen Tests ist, dass sie ein halbwegs einsichtiges Kriterum festlegen, um zu entscheiden, ob eine Maschine menschliches Handeln simulieren kann oder nicht. Gegenargumente für den Turingtest sind vielfältig, z.B. könnte man ein System als riesige Datenbank implementieren, die auf jede Frage vorgefertigte Antworten aus der Datenbank liest und diese präsentiert. Ist ein solches System intelligent?

Andererseits gibt es Programme die manche Menschen in Analogie zum Turingtest täuschen können. Ein solches Programm ist das von J. Weizenbaum entwickelte Programm ELIZA[3], dass (als Softbot) einen Psychotherapeuten simuliert. Zum einen benutzt ELIZA vorgefertigte Phrasen für den Fall, dass das System nichts versteht (z.B. „Erzählen Sie mir etwas aus Ihrer Jugend.“), zum anderen wird mithilfe von Mustererkennung in der Eingabe des Benutzers nach Schlüsselwörtern gesucht und anschließend werden daraus Ausgaben (anhand von Datenbank und Schlussregeln) geniert (z.B. „Erzählen Sie mir etwas über $xyz$“).

Der Turing-Test wird in der Forschergemeinde nicht stark als angestrebtes Ziel verfolgt, aber aufgrund seiner Einfachheit demonstriert er eindrucksvoll, was künstliche Intelligenz bedeutet.

### 1.3.3 Das Gedankenexperiment „Chinesischer Raum“

Das von John Searle vorgeschlagenes Gedankenexperiment zum „Chinesischen Raum“ soll als Gegenargument zur starken KI-Hypothese dienen, d.h. es soll ein Nachweis sein, dass ein technisches System kein Verständnis der Inhalte hat, die verarbeitet werden, bzw. dass es keinen verstehenden Computer geben kann.

Das Experiment lässt sich wie folgt beschreiben: Jemand, der kein chinesisch versteht, sitzt in einem Raum. Es gibt dort chinesisch beschriftete Zettel und ein dickes Handbuch mit Regeln (in seiner Muttersprache), die jedoch ausschließlich angeben, welche chinesische Zeichen als Antwort auf andere chinesische Zeichen gegeben werden sollen. Durch einen Schlitz an der Tür erhält die Person Zettel mit chinesischen Zeichen. Anhand des Handbuchs und der bereits beschrifteten Zettel erzeugt die Person neue Zettel auf seinem Stapel und einen neuen beschrifteten Zettel (mit chinesischen Schriftzeichen), den er nach außen reicht.

Es stellen sich nun die Fragen: Versteht die Person im Raum chinesisch? Versteht das Gesamtsystem chinesisch? John Searles Argument ist, dass kein Teil des Systems etwas vom Inhalt versteht, und daher weder die Person noch das System chinesisch versteht. Das

[3] siehe z.B. http://www.med-ai.com/models/eliza.html.de

Gegenargument (entsprechend dem Behaviorismus) ist, dass das Gesamtsystem etwas versteht, da das Verständnis beobachtbar ist.

### 1.3.4 Das Prothesenexperiment

Dieses Gedankenexperiment geht davon aus, dass die Verbindungen von Neuronen im Gehirn voll verstanden sind und dass man funktional gleiche Bauteile (elektronische Neuronen) herstellen und im Gehirn einsetzen kann. Das Experiment besteht nun darin, einzelne Neuronen durch elektronische Neuronen zu ersetzen. Die Fragen dabei sind: Wird sich die Funktionalität des Gehirns ändern? Ab welcher Anzahl von elektronischen Neuronen wird sich das Prothesen-Gehirn in einen Computer verwandeln, der nichts versteht. Die Folgerung aus diesem Experiment ist, dass entweder das Gehirn nur aus Neuronen besteht (d.h. das Prothesengehirn ändert nichts, die starke KI-Hypothese gilt daher), oder das es etwas gibt, das noch unbekannt ist (Bewusstsein, Geist, etc.).

### 1.3.5 Symbolverarbeitungshypothese vs. Konnektionismus

Ein *physikalisches Symbolsystem* basiert auf Symbolen, denen eine Bedeutung in der Realität zugeordnet werden kann. Aus einer eingegeben Symbolstruktur (z.B. ein String aus Symbolen o.ä.) werden sukzessive weitere Symbolstrukturen erzeugt und evtl. auch ausgegeben.

Die *Symbolverarbeitungshypothese* (von A. Newell and H. Simon) geht davon aus, dass ein physikalisches Symbolsystem so konstruiert werden kann, dass es intelligentes Verhalten zeigt (d.h. den Turingtest besteht). Überlegungen zu Quantencomputern und deren parallele, prinzipielle Fähigkeiten könnten hier zu neuen Ansätzen und Unterscheidungen führen. Im Moment gibt es aber noch keine verwendbaren Computer, die auf diesem Ansatz basieren.

M. Ginsberg charakterisiert die künstliche Intelligenz durch das Ziel der Konstruktion eines physikalischen Symbolsystems, das zuverlässig den Turingtest besteht.

Die *Hypothese des Konnektionismus* geht davon aus, dass man subsymbolische, verteilte, parallele Verarbeitung benötigt, um eine Maschine zu konstruieren, die intelligentes Verhalten zeigt (den Turingtest besteht).

Die technische Hypothese dazu ist, dass man künstliche neuronale Netze benötigt. Als Gegenargument dazu wäre zu nennen, dass man künstliche neuronale Netze auch (als Software) programmieren und auf normaler Hardware simulieren kann.

## 1.4 KI-Paradigmen

Zwei wesentliche Paradigmen der künstlichen Intelligenz sind zum einen das physikalische Symbolsystem (erstellt durch explizites Programmieren und verwenden von Logiken, Schlussregeln und Inferenzverfahren). Die Stärken dieses Paradigmas bestehen dar-

in, dass man gute Ergebnisse in den Bereichen Ziehen von Schlüssen, strategische Spiele, Planen, Konfiguration, Logik, ... erzielen kann.

Ein anderes Paradima konzentriert sich auf Lernverfahren und verwendet insbesondere künstliche neuronale Netze. Dessen Stärken liegen vorallem in der Erkennung von Bildern, der Musterverarbeitung, der Verarbeitung verrauschter Daten, dem maschinelles Lernen, und adaptiven Systeme (Musterlernen).

Trotzallem benötigt man für ein komplexes KI-System im Allgemeinen alle Paradigmen.

### 1.4.1 Analyse und Programmieren von Teilaspekten

Die folgende Arbeitshypothese steht hinter der Untersuchung von Teilaspekten der Künstlichen Intelligenz:

Untersuche und programmiere Teilaspekte an kleinen Beispielen. Wenn das klar verstanden ist, dann untersuche große Beispiele und kombiniere die Lösungen der Teilaspekte.

Das ergibt zwei Problematiken:

**Kombination** der Lösungen von Teilaspekten. Die Lösungen sind oft kombinierbar, aber es gibt genügend viele Beispiele, die inkompatibel sind, so dass die Kombination der Methoden oft nicht möglich ist bzw. nur unter Verzicht auf gute Eigenschaften der Teillösungen.

**realistische Beispiele (scale up)** Wenn die Software-Lösung für kleine Beispiele (Mikrowelten) funktioniert, heißt das noch nicht, dass diese auch für realistische Beispiele (Makrowelten) sinnvoll funktioniert bzw. durchführbar ist.

### 1.4.2 Wissensrepräsentation und Schlussfolgern

Wissensrepräsentationshypothese: (Smith, 1982) "Die Verarbeitung von Wissen läßt sich trennen in: Repräsentation von Wissen, wobei dieses Wissen eine Entsprechung in der realen Welt hat; und in einen Inferenzmechanismus, der Schlüsse daraus zieht."

Dieses Paradigma ist die Basis für alle Programme in Software/Hardware, deren innere Struktur als Modellierung von Fakten, Wissen, Beziehungen und als Operationen, Simulationen verstanden werden kann.

Ein Repräsentations- und Inferenz-System (representation and reasoning system) besteht, wenn man es abstrakt beschreibt, aus folgenden Komponenten:

1. Eine *formale Sprache*: Zur Beschreibung der gültigen Symbole, der gültigen syntaktischen Formen einer Wissensbasis, der syntaktisch korrekten Anfragen usw.
   Im allgemeinen kann man annehmen, dass eine Wissensbasis eine Multimenge von gültigen Sätzen der formalen Sprache ist.

2. Eine *Semantik*: Das ist ein Formalismus, der den Sätzen der formalen Sprache eine Bedeutung zuweist.
   Im allgemeinen ist die Semantik modular, d.h. den kleinsten Einheiten wird eine Bedeutung zugeordnet, und darauf aufbauend den Sätzen, wobei die Bedeutung der Sätze auf der Bedeutung von Teilsätzen aufbaut.

3. Eine *Inferenz-Prozedur* (operationale Semantik) die angibt, welche weiteren Schlüsse man aus einer Wissensbasis ziehen kann. I.a. sind diese Schlüsse wieder Sätze der formalen Sprache.
   Diese Inferenzen müssen korrekt bzgl. der Semantik sein.

Dieses System kann auf Korrektheit geprüft werden, indem man einen theoretischen Abgleich zwischen Semantik und operationaler Semantik durchführt.

Die *Implementierung* des Repräsentations- und Inferenz-Systems besteht aus:

1. Parser für die formale Sprache

2. Implementierung der Inferenzprozedur.

Interessanterweise trifft diese Beschreibung auch auf andere Formalismen zu, wie Programmiersprachen und Logische Kalküle.

## 1.5 Bemerkungen zur Geschichte zur Geschichte der KI

Wir geben einen groben Abriss über die geschichtliche Entwicklung des Gebiets der künstlichen Intelligenz.

**1950:** A. Turing: Imititationsspiel

**1956** J. McCarthy Dartmouth Konferenz: Start des Gebietes “artificial intelligence“ und Formulierung der Ziele.

**1957- 1962** “allgemeiner Problemlöser“
Entwicklung von LISP (higher-order, Parser und Interpreter zur Laufzeit)
Dameprogramm von A. Samuels (adaptierend “lernend“)
Newell & Simon: GPS (General Problem Solver)

**1963-1967** spezialisierte Systeme

- semantische Verarbeitung (Benutzung von Spezialwissen über das Gebiet)
- Problem des Common Sense
- Resolutionsprinzip im automatischen Beweisen

**1968- 1972** Modellierung von Mikrowelten

- MACSYMA mathematische Formel-Manipulation

- DENDRAL Expertensystem zur Bestimmung von organischen Verbindungen mittels Massenspektroskopie
- SHRDLU, ELIZA: erste Versuche zur Verarbeitung natürlicher Sprache

**1972-77** Wissensverarbeitung als Technologie

- medizinisches Expertensystem: MYCIN
- D. Lenats AM (automated mathematics)
- KI-Sprachentwicklungen: PLANNER, KRL PROLOG KL-ONE

**1977-** Entwicklung von Werkzeugen Trend zur Erarbeitung von Grundlagen;

- Expertensystem-Schalen (E-MYCIN)
- Fifth generation project in Japan (beendet)
- Computerlinguistik
- künstliche neuronale Netze
- logisches Programmieren
- 1985 Doug Lenats CYC

**aktuelle Forschungsrichtungen**

- Technologie zum textuellen Sprachverstehen und zur Ermöglichung von Mensch-Computer-Dialogen. (Wahlster: Verbmobil-Projekt)
- Robotik-Ansatz: (Embodied artificial intelligence, R. Pfeifer). sinngemäß: Eine gemeinsame Untersuchung und Entwicklung von Sensorik, Motorik, d.h. physikalische Gesetzmäßigkeiten des Roboters und der Steuerung mittels Computer (neuronalen-Netzen) ist notwendig: „intelligentes Insekt“
- Automatische Deduktion
- Logik
- Robotik
- künstliche neuronale Netze
- …

### 1.5.1 Schlussfolgerungen aus den bisherigen Erfahrungen:

- Das Fernziel scheint mit den aktuellen Methoden, Techniken, Hardware und Software nicht erreichbar
- Motivationen und Visionen der KI sind heute in der Informatik verbreitet.

- Die *direkte Programmierung von Systemen* (ohne sehr gute Lernalgorithmen) hat Grenzen in der Komplexität des einzugebenden Modells (Regeln, Fakten, ...); Auch wird es zunehmend schwerer, die Systeme und die Wissensbasis zu warten und konsistent zu ändern.
  Das Projekt CYC[4] von Doug Lenat zur Erzeugung eines vernünftigen Programms mit Alltagswissen durch Eingabe einer Enzyklopädie hat nicht zum erhofften durchschlagenden Erfolg geführt.

- Manche Gebiete wie (computerunterstützte) „Verarbeitung natürlicher (schriftlicher oder gesprochener) Sprache“, Robotik, Automatische Deduktion, ...haben sich zu eigenständigen Forschungsgebieten entwickelt.

- Forschungsziele sind aktuell eher spezialisierte Aspekte bzw. anwendbare Verfahren:
  Logik und Inferenzen, Mensch-Maschine-Kommunikation, Lernverfahren (adaptive Software), Repräsentationsmechanismen; eingebettete KI, nicht-Standard-Logiken und Schlussfolgerungssysteme, ....

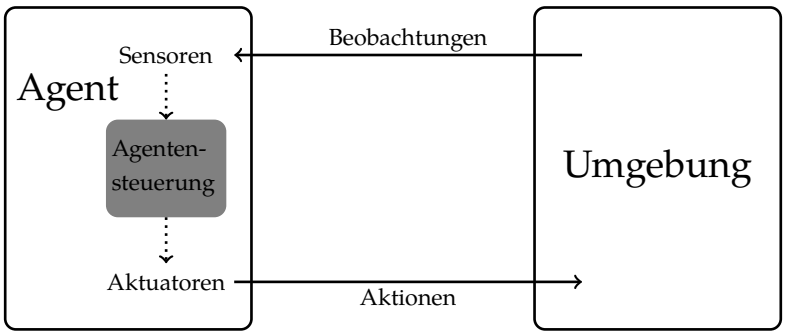
## 1.6 Intelligente Agenten

Unter dem Überbegriff *Agenten* kann man alle KI-Systeme einordnen.

Wir stellen einige einfache Überlegungen zu Agenten und deren Interaktionsmöglichkeiten mit ihrer Umgebung an.

Ein Agent hat

- *Sensoren* zum Beobachten seiner Umgebung und
- *Aktuatoren* (Aktoren; Effektoren) um die Umgebung zu manipulieren.




Er macht *Beobachtungen* (Messungen, percept). Die (zeitliche) Folge seiner Beobachtungen nennt man *Beobachtungssequenz* (percept sequence). Er kann mittels seiner Aktuatoren Aktionen ausführen, die die Umgebung, evtl. auch seine Position, beeinflussen. Wenn man das Verhalten des Agenten allgemein beschreiben will, dann kann man das als

[4] http://www.cyc.com/

*Agentenfunktion*: {Beobachtungsfolgen} → {Aktionen}.

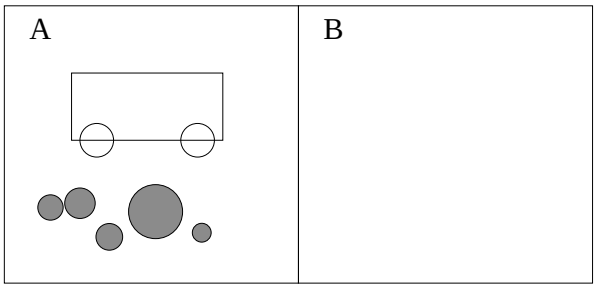
beschreiben.

Diese Agentenfunktion kann man durch das *Agentenprogramm* definieren. Prinzipiell kann man diese Funktion durch eine vollständige Tabelle beschreiben, oder durch einen Algorithmus.

**Beispiel 1.6.1** (Staubsaugerwelt (Russell & Norvig, 2010))**.** *Das ist nur eine Modellwelt, die man variieren kann, um die Problematik und die Effekte zu illustrieren, und anhand derer man sich leichter verdeutlichen kann, welche Konzepte eine Rolle spielen.*

*Der einfachste Fall ist wie folgt modelliert:*

- *Es gibt nur zwei Orte an denen sich der Agent (Staubsauger) aufhalten kann: Quadrate $A$ und $B$ (bzw. Räume $A, B$)*
- *Jedes Quadrat kann Dreck enthalten oder nicht. Der Agent kann nur sehen, ob das Quadrat in dem er sich aufhält, dreckig ist: Sauber/ Dreckig, D.h. seine Beobachtung ergibt einen Booleschen Wert.*
- *Die mögliche Aktionen des Agenten sind: `InsAndereQuadrat`, `Saugen` und `NichtsTun`.*




*Einige implizite Annahmen sind in diesm Modell versteckt:*

- *Der Agent ist nur an einem Ort.*
- *Aktionen kann er nur zur aktuellen Zeit und am aktuellen Ort vornehmen.*
- *Die Umgebung kann sich unabhängig vom Agenten verändern: Hier nur der Zustand „Dreck oder kein Dreck im Quadrat $A$ oder $B$“.*
- *Normalerweise nimmt man auch an, dass der Dreck nicht von selbst verschwindet. Jedoch kann er nicht seinen eigenen Ort beobachten, oder die Anzahl der möglichen Quadrate (Räume).*
- *Der obige Agent kann nicht beobachten, wo er ist, und seine Aktionen sind immer ausführbar. Er kann nur eine Aktion gleichzeitig ausführen.*

Die Aufgabe oder Fragestellung ist nun: Wann ist der Agent (das zugehörige Programm) gut / vernünftig bzw. intelligent ? Dazu braucht man eine Vorgabe: ein (externes) *Performanzmaß*, d.h. eine *Leistungsbewertung* des Agenten. Z.B:

- Alles soll immer maximal sauber sein. Konsequenz wäre, dass der Staubsauger immer hin und her fährt und saugt.
- Alles soll möglichst sauber sein, und der Stromverbrauch soll möglichst klein sein. Hier kann man diskrete Zeit annehmen, pro Zeiteinheit jeweils eine Anzahl Punkte pro sauberes Quadrat vergeben und jeden Wechsel des Quadrats mit einem bestimmten Abzug versehen. Die mittlere Punktzahl pro Zeit wäre dann das passende Maß.
- Alles soll möglichst sauber sein, und der Staubsauger soll möglichst wenig stören. Der Agent kann nicht beobachten, ob er stört.

Der optimale agierende Agent ist der *intelligente* Agent.

Man sieht, dass abhängig vom Performanzmaß jeweils andere Agenten als optimal zählen. Im allgemeinen ist das Performanzmaß nicht vom Agenten selbst zu berechnen, da er nicht ausreichend viele Beobachtungen machen kann. Normalerweise muss man Kompromisse zwischen den verschiedenen Kriterien machen, da nicht alle Kriterien gleichzeitig optimal zu erfüllen sind, und da auch nicht alle Kriterien zu berechnen bzw. effizient zu berechnen sind.

### 1.6.1 Gesichtspunkte, die die Güte des Agenten bestimmen

- Das Performanzmaß
- Das Vorwissen über die Umgebung
- Die möglichen Aktionen
- Die aktuelle Beobachtungsfolge

**Definition 1.6.2.** *Ein* vernünftiger (intelligenter, rationaler) Agent *ist derjenige, der stets die optimale Aktion bzgl des Performanzmaßes wählt, aufgrund seiner Beobachtungsfolge und seines Vorwissens über die Umgebung.*

Allerdings muss man beachten, dass man den vernünftigen Agenten nicht immer implementieren kann, da der Agent oft das Performanzmaß nicht berechnen kann. Zum Vorwissen kann hier z.B. die stochastische Verteilung des Verschmutzung über die Zeit gelten, d.h. wie groß ist die Wahrscheinlichkeit, dass das Quadrat $A$ in der nächsten Zeiteinheit wieder dreckig wird.

### 1.6.2 Lernen

Da normalerweise das Vorwissen über die Umgebung nicht ausreicht, oder sich die Umgebung und deren Langzeitverhalten ändern kann, ist es für einen guten Agenten notwendig

- mittels der Sensoren Wissen über die Umgebung zu sammeln;
- lernfähig zu sein, bzw. sich adaptiv zu verhalten, aufgrund der Beobachtungssequenz.

Z.B. kann eine Erkundung der Umgebung notwendig sein, wenn der Staubsauger in einer neuen Umgebung eingesetzt wird, und das Wissen dazu nicht vorgegeben ist.

Ein Agent wird als *autonom* bezeichnet, wenn der Agent eher aus seinen Beobachtungen lernt und nicht auf vorprogrammierte Aktionen angewiesen ist.

### 1.6.3 Verschiedene Varianten von Umgebungen

Umgebungen können anhand verschiedener Eigenschaften *klassfiziert* werden. Die Eigenschaften sind:

- *Vollständig beobachtbar vs. teilweise beobachtbar*.
  Der Staubsauger kann z.B. nur sein eigenes Quadrat beobachten.
- *Deterministisch vs. Stochastisch*.
  Der Dreck erscheint zufällig in den Quadraten.
- *Episodisch vs. sequentiell*.
  Episodisch: Es gibt feste Zeitabschnitte, in denen beobachtet agiert wird, und die alle unabhängig voneinander sind.
  Sequentiell. Es gibt Spätfolgen der Aktionen.
- *Statisch vs. Dynamisch*
  Dynamisch: die Umgebung kann sich während der Nachdenkzeit des Agenten verändern. Wir bezeichnen die Umgebung als semi-dynamisch, wenn sich während der Nachdenkzeit zwar die Umgebung nicht verändert, jedoch das Performanzmaß. Ein Beispiel ist Schachspielen mit Uhr.
- *Diskret vs. Stetig*.
- *Ein Agent oder Multiagenten*.
  Bei Multiagenten kann man unterscheiden zwischen Gegnern / Wettbewerber / Kooperierenden Agenten. Es gibt auch entsprechende Kombinationen.

**Beispiel 1.6.3.** *Eine Beispieltabelle aus (Russell & Norvig, 2010)*

| Arbeitsumgebung | Beobachtbar | Deterministisch | Episodisch | Statisch | Diskret | Agenten |
|---|---|---|---|---|---|---|
| *Kreuzworträtsel* | *vollst.* | *det.* | *seq.* | *statisch* | *diskret* | *1* |
| *Schach mit Uhr* | *vollst.* | *det.* | *seq.* | *semi* | *diskret* | *n* |
| *Poker* | *teilw.* | *stoch.* | *seq.* | *statisch* | *diskret* | *n* |
| *Backgammon* | *vollst.* | *stoch.* | *seq.* | *statisch* | *diskret* | *n* |
| *Taxifahren* | *teilw.* | *stoch.* | *seq.* | *dyn.* | *stetig* | *n* |
| *Medizinische Diagnose* | *teilw.* | *stoch.* | *seq.* | *dyn.* | *stetig* | *1* |
| *Bildanalyse* | *vollst.* | *det.* | *episod.* | *semi* | *stetig* | *1* |
| *Interaktiver Englischlehrer* | *teilw.* | *stoch.* | *seq.* | *dyn.* | *diskret* | *n* |

### 1.6.4 Struktur des Agenten

Ein Agent besteht aus:

- Physikalischer Aufbau inkl. Sensoren und Aktuatoren; man spricht von der *Architektur des Agenten*.
- dem Programm des Agenten

Wir beschreiben in allgemeiner Form einige Möglichkeiten:

- Tabellengesteuerter Agent: (endlicher Automat) mit einer Tabelle von Einträgen:
  Beobachtungsfolge1 $\mapsto$ Aktion1
  Beobachtungsfolge2 $\mapsto$ Aktion2
  ... ...
  Diese Beschreibung ist zu allgemein und eignet sich nicht als Programmieridee; die Tabelle ist einfach zu groß. Leicht abgeändert und optimiert, nähert man sich „normalen“ Programmen:
- Agent mit Zustand
  implementiert eine Funktion der Form (Zustand, Beobachtung) $\mapsto$ Aktion, Zustand'
  Wobei der Zustand auch die gespeicherte Folge (Historie) der Beobachtungen sein kann.
- Einfacher Reflex-Agent:
  Tabellengesteuert (endlicher Automat) mit einer Tabelle von Einträgen:
  Beobachtung1 $\mapsto$ Aktion1
  Beobachtung2 $\mapsto$ Aktion2

- Modellbasierte Strukturierung:
  Der Zustand wird benutzt, um die Umgebung zu modellieren und deren Zustand zu speichern. Z.B. beim Staubsauger: inneres Modell des Lageplan des Stockwerks; welche Räume wurden vom Staubsauger gerade gesäubert, usw.

  – Zweckbasierte Strukturierung (goalbased, zielbasiert): Die Entscheidung, welche Aktion als nächste ausgeführt wird, hängt davon ab, ob damit ein vorgegebenes Ziel erreicht wird.

    Dr Agent kann erkennen, ob er das Zeil erreicht hat.

  – Nutzenbasierte Strukturierung (utility-based, nutzenbasiert): Die Entscheidung, welche Aktion als nächste ausgeführt wird, hängt davon ab, ob damit eine vorgegebene (interne) Nutzenfunktion (internes Gütemaß) verbessert wird. Da das Performanzmaß vom Agenten selbst meist nicht berechnet werden kann, da ihm die Informationen fehlen, gibt es eine *Bewertungsfunktion* (*utility function*), die der Agent berechnen und als Bewertung verwenden kann.

- Lernende Agenten (s.o.): Das Vorsehen aller möglichen Aktionen auf alle Varianten der Beobachtung ist, auch stark optimiert, sehr aufwändig, manchmal unmöglich, da man leicht Fälle beim Programmieren übersieht, und oft die optimale Aktion nicht so einfach aus der jeweiligen Beobachtungsfolge ausrechnen kann. Die Speicherung der ganzen Beobachtungsfolge kann zudem sehr viel Speicher benötigen.

  *Lernverfahren* bieten hier einen Ausweg und erhöhen die Autonomie des Agenten.

  Da man eine Meta-Ebene ins Programm einführt, muss man unterscheiden zwischen

  – Lernmodul

  – Ausführungmodul

  Das Ausführungsmodul ist das bisher betrachtete Programm, das Aktionen berechnet, das Lernmodul verwendet die Beobachtungsfolge, um das Ausführungsmodul bzw. dessen Parameter (im Zustand gespeichert) zu verändern.

  Zum Lernen benötigt man eine Rückmeldung bzw. Bewertung (critic) der aktuellen Performanz, um dann das Ausführungsmodul gezielt zu verbessern. Das kann eine interne Bewertungsfunktion sein, aber auch ein von Außen gesteuertes Training: Es gibt verschiedene Lernmethoden dazu: online, offline, die sich weiter untergliedern. Offline-Lernen kann man als Trainingsphase sehen, wobei anhand vorgegebener oder zufällig erzeugter „Probleme“ die optimale Aktion trainiert werden kann. Das Lernverfahren kann z.B. durch Vorgabe der richtigen Aktion durch einen Lehrer, oder mittels Belohnung / Bestrafung für gute / schlechte Aktionsauswahl das *Training* durchführen.

# 2
# Suchverfahren

## 2.1 Algorithmische Suche

Oft hat man in einer vorgegebenen Problemlösungs-Situation die Aufgabe, einen bestimmten Zweck zu erreichen, einen (Aktions-)Plan für einen Agenten zu entwickeln, einen Weg zu finden o.ä. Hierzu benötigt man zunächst eine Repräsentation des aktuellen Zustandes sowie der Möglichkeiten, die man hat, um sein Ziel zu erreichen. Im Allgemeinen gibt es kein effizientes Verfahren, um den Plan zu berechnen, denn nicht alle Probleme haben so einfache Lösungen wie z.B. die Aufgabe zu zwei gegebenen Zahlen deren Produkt zu bestimmen. Man benötigt daher *Suchverfahren*. In diesem Kapitel werden wir einige Suchverfahren und deren Anwendung und Eigenschaften erörtern.

Beispiele für solche Suchprobleme sind:

- Spiele: Suche nach dem besten Zug
- Logik: Suche nach einer Herleitung einer Aussage
- Agenten: Suche nach der optimalen nächsten Aktion
- Planen: Suche nach einer Folge von Aktionen eines intelligenten Agenten.
- Optimierung: Suche eines Maximums einer mehrstelligen Funktion auf den reellen Zahlen

Wir beschränken uns auf die Fälle, in denen die Aufgabe eindeutig und exakt formulierbar ist.

Die exakte Repräsentation der Zustände, der Übergänge, der Regeln ist Teil der Programmierung und Optimierung der Suche. Meist hat man:

- Anfangssituationen
- Kalkülregeln, die es erlauben aus einer Situation die nächste zu berechnen (Spielregeln), die sogenannte *Nachfolgerfunktion*
- Ein Test auf Zielsituation

**Beispiel 2.1.1.**

**Schach:** *Gegeben ist als Anfangssituation eine Stellung im Spiel, die Nachfolgerfunktion entspricht den möglichen Zügen. Eine Zielsituation ist erreicht, wenn man gewonnen hat. Gesucht ist nun ein Zug, der zum Gewinn führt.*

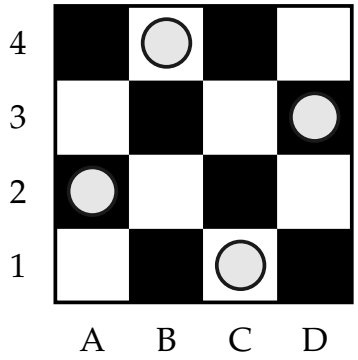
**Deduktion:** *Gegeben ist als Anfangssituation eine zu beweisende Aussage $A$ und einer Menge von Axiomen und evtl. schon bewiesenen Sätzen und Lemmas. Gesucht ist ein Beweis für $A$.*

**Planen:** *Gegeben eine formale Beschreibung des interessierenden Bereichs; z.B. Fahrplan, Anfangsort, Zeit, Zielort der zu planenden Reise. Gesucht ist ein Plan, der die Reise ermöglicht.*

### 2.1.1 Beispiel: das $n$-Damen Problem

Beim $n$-Damen Problem[1] ist ein $n \times n$-Schachfeld gegeben und gesucht ist eine Lösung dafür $n$ Damen auf dem Schachbrett so zu verteilen, dass keine Dame eine andere bedroht (Damen bedrohen sich, wenn sie auf einer horizontalen, vertikalen oder diagonalen Linie stehen).

Eine mögliche Lösung für das $4$-Damen Problem ist:




Ein möglicher Ablauf der Suche, wenn $n = 4$ ist, ist der folgende:

1. Dame A-1
2. Dame A-2; Bedrohung
3. Dame A-2 nach B-2 Bedrohung (Diagonale)
4. Dame B-2 nach C-2
5. Dame A-3; Bedrohung
6. Dame A-3 nach B-3; Bedrohung

usw

9. Dame auf Reihe 3 zurücknehmen, Dame C-2 auf D-2
10. Dame A-3
11. Dame A-3 nach B-3

usw.

[1] Wir betrachten das $n$-Damen Problem hier nur als Demonstrationsbeispiel für die Suche, denn es es gibt bessere, linear berechenbare systematische Lösungen.

### 2.1.2 Beispiel: „Missionare und Kannibalen“

> 3 Missionare und 3 Kannibalen sind auf einer Seite eines Flusses. Es gibt ein Boot, in dem höchstens zwei Personen rudern können.
> Bedingung: Auf keiner Uferseite dürfen jemals mehr Kannibalen als Missionare sein.
> Gesucht: ein Plan zur Überquerung des Flusses, der diese Bedingung einhält.

Kodierungsmöglichkeiten des Problems:
Wir betrachten aufeinanderfolgende Zustände, die aus den Anzahlen der Missionare und der Kannibalen pro Ufer sowie der Position des Bootes bestehen:

| | |
|---|---|
| 1. Am Anfang: | 3M, 3K, L, 0M, 0K |
| Ziel ist die Situation: | 0M, 0K, R, 3M, 3K |

mögliche zweite Situationen:

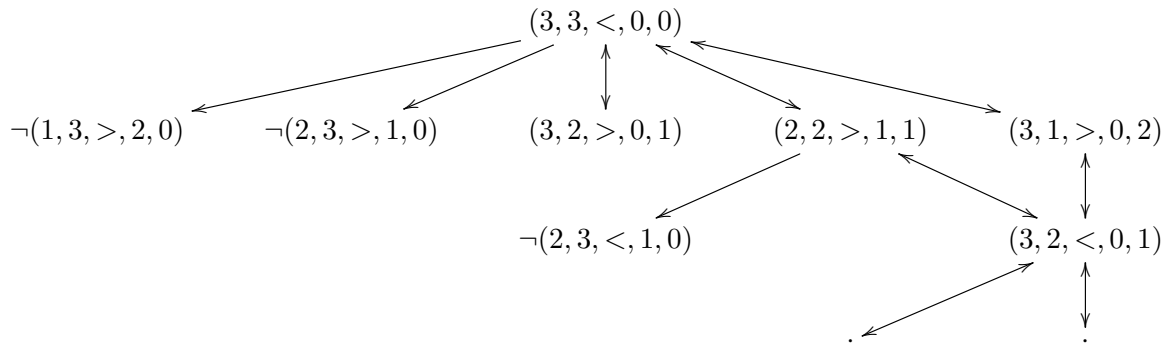
| | |
|---|---|
| 2.1 | 2M, 3K, R, 1M, 0K |
| 2.2 | 1M, 3K, R, 2M, 0K |
| 2.3 | 2M, 2K, R, 1M, 1K |
| 2.4 | 3M, 2K, R, 0M, 1K |
| 2.5 | 3M, 1K, R, 0M, 2K |

Man sieht: 2.1 und 2.2 sind verboten, die anderen erlaubt.

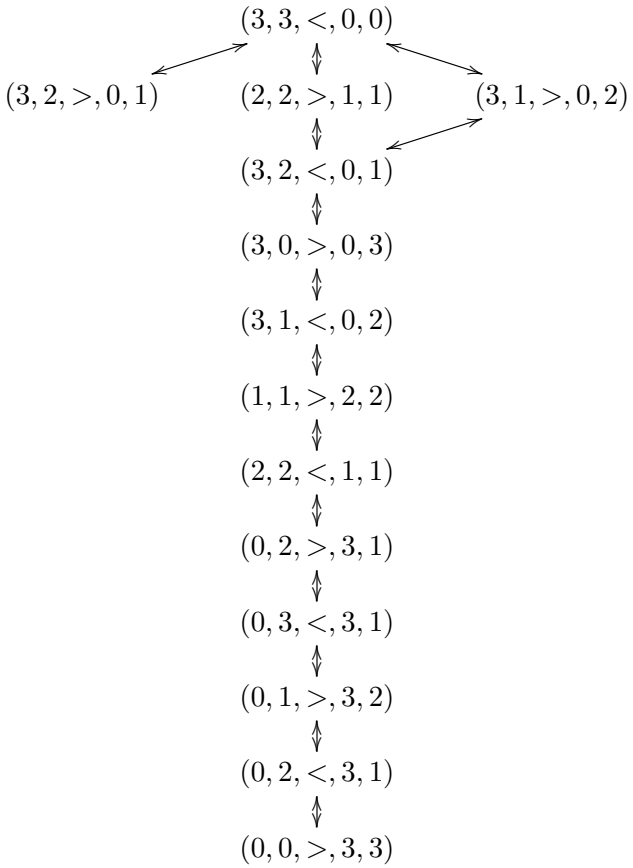
Die naive Suche würde Zyklen enthalten, die dem Hin-und-Her-Rudern entsprechen, ohne dass sich etwas ändert. Es gibt offensichtliche Verbesserungen der naiven Suche:

- Suche nicht in einem Baum aller Möglichkeiten, sondern in einem gerichteten Graphen.
- schon versuchte Situationen werden nicht noch einmal verfolgt.

Ein Diagramm des Zusammenhangs der ersten Zustände. Verbotene Zustände sind mit ¬ markiert.



Der Suchgraph ohne ungültige Situationen ist:




Allerdings ist das Problem doch nicht ganz eindeutig formuliert: Es gibt folgende Probleme der Repräsentation:

1. Die Repräsentation kann Eindeutigkeit herbeiführen, die in der Problemstellung nicht enthalten war. Zum Beispiel ist das Ein-und Aussteigen aus dem Boot nicht mitberücksichtigt: Was passiert, wenn ein Kannibale mit dem Boot am Ufer ankommt, und dort befinden sich ein Missionar und ein Kannibale, und der Missionar will im Boot zurückfahren. Ist das erlaubt oder nicht? Repräsentiert man andere Zustände, z.B. nur die Zustände während des Bootfahrens, kann es erlaubt sein.

2. Die Repräsentation bestimmt die Struktur des Suchraumes mit. Z.B. Wenn man die Missionare und Kannibalen mit Namen benennt, und als Zustand die Menge der Personen auf der Startseite des Flusses speichert und zusätzlich $B$ genau dann in die Menge einfügt, wenn das Boot auf der Startseite des Flusses ist. Dann ist die Anfangssituation: $\{M_1, M_2, M_3, K_1, K_2, K_3, B\}$
   Jetzt gibt es 21 Folgesituationen:
   1. $\{M_2, M_3, K_1, K_2, K_3\}$
   2. $\{M_1, M_3, K_1, K_2, K_3\}$

3. $\{M_1, M_2, K_1, K_2, K_3\}$
4. $\{M_1, M_2, M_3, K_2, K_3\}$
5. $\{M_1, M_2, M_3, K_1, K_3\}$
6. $\{M_1, M_2, M_3, K_1, K_2\}$
7. $\{M_3, K_1, K_2, K_3\}$
8. $\{M_2, K_1, K_2, K_3\}$
9. $\{M_2, M_3, K_2, K_3\}$
10. $\{M_2, M_3, K_1, K_3\}$
11. $\{M_2, M_3, K_1, K_2\}$
12. $\{M_1, K_1, K_2, K_3\}$
13. $\{M_1, M_3, K_2, K_3\}$
14. $\{M_1, M_3, K_1, K_3\}$
15. $\{M_1, M_3, K_1, K_2\}$
16. $\{M_1, M_2, K_2, K_3\}$
17. $\{M_1, M_2, K_1, K_3\}$
18. $\{M_1, M_2, K_1, K_2\}$
19. $\{M_1, M_2, M_3, K_3\}$
20. $\{M_1, M_2, M_3, K_2\}$
21. $\{M_1, M_2, M_3, K_1\}$

Die Anzahl der erlaubten Folgesituationen ist 12.

Offensichtlich ist in dieser Repräsentation die Suche sehr viel schwieriger.

3. Die Repräsentation hat auch eine Wahl bei der Festlegung der Knoten und der Nachfolgerfunktion. Z.B. kann man bei n-Dame auch als Nachfolger *jede* Situation nehmen, bei der eine Dame mehr auf dem Schachbrett steht

Das Finden einer geeigneten Repräsentation und die Organisation der Suche erfordert Nachdenken und ist Teil der Optimierung der Suche.

### 2.1.3 Suchraum, Suchgraph

Wenn man vom Problem und der Repräsentation abstrahiert, kann man die Suche nach einer Lösung als Suche in einem gerichteten Graphen (*Suchgraph, Suchraum*) betrachten. Hier ist zu beachten, dass der gerichtete Graph nicht explizit gegeben ist, er kann z.B. auch unendlich groß sein. Der Graph is daher *implizit* durch Erzeugungsregeln gegeben. Die Suche und partielle Erzeugung des Graphen sind somit verflochten. Der Suchgraph besteht aus:

- *Knoten* Situation, Zustände
- *Kanten* in Form einer Nachfolger-Funktion $N$, die für einen Knoten dessen Nachfolgesituationen berechnet.
- *Anfangssituation*
- *Zielsituationen*: Eigenschaft eines Knotens, die geprüft werden kann.

Diesen Graphen (zusammen mit seiner Erzeugung) nennt man auch *Suchraum*.
Wir definieren einige wichtige Begriffe.

**Definition 2.1.2.** *Die* Verzweigungrate des Knotens $K$ (*branching factor*) *ist die Anzahl der direkten Nachfolger von $K$, also die Mächtigkeit der Menge $N(K)$.*

*Die* mittlere Verzweigunsrate des Suchraumes *ist entsprechend die durchschnittliche Verzweigungrate aller Knoten.*

*Die* Größe des Suchraumes ab Knoten $K$ in Tiefe $d$ *ist die Anzahl der Knoten, die von $K$ aus in $d$ Schritten erreichbar sind, d.h. die Mächtigkeit von $\overline{N}^d(K)$, wobei*

$$\overline{N}^1(M) = \bigcup\{N(L) \mid L \in M\} \text{ und } \overline{N}^i(K) = \overline{N}(\overline{N}^{i-1}(K)).$$

*Eine* Suchstragie *ist eine Vorgehensweise zur Durchmusterung des Suchraumes. Eine Suchstrategie ist* vollständig, *wenn sie in endlich vielen Schritten (Zeit) einen Zielknoten findet, falls dieser existiert.*

**Bemerkung 2.1.3.** *Im Allgemeinen hat man eine mittlere Verzweigungsrate $c > 1$. Damit ist der Aufwand der Suche exponentiell in der Tiefe des Suchraums. Das nennt man auch* kombinatorische Explosion.

*Die meisten Suchprobleme sind NP-vollständig bzw. NP-hart. In diesem Fall benötigen alle bekannten Algorithmen im schlechtesten Fall (mindestens) exponentiellen Zeitaufwand in der Größe des Problems.*

### 2.1.4 Prozeduren für nicht-informierte Suche (Blind search)

Wir betrachten zunächst die nicht-informierte Suche, wobei nicht-informiert bedeutet, dass die Suche nur den Graphen und die Nachfolgerfunktion verwenden darf (kann), aber keine anderen Informationen über die Knoten, Kanten usw. verwenden kann.

Die *Parameter* sind

- Menge der initialen Knoten
- Menge der Zielknoten, bzw. eindeutige Festlegung der Eigenschaften der Zielknoten
- Nachfolgerfunktion $N$

*Algorithmus* **Nicht-informierte Suche**

**Datenstrukturen:** $L$ sei eine Menge von Knoten, markiert mit dem dorthin führenden Weg.

**Eingabe:** Setze $L :=$ Menge der initialen Knoten mit leerem Weg

**Algorithmus**:

1. Wenn $L$ leer ist, dann breche ab.
2. Wähle einen beliebigen Knoten $K$ aus $L$.
3. Wenn $K$ ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn $K$ kein Zielknoten, dann nehme Menge $N(K)$ der direkten Nachfolger von $K$ und verändere $L$ folgendermaßen:
   $L := (L \cup N(K)) \setminus \{K\}$ (Wege entsprechend anpassen)
   Mache weiter mit Schritt 1

Wir betrachten im folgenden Varianten der blinden Suche, die insbesondere das Wählen des Knotens $K$ aus $L$ eindeutig durchführen.

#### 2.1.4.1 Varianten der blinden Suche: Breitensuche und Tiefensuche

Die Tiefensuche verwendet anstelle der Menge der $L$ eine Liste von Knoten und wählt stets den ersten Knoten der Liste als nächsten zu betrachtenden Knoten. Außerdem werden neue Nachfolger stets *vorne* in die Liste eingefügt, was zur Charakteristik führt, dass zunächst in der Tiefe gesucht wird.

*Algorithmus* **Tiefensuche**

**Datenstrukturen:** $L$ sei eine Liste (Stack) von Knoten, markiert mit dem dorthin führenden Weg.

**Eingabe:** Füge die initialen Knoten in die Liste $L$ ein.

**Algorithmus:**

1. Wenn $L$ die leere Liste ist, dann breche ab.
2. Wähle ersten Knoten $K$ aus $L$, sei $R$ die Restliste.
3. Wenn $K$ ein Zielknoten ist, dann gebe aus: Zielknoten und Weg dorthin (d.h. Weg im Graphen dorthin)
4. Wenn $K$ kein Zielknoten, dann sei $N(K)$ die (geordnete) Liste der direkten Nachfolger von $K$, mit dem Weg dorthin markiert
   $L := N(K)$ ++ $R$. (wobei ++ Listen zusammenhängt)
   Mache weiter mit 1.

Eine einfache, rekursive Implementierung in Haskell ist:

```
dfs ::(a -> Bool)    -- Zieltest (goal)
    -> (a -> [a])     -- Nachfolgerfunktion (succ)
    -> [a]            -- Startknoten
    -> Maybe (a, [a]) -- Ergebnis: Just (Zielknoten,Pfad) oder Nothing

dfs goal succ stack =
  -- Einfuegen der Anfangspfade, dann iterieren mit go
  go [(k,[k]) | k <- stack]
 where
  go [] = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,p):r)
   | goal k    = Just (k,p) -- Ziel gefunden
   | otherwise = go ([(k',k':p) | k' <- succ k] ++ r)
```

Beachte, dass die explizite Speicherung der Pfade in Haskell nicht viel Effizienz kostet (da diese lazy ausgewertet werden). In imperativen Sprachen kann man durch geschickte Speicherung der Liste den aktuellen Suchpfad in der Liste $L$ speichern: Jeder Eintrag ist ein Knoten mit Zeiger auf den nächsten Nachbarknoten.

**Bemerkung 2.1.4** (Eigenschaften der Tiefensuche)**.** *Die* Komplexität (*worst-case*) *des Verfahrens ist bei fester Verzweigungsrate $c > 1$:*

**Platz:** *linear in der Tiefe.* (*wenn die Verzweigungsrate eine obere Schranke hat*)

**Zeit:** *entspricht der Anzahl der besuchten Knoten* (*als Baum gesehen*)*, d.h. Aufwand ist exponentiell in der Tiefe.*

*Beachte, dass Tiefensuche nicht vollständig ist, wenn der Suchgraph unendlich groß ist, denn die Suche kann am Ziel vorbeilaufen, und für immer im unendlichen langen Pfad laufen.*

#### 2.1.4.2 Pragmatische Verbesserungsmöglichkeiten der Tiefensuche

Einige offensichtliche, einfache Verbesserungsmöglichkeiten der Tiefensuche sind:

**Tiefensuche mit Tiefenbeschränkung** $k$

Wenn die vorgegebene Tiefenschranke $k$ überschritten wird, werden keine Nachfolger dieser Knoten mehr erzeugt. Die Tiefensuche findet in diesem Fall jeden Zielknoten, der höchstens Tiefe $k$ hat.

Eine Haskell-Implementierung dazu wäre (anstelle der Tiefe wird die noch verfügbare Suchtiefe mit den Knoten gespeichert):

```
dfsBisTiefe goal succ stack maxdepth =
  -- wir speichern die Tiefe mit in den Knoten auf dem Stack:
  go   [(k,maxdepth,[k]) | k <- stack]
 where
  go [] = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,i,p):r)
   | goal k = Just (k,p) -- Ziel gefunden
   | i > 0  = go ([(k',i -1, k':p) | k' <- succ k] ++ r)
   | otherwise =  go r -- Tiefenschranke erreicht
```

**Tiefensuche mit Sharing**

Nutze aus, dass der gerichtete Graph manchmal kein Baum ist. Schon untersuchte Knoten werden nicht nochmal untersucht und redundant weiterverfolgt. Implementierung durch Speichern der bereits besuchten Knoten z.B. in einer Hash-Tabelle. Eine Implementierung in Haskell dazu ist:

```
dfsSharing goal succ stack =
  -- Einfuegen der Anfangspfade, dann iterieren mit go,
  -- letzes Argument ist die Merkliste
  go [(k,[k]) | k <- stack] []
 where
  go [] mem = Nothing -- Alles abgesucht, nichts gefunden
  go ((k,p):r) mem
   | goal k    = Just (k,p) -- Ziel gefunden
   | k `elem` mem = go r mem -- Knoten schon besucht
   | otherwise    = go ([(k',k':p) | k' <- succ k] ++ r) (k:mem)
```

**Platzbedarf** Anzahl der besuchten Knoten (wegen Speicher für schon untersuchte Knoten)

**Zeit:** $O(n * \log(n))$ mit $n =$ Anzahl der untersuchten Knoten ($n$ is exponentiell in der Tiefe bei Verzweigungrate $c > 1$).

**Tiefensuche mit Sharing, aber beschränktem Speicher**

Diese Variante funktioniert wie Tiefensuche mit Sharing, aber es werden maximal nur $B$ Knoten gespeichert.
Der Vorteil gegenüber Tiefensuche ist Ausnutzen von gemeinsamen Knoten; besser als Tiefensuche mit Sharing, da die Suche weiterläuft, wenn die Speicherkapazität nicht für alle besuchten Knoten ausreicht.

#### 2.1.4.3 Effekt des Zurücksetzen: (Backtracking)

Die Tiefensuche führt Backtracking durch, wenn nicht erfolgreiche Pfade erkannt werden. Die Arbeitsweise der Tiefensuche ist: Wenn Knoten $K$ keine Nachfolger hat (oder eine Tiefenbeschränkung) überschritten wurde, dann mache weiter mit dem nächsten Bruder von $K$. Dies wird auch als chronologisches Zurücksetzen – chronological backtracking – bezeichnet.

Abwandlungen dieser Verfahren sind *Dynamic Backtracking* und *Dependency-directed Backtracking*. Diese Varianten schneiden Teile des Suchraumes ab und können verwendet werden, wenn mehr über die Struktur des Suchproblems bekannt ist. Insbesondere dann, wenn sichergestellt ist, dass man trotz der Abkürzungen der Suche auf jeden Fall noch einen Zielknoten finden kann.

Es gibt auch Suchprobleme, bei denen kein Backtracking erforderlich ist (greedy Verfahren ist möglich). Dies kann z.B. der Fall sein, wenn jeder Knoten noch einen Zielknoten als Nachfolger hat. Die Tiefensuche reicht dann aus, wenn die Tiefe der Zielknoten in alle Richtungen unterhalb jedes Knotens beschränkt ist. Anderenfalls reicht Tiefensuche nicht aus (ist nicht vollständig), da diese Suche sich immer den Ast aussuchen kann, in dem der Zielknoten weiter weg ist.

Ein vollständiges Verfahren ist die *Breitensuche*:

*Algorithmus* **Breitensuche**

**Datenstrukturen:** $L$ sei eine Menge von Knoten, markiert mit dem dorthin führenden Weg.

**Eingabe:** Füge die initialen Knoten in die Menge$L$ ein.

**Algorithmus:**

1. Wenn $L$ leer ist, dann breche ab.
2. Wenn $L$ einen Zielknoten $K$ enthält, dann gebe aus: $K$ und Weg dorthin.
3. Sonst sei $N(L)$ Menge aller direkten Nachfolger der Knoten von $L$, mit einem Weg dorthin markiert.
   Mache weiter mit Schritt 1 und $L := N(L)$.

Eine Implementierung in Haskell ist:

```
bfs goal succ start =
  go [(k,[k]) | k <- start] -- Pfade erzeugen
 where
  go [] = Nothing -- nichts gefunden
  go rs =
   case filter (goal . fst) rs of -- ein Zielknoten enthalten?
        -- Nein, mache weiter mit allen Nachfolgern
    [] -> go [(k',k':p)  | (k,p) <- rs, k' <- succ k]
        -- Ja, dann stoppe:
    (r:rs) -> Just r
```

Die Komplexität der Breitensuche bei fester Verzweigungsrate $c > 1$ ist:

**Platz:** $\sim$ Anzahl der Knoten in Tiefe $d$, d.h. $c^d$, und das ist exponentiell in der Tiefe $d$.

**Zeit:** (Anzahl der Knoten in Tiefe $d$) + Aufwand für Mengenbildung: $n + n * log(n)$, wenn $n = \#Knoten$. Bei obiger Annahme einer fester Verzweigungsrate ist das $c^d(1 + d * log(c))$.

Die Breitensuche ist vollständig! D.h. wenn es einen (erreichbaren) Zielknoten gibt, wird sie auch in endlicher Zeit einen finden.

#### 2.1.4.4 Iteratives Vertiefen (iterative deepening)

Dieses Verfahren ist ein Kompromiss zwischen Tiefen- und Breitensuche, wobei man die Tiefensuche mit Tiefenbeschränkung anwendet. Hierbei wird die Tiefenschranke iterativ erhöht. Wenn kein Zielknoten gefunden wurde, wird die ganze Tiefensuche jeweils mit größerer Tiefenschranke nochmals durchgeführt.

Eine Implementierung in Haskell ist:

```
idfs goal succ stack =
 let -- alle Ergebnisse mit sukzessiver Erh"ohung der Tiefenschranke
     alleSuchen = [dfsBisTiefe goal succ stack i | i <- [1..]]
 in
   case filter isJust alleSuchen of
     [] -> Nothing -- Trotzdem nichts gefunden
     (r:rs) -> r   -- sonst erstes Ergebnisse
```

Man beachte, dass man die Tiefe, in welcher der erste Zielknoten auftritt, nicht vorhersagen kann.

Der Vorteil des iterativen Vertiefens ist der viel geringere Platzbedarf gegenüber der Breitensuche bei Erhaltung der Vollständigkeit des Verfahrens. Man nimmt eine leichte Verschlechterung des Zeitbedarfs in Kauf, da Knoten mehrfach untersucht werden.

**Komplexitätsbetrachtung:** bei fester Verzweigungsrate $c > 1$.

**Platzbedarf:** linear in der Tiefe.

**Zeitbedarf:** Wir zählen nur die im Mittel besuchten Knoten, wenn der erste Zielknoten in Tiefe $k$ ist, wobei wir bei der Rechnung die Näherung $\sum_{i=1}^{n} a^i \approx \frac{a^{n+1}}{a-1}$ für $a > 1$ verwenden.

Tiefensuche mit Tiefenbeschränkung $k$:

$$0.5 * (\sum_{i=1}^{k} c^i) \approx 0.5 * (\frac{c^{k+1}}{c-1})$$

Iteratives Vertiefen bis Tiefe $k$:

$$\begin{aligned}
&\sum_{i=1}^{k-1} \frac{c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) = \frac{\sum_{i=1}^{k-1} c^{i+1}}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \\
= \quad & \frac{(\sum_{i=1}^{k} c^i) - c^k}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx \frac{1}{c-1}((\frac{c^{k+1}}{c-1}) - c^1) + 0.5 * (\frac{c^{k+1}}{c-1}) \\
= \quad & (\frac{c^{k+1}}{(c-1)^2}) - \frac{c}{c-1} + 0.5 * (\frac{c^{k+1}}{c-1}) \approx (\frac{c^{k+1}}{(c-1)^2}) + 0.5 * (\frac{c^{k+1}}{c-1})
\end{aligned}$$

Der Faktor des Mehraufwandes für iteratives Vertiefen im Vergleich zur Tiefensuche (mit der richtigen Tiefenbeschränkung) ergibt sich zu:

$$\frac{\frac{c^{k+1}}{(c-1)^2} + 0.5 * (\frac{c^{k+1}}{c-1})}{0.5 * (\frac{c^{k+1}}{c-1})} = \frac{\frac{c^{k+1}}{(c-1)^2}}{0.5 * (\frac{c^{k+1}}{c-1})} + 1 = \frac{2}{c-1} + 1$$

Ein Tabelle der ca.-Werte des Faktors $d = \frac{2}{c-1} + 1$ ist

| $c$ | 2 | 3 | 4 | 5 | ... | 10 |
|---|---|---|---|---|---|---|
| $d$ | 3 | 2 | $1,67$ | $1,5$ | ... | $1,22$ |

D.h. der eigentliche Aufwand ergibt sich am Horizont der Suche. Der Vergleich ist zudem leicht unfair für iteratives Vertiefen, da die Tiefensuche ja den Wert von $k$ nicht kennt.

Das Verfahren des iteratives Vertiefen wird z.B. im Automatischen Beweisen verwendet (siehe z.B. M. Stickel: A PROLOG-technology theorem prover.) Die Entscheidungsalternativen, die man beim Optimieren abwägen muss, sind:

- Speichern
- Neu Berechnen.

Stickels Argument: in exponentiellen Suchräumen greift die Intuition nicht immer: Neuberechnen kann besser sein als Speichern und Wiederverwenden.

**Bemerkung 2.1.5.** *Beim dynamischen Programmieren ist in einigen Fällen der Effekt gerade umgekehrt. Man spendiert Platz für bereits berechnete Ergebnisse und hat dadurch eine enorme Ersparnis beim Zeitbedarf.*

#### 2.1.4.5 Iteratives Vertiefen (iterative deepening) mit Speicherung

Die Idee hierbei ist, den freien Speicherplatz beim iterativen Vertiefen sinnvoll zu nutzen. Hierbei kann man zwei verschiedene Verfahren verwenden:

1. Bei jedem Restart der Tiefensuche wird der zugehörige Speicher initialisiert. Während der Tiefensuchphase speichert man Knoten, die bereits expandiert waren und vermeidet damit die wiederholte Expansion. Hier kann man soviele Knoten speichern wie möglich. Man spart sich redundante Berechnungen, aber der Nachteil ist, dass bei jeder Knotenexpansion nachgeschaut werden muss, ob die Knoten schon besucht wurden.

2. Man kann Berechnungen, die in einer Tiefensuchphase gemacht wurden, in die nächste Iteration der Tiefensuche retten und somit redundante Berechnungen vermeiden. Wenn alle Knoten der letzten Berechnung gespeichert werden können, kann man von dort aus sofort weiter machen (d.h. man hat dann Breitensuche). Da man das i.a. nicht kann, ist folgendes sinnvoll:
   Man speichert den linken Teil des Suchraum, um den Anfang der nächsten Iteration schneller zu machen, und Knoten, die schon als Fehlschläge bekannt sind, d.h. solche, die garantiert keinen Zielknoten als Nachfolger haben.

### 2.1.5 Rückwärtssuche

Die Idee der Rückwärtssuche ist:

> man geht von einem (bekannten) Zielknoten aus und versucht den Anfangsknoten zu erreichen.

Voraussetzungen dafür sind:

- Man kann Zielknoten explizit angeben (nicht nur eine Eigenschaft, die der Zielknoten erfüllen muss).
- man kann von jedem Knoten die direkten Vorgänger berechnen.

Rückwärtssuche ist besser als Vorwärtssuche, falls die Verzweigungsrate in Rückwärtsrichtung kleiner ist die Verzweigungsrate in Vorwärtsrichtung.

Man kann normale Suche und Rückwärtssuche kombinieren zur *Bidirektionalen Suche*, die von beiden Seiten sucht.

## 2.2 Informierte Suche, Heuristische Suche

Die Suche nennt man „informiert“, wenn man zusätzlich eine Bewertung von allen Knoten des Suchraumes angeben kann, d.h. eine *Schätzfunktion*, die man interpretieren kann als Ähnlichkeit zu einem Zielknoten, oder als Schätzung des Abstands zum Zielknoten. Der Zielknoten sollte ein Maximum bzw. Minimum der Bewertungsfunktion sein.

Eine Heuristik („*Daumenregel*“) ist eine Methode, die oft ihren Zweck erreicht, aber nicht mit Sicherheit. Man spricht von heuristischer Suche, wenn die Schätzfunktion in vielen praktisch brauchbaren Fällen die richtige Richtung zu einem Ziel angibt, aber möglicherweise manchmal versagt.

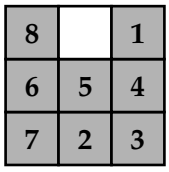
Das Suchproblem kann jetzt ersetzt werden durch:

> Minimierung (bzw. Maximierung) einer Knotenbewertung (einer Funktion) auf einem gerichteten Graphen (der aber erst erzeugt werden muss)

Variante: Maximierung in einer Menge oder in einem $n$-dimensionaler Raum.

Beispiel: 8-Puzzle

Wir betrachten als Beispiel das sogenannte 8-Puzzle. Dabei sind 8 Plättchen mit den Zahlen 1 bis beschriftet und in einem 3x3 Raster angeordnet, d.h. es gibt ein leeres Feld. Mögliche Züge sind: Plättchen können auf das freie Feld verschoben werden, wenn sie direkter Nachbar des freien Feldes sind. Ziel des Puzzles ist es, die Zahlen der Reihe nach anzuordnen.




Wir repräsentieren das 8-Puzzle als als $3 \times 3$-Matrix, wobei $a_{ij} = n$, wenn $n$ die Nummer des Plättchens ist, das an Position $(i, j)$ ist. $B$ bezeichne das leere Feld.

**Anfang:** beliebige Permutation der Matrix, z.B.

$$\begin{pmatrix} 8 & B & 1 \\ 6 & 5 & 4 \\ 7 & 2 & 3 \end{pmatrix}$$

**Ziel:**

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & B \end{pmatrix}$$

**Bewertungsfunktionen:** zwei Beispiele sind:

1. $f_1()$ Anzahl der Plättchen an der falschen Stelle
2. $f_2()$ Anzahl der Züge (ohne Behinderungen zu beachten), die man braucht, um Endzustand zu erreichen.

$$S_1 = \begin{pmatrix} 2 & 3 & 1 \\ 8 & 7 & 6 \\ B & 5 & 4 \end{pmatrix} \quad \begin{array}{l} f_1(S_1) = 7 \text{ (nur Plättchen 6 ist richtig)} \\ f_2(S_1) = 2+1+1+3+1+0+2+2 = 12 \\ \qquad \text{(Die Summe ist in der Reihenfolge } 1, \ldots, n) \end{array}$$

$$S_2 = \begin{pmatrix} 2 & 3 & 1 \\ B & 7 & 6 \\ 8 & 5 & 4 \end{pmatrix} \quad \begin{array}{l} f_1(S_2) = 7 \\ f_2(S_2) = 2+1+1+3+1+0+2+1 = 11 \end{array}$$

Man sieht, dass die zweite Bewertungsfunktion genauer angibt als die erste, wie gut die erreichte Situation ist.

Aber: es ist bekannt, dass es Ausgangssituationen gibt, von denen aus man das Ziel nicht erreichen kann (Permutation ist ungerade).

In den Unterlagen ist ein Haskell-Programm, das mit Best-First und verschiedenen Bewertungsfunktionen, nach nicht allzu langer Zeit eine Zugfolge für $3 \times 3$ findet. Wenn man dieses Puzzle für größere $n$ per Hand und per Suche probiert und vergleicht, so erkennt man, dass der Rechner sich in lokalen Optimierungen verliert, während man von Hand eine Strategie findet, die die Zielsituation herstellt, falls es überhaupt möglich ist.

Diese Strategie ist in etwa so:

- Verschiebe zunächst die 1 so, dass diese an ihrem Platz ist.
- Verschiebe zunächst die 2 so, dass diese an ihrem Platz ist.
- ……
- bei den letzten beiden Ziffern der ersten Zeile: $n-1, n$ erlaube eine Suche durch Verschieben innerhalb eines $3 \times 2$-Kästchens.
- Benutze diese Strategie bis zur vorletzten Zeile.
- Bei der vorletzten Zeile, fange von links an, jeweils zwei Plättchen korrekt zu schieben, bis das letzte $2 \times 3$ Rechteck bleibt.
- Innerhalb dieses Rechtecks probiere, bis die Plättchen an ihrem Platz sind.

Man sieht, dass einfache Bewertungsfunktionen diese Strategie nicht erzeugen. Kennt man die Strategie, kann man eine (komplizierte) Bewertungsfunktion angeben, die dieses Verhalten erzwingt. Beim n-Puzzle ist es vermutlich schwieriger diese Bewertungsfunktion zu finden, als das Problem direkt zu programmieren, wenn man eine Strategie erzwingen will. Hier braucht man eigentlich einen guten Planungsalgorithmus mit Zwischen- und Unterzielen und Prioritäten.

### 2.2.1 Bergsteigerprozedur (Hill-climbing)

Dies ist auch als Gradientenaufstieg (-abstieg)[2] bekannt, wenn man eine $n$-dimensionale Funktion in die reellen Zahlen maximieren will (siehe auch Verfahren im Bereich Operations Research).

Parameter der Bergsteigerprozedur sind

- Menge der initialen Knoten
- Nachfolgerfunktion (Nachbarschaftsrelation)
- Bewertungsfunktion der Knoten, wobei wir annehmen, dass Zielknoten maximale Werte haben (Minimierung erfolgt analog)
- Zieltest

[2] Gradient: Richtung der Vergrößerung einer Funktion; kann berechnet werden durch Differenzieren

*Algorithmus* **Bergsteigen**

**Datenstrukturen:** $L$ : Liste von Knoten, markiert mit Weg dorthin
$h$ sei die Bewertungsfunktion der Knoten
**Eingabe:** $L$ sei die Liste der initialen Knoten, absteigend sortiert entsprechend $h$

**Algorithmus**:

1. Sei $K$ das erste Element von $L$ und $R$ die Restliste
2. Wenn $K$ ein Zielknoten, dann stoppe und geben $K$ markiert mit dem Weg zurück
3. Sortiere die Liste $NF(K)$ absteigend entsprechend $h$ und entferne schon besuchte Knoten aus dem Ergebnis. Sei dies die Liste $L'$.
4. Setze $L := L'$++$R$ und gehe zu 1.

Bergsteigen verhält sich analog zur Tiefensuche, wobei jedoch stets als nächster Knoten der Nachfolger expandiert wird, der heuristisch den besten Wert besitzt. Bei *lokalen Maxima* kann die Bergsteigerprozedur eine zeitlang in diesen verharren, bevor Backtracking durchgeführt wird. Beachte, dass das Vermeiden von Zyklen unabdingbar ist, da ansonsten die Bergsteigerprozedur in lokalen Maxima hängen bleibt. Der Erfolg der Bergsteigerprozedur hängt stark von der Güte der Bewertungsfunktion ab. Ein anderes Problem ergibt sich bei *Plateaus*, d.h. wenn mehrere Nachfolger die gleiche Bewertung haben, in diesem Fall ist nicht eindeutig, welcher Weg gewählt werden soll.

Durch die Speicherung der besuchten Knoten ist der Platzbedarf linear in der Anzahl der besuchten Knoten, also exponentiell in der Tiefe.

In Haskell könnte man die Bergsteigerprozedur implementieren durch folgenden Code:

```
hillclimbing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
   list = map (\k -> (k,[k])) (sortByHeuristic start)
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger (nur neue Knoten)
           nf = (successor k) \\ mem
           -- Sortiere die Nachfolger entsprechend der Heuristik
           l' = map (\k -> (k,k:path)) (sortByHeuristic nf)
       in go  (l' ++ r) (k:mem)
   sortByHeuristic = sortBy (\a b  -> cmp (heuristic a) (heuristic b))
```

Dabei ist das erste Argument `cmp` die Vergleichsfunktion (z.B. `compare`), die es daher flexibel ermöglich zu minimieren oder zu maximieren. Das zweite Argument ist die Heuristik als Funktion von Zuständen nach Werten, das dritte Argument ist der Zieltest, das vierte die Nachfolgerfunktion und `start` ist eine Liste von initialen Zuständen.

### 2.2.2 Der Beste-zuerst (Best-first) Suche

Wir betrachten als nächste die Best-first-Suche. Diese wählt immer als nächsten zu expandierenden Knoten, denjenigen Knoten aus, der den *besten Wert* bzgl. der Heuristik hat. Der Algorithmus ist:

*Algorithmus* **Best-First Search**

**Datenstrukturen:**
Sei $L$ Liste von Knoten, markiert mit dem Weg dorthin.
$h$ sei die Bewertungsfunktion der Knoten

**Eingabe:** $L$ Liste der initialen Knoten, sortiert, so dass die besseren Knoten vorne sind.

**Algorithmus**:

1. Wenn $L$ leer ist, dann breche ab
2. Sei $K$ der erste Knoten von $L$ und $R$ die Restliste.
3. Wenn $K$ ein Zielknoten ist, dann gebe $K$ und den Weg dahin aus.
4. Sei $N(K)$ die Liste der Nachfolger von $K$. Entferne aus $N(K)$ die bereits im Weg besuchten Knoten mit Ergebnis $\mathcal{N}$
5. Setze $L := \mathcal{N}$ ++ $R$
6. Sortiere $L$, so dass bessere Knoten vorne sind und gehe zu 1.

Die best-first Suche entspricht einer Tiefensuche, die jedoch den nächsten zu expandierenden Knoten anhand der Heuristik sucht. Im Unterschied zum Bergsteigen, bewertet die Best-First-Suche stets alle Knoten im Stack und kann daher schneller zurücksetzen und daher lokale Maxima schneller wieder verlassen. In der angegebenen Variante ist die Suche nicht vollständig (analog wie die Tiefensuche). Der Platzbedarf ist durch die Speicherung der besuchten Knoten exponentiell in der Tiefe.

Der Vollständigkeit halber geben wir eine Implementierung in Haskell an:

```
bestFirstSearchMitSharing cmp heuristic goal successor start =
 let -- sortiere die Startknoten
     list = sortByHeuristic (map (\k -> (k,[k])) (start))
 in go list []
  where
   go ((k,path):r) mem
    | goal k    = Just (k,path) -- Zielknoten erreicht
    | otherwise =
       let -- Berechne die Nachfolger und nehme nur neue Knoten
           nf = (successor k) \\ mem
           -- aktualisiere Pfade
           l' = map (\k -> (k,k:path)) nf
           -- Sortiere alle Knoten nach der Heuristik
           l'' = sortByHeuristic (l' ++ r)
       in go l'' (k:mem)
   sortByHeuristic =
    sortBy (\(a,_) (b,_)-> cmp (heuristic a) (heuristic b))
```

### 2.2.3 Simuliertes Ausglühen (simulated annealing)

Die Analogie zum Ausglühen ist:
Am Anfang hat man hohe Energie und alles ist beweglich,
dann langsame Abkühlung, bis die Kristallisation eintritt (bei minimaler Energie).

Die Anwendung in einem Suchverfahren ist sinnvoll für Optimierung von $n$-dimensionalen Funktionen mit reellen Argumenten, weniger bei einer Suche nach einem Zielknoten in einem gerichteten Suchgraphen.
Bei Optimierung von $n$-dimensionalen Funktionen kann man die Argumente verändern, wobei man den Abstand der verschiedenen Argumenttupel abhängig von der Temperatur definieren kann. Bei geringerer Temperatur sinkt dieser Abstand. Der Effekt ist, dass man zunächst mit einem etwas groben Raster versucht, Anhaltspunkte für die Nähe zu einem Optimum zu finden. Hat man solche Anhaltspunkte, verfeinert man das Raster, um einem echten Maximum näher zu kommen, und um nicht über den Berg drüber zu springen bzw. diesen zu untertunneln.

Implementierung in einem Optimierungsverfahren:

1. Am Anfang große Schrittweite (passend zum Raum der Argumente). Mit diesen Werten und der Schrittweite Suche analog zu Best-First; dann allmähliche Verminderung der Schrittweite, hierbei Schrittweite und Richtung zufällig auswählen, bis sich das System stabilisiert. Die besten Knoten werden jeweils weiter expandiert.

2. Alternativ: Das System läuft getaktet. Am Anfang wird eine zufällige Menge von Knoten berücksichtigt. Bessere Knoten dürfen mit größerer Wahrscheinlichkeit im nächsten Takt Nachfolger erzeugen als schlechte Knoten. Die Wahrscheinlichkeit

für schlechte Knoten wird allmählich gesenkt, ebenso die Schrittweite.

Analoge Verfahren werden auch in künstlichen neuronalen Netzen beim Lernen verwendet. Variante 2 hat Analogien zum Vorgehen bei evolutionären Algorithmen. Ein Vorteil des Verfahrens liegt darin, dass lokale Maxima können verlassen werden. Ein Probleme des Verfahrens ist, dass es einige Parameter zu justieren sind, die stark vom Problem abhängen können:

- Schnelligkeit der „Abkühlung“
- Abhängigkeit der Wahrscheinlichkeit von der „Temperatur“ (bzw. Wert der Schätzfunktion)

## 2.3 $A^*$-Algorithmus

Die vom $A^*$-Algorithmus zu lösenden Suchprobleme bestehen aus den Eingaben:

- ein (gerichteter) Graph (wobei die Nachfolgerbeziehung i.a. algorithmisch gegeben ist), d.h. es gibt eine Nachfolgerfunktion NF, die zu jedem Knoten $N$ die Nachfolger NF$(N)$ berechnet. Wir nehmen an, dass der Graph schlicht ist, d.h. es gibt höchstens eine Kante zwischen zwei Knoten.
- eine (i.a. reellwertige) Kostenfunktion $c$ für Kanten, sodass $c(N_1, N_2) \in \mathbb{R}$ gerade die Kosten der Kante von Knoten $N_1$ zu Knoten $N_2$ festlegt.
  Für einen Weg $N_1 \to N_2 \to \ldots \to N_k$ bezeichne $c_W(N_1 N_2 \ldots N_k)$ gerade die Summe der Kosten $\sum_{i=1}^{k-1} c(N_i, N_{i+1})$.
- ein Startknoten $S$
- eine Schätzfunktion $h(\cdot)$, die die Kosten von einem bereits erreichten Knoten $N$ bis zum nächsten Zielknoten abschätzt.
- einen Test auf Zielknoten $Z$. Die Zielknoten sind meist algorithmisch definiert, d.h. man hat nur einen Test, ob ein gegebener Knoten ein Zielknoten ist.

**Ziel** ist das Auffinden eines optimalen (d.h. mit minimalen Kosten) Weges vom Startknoten zu einem der Zielknoten,

**Beispiel 2.3.1.** *Ein typisches Beispiel ist die Suche nach einem kürzesten Weg von A nach B in einem Stadtplan. Hier sind die Knoten die Kreuzungen und die Kanten die Straßenabschnitte zwischen den Kreuzungen; die Kosten entsprechen der Weglänge.*

Der $A^*$-Algorithmus stützt sich auf eine Kombination der bereits verbrauchten Kosten $g(N)$ vom Start bis zu einem aktuellen Knoten $N$ und der noch geschätzten Kosten $h(N)$ bis zu einem Zielknoten $Z$, d.h. man benutzt die Funktion $f(N) = g(N) + h(N)$. Die Funktion $h(\cdot)$ sollte leicht zu berechnen sein, und muss nicht exakt sein.

Es gibt zwei Varianten des $A^*$-Algorithmus (abhängig von den Eigenschaften von $h$).

- Baum-Such-Verfahren: Kein Update des minimalen Weges bis zu einem Knoten während des Ablaufs.
- Graph-Such-Verfahren: Update des minimalen Weges bis zu einem Knoten während des Ablaufs

Im Folgenden ist der $A^*$-Algorithmus als Graph-Such-Verfahren angegeben.

*Algorithmus* **$A^*$-Algorithmus**

**Datenstrukturen:**
- Mengen von Knoten: `Open` und `Closed`
- Wert $g(N)$ für jeden Knoten (markiert mit Pfad vom Start zu $N$)
- Heuristik $h$, Zieltest $Z$ und Kantenkostenfunktion $c$

**Eingabe:**
- `Open` $:= \{S\}$, wenn $S$ der Startknoten ist
- $g(S) := 0$, ansonsten ist $g$ nicht initialisiert
- `Closed` $:= \emptyset$

**Algorithmus**:
**repeat**
  Wähle $N$ aus `Open` mit minimalem $f(N) = g(N) + h(N)$
  **if** $Z(N)$ **then**
    **break**; // Schleife beenden
  **else**
    Berechne Liste der Nachfolger $\mathcal{N} := NF(N)$
    Schiebe Knoten $N$ von `Open` nach `Closed`
    **for** $N' \in \mathcal{N}$ **do**
      **if** $N' \in$ `Open` $\cup$ `Closed` und $g(N) + c(N, N') > g(N')$ **then**
        **skip** // Knoten nicht verändern
      **else**
        $g(N') := g(N) + c(N, N')$; // neuer Minimalwert für g(N')
        Füge $N'$ in `Open` ein und (falls vorhanden) lösche $N'$ aus `Closed`;
      **end-if**
    **end-for**
  **end-if**
**until** `Open` $= \emptyset$
**if** `Open` $= \emptyset$ **then** Fehler, kein Zielknoten gefunden
**else** $N$ ist der Zielknoten mit $g(N)$ als minimalen Kosten
**end-if**

Bei der Variante als *Baumsuche* gibt es keine `Closed`-Menge. Daher kann der gleiche Knoten mehrfach in der Menge `Open` mit verschiedenen Wegen stehen. Bei Graphsuche

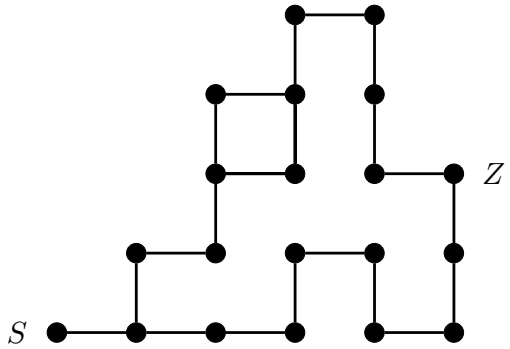
hat man immer den aktuell bekannten minimalen Weg, und somit den Knoten nur einmal in `Open`.

Beachte, dass der (bzw. ein) Zielknoten in der `Open`-Menge sein kann, dieser aber erst als Ziel erkannt wird, wenn er als zu expandierender Knoten ausgewählt wird. Ebenso ist zu beachten, dass ein Knoten in der `Closed`-Menge sein kann, und wieder in die `Open`-Menge eingefügt wird, weil ein kürzerer Weg entdeckt wird.

Eine rekursive Implementierung der $A^*$-Suche in Haskell ist:

```
-- Eintr"age in open / closed: (Knoten, (g(Knoten), Pfad zum Knoten))
aStern heuristic goal successor open closed
  | null open = Nothing -- Kein Ziel gefunden
  | otherwise =
     let n@(node,(g_node,path_node)) = Knoten mit min. f-Wert
                 minimumBy (\(a,(b,_)) (a',(b',_))
                    -> compare  ((heuristic a) + b) ((heuristic a') + b')) open
     in
      if goal node then Just n else  -- Zielknoten expandiert
       let nf = (successor node)   -- Nachfolger
           -- aktualisiere open und closed:
           (open',closed')  = update nf (delete n open) (n:closed)
           update [] o c = (o,c)
           update ((nfnode,c_node_nfnode):xs) o c =
            let (o',c') = update xs o c -- rekursiver Aufruf
                -- m"oglicher neuer Knoten, mit neuem g-Wert und Pfad
                newnode = (nfnode,(g_node + c_node_nfnode,path_node ++ [node]))
            in  case lookup nfnode open of -- Knoten in Open?
                 Nothing -> case lookup nfnode closed of -- Knoten in Closed?
                             Nothing -> (newnode:o',c')
                             Just (curr_g,curr_path) ->
                              if curr_g > g_node + c_node_nfnode
                               then (newnode:o',delete (nfnode,(curr_g,curr_path)) c')
                               else (o',c')
                 Just (curr_g,curr_path) ->
                  if curr_g > g_node + c_node_nfnode
                   then (newnode:(delete (nfnode,(curr_g,curr_path)) o'),c')
                   else  (o',c')
       in aStern heuristic goal successor open' closed'
```

**Beispiel 2.3.2.** *Im Beispiel kann man sich den Fortschritt des $A^*$-Algorithmus leicht klarmachen, wenn man eine Kante als 1 zählt, weiß wo das Ziel ist, und als Schätzfunktion des Abstands die Rechteck-Norm $|y_2 - y_1| + |x_2 - x_1|$ des Abstands vom aktuellen Knoten zum Ziel nimmt.*




*Rechnet man das Beispiel durch, dann sieht man, dass der $A^*$-Algorithmus sowohl den oberen als auch den unteren Weg versucht.*

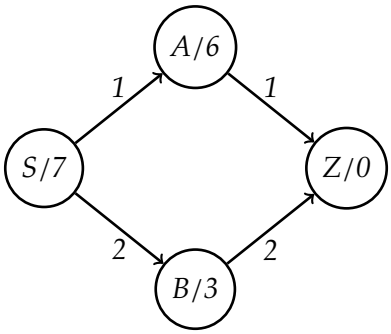
### 2.3.1 Eigenschaften des $A^*$-Algorithmus

Im folgenden einige Bezeichnungen, die wir zur Klärung der Eigenschaften benötigen.

| | | |
|---|---|---|
| $g^*(N, N')$ | = | Kosten des optimalen Weges von $N$ nach $N'$ |
| $g^*(N)$ | = | Kosten des optimalen Weges vom Start bis zu $N$ |
| $c^*(N)$ | = | Kosten des optimalen Weges von $N$ bis zum nächsten Zielknoten $Z$. |
| $f^*(N)$ | = | $g^*(N) + c^*(N)$ |
| | | (Kosten des optimalen Weges durch $N$ bis zu einem Zielknoten $Z$) |

Der $A^*$-Algorithmus hat gute Eigenschaften, wenn die Schätzfunktion $h$ die Kosten unterschätzt. D.h., wenn für jeden Knoten $N$: $h(N) \leq c^*(N)$ gilt.

**Beispiel 2.3.3.** *Wir betrachten das folgende Beispiel, mit einer* überschätzenden *Schätzfunktion:*




*Die Notation dabei ist $N/h(N)$, d.h. in jedem Knoten ist die heuristische Bewertung angegeben. Führt man den $A^*$-Algorithmus für dieses Beispiel mit Startknoten $S$ und Zielknoten $Z$ aus, erhält man den Ablauf:*

**Am Anfang**:

`Open` : $\{S\}$

`Closed` : $\{\emptyset\}$

**Nach Expansion von** $S$*:*

`Open` : $\{A, B\}$ *mit* $g(A) = 1$ *und* $g(B) = 2$

`Closed` : $\{S\}$
**Nächster Schritt**:
*Da* $f(A) = g(A) + h(a) = 1 + 6 = 7$ *und* $f(B) = g(B) + h(B) = 2 + 3 = 5$, *wird* $B$ *als nächster Knoten expandiert und man erhält:*
`Open` : $\{Z, A\}$ *mit* $g(Z) = 4$
`Closed` : $\{B, S\}$
**Nächster Schritt**:
*Da* $f(A) = g(A) + h(A) = 1 + 6 = 7$ *und* $f(Z) = g(Z) + h(Z) = 4 + 0 = 0$, *wird der Zielknoten* $Z$ *expandiert und der* $A^*$*-Algorithmus gibt den Weg* $S \to B \to Z$ *mit Kosten* $4$ *als optimalen Weg aus, obwohl ein kürzerer Weg* $S \to A \to Z$ *existiert.*

**Beispiel 2.3.4.** *Die Länge jedes Weges von* $A$ *nach* $B$ *in einem Stadtplan ist stets länger als die direkte Entfernung (Luftlinie)* $\sqrt{(b_2 - a_2)^2 + (b_1 - a_1)^2}$. *Diese einfach zu berechnende Schätzfunktion unterschätzt die echten Kosten.*

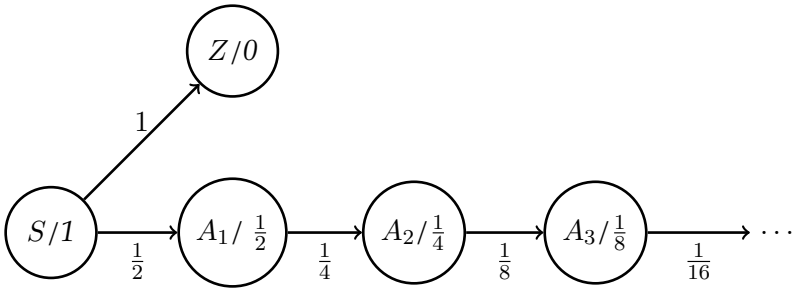
*Wenn alle Straßen rechtwinklig aufeinander zulaufen, und es keine Abkürzungen gibt, kann man auch die Rechtecknorm als Schätzung nehmen:* $|b_2 - a_2| + |b_1 - a_1|$, *denn die echten Wege können dann nicht kürzer als die Rechtecknorm sein (auch Manhattan-Abstand genannt).*

**Definition 2.3.5.** *Voraussetzungen für den* $A^*$*-Algorithmus:*

1. *es gibt nur endlich viele Knoten* $N$ *mit* $g^*(N) + h(N) \leq d$, *wobei* $d = inf\{$*Kosten aller Wege von* $S$ *zu einem Zielknoten* $Z\}$.
2. *Für jeden Knoten* $N$ *gilt:* $h(N) \leq c^*(N)$, *d.h. die Schätzfunktion ist unterschätzend.*
3. *Für jeden Knoten* $N$ *ist die Anzahl der Nachfolger endlich.*
4. *Alle Kanten kosten etwas:* $c(N, N') > 0$ *für alle* $N, N'$.
5. *Der Graph ist schlicht*[3].

Die Voraussetzung (1) der Endlichkeit der Knoten und (4) sind z.B. erfüllt, wenn es eine untere Schranke $\delta > 0$ für die Kosten einer Kante gibt.

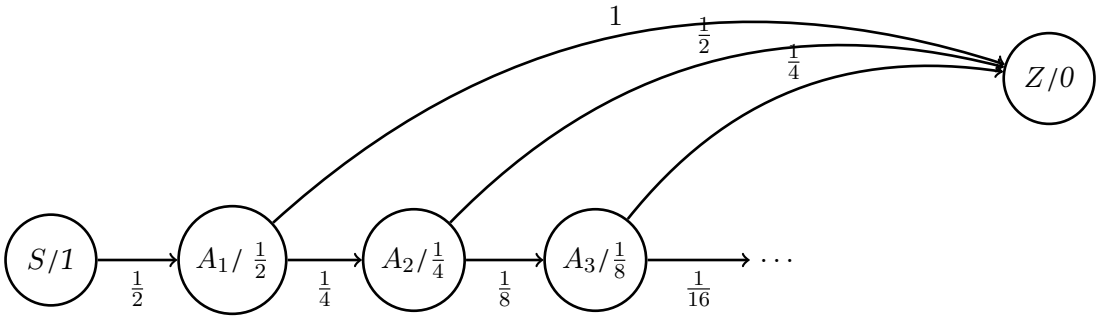
**Beispiel 2.3.6.** *Betrachte den folgenden Suchgraph, der Bedingung 1 aus Definition 2.3.5 verletzt:*




[3] schlicht = zwischen zwei Knoten gibt es höchstens eine Kante

*Dabei gehen wir davon aus, dass der Pfad $A_1 \to A_2 \to A_3 \to \ldots$ entsprechend unendlich weiter verläuft, mit $c(A_i, A_{i+1}) = \frac{1}{2^{i+1}}$ und $h(A_i) = \frac{1}{2^i}$. In diesem Fall gibt es nur den direkten Pfad $S \to Z$ als mögliche und optimale Lösung, insbesondere ist $d = 1$. Der A*-Algorithmus startet mit $S$ und fügt die Knoten $A_1$ und $Z$ in die* `Open`*-Menge ein. Anschließend wird stets für $i = 1, \ldots$ $A_i$ expandiert und $A_{i+1}$ in die* `Open`*-Menge eingefügt. Da $Z$ nie expandiert wird, terminiert der A*-Algorithmus nicht.*

**Beispiel 2.3.7.** *Das folgende Beispiel ist ähnlich zum vorhergehenden, es verletzt ebenfalls Bedingung 1 aus Definition 2.3.5. Es zeigt jedoch, dass es zum Infimum $d$ nicht notwendigerweise auch einen endlichen Weg im Suchgraphen geben muss:*




*Der Pfad $A_1 \to A_2 \to A_3 \to \ldots$ laufe entsprechend unendlich weiter, wobei $c(A_i, Z) = \frac{1}{2^{i-1}}$, $c(A_i, A_{i+1}) = \frac{1}{2^{i+1}}$ und $h(A_i) = \frac{1}{2^i}$. In diesem Fall gibt es unendlich viele Wege von $S$ nach $Z$, der Form $S \to A_1 \to \ldots A_k \to Z$ für $k = 1, 2, \ldots$. Die Kosten der Pfade sind $1\frac{1}{2}, 1\frac{1}{4}, 1\frac{1}{8}, \ldots$. Das Beispiel zeigt, daher das $d = 1$ als Infimum definiert werden muss, und dass es keinen Weg von $S$ zu $Z$ mit Kosten $1$ gibt, man aber für jedes $\epsilon > 0$ einen Weg findet, dessen Kosten kleiner $1 + \epsilon$ sind. Der A*-Algorithmus würde für obigen Graphen, nur die $A_i$-Knoten expandieren, aber nie den Zielknoten $Z$ expandieren.*

Wir zeigen nun zunächst, dass die Bedingungen aus Definition 2.3.5 ausreichen, um zu folgern, dass zum Infimum $d$ auch stets ein endlicher Weg mit Kosten $d$ existiert.

Für einen Knoten $N$ sei $infWeg(N) := inf\{$Kosten aller Wege von $N$ zu einem Zielknoten$\}$

**Satz 2.3.8.** *Es existiere ein Weg vom Start $S$ bis zu einem Zielknoten. Sei $d = infWeg(S) = inf\{$Kosten aller Wege von $S$ zu einem Zielknoten $Z\}$. Die Voraussetzungen in Definition 2.3.5 seien erfüllt.*
*Dann existiert ein optimaler Weg von $S$ zum Ziel mit Kosten $d$.*

*Beweis.* Wir zeigen mit Induktion, dass wir für jedes $n \geq 1$ einen der folgenden beiden Wege konstruieren können:

1. $S = K_1 \to \ldots \to K_i$ mit $i \leq n$, $c_W(K_1, \ldots, K_i) = d$ und $K_i$ ist ein Zielknoten
2. $S = K_1 \to \ldots \to K_n$, wobei $c_W(K_1, \ldots K_j) + infWeg(K_j) = d$ für alle $j \in \{1, \ldots, n\}$

Basisfall $n = 1$. Dann erfüllt der „Weg“ $S$ die Voraussetzung, da $infWeg(S) = d$ angenommen wurde. Für den Induktionsschritt, sei die Bedingung für $n-1$ erfüllt. Wir betrachten zwei Fälle:

- Die Induktionsannahme erfüllt Bedingung 1, d.h. es gibt einen Weg $K_1 \to \dots K_i$ (für ein $i \leq n-1$) mit $c_W(K_1, \dots, K_i) = d$ und $K_i$ ist Zielknoten. Dann erfüllt der gleiche Weg Bedingung 1 für $n$.
- Die Induktionsannahme erfüllt Bedingung 2. Sei $K_1 \to \dots \to K_{n-1}$ der entsprechende Weg. Insbesondere gilt $c_W(K_1, \dots, K_{n-1}) + infWeg(K_{n-1}) = d$. Wenn $K_{n-1}$ ein Zielknoten ist, ist Bedingung 1 erfüllt und wir sind fertig. Anderenfalls betrachte alle Nachfolger $N_1, \dots, N_m$ von $K_j$. Da es nur endliche viele Nachfolger gibt, gibt es mindestens einen Nachfolger $N_h$, so dass $c_W(K_1, \dots, K_{n-1}) + c(K_j, N_h) + infWeg(N_h) = d$. Mit $K_n := N_h$ erfüllt der Weg $K_1 \to \dots \to K_n$ Bedingung 2.

Zum Beweis der eigentlichen Aussage, müssen wir jetzt nur noch zeigen, dass es ein $n$ gibt, dass Bedingung 1 erfüllt, d.h. es gibt keinen unendlichen langen Weg der Bedingung 2 erfüllt.

Betrachte die konstruierte Folge $K_1 \to K_2 \to \dots$ gem. Bedingung 2. Ein Knoten kann nicht mehrfach in der Folge $K_0, K_1, \dots$ vorkommen, da man sonst das Infimum $d$ echt verkleinern könnte (den Zyklus immer wieder gehen). Weiterhin muss für jeden Knoten $K_j$ gelten, dass $h(K_j) \leq infWeg(K_j)$ ist, da $h$ unterschätzend ist und $infWeg(K_j)$ das entsprechende Infimum der existierenden Wege zum Ziel ist. Gäbe es unendlich viele Knoten $K_j$, dann wäre daher Bedingung 1 aus Definition 2.3.5 verletzt. Somit folgt, dass es keine unendliche Folge gibt und man daher stets einen endlichen optimalen Weg findet. □

**Lemma 2.3.9.** *Es existiere ein optimaler Weg $S = K_0 \to \dots \to K_n = Z$ vom Startknoten $S$ bis zu einem Zielknoten $Z$ . Die Voraussetzungen in Definition 2.3.5 seien erfüllt. Dann ist während der Ausführung des A*-Algorithmus stets ein Knoten $K_i$ in* `Open`*, markiert mit $g(K_i) = g^*(K_i)$, d.h. mit einem optimalen Weg von $S$ zu $K_i$.*

*Beweis.* Induktion über die Iterationen des $A^*$-Algorithmus. Für den Basisfall, genügt es zu beachten, dass $S \in$ `Open` und $g(S) = 0$ gilt, und im Anschluss $K_1$ in `Open` eingefügt wird. Da $S \to K_1$ und $K_1$ auf einem optimalen Weg liegt, muss $K_1$ mit $g(K_1) = c(S, K_1) = g^*(S, K_1)$ markiert werden (sonst gäbe es einen kürzeren Weg von $S$ zu $Z$).

Induktionsschritt: Als Induktionshypothese verwenden wir, dass `Open` mindestens einen der Knoten $K_i$ enthält, der mit $g(K_i) = g^*(K_i)$ markiert ist. Sei $j$ maximal, so dass $K_j$ diese Bedingungen erfüllt. Wir betrachten unterschiedliche Fälle:

- Ein Knoten $\neq K_j$ wird expandiert. Dann verbleibt $K_j$ in `Open` und der Wert $g(K_j)$ wird nicht verändert, da er bereits optimal ist.

- Der Knoten $K_j$ wird expandiert. Dann wird $K_{j+1}$ in `Open` eingefügt und erhält den Wert $g(K_{j+1}) = g(K_j) + c(K_j, K_{j+1}) = g^*(K_j) + c(K_j, K_{j+1})$ Dieser Wert muss optimal sein, da ansonsten der Weg $S \to K_1 \to \ldots \to K_n$ nicht optimal wäre.

□

**Lemma 2.3.10.** *Unter der Annahme, dass die Bedingungen aus 2.3.5 erfüllt sind, gilt: Wenn $A^*$ einen Zielknoten expandiert, dann ist dieser optimal.*

*Beweis.* Nehme an $S = K_0 \to K_1 \ldots K_n = Z$ ist ein optimaler Weg. Angenommen, ein Zielknoten $Z'$ wird expandiert. Dann ist $Z'$ der Knoten aus `Open` mit dem minimalen $f(Z') = g(Z') + h(Z') = g(Z')$. Betrachte den Knoten $K_j$ mit maximalem $j$ aus dem optimalen Weg, der noch in `Open` ist und mit $g(K_j) = g^*(K_j)$ markiert ist (gem. Lemma 2.3.9). Wenn $Z = Z'$ und genau auf dem Weg erreicht wurde, dann ist genau der optimale Weg $S \to K_1 \to \ldots \to Z$ gefunden worden. Wenn $K_j \neq Z'$, dann gilt $f(K_j) = g(K_j) + h(K_j) \leq d$ da $h$ die Kosten bis zum Ziel unterschätzt. Da $Z'$ expandiert wurde, gilt $f(Z') = g(Z') + h(Z') = g(Z') \leq g(K_j) + h(K_j) \leq d$, und daher ist der gefundene Weg optimal. □

**Lemma 2.3.11.** *Unter der Annahme, dass die Bedingungen aus 2.3.5 erfüllt sind, und ein Weg vom Start zum Zielknoten existiert gilt: Der $A^*$-Algorithmus expandiert einen Zielknoten nach endlich vielen Schritten.*

*Beweis.* Seien $d$ die Kosten des optimalen Wegs $S = K_0 \to K_1 \ldots \to K_n = Z$. Aufgrund von Lemma 2.3.9 genügt es zu zeigen, dass jeder der Knoten $K_i$ nach endlich vielen Schritten expandiert wird, oder ein anderer Zielknoten wird expandiert. Da stets ein $K_i$ in `Open` mit Wert $g(K_i) = g^*(K_i)$ markiert ist und $g^*(K_i) + h(K_i) \leq d$ gelten muss (die Heuristik ist unterschätzend), werden nur Knoten $L$ expandiert für die zum Zeitpunkt ihrer Expansion gilt: $g(L) + h(L) \leq g^*(K_i) + h(K_i) \leq d$. Da $g^*(L) + h(L) \leq g(L) + h(L)$, gilt auch $g^*(L) + h(L) \leq d$ für all diese Knoten $L$. Gemäß Bedingung 1 aus Definition 2.3.5 kann es daher nur endlich viele solcher Knoten $L$ geben, die vor $K_i$ expandiert werden. □

Aus den Lemmas 2.3.10 und 2.3.11 und Satz 2.3.8 folgt direkt:

**Theorem 2.3.12.** *Es existiere ein Weg vom Start bis zu einem Zielknoten. Die Voraussetzungen in Definition 2.3.5 seien erfüllt. Dann findet der $A^*$-Algorithmus einen optimalen Weg zu einem Zielknoten.*

Beachte, dass der $A^*$-Algorithmus auch einen anderen optimalen Weg finden kann, der von $K_1, \ldots, K_n$ verschieden ist.

**Beispiel 2.3.13.** *Für das Stadtplan-Beispiel können wir folgern, dass der $A^*$-Algorithmus im Falle des Abstandsmaßes $h(.)$, das die Luftlinienentfernung zum Ziel misst, einen optimalen Weg finden wird. Hat der Stadtplan nur Straßen in zwei zueinander senkrechten Richtungen, dann kann man auch die Rechtecknorm nehmen.*

Der Aufwand des $A^*$-Algorithmus ist i.a. exponentiell in der Länge des optimalen Weges (unter der Annahme einer konstanten Verzweigungsrate).

**Bemerkung 2.3.14.** *Wenn man den endlichen Graphen als gegeben annimmt, mit Bewertungen der Kanten, dann kann man mit dynamischem Programmieren in polynomieller Zeit (in der Größe des Graphen) alle kostenoptimalen Wege berechnen, indem man analog zur Berechnung der transitiven Hülle eine Matrix von Zwischenergebnissen iteriert (Floyd-Algorithmus für kürzeste Wege, Dijkstra-Algorithmus).*

*Der Dijkstra-Algorithmus entspricht dem $A^*$-Algorithmus, wenn man die Schätzfunktion $h(N) = 0$ setzt für alle Knoten $N$. Dieser hat als Aufgabe, einen Weg mit minimalen Kosten zwischen zwei Knoten eines gegebenen endlichen Graphen zu finden.*
*In diesem Fall hat man stets eine Front (`Open`) der Knoten mit minimalen Wegen ab dem Startknoten.*

### 2.3.2 Spezialisierungen des $A^*$-Algorithmus:

**Beispiel 2.3.15.** *Weitere Beispiele für Graphen und Kostenfunktionen sind:*

*Wenn $h(N) = 0$, dann ist der $A^*$-Algorithmus dasselbe wie die sogenannte Gleiche-Kostensuche (branch-and-bound)*
*Wenn $c(N_1, N_2) = k$ ($k$ Konstante, z.B. 1), und $h(N) = 0$, dann entspricht der $A^*$-Algorithmus der Breitensuche.*

Die Variante $A^{*o}$ des $A^*$-Algorithmus, die alle optimalen Weg finden soll, hat folgende Ergänzung im Algorithmus: sobald ein Weg zum Ziel gefunden wurde, mit einem Wert $d$, werden nur noch Knoten in `Open` akzeptiert (d.h. die anderen kommen nach `Closed`), bei denen $g(N) + h(N) \leq d$. Der Algorithmus stoppt erst, wenn keine Knoten mehr in `Open` sind.

**Theorem 2.3.16.** *Es gelten die Voraussetzung von Satz 2.3.12. Dann findet der Algorithmus $A^{*o}$ alle optimalen Wege von $S$ nach $Z$.*

*Beweis.* Zunächst findet der $A^*$-Algorithmus einen optimalen Weg zu einem Ziel. Daraus ergibt sich eine obere Schranke $d$ für die Kosten eines optimalen Weges. Aus den Voraussetzungen ergibt sich, dass der $A^{*o}$-Algorithmus nur endlich viele Knoten und Wege inspiziert. □

**Definition 2.3.17.** *Wenn man zwei Schätzfunktionen $h_1$ und $h_2$ hat mit:*

1. *$h_1$ und $h_2$ unterschätzen den Aufwand zum Ziel*
2. *für alle Knoten $N$ gilt: $h_1(N) \leq h_2(N) \leq c^*(N)$*

*dann nennt man* $h_2$ besser informiert *als* $h_1$.

Hieraus alleine kann man noch nicht folgern, dass der $A^*$-Algorithmus zu $h_2$ sich besser verhält als zu $h_1$. (Übungsaufgabe)

Notwendig ist: Die Abweichung bei Sortierung der Knoten mittels $f$ muss klein sein. D.h. optimal wäre $f(k) \leq f(k') \Leftrightarrow f^*(k) \leq f^*(k')$.

Der wichtigere Begriff ist:

**Definition 2.3.18.** *Eine Schätzfunktion* $h(.)$ *ist* monoton, *gdw.*
$h(N) \leq c(N, N') + h(N')$ *für alle Knoten* $N$ *und deren Nachfolger* $N'$ *und wenn* $h(Z) = 0$ *ist für Zielknoten* $Z$.

Offenbar gilt die Monotonie-Ungleichung auch für die optimale Weglänge von $N$ nach $N'$, wenn $N'$ kein direkter Nachfolger von $N$ ist. Dies kann man so sehen:
$h(N) \leq c(N, N_1) + h(N_1) \leq c(N, N_1) + c(N_1, N_2) + h(N_2)$
Iteriert man das, erhält man: $h(N) \leq c_W(N, N') + h(N')$
Und damit auch $h(N) \leq g^*(N, N') + h(N')$
Für den Weg zum Ziel $Z$ gilt damit $h(N) \leq g^*(N, Z) + h(Z) = g^*(N, Z)$ , da $h(Z) = 0$ ist.

**Lemma 2.3.19.** *Eine monotone Schätzfunktion* $h(.)$ *ist unterschätzend.*

Die Monotonie der Schätzfunktion entspricht der Dreiecksungleichung in Geometrien und metrischen Räumen.
Die Monotonieeigenschaft beschreibt das Verhalten der Funktion $f$ beim Expandieren: Wenn die Schätzfunktion monoton ist, dann wächst der Wert von $f$ monoton, wenn man einen Weg weiter expandiert.

**Satz 2.3.20.** *Ist die Schätzfunktion* $h$ *monoton, so expandiert der* $A^*$*-Algorithmus jeden untersuchten Knoten beim ersten mal bereits mit dem optimalen Wert. D.h.* $g(N) = g^*(N)$ *für alle expandierten Knoten.*

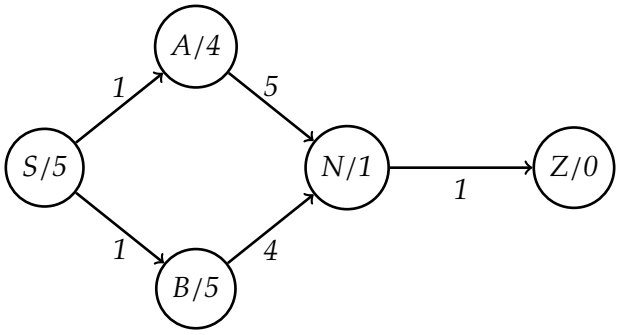
*Beweis.* Sei $S = K_1 \to \ldots \to K_n$ ein optimaler Weg von $S$ nach $K_n$. Wir nehmen an die Aussage sei falsch und es gelte $g(K_n) > g^*(K_n)$ und $K_n$ wird expandiert. Aus Lemma 2.3.9 folgt: Es gibt einen Knoten $K_i \neq K_n$ und $(i < n)$, der in `Open` ist und für den gilt $g(K_i) = g^*(K_i)$.

$$\begin{aligned} f(K_i) &= g^*(K_i) + h(K_i) \\ &\leq g^*(K_i) + g^*(K_i, K_N) + h(K_N) \quad \text{folgt aus Monotonie} \\ &= g^*(K_N) + h(K_N) \\ &< g(K_N) + h(K_N) = f(K_N). \end{aligned}$$

Da $f(K_i) < f(K_N)$ müsste aber $K_i$ anstelle von $K_n$ expandiert werden. D.h. wir haben einen Widerspruch gefunden. □

Diese Aussage erlaubt es bei monotonen Schätzfunktionen, den $A^*$-Algorithmus zu vereinfachen: man braucht kein Update der optimalen Weglänge in `Closed` durchzuführen. D.h. die Baumsuchvariante des $A^*$-Algorithmus reicht aus.

**Beispiel 2.3.21.** *Die Aussage bedeutet nicht, dass nicht schon Knoten des optimalen Weges vorher in `Open` sind. Diese könnten auf einem nichtoptimalen Weg erreicht werden, werden dann aber nicht expandiert. Das folgende Beispiel zeigt dies.*




*Der $A^*$-Algorithmus expandiert zunächst $S$, dann $A$. Bei der Expansion von $A$ wird $N$ in `Open` eingefügt (mit dem bis dahin gefundenden Weg $S \to A \to N$), aber der optimale Weg von $S$ zu $N$ ist noch nicht entdeckt. Im nächsten Schritt wird $B$ expandiert, was zur Entdeckung des Weges $S \to B \to N$ führt. Erst jetzt wird der Knoten $N$ expandiert.*

**Theorem 2.3.22.** *Ist die Schätzfunktion $h$ monoton, so findet der $A^*$-Algorithmus auch in der Baum-Variante (ohne update) den optimalen Weg zum Ziel.*

**Bemerkung 2.3.23.** *Der Dijkstra-Algorithmus zum Finden optimaler Wege in einem Graphen entspricht (wenn man von der Speicherung des optimalen Weges absieht) folgender Variante:*
*Man wählt $h() = 0$ und nimmt den $A^*$-Algorithmus.*
*Dann ist $h()$ monoton und es gelten die Aussagen für monotone Schätzfunktionen.*

Es gelten folgende weitere Tatsachen für monotone Schätzfunktionen:

**Satz 2.3.24.** *Unter den Voraussetzungen von Aussage 2.3.20 (d.h. Monotonie von $h$) gilt:*

1. *Wenn $N$ später als $M$ expandiert wurde, dann gilt $f(N) \geq f(M)$.*
2. *Wenn $N$ expandiert wurde, dann gilt $g^*(N) + h(N) \leq d$ wobei $d$ der optimale Wert ist.*
3. *Jeder Knoten mit $g^*(N) + h(N) \leq d$ wird von $A^{*o}$ expandiert.*

**Theorem 2.3.25.** *Wenn eine monotone Schätzfunktion gegeben ist, die Schätzfunktion in konstanter Zeit berechnet werden kann, dann läuft der $A^*$-Algorithmus in Zeit $O(|D|)$, wenn $D = \{N \mid g^*(N) + h(N) \leq d\}$.*
*Nimmt man die Verzweigungsrate $c$ als konstant an, $d$ als Wert des optimalen Weges und $\delta$ als der kleinste Abstand zweier Knoten, dann ist die Komplexität $O(c^{\frac{d}{\delta}})$.*

In einem rechtwinkligen Stadtplan, in dem die Abstände zwischen zwei Kreuzungen alle gleich sind, gilt, dass der $A^*$-Algorithmus in quadratischer Zeit abhängig von der Weglänge des optimalen Weges einen Weg findet. Das liegt daran, dass es nur quadratisch viele Knoten in allen Richtungen gibt. Im allgemeinen kann die Suche exponentiell dauern (z.B. mehrdimensionale Räume)

**Beispiel 2.3.26.** *Die Suche eines kürzesten Weges im Stadtplan mit dem Luftlinienabstand als Schätzfunktion ist monoton: Mit $ll()$ bezeichnen wir den Luftlinienabstand. Es gilt wegen der Dreiecksungleichung: $ll(N, Z) < ll(N, N') + ll(N', Z)$. Da der echte Abstand stets größer als Luftlinie ist, gilt: $ll(N, N') \leq c(N, N')$, und mit der Bezeichnung $h(\cdot) := ll(\cdot)$ gilt $h(N) \leq c(N, N') + h(N')$.*

*Dasselbe gilt für das Beispiel des gitterförmigen Stadtplans, bei dem man die Rechtecknorm nimmt, da auch diese die Dreiecksungleichung erfüllt.*

*Man sieht auch, dass in einem quadratischen Stadtplan die Rechtecknorm besser informiert ist als die Luftlinienentfernung.*

**Satz 2.3.27.** *es gelten die Voraussetzungen in Satz 2.3.12. Seien $d$ die Kosten des optimalen Weges. Seien $h_1, h_2$ zwei monotone Schätzfunktionen, so dass $h_2$ besser informiert sei als $h_1$.*

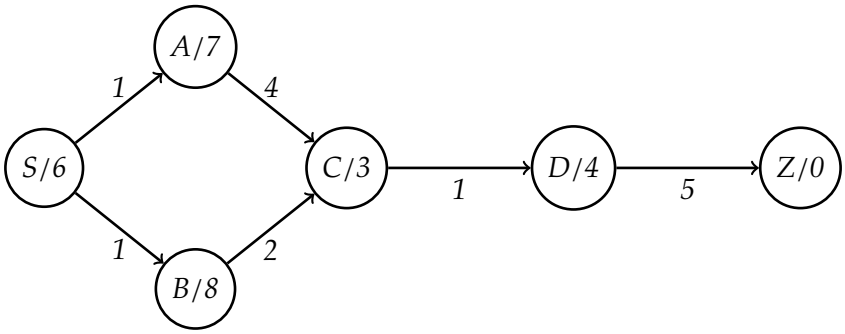
*Sei $A_1$ der $A^{*o}$-Algorithmus zu $h_1$ und $A_2$ sei der $A^{*o}$-Algorithmus zu $h_2$.*
*Dann gilt: Alle Knoten $N$ mit $g^*(N) + h_2(N) \leq d$ die von $A_2$ expandiert werden, werden auch von $A_1$ expandiert.*

*Beweis.* Der $A^{*o}$-Algorithmus stoppt erst, wenn er keine Knoten mehr findet, die unter der Grenze sind. Ein Knoten $N$ wird von $A_1$ expandiert, wenn es einen Weg von $S$ nach $N$ gibt, so dass für alle Knoten $M$ auf dem Weg die Ungleichung $g^*(M) + h_2(M) \leq d$ gilt. Da wegen $h_1(M) \leq h_2(M)$ dann $g^*(M) + h_1(M) \leq d$ gilt, wird der Knoten auch von $A_1$ expandiert

□

Was macht man, wenn die Schätzfunktion nicht monoton ist? Im Fall einer unterschätzenden Funktion $h$ gibt es eine Methode der Korrektur während des Ablaufs des Algorithmus.
Angenommen, $N$ hat Nachfolger $N'$ und es gilt $h(N) > c(N, N') + h(N')$. Nach Voraussetzung ist der optimale Wert eines Weges von $N$ zum Zielknoten größer als $h(N)$. Wenn es einen Weg von $N'$ zu einem Ziel gäbe der besser als $h(N) - c(N, N')$ ist, dann hätte man einen Widerspruch zur Unterschätzung. Damit kann man $h'(N') := h(N) - c(N, N')$ definieren. Danach gilt $h(N) = c(N, N') + h'(N')$.

**Beispiel 2.3.28.** *Es kann vorkommen, dass ein Knoten im normalen $A^*$-Algorithmus von `Open` nach `Closed` geschoben wird, aber danach nochmal nach `Open` bewegt wird, da man doch einen besseren Weg gefunden hat. Betrachte den Suchgraph:*




*Die Expansionsreihenfolge ist:*

| *expandiert* | *Nachfolger* | `Open` | `Closed` |
|---|---|---|---|
| *S* | *A,B* | *S* | |
| *A* | *C* | *A,B* | *S* |
| *C* | *D* | *B,C* | *S,A* |
| *B* | *C* | *B,D* | *S,A,C* |
| *C* | *D* | *C,D* | *S,A,B* |
| ... | | | |

*Der Knoten $C$ wird daher von* `Closed` *erneut nach* `Open` *geschoben. Diese Schätzfunktion kann nicht monoton sein. Es gilt* $8 = h(B) > c(B, C) + h(C) = 5$.

*Was dieses Beispiel auch noch zeigt, ist dass der Reparaturmechanismus, der dynamisch $h$ zu einer monotonen Schätzfunktion verbessert, das Verhalten des Algorithmus nicht so verbessert, dass Aussage 2.3.20 gilt, denn die Aussage gilt nicht für dieses Beispiel unter der Annahme der nachträglichen Korrektur von $h$.*

### 2.3.3 IDA*-Algorithmus und SMA*-Algorithmus

Da der A*-Algorithmus sich alle jemals besuchten Knoten merkt, wird er oft an die Speichergrenze stoßen und aufgeben. Um das zu umgehen, nimmt man eine Kombination des abgeschwächten $A^*$-Algorithmus und des iterativen Vertiefens. Die Rolle der Grenze spielt in diesem Fall der maximale Wert $d$ eines Weges, der schon erzeugt wurde.

IDA* mit Grenze $d$:

- Ist analog zu $A^*$.
- es gibt keine `Open/Closed`-Listen, nur einen Stack mit Knoten und Wegekosten.
- der Wert $g(N)$ wird bei gerichteten Graphen nicht per Update verbessert.
- Der Knoten $N$ wird nicht expandiert, wenn $f(N) > d$.
- das Minimum der Werte $f(N)$ mit $f(N) > d$ wird das $d$ in der nächsten Iteration.

Dieser Algorithmus benötigt nur linearen Platz in der Länge des optimalen Weges, da er nur einen Stack (Rekursions-Stack des aktuellen Weges) verwalten muss.

Da dieser Algorithmus in einem gerichteten Graphen möglicherweise eine exponentielle Anzahl von Wegen absuchen muss, obwohl das beim $A^*$-Algorithmus durch Update zu vermeiden ist, gibt es eine pragmatische Variante, den SMA$^*$-Algorithmus, der wie der $A^*$-Algorithmus arbeitet, aber im Fall, dass der Speicher nicht reicht, bereits gespeicherte Knoten löscht. Der Algorithmus nutzt hierzu die verfügbare Information und löscht die Knoten, der am ehesten nicht zu einem optimalen Weg beiträgt. Das sind Knoten, die hohe $f$-Werte haben. Es gibt weitere Optimierungen hierzu (siehe Russel-Norvig).

## 2.4 Suche in Spielbäumen

Ziel dieses Kapitels ist die Untersuchung von algorithmischen Methoden zum intelligenten Beherrschen von strategischen *Zweipersonenspielen*, wie z.B. Schach, Dame, Mühle, Go, Tictactoe, bei dem der Zufall keine Rolle spielt (siehe auch 2 Artikel aus (Wegener, 1996)).

Spiele mit mehr als zwei Personen sind auch interessant, aber erfordern andere Methoden und werden hier nicht behandelt.

Zum Schachspiel existierten bereits vor Benutzung von Computern Schachtheorien, Untersuchungen zu Gewinnstrategien in bestimmten Spielsituationen (Eröffnungen, Endspiel, usw) und zum Bewerten von Spielsituationen. Mittlerweile gibt es zahlreiche Varianten von recht passabel spielenden Schachprogrammen, und auch einige sehr spielstarke Programme.

Es gibt zwei verschiedene Ansätze, ein Programm zum Schachspielen zu konzipieren:

1. Man baut auf den bekannten Schachtheorien und Begriffen auf, wie Verteidigung, Angriff, Bedrohung, Decken, Mittelfeld, ..., und versucht die menschliche Methode zu programmieren.

2. Man versucht mit reiner Absuche aller Möglichkeiten unter Ausnutzung der Geschwindigkeit eines Computers, einen guten nächsten Zug zu finden.

Betrachtet man heutige, erfolgreiche Schachprogramme, dann ist die wesentliche Methode das Absuchen der Varianten, wobei auf einer (einfachen) statischen Bewertung von Spielsituationen aufgebaut wird. Ergänzend dazu wird eine Eröffnungsbibliothek verwendet und Endspielvarianten werden getrennt programmiert. Die Programmierung der meisten Schachtheorien scheitert daran, dass sie nicht auf Computer übertragbar sind, da die Begriffe unscharf definiert sind.

Wir betrachten zunächst sogenannte Spielbäume, die ein Zweipersonenspiel repräsentieren:

**Definition 2.4.1** (Spielbaum)**.** *Es gibt zwei Spieler, die gegeneinander spielen. Diese werden aus bald ersichtlichen Gründen* Maximierer *und* Minimierer *genannt. Wir nehmen an, es gibt*

*eine* Darstellung *einer* Spielsituation *(z.B. Schachbrett mit Aufstellung der Figuren), und die Angabe* welcher der beiden Spieler am Zug ist. *Der Spielbaum dazu ist wiefolgt definiert:*

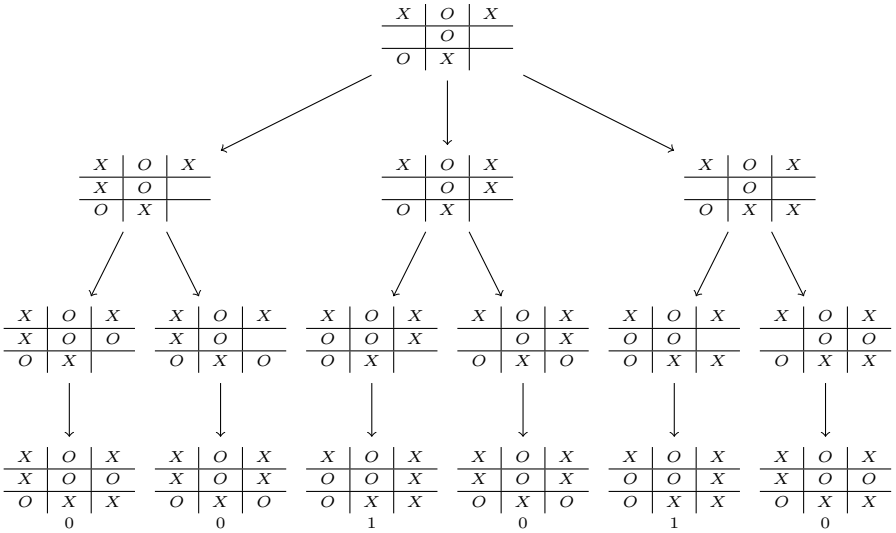
- *Die Wurzel ist mit der aktuellen Spielsituation markiert.*
- *Für jeden Knoten sind dess Kinder genau die Spielsituationen die durch den nächsten Zug des aktuellen Spielers möglich sind. Pro Ebene wechselt der Spieler*
- *Blätter sind Knoten, die keine Nachfolger mehr haben, d.h. Endsituation des Spiels*
- *Blätter werden mit einem* Gewinnwert *bewertet. Normalerweise ist die eine ganze Zahl. Bei Schach z.B.: 1, wenn der Maximierer gewonnen hat, -1, wenn der Minimierer gewonnen hat, 0 bei Remis. Bei anderen Spielen kann dies auch eine Punktzahl o.ä. sein.*

Als Beispiel betrachten wir TicTacToe: In diesem Spiel wird eine Anfangs leere (3x3)-Matrix abwechselnd durch die beiden Spieler mit X und O gefüllt. Ein Spieler gewinnt, wenn er drei seiner Symbole in einer Reihe hat (vertikal, horizontal oder Diagonal). Wir nehmen an, dass Maximierer X als Symbol benutzt und Minimierer O.

Betrachte z.B. den Spielbaum zur Situation

| | | |
|---|---|---|
| X | O | X |
| | O | |
| O | X | |

wobei der Maximierer am Zug ist.




Der Spielbaum zeigt schon, dass der Maximierer keinesfalls verlieren wird, aber kann er sicher gewinnen? Und wenn ja, welchen Zug sollte er als nächsten durchführen?

Das Ziel der Suche besteht darin, den optimalen nächsten Zug zu finden, d.h. das Finden einer Gewinnstrategie. Dabei soll zu jeder Spielsituation der optimale Zug gefunden

werden. Als Sprechweise kann man verwenden, dass man einen einzelnen Zug Halbzug nennt; ein Zug ist dann eine Folge von zwei Halbzügen.

Für die betrachteten Spiele ist die Menge der Spielsituationen endlich, also kann man im Prinzip auch herausfinden, welche Spielsituationen bei optimaler Spielweise zum Verlust, Gewinn bzw. Remis führen.

Wenn man Spiele hat, die zyklische Zugfolgen erlauben, bzw. wie im Schach bei dreifach sich wiederholender Stellung (bei gleichem Spieler am Zug) die Stellung als Remis werten, dann ist die folgende Minmax-Prozedur noch etwas zu erweitern.

Die Suche im Spielbaum der Abspielvarianten mit einer Bewertungsfunktion wurde schon von den Pionieren der Informatik vorgeschlagen (von Shannon, Turing; siehe auch die Übersicht in Russel und Norvig). Dies nennt man die **Minmax-Methode**.

*Algorithmus* **MinMax-Suche**

**Datenstrukturen:** Nachfolgerfunktion, Wert(Zustand) für Endsituationen, Datenstruktur für Spielzustand, Spieler

**Funktion** Minmax(Zustand,Spieler):
$NF :=$ alle Nachfolgezustände zu (Zustand,Spieler);
`if` $NF = \emptyset$ `then return` Wert(Zustand) `else`
  `if` Spieler == Maximierer `then`
    `return` $max\{\text{Minmax}(Z, \overline{\text{Spieler}}) \mid Z \in NF\}$
  `else` // Spieler ist Minimierer
    `return` $min\{\text{Minmax}(Z, \overline{\text{Spieler}}) \mid Z \in NF\}$
  `endif`
`endif`
Wobei $\overline{\text{Spieler}}$ der jeweils anderen Spieler bedeuten soll.

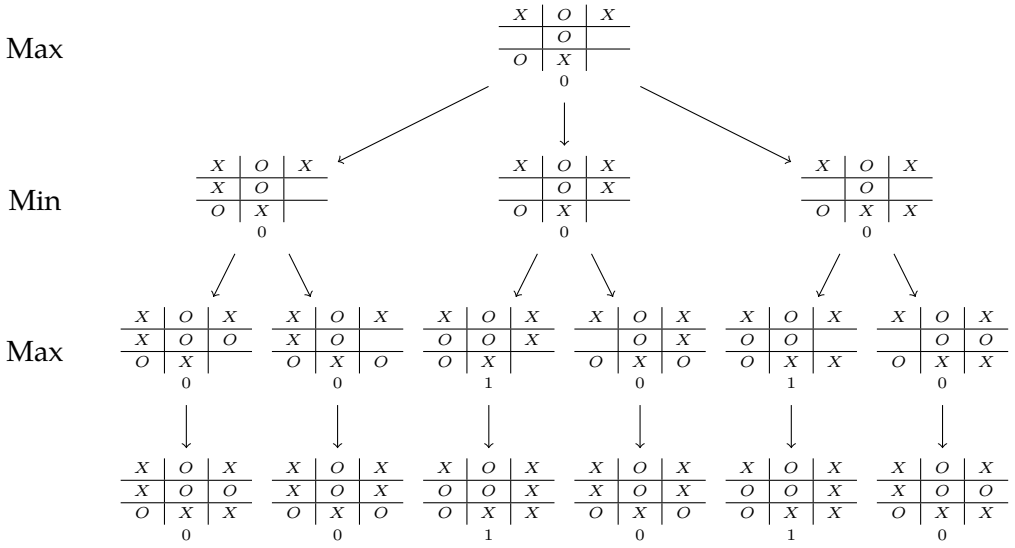
In einer echten Implementierung, sollte man zusätzlich den gegangenen Weg speichern, damit der nächste optimale Zug ebenfalls als Ausgabe zur Verfügung steht.

Im Spielbaum entspricht die MinMax-Suche gerade dem Verfahren:

- Bewerte alle Blätter
- Berechne den Wert der anderen Knoten Stufenweise von unten nach oben, wobei über die Werte der Kinder maximiert bzw. minimiert wird, je nachdem welcher Spieler am Zug ist.

Beachte, dass die Implementierung des MinMax-Algorithmus jedoch als *Tiefensuche* erfolgen kann, und daher die Ebenenweise Bewertung (Minimierung / Maximierung) nach und nach vornimmt.

Wir betrachten erneut das TicTacToe-Beispiel und markieren alle Knoten entsprechend:




D.h. der MinMax-Algorithmus wir 0 als Ergebnis liefern, d.h. der Maximierer wird nicht gewinnen, wenn sich beide Spieler optimal verhalten, da der Minimierer stets eine Möglichkeit hat, das Ergebnis auf ein Unentschieden zu lenken.

Eine Implementierung des MinMax-Verfahrens in Haskell (mit Merken der Pfade) ist:

```
minmax endZustand wert nachfolger spieler zustand  =
  go (zustand,[]) spieler
  where
   go (zustand,p) spieler
     | endZustand zustand = (wert zustand,reverse $ zustand:p)
     | otherwise =
        let l = nachfolger spieler zustand
        in case spieler of
             Maximierer ->  maximumBy
                              (\(a,_) (b,_) -> compare a b)
                              [go (z,zustand:p) Minimierer | z <- l]
             Minimierer ->  minimumBy
                              (\(a,_) (b,_) -> compare a b)
                              [go (z,zustand:p) Maximierer | z <- l]
```

Das *praktische Problem* ist jedoch, dass ein vollständiges Ausprobieren aller Züge und die Berechnung des Minmax-wertes i.A. nicht machbar ist. Selbst bei TicTacToe gibt es am Anfang schon: $9! = 362880$ mögliche Zugfolgen, was auf einem Computer gerade noch machbar ist. Wir werden später im Verlauf des Kapitels noch die Alpha-Beta-Suche kennenlernen, die es ermöglicht etwas besser zu suchen, indem einzelne Teilbäume nicht betrachtet werden müssen. Allerdings kann führt auch diese Methode nicht dazu, dass man bei anspruchsvollen Spielen (wie z.B. Schach etc.) bis zur Endsituation suchen kann.

Daher wendet man eine Abwandlung der MinMax-Suche (bzw. später auch Alpha-Beta-Suche) an: Man sucht nur bis zu einer festen Tiefe $k$, d.h. man schneidet den Spielbaum ab einer bestimmten Tiefe ab. Nun bewertet man die Blattknoten (die i.A. keine Endsituationen sind!) mithilfe einer *heuristische Bewertungsfunktion* (Heuristik). Anschließend verfährt man wie bei der MinMax-Methode und berechnet die Minima bzw. Maxima der Kinder, um den aktuellen Wert des Knotens zu ermitteln. D.h. man minimiert über die Situationen nach Zug des Gegners und maximiert über die eigenen Zugmöglichkeiten. Die Zugmöglichkeit, mit dem optimalen Wert ist dann der berechnete Zug.

Eine solche heuristische Bewertung von Spielsituationen muss algorithmisch berechenbar sein, und deren Güte bestimmt entsprechend auch die Güte des MinMax-Verfahrens.

Beispiele für die Bewertungsfunktion sind:

**Schach:** Materialvorteil, evtl. Stellungsvorteil, Gewinnsituation, ...

**Mühle:** Material, #Mühlen, Bewegungsfreiheit, …

**Tictactoe:** am einfachsten: Gewinn = 1, Verlust = -1, Remis = 0
nur eindeutige Spielsituationen werden direkt bewertet.
Rein theoretisch reicht diese Bewertung bei allen diesen Spielen aus, wenn man bis zum Ende suchen könnte.

Beachte, dass in Spielen ohne Zufall gilt: Die Berechnung des besten Zuges ist unabhängig von den exakten Werten der Bewertungsfunktion; es kommt nur auf die erzeugte *Ordnung* zwischen den Spielsituationen an. Z.B. ist bei TicTacToe egal, ob wir mit Sieg, Niederlage, Unentschieden mit 1, -1, 0 bewerten oder 100, 10, 50. Allerdings ändert sich die Suche, wenn wir die Ordnung ändern und bspw. Sieg, Niederlage, Unentschieden mit 1,-1,1 bewerten (MinMax unterscheidet dann nicht zwischen Sieg und Unentschieden).

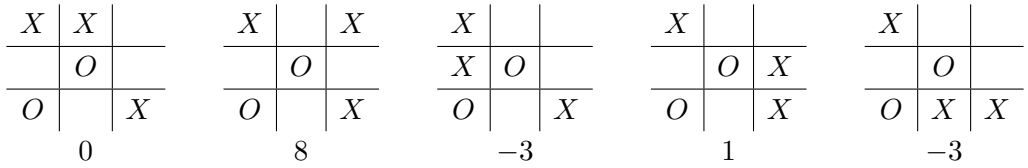
**Beispiel 2.4.2.** *Heuristische Bewertung für Tictactoe. Eine komplizierte Bewertung ist:*

(*#einfach X-besetzte Zeilen/Spalten/Diag*) * *1*
\+ (*# doppelt X-besetzte Zeilen/Spalten/Diag*) * *5*
\+ (*20, falls Gewinnsituation*)
\- (*#einfach O-besetzte Zeilen/Spalten/Diag*) * *1*
\- (*# doppelt O-besetzte Zeilen/Spalten/Diag*) * *5*
\- (*20, falls Verlustsituation*)

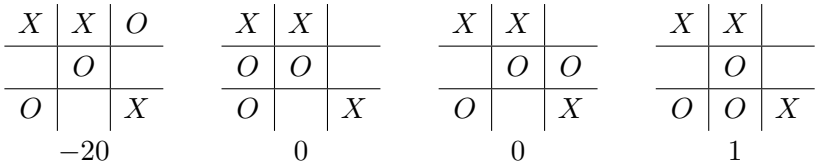
*Die aktuelle Spielsituation sei*

| | | |
|---|---|---|
| X | | |
| | O | |
| O | | X |

*. Die nächsten Spielsituationen zusammen mit den Bewertungen, wenn X-Spieler dran ist, sind:*




*Die Weiterführung der ersten Situation nach möglichen Zügen von O-Spieler ergibt:*




*Über diese ist zu minimieren, so dass sich hier ein Wert von* $-20$ *ergibt.*
*In der ersten Ebene, nach der ersten Zugmöglichkeit von X, ist zu maximieren. Der erste Zug trägt* $-20$ *dazu bei.*

*Auf der Webseite zu dieser Veranstaltung ist ein Haskell-Programm, das zur einfachen Bewertung* $+1, 0, -1$ *eine Berechnung des nächsten Gewinnzuges bzw. falls es keinen solchen gibt, den Zug ausrechnet, der zum Remis führt, oder aufgibt. Dieses Programm findet in annehmbarer Zeit jeweils den besten Zug, und erkennt auch, dass es keinen garantierten Gewinn für den X-Spieler oder O-Spieler am Anfang gibt. Z.B.*

```
*Main> nzug 'X'  "XBBBBOBBB"
"Siegzug Feld: 3"
*Main> nzug 'X'  "XBBBBBOBB"
"Siegzug Feld: 2"
*Main> nzug 'X'  "XBBBBBBOB"
"Siegzug Feld: 3"
*Main> nzug 'X'  "XBBBBBBBO"
```

Man könnte die Bewertung verbessern, wenn man beachtet, wer am Zug ist, allerdings ergibt sich der gleiche Effekt, wenn man einen Zug weiter sucht und bewertet.

Praktisch erwünschte Eigenschaften der Bewertungsfunktion sind:

- hoher Wert $\equiv$ hohe Wahrscheinlichkeit zu gewinnen
- schnell zu berechnen

Dies verdeckt manchmal den Weg zum Sieg. z.B. im Schach: Ein Damenopfer, das zum Gewinn führt, würde zwischenzeitlich die Bewertung bzgl. Material verringern.

Beachte, dass der folgende Einwand nicht ganz aus der Luft gegriffen ist: Wenn man ein sehr gute Bewertungsfunktion hat: Warum sollte man überhaupt mit MinMax in die Tiefe schauen, und nicht direkt die Nachfolger der Wurzel bewerten, und den am besten

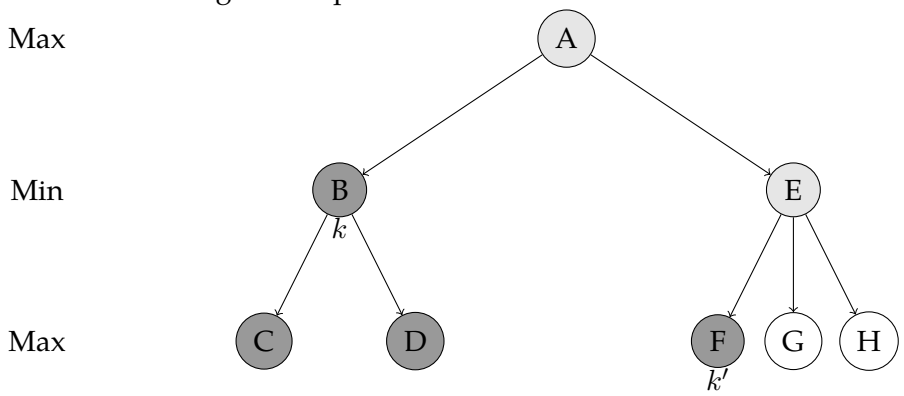
bewerteten Zug übernehmen? Das Gegenargument ist, dass i.A. die Bewertung besser wird, je weiter man sich dem Ziel nähert. Betrachtet man z.B. Schach, dann kann man aus der Bewertung des Anfangszustand und dessen direkte Nachfolger vermutlich nicht viel schließen.

Eine Tiefenbeschränkte Min-Max-Suche braucht $O(c^m)$ Zeit, wenn $c$ die mittlere Verzweigungrate, $m$ die Tiefenschranke ist und die Bewertungsfunktion konstante Laufzeit hat

### 2.4.1 Alpha-Beta Suche

Betrachtet man den Ablauf der MinMax-Suche bei Verwendung der (i.A. tiefenbeschränkten) Tiefensuche, so stellt man fest, dass man manche Teilbäume eigentlich nicht betrachten muss.

Betrachte den folgenden Spielbaum




Dann ergibt sich der Wert für die Wurzel aus

$$\begin{aligned} & \text{MinMax}(A, Max) \\ = \; & \texttt{max}\{\text{MinMax}(B, Min), \text{MinMax}(E, Min)\} \\ = \; & \texttt{max}\{\texttt{min}\{\text{MinMax}(C, Max), \text{MinMax}(D, Max)\}, \\ & \quad \texttt{min}\{\text{MinMax}(F, Max), \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}\} \end{aligned}$$

Nehmen wir an, die Tiefensuche, hat bereits die Werte für die Knoten $B$ und $F$ als $k$ und $k'$ berechnet (die Schattierungen der Knoten sollen dies gerade verdeutlichen: ganz dunkle Knoten sind fertig bearbeitet durch die Tiefensuche, hellgraue Knoten noch in Bearbeitung, und weiße Knoten noch unbesucht).

Dann ergibt sich:

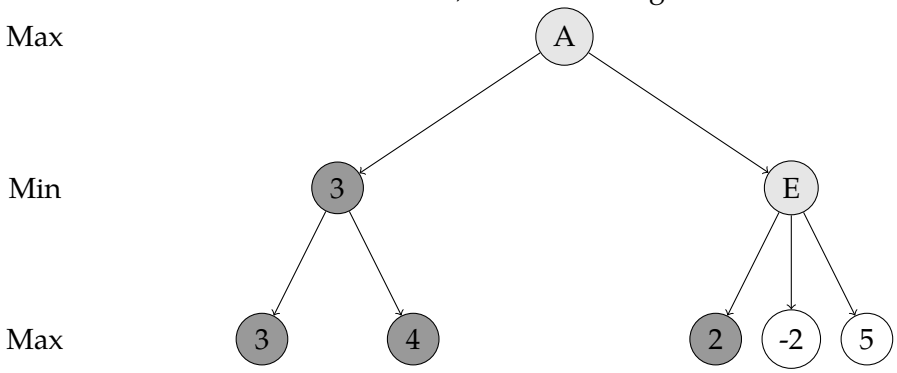
$$\text{MinMax}(A, Max) = \texttt{max}\{k, min\{k', \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}\}$$

Wenn $k' \leq k$, dann „weiß“ der Minimierer, dass die komplette Berechnung von $\texttt{min}\{k', \text{MinMax}(G, Max), \text{MinMax}(H, Max)\}$ (also dem Wert des Knotens $E$), dem Maximierer bei der Berechnung des Wertes für $A$ irrelevant sein wird, da der Maximierer

von $E$ sicher einen Wert kleiner gleich $k'$ (und kleiner gleich $k$) geliefert bekommt, und daher der Wert $k$ für die Knoten $B$ und $E$ der bessere (maximale) Wert ist.

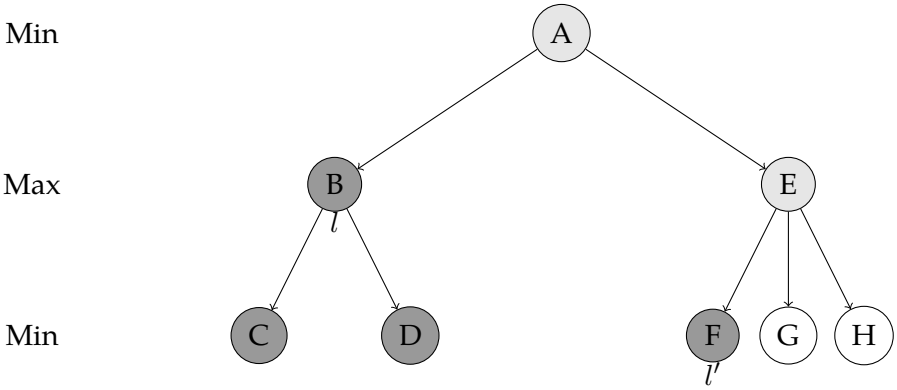
Aus diesem Grund ist es nicht mehr nötig, die Werte von $G$ und $H$ zu berechnen. D.h. diese Äste ($G$ und $H$ können größe Unterbäume besitzen!) braucht nicht betrachtet zu werden.

Um es nochmals zu verdeutlichen, betrachte den gleichen Baum mit konkreten Werten:




Obwohl das Minimum für die Berechnung des Wertes von $E$ eigentlich $-2$ ergibt, kann die Suche beim Entdecken des Werts 2 abgebochen werden, da die Bewertung für $A$ (Maximieren), in jedem Fall die 3 aus dem linken Teilbaum bevorzugen wird (da $3 > 2$).

Schließlich gibt es den symmetrischen Fall, wenn an einem Knoten minimiert statt maximiert wird. Betrachte den Baum:




Der Baum unterscheidet sich vom vorherigen lediglich in der Minimierungs- und Maximierungsreihenfolge. Die Berechnung des Werts von $A$ ist:

$$
\begin{aligned}
& \text{MinMax}(A, Min) \\
= \; & \texttt{min}\{\text{MinMax}(B, Max), \text{MinMax}(E, Max)\} \\
= \; & \texttt{min}\{\text{MinMax}(B, Max), \\
& \quad \texttt{max}\{\text{MinMax}(F, Min), \text{MinMax}(G, Min), \text{MinMax}(H, Min)\}\}
\end{aligned}
$$

Wenn $\text{MinMax}(B, Max) = l$ und $\text{MinMax}(F, Min) = l'$ schon berechnet sind, ergibt das $\texttt{min}\{l, \texttt{max}\{\text{l'}, \text{MinMax}(G, Min), \text{MinMax}(H, Min)\}\}$.

Wenn $l' \geq l$, dann brauchen die Werte von $G$ und $H$ nicht berechnet werden, da der

Minimierer an $A$ sowieso niedrigeren Wert $k$ wählen wird.

Die Alpha-Beta-Suche verwendet genau diese Ideen und führt bei der Suche zwei Schranken mit, die das Suchfenster festlegen (und den Werten $k$ und $l$ in den obigen Beispielen entsprechen). Als untere Begrenzung des Suchfensters wird der Wert der Variablen $\alpha$ verwendet, als obere Begrenzung der aktuelle Wert der Variablen $\beta$. Das ergibt das Fenster $[\alpha, \beta]$. Der Algorithmus startet mit $\alpha = -\infty$ und $\beta = \infty$ und passt während der Suche die Wert von $\alpha$ und $\beta$ an. Sobald an einem Knoten $\alpha \geq \beta$ gilt, wird der entsprechende Teilbaum nicht mehr untersucht.

Beachte, dass die Alpha-Beta-Suche eine Tiefensuche mit Tiefenschranke durchführt (um die Terminierung sicherzustellen).

*Algorithmus* $\alpha - \beta$**-Suche**

**Eingaben:** Nachfolgerfunktion, Bewertungsfunktion, Tiefenschranke, Anfangsspielsituation und Spieler

**Aufruf:** AlphaBeta(Zustand,Spieler,$-\infty$,$+\infty$)

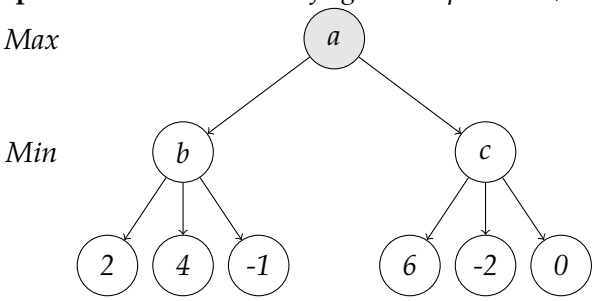
**Funktion**: AlphaBeta(Zustand,Spieler,$\alpha$,$\beta$)

1. Wenn Tiefenschranke erreicht, bewerte die Situation, `return` Wert
2. Sei $NF$ die Folge der Nachfolger von (Zustand,Spieler)
3. Wenn Spieler = Minimierer:
   $\beta_l := \infty$;
   `for-each` $L \in NF$
   $\beta_l :=$ `min`$\{\beta_l,$ AlphaBeta($L$,Maximierer,$\alpha$,$\beta$)$\}$
   `if` $\beta_l \leq \alpha$ `then return` $\beta_l$ `endif`// verlasse Schleife
   $\beta :=$ `min`$\{\beta, \beta_l\}$
   `end-for`
   `return` $\beta_l$
4. Wenn Spieler = Maximierer
   $\alpha_l := -\infty$;
   `for-each` $L \in NF$
   $\alpha_l :=$ `max`$\{\alpha_l,$ AlphaBeta($L$,Minimierer,$\alpha$,$\beta$)$\}$
   `if` $\alpha_l \geq \beta$ `then return` $\alpha_l$ `endif`// verlasse Schleife
   $\alpha :=$ `max`$\{\alpha, \alpha_l\}$
   `end-for`
   `return` $\alpha_l$

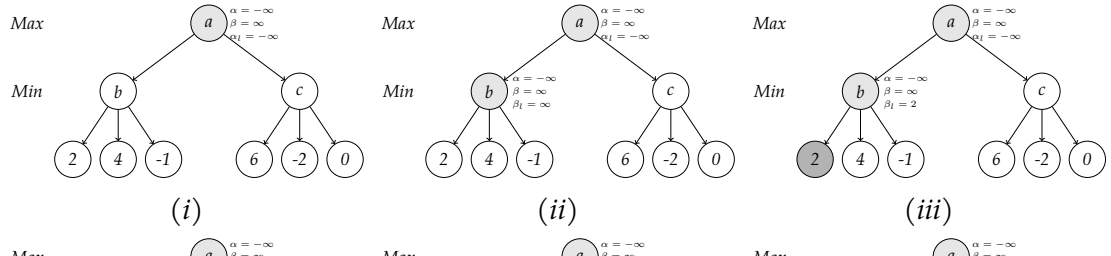
Beachte, dass der Algorithmus lokale $\alpha_l$ und $\beta_l$ Werte verwendet, und dass die rekursiven Aufrufe auch entsprechend die aktuelle Tiefe anpassen müssen (was der Einfachheithalber in der Beschreibung weggelassen wurde).

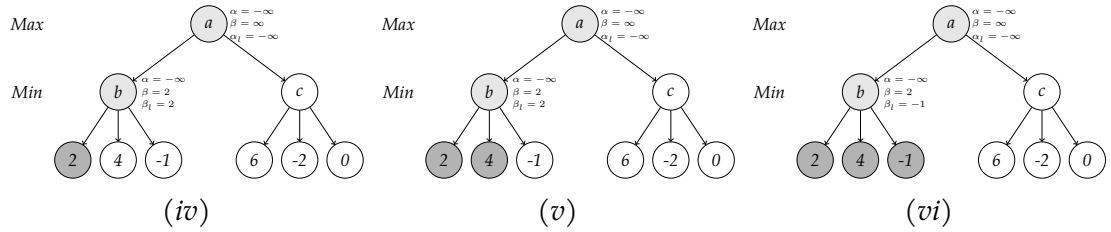
Wir betrachten ein Beispiel:

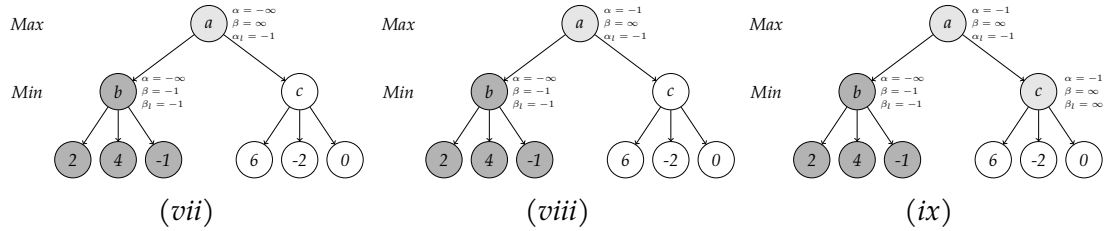
**Beispiel 2.4.3.** *Betrachte den folgenden Spielbaum, wobei der Maximierer an der Wurzel am Zug ist*

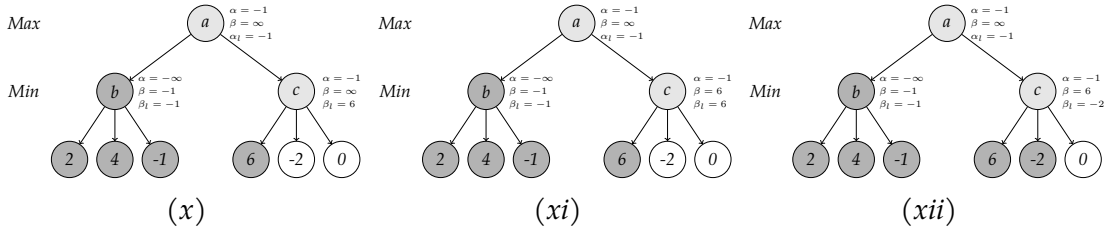


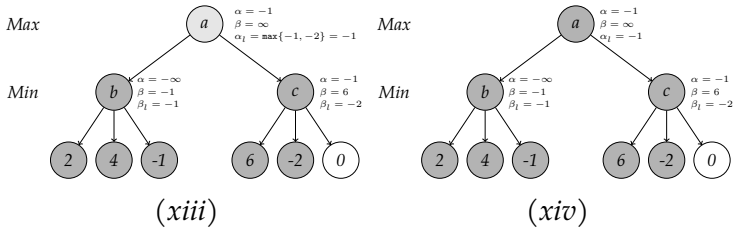

*Die folgenden Abbildungen zeigen die Schrittweise Abarbeitung durch die Alpha-Beta-Suche:*




*Am Knoten c kann die Suche abgebrochen worden, nachdem die $-2$ als Kindwert entdeckt wurde. Daher kann braucht der mit 0 markierte Knoten (der auch ein großer Baum sein könnte) nicht*

*untersucht werden. Der zugehörige Schritt ist* (*xii*)*, da dort* $\alpha \geq \beta_l$ ($\beta_l$ *ist der lokal beste gefundene* $\beta$*-Wert*) *festgestellt wird. Anschließend wird der* $\beta_l$*-Wert* (-2) *direkt nach oben gegeben, und zur Aktualisierung des* $\alpha_l$*-Wertes von Knoten* (*a*) *verwendet* (*Schritt* (*xiii*))*. Da dieser schlechter ist, als der Bereits erhaltene liefert die AlphaBeta-Suche den Wert* $-1$ *insgesamt als Ergebnis zurück.*

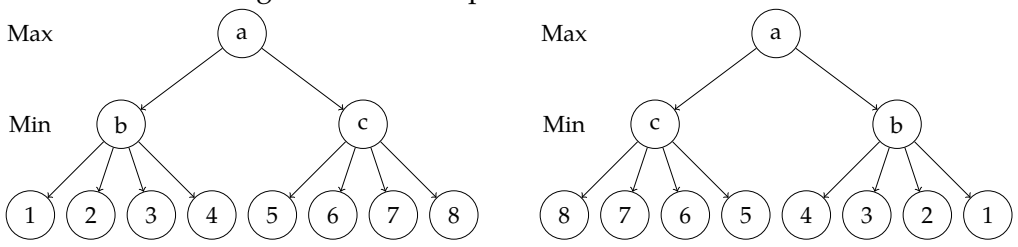
Die Alpha-Beta-Prozedur kann bei geeignetem Suchfenster auf jeder Ebene die Suche abbrechen, es muss nicht jeder innere Konten untersucht werden!.

Eine Implementierung der Alpha-Beta-Suche in Haskell (ohne Merken des Pfades) ist:

```
alphaBeta wert nachfolger maxtiefe neginfty infty spieler zustand  =
  ab 0 neginfty infty spieler zustand
 where
  ab tiefe alpha beta spieler zustand
   | null (nachfolger spieler zustand) || maxtiefe <= tiefe = wert zustand
   | otherwise =
      let l = nachfolger spieler zustand
      in
       case spieler of
         Maximierer -> maximize tiefe alpha beta l
         Minimierer -> minimize tiefe alpha beta l
  maximize tiefe alpha beta xs = it_maximize alpha neginfty xs
   where
      it_maximize alpha alpha_l []     = alpha_l
      it_maximize alpha alpha_l (x:xs) =
       let alpha_l' = max alpha_l (ab (tiefe+1) alpha beta Minimierer x)
       in  if alpha_l' >= beta then alpha_l' else
             it_maximize (max alpha alpha_l') alpha_l' xs
  minimize tiefe alpha beta xs = it_minimize beta infty xs
   where
      it_minimize beta beta_l []     = beta_l
      it_minimize beta beta_l (x:xs) =
        let beta_l' = min beta_l (ab (tiefe+1) alpha beta Maximierer x)
        in if beta_l' <= alpha then beta_l' else
             it_minimize (min beta beta_l') beta_l' xs
```

Beachte, dass die Güte der Alpha-Beta-Suche (gegenüber dem Min-Max-Verfahren) davon abhängt in welcher Reihenfolge die Nachfolgerknoten generiert und daher abgesucht werden.

Betrachte z.B. die folgenden beiden Spielbäume:



Beide Bäume stellen das selbe Spiel dar, lediglich die Reihenfolge der Nachfolgergenerierung ist verschieden. Im linken Baum, kann die Alpha-Beta-Suche keine Pfade abschneiden: Der für $b$ ermittelte Wert 1 wird von allen Kindern von $c$ übertroffen, daher muss die Suche sämtliche Kinder von $c$ explorieren (stets in der Hoffnung einen kleineren Wert zu finden). Im rechten Baum hingegen wird für $c$ der Wert 5 ermittelt, da das erste Kind von $b$ bereits den niedrigeren Wert $4$ hat, wird die Alpha-Beta-Suche die restlichen Kinder von $b$ nicht mehr explorieren.

Abhilfe kann hierbei wieder eine Heuristik schaffen, die entscheidet welche Kindknoten zuerst untersucht werden sollen (d.h. die Nachfolger werden anhand der Heuristik vorsortiert). Z.B. kann man bei Schach die verschiedenen Stellungen berücksichtigen und bspw. schlagende Züge reinem Ziehen vorziehen usw.

Schließlich stellt sich die Frage, wie groß die Wirkung der Alpha-Beta-Suche als Verbesserung der Min-Max-Suche ist. Wie bereits obiges Beispiel zur Reihenfolge der Nachfolger zeigt, ist im worst case die Alpha-Beta-Suche gar nicht besser als die Min-Max-Suche und muss alles inspizieren und benötigt daher $O(c^d)$ Zeit, wenn $c$ die mittlere Verzweigungrate, $d$ die Tiefenschranke ist und die Bewertungsfunktion konstante Laufzeit hat.

Für den best case kann man nachweisen, dass Alpha-Beta nur $O(c^{\frac{d}{2}})$ Knoten explorieren muss. Diese Schranke gilt auch im Mittel, wenn man eine gute Heuristik verwendet. Umgekehr bedeutet das, dass man mit Alpha-Beta-Suche bei guter Heuristik bei gleichen Ressourcen (Zeit und Platz) *doppelt so tief* suchen kann, wie bei Verwendung der Min-Max-Suche. Damit wird auch die Spielstärke eines entsprechenden Programms gesteigert, denn tiefere Suche führt zur Verbesserung der Spielstärke.

Verwendet man zufälliges Raten beim Explorieren der Nachfolger, so kann man als Schranke für die Laufzeit $O(c^{\frac{3}{4}d})$ herleiten.

Wichtige Optimierungen der Suche, die z.B. beim Schach vorgenommen werden, sind die Speicherung bereits bewerteter Stellungen, um bei Zugvertauschungen nicht zuviele Stellungen mehrfach zu bewerten. Damit kann man die Vorsortierung von Zügen unterstützen.

Auftretende Probleme bei der Suche sind:

- Tiefere Suche kann in seltenen Fällen zur schlechteren Zugauswahl führen. Dies tritt mit immer geringerer Wahrscheinlichkeit ein, falls man eine gute Bewertungsfunktion hat

- Horizont-Effekt: (z.B. durch Schach bieten gerät ein wichtiger Zug außerhalb des Suchhorizonts)

Um den Horizont-Effekt in der Hauptvariante auszuschalten, kann man den Horizont unter gewissen Bedingungen erweitern.

Idee: falls bei Suche mit Tiefe $d$ ein einzelner Zug $k_0$ besonders auffällt (d.h. gut beim Maximierer und schlecht beim Minimierer):

Wenn $v_d(k_0) > v_d(k) + S$ für alle Brüder $k \neq k_0$ und für feste, vorgegebene Schranke $S$, wobei $v_d$ die Bewertung bei Suchtiefe $d$ bezeichnet; Dann wird dieser Zug eine Tiefe weiter untersucht. Die tiefere Suche ist auch sinnvoll bei Zugzwang, d.h. wenn nur ein weiterer Zug möglich ist.

Im Falle des Zugzwangs vergrößert das den Suchraum nicht, aber erhöht die Vorausschau

Die Implementierung dieser Idee ist eine wichtige Erweiterung der Suchmethoden in Programmen zu strategischen Spielen.
Folgendes Vorgehen ist üblich: es gibt mehrere Durchgänge:

1. normale Bewertung

2. Erkennen aller besonderen Züge:

3. Alle auffallenden Züge dürfen um 1 weiter expandieren.

**Effekte:**

- Die Suche kann instabil werden:
  ob ein Zug auf Ebene $n$ als interessant auffällt, kann davon abhängen, wie tief die Suche unterhalb dieses Knotens im Baum ist.

- Der Suchraum kann beliebig wachsen, falls man mehrere Durchgänge macht: jeweils ein anderer Knoten wird als wichtig erkannt.

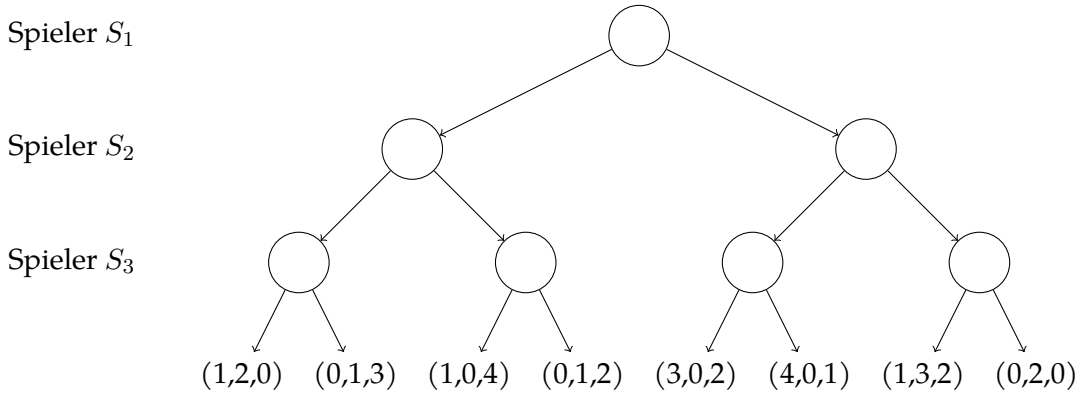
- Durch die tiefere Suche kann ein guter Zug wieder schlechter bewertet werden.

### 2.4.2 Mehr als 2 Spieler

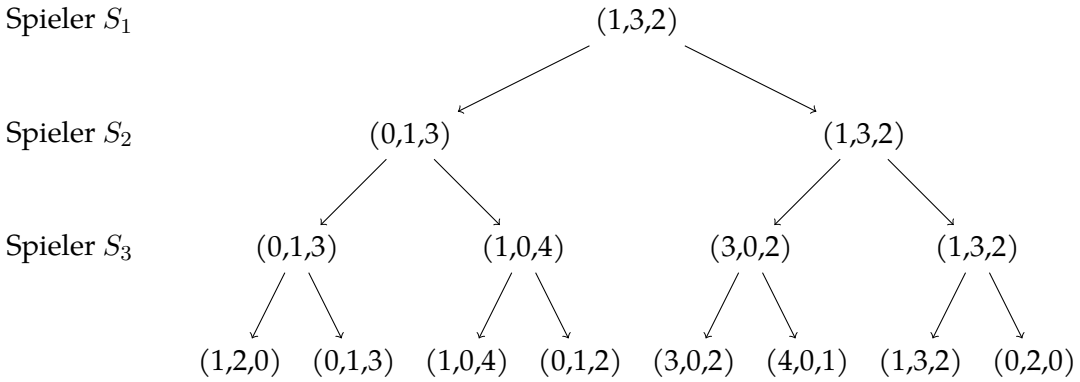
In diesem kurzen Abschnitt werden wir erläutern, warum die Min-Max-Methode (analog die Alpha-Beta-Suche) bei Spielen mit mehr als 2 Spielern nicht mehr funktioniert. Wir geben keine Lösungen für diese Spiele an, da die Verfahren komplizierte und aufwändig sind und in der Literatur nachgelesen werden können.

Wir betrachten ausschließlich Spiele mit drei Spielern (sagen wir $S_1$, $S_2$ und $S_3$), wobei alle Spieler nacheinander ziehen. Zur Blattbewertung macht es in diesem Fall keinen Sinn einen einzigen Wert zu verwenden, daher benutzen wir ein Drei-Tupel (Ergebnis für $S_1$, Ergebnis für $S_2$, Ergebnis für $S_3$). Jede Komponente stellt das Ergebnis aus Sicht des Spieler dar. Zunächst scheint das Min-Max-Verfahren einfach übertragbar: Jenachdem welcher Spiel an einem Knoten am Zug ist, wird entsprechend seiner Komponente maximiert.

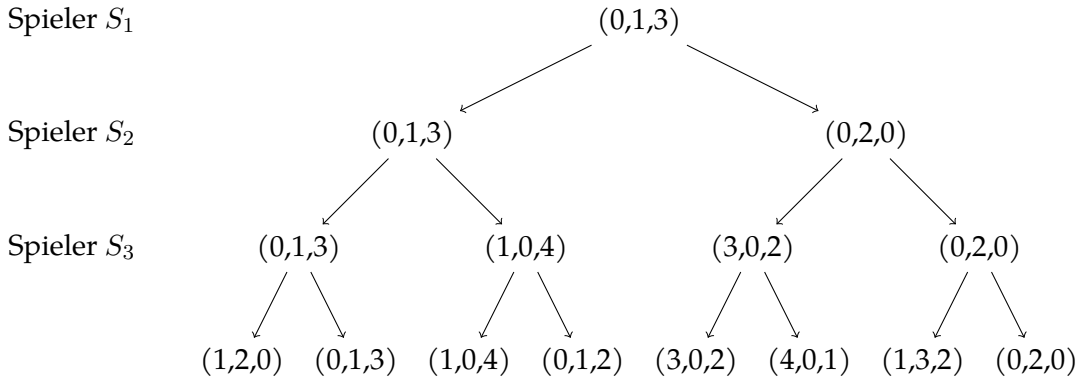
Betrachte z.B. den Spielbaum:




Bewertet man den Baum von unten nach oben, indem jeder Spieler sein Ergebnis maximiert, so erhält man:

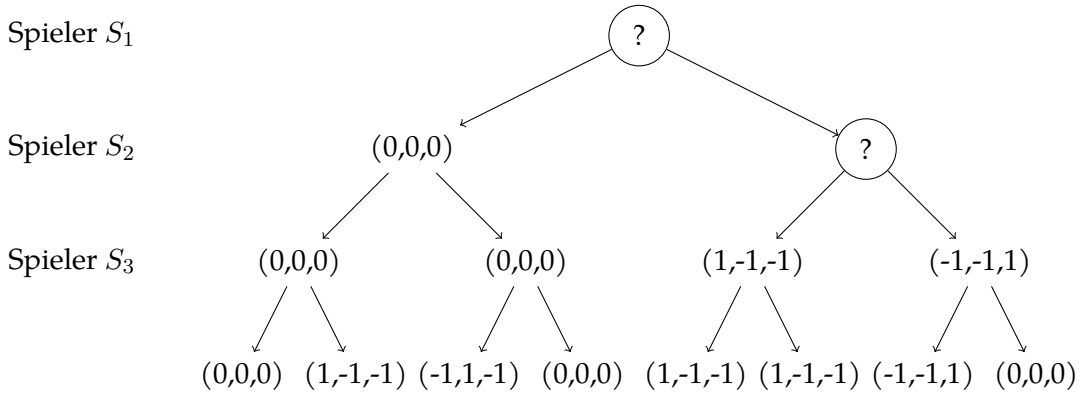



Hierbei sind wir jedoch davon ausgegangen, dass jeder Spieler nur nach der eigenen Gewinnmaximierung vorgeht. Nehmen wir an, Spieler 1 und Spieler 3 *verbünden sich*, mit dem Ziel, das Ergebnis von Spieler 2 zu minimieren, so ergibt sich eine andere Bewertung:




Noch gravierender ist der Fall, dass bei keiner Kooperation ein Spieler quasi zufällig einen Zug auswählt, da mehrere Nachfolger den gleichen Wert für ihn haben, aber diese Wahl die Bewertung der anderen Spieler beeinflussen kann.

Betrachte z.B. den Baum:




Spieler 2 hat zunächst keine Präferenz, wie er den zweiten Kindknoten bewerten soll, da die Bewertungen (1,-1,-1) und (-1,-1,1) aus seiner Sicht beide gleich schlecht sind. Allerdings nimmt er (1,-1,-1), so würde Spieler 1 an der Wurzel gewinnen, nimmt er (-1,-1,1) ist dies rational besser, da Spieler 1 an der Wurzel (0,0,0) also unentschieden nimmt.

Allgemein scheint es jedoch keine einfache Strategie zu geben, die stets zu einer rationalen Entscheidung führt.

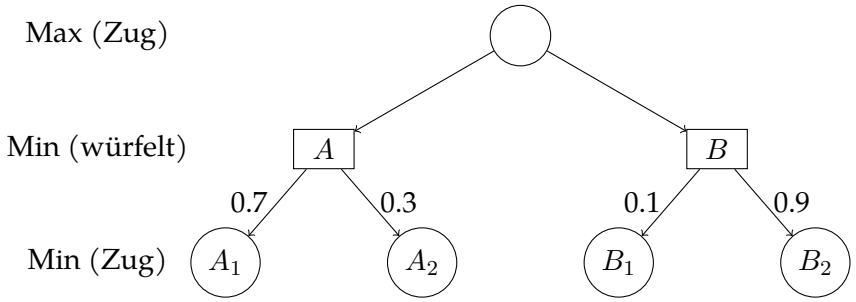
### 2.4.3 Spiele mit Zufallsereignissen

In diesem Abschnitt betrachten wir wieder Zwei-Spieler-Spiele, jedoch Spiele, bei denen der Zufall eine Rolle spielt. Auch in dieser Variante kann die Minmax-Suche mit leichten Anpassungen verwendet werden.

Z.B. beim Backgammon besteht ein Spielzug darin, zunächst mit zwei Würfeln zu würfeln und dann zu ziehen, wobei man bei der Zugwahl selbst entscheiden kann, welche Steine zu ziehen sind. Da der Gegner auch würfelt und erst dann eine Entscheidungsmöglichkeit für seinen Zug hat, das gleiche aber auch für weitere eigene Züge gilt, hat man keinen festen Baum der Zugmöglichkeiten. Man könnte einen Zug des Gegners vorausschauend über alle seine Würfel- und Zugmöglichkeiten minimieren, aber sinnvoller erscheint es, über die bekannte Wahrscheinlichkeit zu argumentieren und den Erwartungswert des Gegners zu minimieren und den eigenen Erwartungswert zu maximieren.

**Theorem 2.4.4.** *Es gilt: Wenn Wahrscheinlichkeit eine Rolle spielt, dann ist die Berechnung des Zuges mit dem besten Erwartungswert abhängig von den exakten Werten der Bewertungsfunktion; nicht nur von der relativen Ordnung auf den Situationen.*

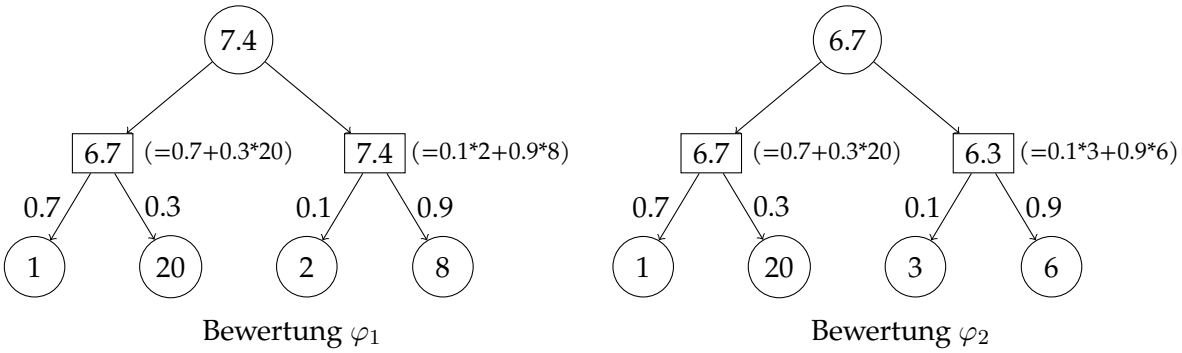
*Beweis.* Wir betrachten als Gegenbeispiel, den folgenden Spielbaum, wobei der Maximierer zunächst am Zug ist, und zwischen Zuständen $A$ und $B$ wählen kann. Im Anschluss beginnt der Zug des Minimierer, wobei dieser zunächst „würfelt“ um für Situation $A$ zwischen $A_1$ (mit Wahrscheinlichkeit $0,7$) und $A_2$ (mit Wahrscheinlichkeit $0,3$) und für Situation $B$ zwischen $B_1$ (mit Wahrscheinlichkeit $0,1$) und $B_2$ (mit Wahrscheinlichkeit $0,9$) zu entscheiden.




Wir betrachten zwei Bewertungen $\varphi_1$ und $\varphi_2$:

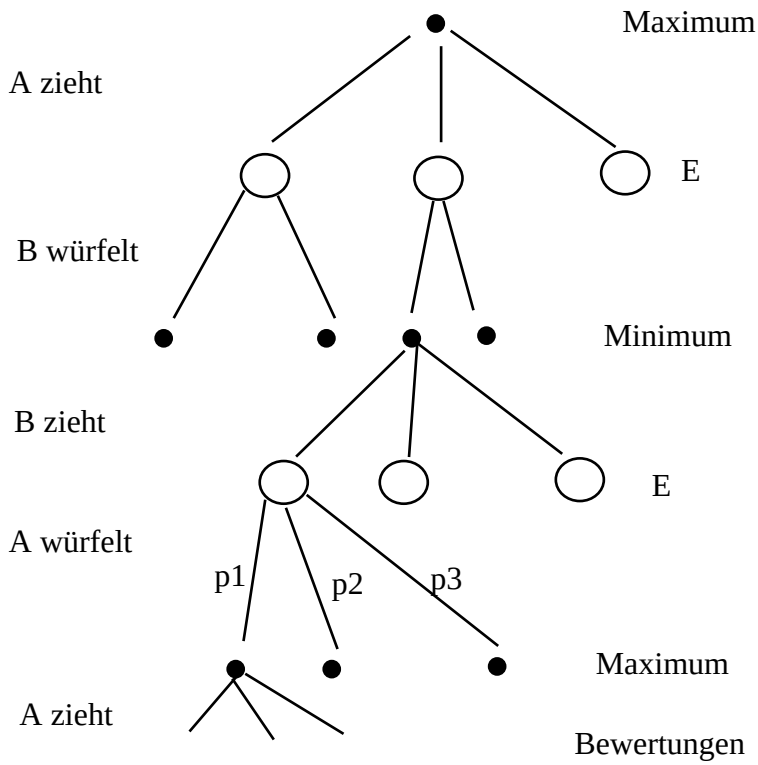
| Bewertung | $A_1$ | $A_2$ | $B_1$ | $B_2$ |
|---|---|---|---|---|
| $\varphi_1$ | 1 | 20 | 2 | 8 |
| $\varphi_2$ | 1 | 20 | 3 | 6 |

Beachte, dass die relative Ordnung Bewertung der Situationen bei $\varphi_1$ und $\varphi_2$ die gleiche ist ($\varphi_i(A_1) < \varphi_i(B_1) < \varphi_i(B_2) < \varphi_i(A_2)$ für $i = 1, 2$), aber die Erwartungswerte verschieben sich:




Damit ist einmal der rechte, einmal der linke Erwartungswert größer. D.h. einmal ist $B$, einmal $A$ zu bevorzugen. □

D.h., die Bewertungsfunktion muss auch von den Werten her vorsichtig definiert werden. Beim Backgammon wäre die richtige Intuition: die Anzahl der benötigten Züge bis alle Steine das Ziel erreicht haben. Diese ist aber vermutlich schwer zu schätzen, da der Spielbaum viel zu groß ist.




Bei Vorausschau mit Tiefe 2 wird der Baum der Möglichkeiten aufgebaut. Die Blätter werden bewertet. Im Bild sind es die Situationen nach einem Zug von $A$. Man maximiert über die Geschwisterknoten und erhält den besten Zug. Im nächsten Schritt nach oben berechnet man den Erwartungswert für den Wert des besten Zuges. Geht man weiter nach oben, ist das gleiche zu tun für gegnerische Züge und Würfeln, wobei man minimieren muss. An der Wurzel ist das Würfelergebnis bekannt, so dass man als besten Zug denjenigen mit dem höchsten Erwartungswert ermittelt.

Hat man eine gute Bewertungsfunktion, dann kann man die Minmax-Suche verbessern durch eine Variante der Alpha-Beta-Suche. Diese Optimierung spielt allerdings in der Praxis keine so große Rolle wie in deterministischen Spielen, da der Spielbaum i.a. viel zu groß ist, um mehrere Züge voraus zu schauen.

## 2.5 Evolutionäre (Genetische) Algorithmen

Das Ziel bzw. die Aufgabe von evolutionären Algorithmen ist eine Optimierung von Objekten mit komplexer Beschreibung, wobei es variable Parameter gibt. Dabei werden die Objekte (Zustände) als Bitstrings kodiert, so dass die Aufgabe die Optimierung einer reellwertigen Bewertungsfunktion auf Bitfolgen fester Länge ist. Dabei wird stets eine Multimenge solcher Objekte betrachtet, d.h. es wird eine parallele Suche durchgeführt.

Genetische bzw. evolutionäre Algorithmen orientieren sich dabei an der Evolution von Lebewesen, daher werden entsprechende andere Sprechweisen benutzt, die wir gleich einführen:

Die Eingaben eines evolutionären Algorithmus sind:

- Eine *Anfangspopulation*, d.h. eine (Multi-)Menge von Individuen (Zuständen, Objekten), die üblicherweise als Bitstring dargestellt sind. Ein Bit oder eine Teilfolge von Bits entspricht dabei einem Gen.
- Eine Bewertungsfunktion, welche Fitnessfunktion genannt wird.

Gesucht (als Ausgabe) ist ein *optimaler Zustand*

Die Anzahl der möglichen Zustände ist im Allgemeinen astronomisch, so dass eine Durchmusterung aller Zustände aussichtslos ist. Da die Evolution Lebewesen „optimiert", versucht man die Methoden der Evolution und zwar *Mutation*, *Crossover* (Rekombination) und *Selektion* analog auf diese Optimierungen zu übertragen, und mittels Simulationsläufen ein Optimum zu finden.

Die Idee ist daher

- Zustände (Individuen) werden als Bitfolgen (Chromosomen) kodiert.
- Die Bewertung entspricht der Fitness der Bitfolge (dieses Chromosomensatzes bzw. Individuums),
- Höhere Fitness bedeutet mehr Nachkommen
- Man beobachtet die Entwicklung einer Menge von Bitfolgen (*Population*). D.h. aus einer aktuellen Population wird die nächste Generation erzeugt.
- Nachkommen werden durch zufällige Prozesse unter Verwendung von Operationen analog zu Mutation und Crossover erzeugt

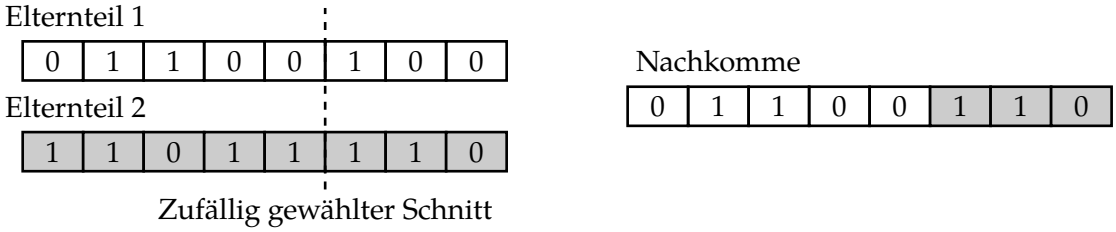
### 2.5.1 Genetische Operatoren

Um von einer Population der Generation $n$ zur Population $n + 1$ zu kommen, werden verschiedene Operatoren verwendet, die auf der Basis der Population zu $n$ die Population $n+1$ ausrechnen. Wir beschreiben diese Operationen allgemein. Für spezifische Probleme können diese Operationen angepasst (oder auch verboten) sein.

**Wahl der Anfangspopulation:** Möglichst breit und gut verteilt.

**Selektion:** Unter *Selektion* versteht man die Auswahl eines Individuums (Chromoson) aus der Population zur Generation $n$ Diese Auswahl kann zum Zwecke der Fortpflanzung oder der Mutation, des Weiterlebens usw. passieren. Diese Auswahl erfolgt für ein Individuum mit hoher Fitness mit höherer Wahrscheinlichkeit, wobei die Wahrscheinlichkeit auch vom Zweck der Selektion abhängen kann.

**Rekombination** Zwei Chromosomen werden ausgewählt mittels Selektion. Danach wird innerhalb der Chromosomen (zufällig) eine Stelle zum Zerschneiden ermittelt, ab dieser werden die Gene ausgetauscht. Es entstehen zwei neue Chromosomensätze. Alternativ kann man auch mehrere Abschnitte der Chromosomen austauschen. Die folgende Darstellung illustriert die Rekombination:




**Mutation.** mit einer gewissen Wahrscheinlichkeit werden zufällig ausgewählte Bits in den Chromosomen verändert. ($0 \to 1$ bzw. $1 \to 0$)

**Aussterben** Wenn die Populationsgröße überschritten wird, stirbt das schlechteste Individuum. Alternativ ein zufällig gewähltes, wobei das schlechteste mit größerer Wahrscheinlichkeit als des beste ausstirbt.

**Ende der Evolution** Wenn vermutlich das Optimum erreicht ist.

Ein genetischer / evolutionärer Algorithmus beginnt daher mit einer initialen Population (1. Generation) und erzeugt durch die genetischen Operationen Nachfolgegenerationen. Er stoppt, wenn ein optimales Individuum gefunden wurde, oder nach einer Zeitschranke. Im letzteren Fall wird nicht notwendigerweise ein Optimum gefunden, aber häufig (z.B. bei Verteilungsproblemen) ein ausreichend gutes Ergebnis.

Ein einfaches Grundgerüst für einen genetischen Algorithmus ist:

*Algorithmus* **Genetische Suche**

**Eingabe:** Anfangspopulation, Fitnessfunktion $\phi$, $K$ die Populationsgröße

**Datenstrukturen:** $S, S'$ : Mengen von Individuen
**Algorithmus**:
$S :=$ Anfangspopulation;
`while` ($S$ enthält kein Individuum mit maximalem $\phi$) `do`:
 $S' := \emptyset$
 `for` $i := 1$ `to` $K$ `do`:
  Wähle zufällig (mit $\phi$ gewichtet) zwei Individuen $A$ und $B$ aus $S$;
  Erzeuge Nachkommen $C$ aus $A$ und $B$ durch Rekombination;
  $C' :=$ Mutation (geringe Wahrscheinlichkeit) von $C$;
  $S' := S' \cup \{C'\}$;
 `end for`
 $S := S'$
`end while`
Gebe Individuum mit maximaler Fitness aus

Allerdings sind weitere genetische Operatione und Anpassungen von Mutation, Rekombination und Selektion möglich. Außerdem ist es evtl. ratsam besonders gute Individuen ohne Rekombination in die nächste Generation zu übernehmen (weiterleben bzw. Klonen).

Im Allgemeinen sind die adjustierende *Parameter* vielfältig:

- Erzeugung der initialen Population und Kriterien zum Beenden
- Mutationswahrscheinlichkeit: wieviel Bits werden geflippt?, welche Bits?
- Crossover-Wahrscheinlichkeit: an welcher Stelle wird zerschnitten? Welche Individuen dürfen Nachkommen zeugen?
- Abhängigkeit der Anzahl der Nachkommen von der Fitness
- Größe der Population

Wie bereits in den vorherigen Abschnitten zur Suche sind auch folgende Aspekte hier wichtig:

- Finden einer geeigneten Repräsentation der Zustände
- Finden geeigneter genetischer Operatoren (auf der Repräsentation).

Problem: auch bei genetischen Algorithmen kann sich die Population um ein lokales Maximum versammeln. Z.B. durch Inzucht oder durch einen lokalen Bergsteigereffekt, bei dem die Mutation das lokale Optimum nicht verlassen kann.

Die Codierung der Chromosomen ist im allgemeinen eine Bitfolge, in diesem Fall spricht man von *genetischen Algorithmen.*
Die Kodierung kann aber auch dem Problem angepasst sein, so dass nicht mehr alle Kodierungen einem Problem entsprechen (ungültige Individuen). In dem Fall ist es meist auch sinnvoll, Operatoren zu verwenden, die nur gültige Nachkommen erzeugen. In diesem Fall spricht man von *evolutionären Algorithmen*

### 2.5.2 Ermitteln der Selektionswahrscheinlichkeit

Alternativen:

1. Proportional zur Fitness.
   Dies macht die Optimierung sehr sensibel für den genauen Wert und der Steigung der Fitness. Flacht die Fitness ab – z.B. in der Nähe zum Optimum oder auf einem Plateau– dann gibt es keine bevorzugte Auswahl mehr.

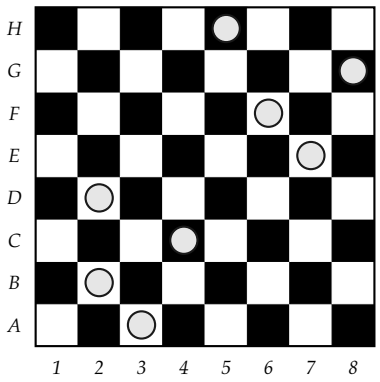
2. Proportional zur Reihenfolge innerhalb der Population.
   z.B bestes Individuum 2-3 fach wahrscheinlicher als schlechtestes Individuum. In diesem Fall ist man flexibler in der Auswahl der Fitness-Funktion. Die Auswahl hängt nicht vom genauen Wert ab.

3. Proportional zu einer normierten Fitness (z.B. Fitness $-c$), wobei man die Fitness $c$ als das Minimum wählen kann.

**Beispiel 2.5.1.** *n-Dame mit evolutionären Algorithmen.*
*Wir geben zwei verschiedene Kodierungen an.*
*Kodierung 1:*

**Codierung der Individuen** *: Ein Individuum ist eine Folge von $n$ Zahlen, wobei die $i$. Zahl die Position (Spalte) der Dame in Zeile $i$ angibt.*

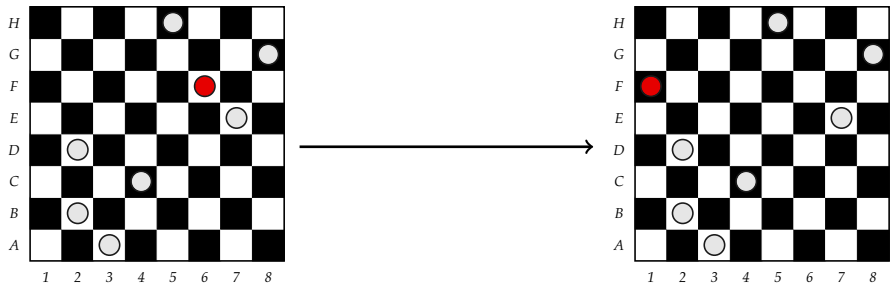
*Z.B. für das 8-Damenproblem kodiert die Folge [3,2,4,2,7,6,8,5] gerade die Belegung:*



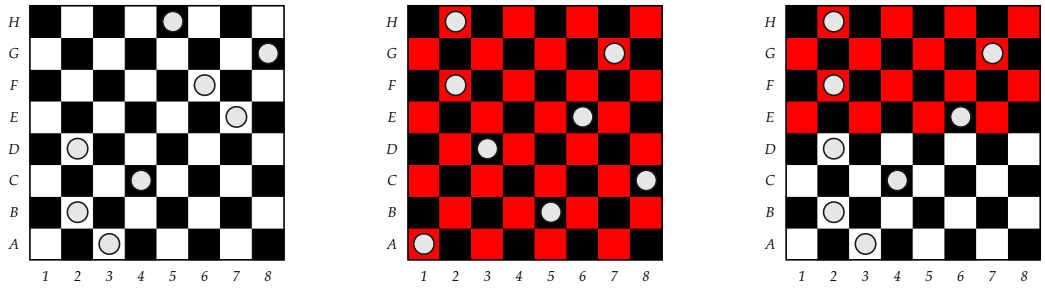
**Fitness:** *Anzahl der Damenpaare, die sich nicht gegenseitig bedrohen. Im schlechtesten Fall bedrohen sich alle Paare gegenseitig, und die Fitness ist daher 0. Im besten Fall (Optimum, Ziel) sind dies alle Damenpaare (keine Dame bedroht eine andere). Der Wert hierfür ist* $\binom{n}{2} = \frac{n*(n-1)}{2}$.

**Operationen:** *Mutation ändert einen Eintrag in der Folge auf einen beliebigen Wert zwischen 1 und* $n$. *Dies entspricht gerade dem Verschieben einer einzelnen Dame in der entsprechenden Zeile.*

*Z.B. entspricht die Mutation des Individuums* [3,2,4,2,7,6,8,5] *zum* [3,2,4,2,7,1,8,5] *gerade der folgenden Bewegung:*




*Rekombination entspricht gerade dem horizontalen Teilen der zwei Schachfelder. Z.B. beim 8-Damenproblem und den Individuen* [3,2,4,2,7,6,8,5] *und* [1,5,8,3,6,2,7,2] *ensteht durch Rekombination mit Schnitt in der Mitte das Individuum* [3,2,4,2,6,2,7,2]*:*




*Da man jedoch beim Damespiel schon weiß, das Zustände mit doppelten Zahlen in der Folge nicht gültig sind kann man alternativ wie folgt kodieren:*

*Man lässt nur* Permutationen *der Zahlen von 1 bis* $n$ *als Individuen zu. Als Mutationsoperator verwendet man das Austauschen zweier Zahlen, Die Rekombination scheint für diese Kodierung wenig sinnvoll, da sie im Allgemeinen keinen gültigen Individuen erzeugt. Wenn man sie verwendet, sollte man verhindern, dass verbotene Individuen entstehen:*

1. *großer negativer Wert der Fitnessfunktion*

2. *sofortiges Aussterben*

3. *deterministische oder zufällige Abänderung des Individuums in ein erlaubtes*

*Eine beispielhafte Veränderung einer Population für das 5-Damenproblem, wobei die Population nur aus 1-2 Elementen besteht:*

| | |
|---|---|
| *Population* | $\{[3,2,1,4,5]\}$ |
| *Fitness* | $\varphi([3,2,1,4,5]) = 4$ |
| *Mutationen* | $[3,2,1,4,5] \rightarrow [5,2,1,4,3]$; $[3,2,1,4,5] \rightarrow [3,4,1,2,5]$ |
| *Population* | $\{[5,2,1,4,3],[3,4,1,2,5]\}$ |
| *Bewertung* | $\varphi([5,2,1,4,3]) = 6$, $\varphi([3,4,1,2,5]) = 6$ |
| *Mutationen* | $[3,4,1,2,5] \rightarrow [2,5,1,4,3]$, $[5,2,1,4,3] \rightarrow [5,2,4,1,3]$ |
| *Population* | $\{[5,2,1,4,3],[3,4,1,2,5],[2,5,1,4,3],[5,2,4,1,3]\}$ |
| *Bewertung* | $\varphi([2,5,1,4,3]) = 8$, $\varphi([5,2,4,1,3]) = 10$ |
| *Selektion* | $\{[2,5,1,4,3],[5,2,4,1,3]\}$ |

*Die Kodierung* $[5,2,4,1,3]$ *ist eine Lösung.*

Reine Mutationsveränderungen sind analog zu Bergsteigen und Best-First.

#### 2.5.2.1 Bemerkungen zu evolutionären Algorithmen:

Wesentliche Unterschiede zu Standardsuchverfahren

- Evolutionäre Algorithmen benutzen Codierungen der Lösungen
- Evolutionäre Algorithmen benutzen eine Suche, die parallel ist und auf einer Menge von Lösungen basiert.
- Evolutionäre Algorithmen benutzen nur die Zielfunktion zum Optimieren
- Evolutionäre Algorithmen benutzen probabilistische Übergangsregeln. Der Suchraum wird durchkämmt mit stochastischen Methoden.

Eine gute Kodierung erfordert *annähernde Stetigkeit*. D.h. es sollte einen annähernd stetigen Zusammenhang zwischen Bitfolge und Fitnessfunktion geben. D.h. optimale Lösungen sind nicht singuläre Punkte, sondern man kann sie durch Herantasten über gute, suboptimale Lösungen erreichen.

Damit Crossover zur Optimierung wesentlich beiträgt, muss es einen (vorher meist unbekannten) Zusammenhang geben zwischen Teilfolgen in der Kodierung (Gene) und deren Zusammenwirken bei der Fitnessfunktion. Es muss so etwas wie „gute Gene “geben, deren Auswirkung auf die Fitness eines Individuums sich ebenfalls als gut erweist.

Baustein-Hypothese (Goldberg, 1989) (zur Begründung des Crossover)
(Building block hypothesis)
Genetische Algorithmen verwenden einfache Chromosomenbausteine und sammeln und mischen diese um die eigene Fitness zu optimieren.

Hypothese: das Zusammenmischen vieler guter (kleiner) Bausteine ergibt das fitteste Individuum

#### 2.5.2.2 Parameter einer Implementierung

Es gibt viele Varianten und Parameter, die man bei genetischen/evolutionären Algorithmen verändern kann.

- Welche Operatoren werden verwendet? (Mutation, Crossover, ...)
- Welche Wahrscheinlichkeiten werden in den Operatoren verwendet? Wahrscheinlichkeit einer Mutation, usw.
- Welcher Zusammenhang soll zwischen Fitnessfunktion und Selektionswahrscheinlichkeit bestehen?
- Wie groß ist die Population? Je größer, desto breiter ist die Suche angelegt, allerdings dauert es dann auch länger.
- Werden Individuen ersetzt oder dürfen sie weiterleben?
- Gibt es Kopien von Individuen in der Population?
- ...

### 2.5.3 Statistische Analyse von Genetischen Algorithmen

Vereinfachende Annahme: Fitness = Wahrscheinlichkeit, dass ein Individuum 1 Nachkommen in der nächsten Generation hat.

Es soll die Frage untersucht werden: „wie wächst die Häufigkeit einzelner Gene?“ von Generation zu Generation?

D.h. wie ist die „genetische Drift“?

Sei $S$ die Menge der Individuen, die ein bestimmtes Gen G enthält.
Die „Schema-theorie“ stellt diese mittels eines Schemas dar:
z.B. durch $[****10101****]$. Die Folge $1\,0\,1\,0\,1$ in der Mitte kann man als gemeinsames Gen der Unterpopulation $S$ ansehen. Die ebenfalls vereinfachende Annahme ist, dass das Gen vererbt wird: die Wahrscheinlichkeit der Zerstörung muss gering sein.

Sei $p()$ die Fitnessfunktion (und Wahrscheinlichkeit für Nachkommen), wobei man diese auch auf Mengen anwenden darf, und in diesem Fall die Fitness aller Individuen summiert wird. Sei $V$ die Gesamtpopulation.

Der Erwartungswert der Anzahl der Nachkommen aus der Menge $S$ ist, wenn man annimmt, dass $|V|$ mal unabhängig ein Nachkomme ermittelt wird.
$p(s_1) * |V| + \ldots + p(s_{|S|}) * |V| = p(S) * |V|$

Verbreitung tritt ein, wenn $p(S) * |V| > |S|$, d.h. wenn $\frac{p(S)}{|S|} > 1/|V|$ ist. D.h. wenn die mittlere Wahrscheinlichkeit in der Menge $|S|$ erhöht ist:

$$\frac{p(S)}{|S|} > 1/|V|$$

Damit kann man $\frac{p(S)}{|S|}$ als Wachstumsfaktor der Menge $S$ ansehen. Wenn dieser Vorteil in der nächsten Generation erhalten bleibt, dann verbreitet sich dieses Gen exponentiell. Es tritt aber eine Sättigung ein, wenn sich das Gen schon ausgebreitet hat. Analog ist das Aussterben von schlechten Genen exponentiell.

Eine etwas andere Analyse ergibt sich, wenn man annimmt, dass über die Generationen die Fitness $a$ der $S$-Individuen konstant ist, ebenso wie die Fitness $b$ der Individuen in $V \setminus S$. Dann erhält man die Nachkommenswahrscheinlichkeit durch Normierung der Fitness auf 1. Wir nehmen an, dass $S_t$ die Menge der Individuen zum Gen $S$ in der Generation $t$ ist.

Das ergibt

$$p_S := \frac{a}{a * |S_t| + b * (|V| - |S_t|)}$$

Damit ist der Erwartungswert für $|S_{t+1}|$:

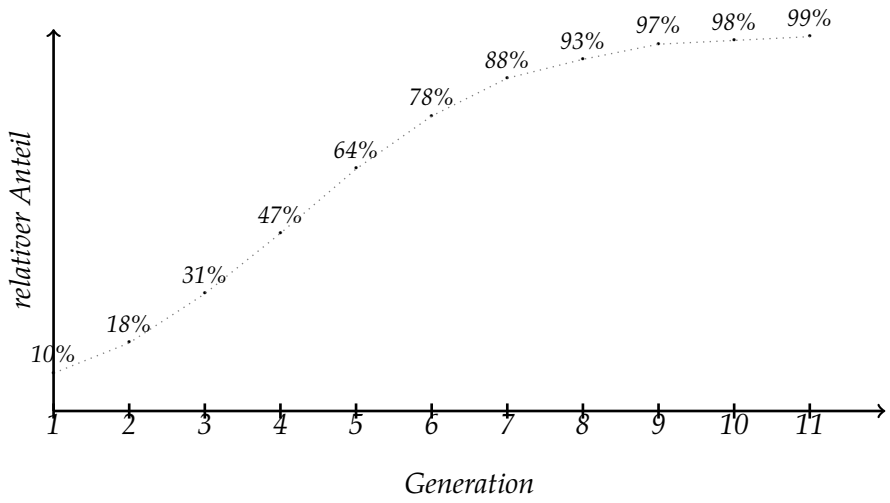
$$|S_{t+1}| = \frac{a * |S_t|}{a * |S_t| + b * (|V| - |S_t|)} * |V|$$

Wenn wir $r(S,t)$ für den relativen Anteil von $S$ schreiben, dann erhalten wir:

$$|r(S,t+1)| = \frac{a}{a * r(S,t) + b * (1 - r(S,t))} * r(S,t).$$

**Beispiel 2.5.2.** *Wählt man als Wert $a = 2b$, dann erhält man als Funktion $\frac{2r}{r+1}$, folgende Werte und Kurve für die Anteile, wenn man mit $0,1$ startet*

| $t$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $r(S,t)$ | $0,1$ | $0,18$ | $0,31$ | $0,47$ | $0,64$ | $0,78$ | $0,88$ | $0,93$ | $0,97$ | $0,98$ | $0,99$ |




### 2.5.4 Anwendungen

Man kann evolutionäre Algorithmen verwenden für Optimierungen von komplexen Problemstellungen, die

- keinen einfachen Lösungs- oder Optimierungsalgorithmus haben,
- bei denen man relativ leicht sagen kann, wie gut diese Lösungen sind
- die nicht in Echt-Zeit lösbar sein müssen, d.h. man hat ausreichend (Rechen-)Zeit um zu optimieren.
- bei denen man aus einer (bzw. zwei) (Fast-)Lösungen neue Lösungen generieren kann.
- wenn man die Problemstellung leicht mit weiteren Parametern versehen will, und trotzdem noch optimieren können will.

Beispiele:

Handlungsreisenden-Problem.

Es gibt verschiedene Kodierung, und viele Operatoren, die untersucht wurden. Die einfachste ist eine Kodierung als Folge der besuchten Städte. Die Fitnessfunktion ist die Weglänge (evtl. negativ).
**Mutationen**, die gültige Individuen (Wege) erzeugen soll, braucht Reparaturmechanismen. Man kann einen algorithmischen Schritt dazwischen schalten, der **lokale Optimierungen** erlaubt, z.B. kann man alle benachbarten Touren prüfen, die sich von der zu optimierenden Tour nur um 2 Kanten unterscheiden.
Ein vorgeschlagener Operator ist z.B. das partielle Invertieren: Man wählt zwei Schnittpunkte in einer gegebenen Tour aus, und invertiert eine der Teiltouren mit anschließender Reparatur.

Ähnliches Bemerkungen wie für Mutation gelten für **Crossover**, das eine neue Tour aus Teiltouren anderer Touren der Population zusammensetzt.

Operations Research-Probleme: Schedulingprobleme, Tourenplanung.

SAT (Erfüllbarkeit von aussagenlogischen Formeln)
Bei SAT kann man die Interpretation als Bitstring kodieren: Man nimmt eine feste Reihenfolge der Aussagenvariablen und schreibt im Bitstring die Belegung mit 0/1 hin. Mutation und Crossover erzeugen immer gültige Individuen. Als Fitnessfunktion kann man die Anzahl der wahren Klauseln nehmen.

Optimale Verteilung der Studenten auf Proseminare (Bachelor-Seminare). Dies wird im Institut mit einem evolutionären Algorithmus optimiert (geschrieben von Herrn Björn Weber)

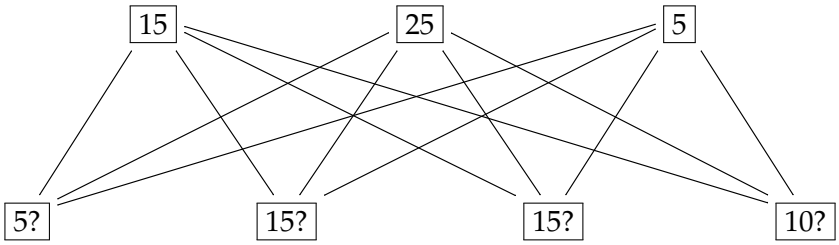
### 2.5.5 Transportproblem als Beispiel

Wir betrachten als Beispiel ein einfaches Transportproblem mit 3 Lagern $L_1, L_2, L_3$ von Waren und 4 Zielpunkten $Z_1, \ldots, Z_4$. [4]. Man kennt die Kosten für den Transport von Waren von $L_i$ nach $Z_i$ pro Einheit.

Folgende Daten sind gegeben:

| Lagerbestand | | | Warenwünsche | | | |
|---|---|---|---|---|---|---|
| $L_1$ | $L_2$ | $L_3$ | $Z_1$ | $Z_2$ | $Z_3$ | $Z_4$ |
| 15 | 25 | 5 | 5 | 15 | 15 | 10 |

Die Wünsche sind erfüllbar. Ein Bild ist:




Die Kosten pro Einheit sind:

| | $Z_1$ | $Z_2$ | $Z_3$ | $Z_4$ |
|---|---|---|---|---|
| $L_1$ | 10 | 0 | 20 | 11 |
| $L_2$ | 12 | 7 | 9 | 20 |
| $L_3$ | 0 | 14 | 16 | 18 |

[4] siehe (Michalewicz, 1992)

Eine optimale Lösung ist:

| | 5 | 15 | 15 | 10 |
|---|---|---|---|---|
| 15 | 0 | 5 | 0 | 10 |
| 25 | 0 | 10 | 15 | 0 |
| 5 | 5 | 0 | 0 | 0 |

Die Kosten sind:
$5 * 0 + 10 * 11 + 10 * 7 + 15 * 9 + 5 * 0 = 315$
Will man das Problem mittels evolutionärer Algorithmen lösen, so kann man verschiedene Kodierungen wählen.

1. Nehme die Lösungsmatrix als Kodierung.
2. Nehme eine Matrix der gleichen Dimension wie die Lösungsmatrix, Prioritäten 1,2,3,...als Einträge.

Als Fitnessfunktion wählt man die Kosten (bzw. negativen Kosten: Ersparnisse).

1. hat den Vorteil, dass man eine exakte Kodierung hat. Allerdings weiss man bei diesem Problem, dass man die billigsten Fuhren zuerst nehmen kann, so dass man damit evtl. Lösungen variiert, die man mit etwas Einsicht in das Problem als äquivalent ansehen kann. Als Bedingung hat man noch, dass es keine negativen Werte geben soll und dass die Spaltensummen den Wünschen entsprechen sollen, die Zeilensummen sollen nicht größer sein als der jeweilige Lagerbestand.
2. hat den Vorteil, dass alle Kodierung gültig sind und man das Wissen über das Problem nutzt. Aus den Prioritäten wird nach folgendem festgelegten Schema eine Lösung berechnet: zunächst wird die Fuhre mit der Priorität 1 so voll wie möglich gemacht. Danach unter Beachtung, dass schon geliefert wurde, die Fuhre mit Priorität 2. usw. Hierbei verliert man keine optimalen Lösungen. Zudem kann man leichter neue Kodierungen erzeugen.

   Eine Kodierung entspricht einer Permutation der 12 Matrixeinträge, die als Folge von 12 Zahlen geschrieben werden kann.

**Mutation in 1.** : verschiebe eine Wareneinheit auf eine andere Fuhre. Hierbei kann aber eine ungültige Lösung entstehen. Crossover: unklar.

**Mutation in 2)** : permutiere die Folge durch Austausch von 2 Prioritäten. Hierbei entsteht eine gültige Kodierung.

**Inversion:** invertiere die Reihenfolge der Prioritäten:

**Crossover:** Wähle zwei Permutation $p_1, p_2$ aus der Population aus, und wähle aus $p_1$ ein Unterfolge aus. Entferne aus $p_2$ die Prioritäten, die in $p_2$ vorkommen, ergibt Folge

$p'_2$. Dann setze die Prioritäten an den freien Stellen in $p'_2$ wieder ein.
Alternativ (aber ohne bezug: setze die Prioritäten an irgendwelchen Stellen wieder ein.

### 2.5.6 Schlußbemerkungen

**Bemerkung 2.5.3.** *Es gibt generische Programme, die die Berechnung der Generationen usw. bereits ausführen, und denen man nur noch die Kodierung der Problemstellung geben muss, die Kodierung der Operatoren und die Parameter.*

*Es gibt auch verschiedene eingebettete implementierte Anwendungen*

**Bemerkung 2.5.4.** *Für komplexe Probleme und deren Optimierung gilt die Heuristik: mit einem Evolutionären Algorithmus kann man immer ganz brauchbar verbessern, wenn man wenig über die Struktur und Eigenschaft der Problemklasse sagen kann.*

*Aber sobald man sich mit der Problemklasse genauer beschäftigt und experimentiert, wird ein Optimierungsalgorithmus informierter sein, und sich mehr und mehr zu einem eigenständigen Verfahren entwickeln, das bessere Ergebnisse produziert als allgemeine Ansätze mit evolutionären Algorithmen.*

*Probleme der genetischen/evolutionären Algorithmen ist die möglicherweise sehr lange Laufzeit Man braucht mehrere Durchläufe, bis man eine gute EInstellung der Parameter gefunden hat. Auch problematisch ist, dass unklar ist, wann man mit Erfolg aufhören kann und wie nahe man einem Optimum ist.*

# 3 Aussagenlogik

In diesem Kapitel werden die Aussagenlogik und dort angesiedelte Deduktionsverfahren erörtert. Wir zeigen, wie diese (relativ einfache) Logik verwendet werden kann, um intelligente Schlüsse zu ziehen und Entscheidungsverfahren zu implementieren. Bereits in Kapitel 1 haben wir in Abschnitt 1.4.2 die Wissenrepräsentationshypothese nach Brian Smith erörtert. Wir wiederholen sie an dieser Stelle, da sie als Paradigma insbesondere dieses Kapitel motiviert:

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

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

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

## 3.1 Syntax und Semantik der Aussagenlogik

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

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

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

**Definition 3.1.1** (Syntax der Aussagenlogik)**.** *Sei $X$ ein Nichtterminal für aussagenlogische* Variablen (*aus einer Menge von abzählbar unendlich vielen Variablen*)*. Die folgende Grammatik legt die* Syntax *aussagenlogischer Formeln fest:*

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

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

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

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

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

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

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

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

**Definition 3.1.2** (Semantik der Junktoren)**.** *Zunächst definiert man für jeden Junktor $\neg, \wedge, \vee, \Rightarrow$ und $\Leftrightarrow$ Funktionen $\{0,1\} \to \{0,1\}$ bzw. $\{0,1\}^2 \to \{0,1\}$. Die Funktion $f_\neg$ ist definiert als $f_\neg(0) = 1, f_\neg(1) = 0$. Ebenso definiert man $f_0 := 0, f_1 := 1$. Die anderen Funktionen $f_{op}$ sind definiert gemäß folgender Tabelle:*

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

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

**Definition 3.1.3** (Interpretation)**.** *Eine* Interpretation (*Variablenbelegung*) $I$ *ist eine Funktion:*

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

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

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

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

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

**Definition 3.1.4.** *Sei $A$ ein Aussage.*

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

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

**Beispiel 3.1.5.** *Wir betrachten einige Beispiele:*

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

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

- $(X \Rightarrow Y) \Rightarrow ((Y \Rightarrow Z) \Rightarrow (X \Rightarrow Z))$ *ist eine Tautologie.*

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

- $I$ *mit* $I(X) = 1, I(Y) = 0$ *ist ein Modell für* $X \wedge \neg Y$

**Beispiel 3.1.6.** *Bauernregeln:*

- *„Abendrot Schlechtwetterbot'" kann man übersetzen in*
  *Abendrot* $\Rightarrow$ *Schlechtes_Wetter. Diese Aussage ist weder Tautologie noch Widerspruch, aber erfüllbar.*

- *„Wenn der Hahn kräht auf dem Mist, ändert sich das Wetter oder es bleibt wie es ist. " kann man übersetzen in*
  *Hahn_kraeht_auf_Mist* $\Rightarrow$ *(Wetteraenderung* $\vee$ $\neg$*Wetteraenderung).*
  *Man sieht, dass das eine Tautologie ist.*

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

**Theorem 3.1.7.**

- *Es ist entscheidbar, ob eine Aussage eine Tautologie (Widerspruch, erfüllbar) ist.*

- *Die Frage „Ist A erfüllbar?" ist* $\mathcal{NP}$*-vollständig.*

- *Die Frage „Ist A falsifizierbar?" ist ebenso* $\mathcal{NP}$*-vollständig.*

- *Die Frage „Ist A Tautologie?" ist* $Co\mathcal{NP}$*-vollständig.*

- *Die Frage „Ist A ein Widerspruch?" ist* $Co\mathcal{NP}$*-vollständig.*

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

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

Eine Problemklasse ist $CoNP$-vollständig, wenn die Problemklasse, die aus dem Komplement gebildet wird, $NP$-vollständig ist. [1]

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

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

## 3.2 Folgerungsbegriffe

Man muß zwei verschiedene Begriffe der Folgerungen für Logiken unterscheiden:

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

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

**Definition 3.2.1.** *Sei $\mathcal{F}$ eine Menge von (aussagenlogischen) Formeln und $G$ eine weitere Formel.*

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

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

Das Wortproblem ist die Frage, ob eine gegebene Formel in der Sprache SAT liegt oder nicht.

Die Komplexitätsklasse $NP$ umfasst alle Sprachen, deren Wortproblem auf einer nichtdeterministischen Turingmaschine in polynomieller Zeit lösbar ist. Die Komplexitätsklasse $CoNP$ ist definiert als:

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

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

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

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

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

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

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

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

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

**Satz 3.2.2.** *Wenn ein $F_i$ ein Widerspruch ist, dann kann man alles folgern: Es gilt dann für jede Formel G: $F_1, \ldots, F_n \models G$.*
*Wenn ein $F_i$ eine Tautologie ist, dann kann man dies in den Voraussetzungen weglassen: Es gilt dann für alle Formeln G: $F_1, \ldots, F_n \models G$ ist dasselbe wie $F_1, \ldots, F_{i-1}, F_{i+1}, \ldots, F_n \models G$*

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

**Theorem 3.2.3** (Deduktionstheorem der Aussagenlogik)**.**
*$\{F_1, \ldots, F_n\} \models G$ gdw. $F_1 \wedge \ldots \wedge F_n \Rightarrow G$ ist Tautologie.*

Zwei Aussagen $F, G$ nennen wir *äquivalent* ($F \sim G$), gdw. $F \iff G$ eine Tautologie ist. Beachte, dass $F$ und $G$ genau dann äquivalent sind, wenn für alle Interpretationen $I$ gilt: $I \models F$ gdw. $I \models G$ gilt.
Es ist auch zu beachten, dass z.B. $X \wedge Y$ nicht zu $X' \wedge Y'$ äquivalent ist. D.h. bei diesen Beziehungen spielen die Variablennamen eine wichtige Rolle.

**Definition 3.2.4.** *Gegeben sei ein (nicht-deterministischer) Algorithmus $\mathcal{A}$ (ein Kalkül), der aus einer Menge von Formeln $\mathcal{H}$ eine neue Formel $H$ berechnet, Man sagt auch, dass $H$* syntaktisch *aus $\mathcal{H}$* folgt *(bezeichnet mit $\mathcal{H} \vdash_A H$ oder auch $\mathcal{H} \to_A H$).*

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

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

Es gibt verschiedene Methoden, aussagenlogische Tautologien (oder auch erfüllbare Aussagen) algorithmisch zu erkennen: Z.B. BDDs (binary decision diagrams) : eine Methode, Aussagen als Boolesche Funktionen anzusehen und möglichst kompakt zu repräsentieren; Genetische Algorithmen zur Erkennung erfüllbarer Formeln; Suchverfahren die mit statischer Suche und etwas Zufall und Bewertungsfunktionen operieren; DPLL-Verfahren: Fallunterscheidung mit Simplifikationen (siehe unten 3.7); Tableaukalkül, der Formeln syntaktisch analysiert (analytic tableau).

## 3.3 Tautologien und einige einfache Verfahren

Wir wollen im folgenden Variablen in Aussagen nicht nur durch die Wahrheitswerte $0, 1$ ersetzen, sondern auch durch Aussagen. Wir schreiben dann $A[B/x]$, wenn in der Aussage $A$ die Aussagenvariable $x$ durch die Aussage $B$ ersetzt wird. In Erweiterung dieser Notation schreiben wir auch: $A[B_1/x_1, \ldots, B_n/x_n]$, wenn mehrere Aussagevariablen ersetzt werden sollen. Diese Einsetzung passiert gleichzeitig, so dass dies kompatibel zur Eigenschaft der Komposition als Funktionen ist: Wenn $A$ $n$-stellige Funktion in den Aussagevariablen ist, dann ist $A[B_1/x_1, \ldots, B_n/x_n]$ die gleiche Funktion wie $A \circ (B_1, \ldots, B_n)$.

**Theorem 3.3.1.** *Gilt $A_1 \sim A_2$ und $B_1, \ldots, B_n$ sind weitere Aussagen, dann gilt auch $A_1[B_1/x_1, \ldots, B_n/x_n] \sim A_2[B_1/x_1, \ldots, B_n/x_n]$.*

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

**Theorem 3.3.2.** *Sind $A, B$ äquivalente Aussagen, und $F$ eine weitere Aussage, dann sind $F[A]$ und $F[B]$ ebenfalls äquivalent.* [2]

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

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

Weiterhin gibt es für Aussagen noch Rechenregeln und Gesetze, die wir im folgenden auflisten wollen. Alle lassen sich mit Hilfe der Wahrheitstafeln beweisen, allerdings erweist sich das bei steigender Variablenanzahl als mühevoll, denn die Anzahl der Überprü-

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

fungen 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 3.3.3** (Äquivalenzen)**.**

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

Diese Regeln sind nach Satz 3.3.1 nicht nur verwendbar, wenn $A, B, C$ Aussagevariablen sind, sondern auch, wenn $A, B, C$ für Aussagen stehen.

## 3.4 Normalformen

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

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

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

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

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

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

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

Die Klauselnormalform wird oft als Menge von Mengen notiert und auch behandelt. Dies ist gerechtfertigt, da sowohl $\wedge$ als auch $\vee$ assoziativ, kommutativ und idempotent

sind, so dass Vertauschungen und ein Weglassen der Klammern erlaubt ist. Wichtig ist die Idempotenz, die z.B. erlaubt, eine Klausel $\{A, B, A, C\}$ als unmittelbar äquivalent zur Klausel $\{A, B, C\}$ zu betrachten. Eine Klausel mit einem Literal bezeichnet man auch als *1-Klausel* (Unit-Klausel). Eine Klausel (in Mengenschreibweise) ohne Literale wird als *leere Klausel* bezeichnet. Diese ist äquivalent zu $0$, dem Widerspruch.

**Lemma 3.4.1.** *Eine Klausel $C$ ist genau dann eine Tautologie, wenn es eine Variable $A$ gibt, so dass sowohl $A$ als auch $\neg A$ in der Klausel vorkommen*

*Beweis.* Übungsaufgabe.

Es gilt:

**Lemma 3.4.2.** *Zu jeder Aussage kann man eine äquivalente CNF finden und ebenso eine äquivalente DNF. Allerdings nicht in eindeutiger Weise.*

**Lemma 3.4.3.**

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

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

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

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

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

**Lemma 3.4.4.** *Sei $F$ eine Formel. Dann ist $F$ allgemeingültig gdw. $\neg F$ ein Widerspruch ist*

**Bemerkung 3.4.5.** *Aus Lemma 3.4.3 kann man Schlussfolgerungen ziehen über die zu erwartende Komplexität eines Algorithmus, der eine Formel (unter Äquivalenzerhaltung) in eine DNF (CNF) transformiert. Wenn dieser Algorithmus polynomiell wäre, dann könnte man einen polynomiellen Tautologietest durchführen:*

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

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

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

**Definition 3.4.6** (Transformation in Klauselnormalform)**.** *Folgende Prozedur wandelt jede aussagenlogische Formel in konjunktive Normalform (CNF, Klauselnormalform) um:*

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

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

2. *Negation ganz nach innen schieben:*

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

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

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

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

*oder in (Multi-)Mengenschreibweise:*

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

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

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

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

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

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

**Beispiel 3.4.7.**

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

## 3.5 Lineare CNF

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

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

**Lemma 3.5.1.** *$F[G]$ ist erfüllbar gdw. $(G \iff X) \wedge F[X]$ erfüllbar ist. Hierbei muß $X$ eine Variable sein, die nicht in $F[G]$ vorkommt.*

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

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

Analog dazu erhält diese Transformation auch die Widersprüchlichkeit.

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

**Definition 3.5.2.** *Die Tiefe einer Subformel in Tiefe $n$ definiert man entsprechend dem Aufbau der Syntax: Zunächst ist jede Formel $F$ eine Subformel der Tiefe $0$ von sich selbst. Sei $H$ eine Subformel von $F$ der Tiefe $n$. Dann definieren wir Subformeln von $F$ wie folgt:*

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

*Die Tiefe einer Formel sei die maximale Tiefe einer Subformel.*

*Algorithmus* **Schnelle CNF-Berechnung**

**Eingabe:** Aussagenlogische Formel $F$

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

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

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

**Satz 3.5.3.** *Zu jeder aussagenlogischen Formel kann unter Erhaltung der Erfüllbarkeit eine CNF in Zeit $O(n)$ berechnet werden, wobei $n$ die Größe der aussagenlogischen Formel ist.*

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

**Bemerkung 3.5.4.** *Sei $F[G]$ eine Tautologie. Dann ist die Formel $F[X] \wedge (X \iff G)$, wobei $X$ eine neue aussagenlogische Variable, keine Tautologie: Seie $I$ eine Interpretation, die $F[G]$ wahr macht (d.h. $I(F[G]) = 1$).*

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

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

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

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

**Beispiel 3.5.5.** *Wandle eine DNF in eine CNF auf diese Art und Weise um unter Erhaltung der Erfüllbarkeit. Sei $(D_{11} \wedge D_{12}) \vee \ldots \vee (D_{n1} \wedge D_{n2})$ die DNF. Wenn man das Verfahren leicht abwandelt, damit man besser sieht, was passiert: ersetzt man die Formeln $(D_{i1} \wedge D_{i2})$ durch neue Variablen $A_i$ und erhält am Ende:*
*$(A_1 \vee \ldots \vee A_n) \wedge (\neg A_1 \vee D_{11}) \wedge (\neg A_1 \vee D_{12}) \wedge (\neg D_{11} \vee \neg D_{12} \vee A_1) \wedge \ldots$. Diese Formel hat $8n$ Literale.*

**Beispiel 3.5.6.** *Wandle die Formel*

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

*um. Das ergibt:*

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

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

**Bemerkung 3.5.7.** *Es gibt zahlreiche Implementierungen von Algorithmen, die Formeln in Klauselmengen verwandelt. Z.B. siehe die www-Seite* `http://www.spass-prover.org/`.

## 3.6 Resolution für Aussagenlogik

Das Resolutionsverfahren dient zum Erkennen von Widersprüchen, wobei statt des Testes auf Allgemeingültigkeit einer Formel $F$ die Formel $\neg F$ auf Unerfüllbarkeit getestet wird. Eine Begründung wurde bereits gegeben. Eine erweiterte liefert das folgende Lemma zum Beweis durch Widerspruch:

**Lemma 3.6.1.** *Eine Formel $A_1 \wedge \ldots \wedge A_n \Rightarrow F$ ist allgemeingültig gdw. $A_1 \wedge \ldots \wedge A_n \wedge \neg F$ widersprüchlich ist.*

*Beweis.* Übungsaufgabe □

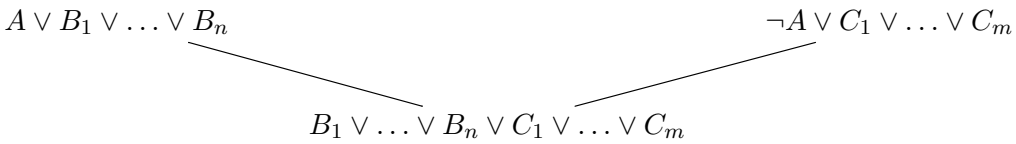
Die semantische Entsprechung ist:

**Lemma 3.6.2.** *$\{A_1, \ldots, A_n\} \models F$ gdw. es keine Interpretation $I$ gibt, so dass $I \models \{A_1, \ldots, A_n, \neg F\}$.*

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

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

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




Auf der Ebene der Klauselmengen sieht das Verfahren so aus:

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

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

Eingesetzt wird die Resolution zur Erkennung unerfüllbarer Klauselmengen. Es werden solange Resolventen hergeleitet, bis entweder die leere Klausel hinzugefügt wurde

oder keine neue Resolvente mehr herleitbar ist. Dies geschieht meist in der Form, dass man Axiome (als Konjunktion) eingibt und ebenso eine negierte Folgerung, so dass die Unerfüllbarkeit bedeutet, dass man einen Widerspruch hergeleitet hat.

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

**Lemma 3.6.3.** *Wenn $C \to C'$ mit Resolution, dann ist $C$ äquivalent zu $C'$.*

*Beweis.* Wir zeigen den nichttrivialen Teil:

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

**Lemma 3.6.4.** *Die Resolution auf einer aussagenlogischen Klauselmenge terminiert, wenn man einen Resolutionsschritt nur ausführen darf, wenn sich die Klauselmenge vergrößert.*

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

**Übungsaufgabe 3.6.5.** *Gebe ein Beispiel an, so dass $R$ sich aus $C_1, C_2$ mit Resolution herleiten läßt, aber $C_1 \wedge C_2 \iff R$ ist falsch*

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

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

D.h. es wurden zwei tautologische Klauseln hinzugefügt. (Eine Klausel ist eine Tautologie, wenn $P, \neg P$ vorkommt für ein $P$.) Ein Modell läßt sich nicht direkt ablesen. Zum Generieren von Modellen ist z.B. die DPLL-Prozedur (siehe 3.7) geeigneter.

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

**Theorem 3.6.6.** *Für eine unerfüllbare Klauselmenge findet Resolution nach endlich vielen Schritten die leere Klausel.*

*Beweis.* Wir verwenden das folgende Maß für Klauselmengen $C$:

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

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

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

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

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

**Theorem 3.6.7.** *Resolution erkennt unerfüllbare Klauselmengen.*

Die Komplexität im schlimmsten Fall wurde von A. Haken (Haken, 1985) (siehe auch (Eder, 1992) nach unten abgeschätzt: Es gibt eine Folge von Formeln (die sogenannten Taubenschlag-formeln (pigeon hole formula, Schubfach-formeln), für die gilt, dass die

kürzeste Herleitung der leeren Klausel mit Resolution eine exponentielle Länge (in der Größe der Formel) hat.

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

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

### 3.6.1 Optimierungen des Resolutionskalküls

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

#### 3.6.1.1 Tautologische Klauseln

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

**Definition 3.6.8.** *Eine Klausel ist* tautologisch, *wenn sie ein Literal sowohl positiv als auch negativ enthält.*

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

**Satz 3.6.9.** *Sei $C \cup \{K\}$ eine Klauselmenge, wobei $K$ eine tautologische Klausel ist. Dann sind $C \cup \{K\}$ und $C$ äquivalente Formeln.*

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

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

#### 3.6.1.2 Klauseln mit isolierten Literalen

**Definition 3.6.10.** *Ein Literal $L$ ist* isoliert *in einer Klauselmenge $C$, wenn das Literal $\overline{L}$ in $C$ nicht vorkommt. Dabei ist $\overline{L}$ das zu $L$ negierte Literal, d.h.*

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

**Beispiel 3.6.11.** *In $\{\{A, \neg B\}, \{A, \neg B, C\}, \{\neg A, C\}\}$ ist das Literal $C$ isoliert, da $\neg C$ nicht in der Klauselmenge vorkommt. Ebenso ist $\neg B$ isoliertes Literal, da $B$ nicht in der Klauselmenge vorkommt.*

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

**Satz 3.6.12.** *Sei $C$ eine Klauselmenge und $C'$ die Klauselmenge, die aus $C$ entsteht, indem alle Klauseln entfernt werden, die isolierte Literale enthalten. Dann ist $C$ erfüllbar gdw. $C'$ erfüllbar ist.*

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

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

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

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

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

**Bemerkung 3.6.13.** *Betrachte die Klauselmenge $C = \{\{A, \neg A\}, \{B\}\}$. Das Literal $B$ ist isoliert. Nach Löschen der Klausel entsteht $C' = \{\{A, \neg A\}\}$. Während $C'$ eine Tautologie ist, ist $C$ falsifizierbar (mit $I(B) = 0$).*

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

#### 3.6.1.3 Subsumtion

**Definition 3.6.14.** *Eine Klausel $K_1$* subsumiert *eine Klausel $K_2$, wenn $K_1 \subseteq K_2$ gilt. Man sagt in diesem Fall auch: „$K_2$ wird von $K_1$ subsumiert“.*

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

**Satz 3.6.15.** *Sei $C \cup \{K_1\} \cup \{K_2\}$ eine Klauselmenge, wobei $K_1$ die Klausel $K_2$ subsumiert. Dann sind die Formeln $C \cup \{K_1\} \cup \{K_2\}$ und $C \cup \{K_1\}$ äquivalent.*

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

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

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

□

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

## 3.7 DPLL-Verfahren

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

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

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

*Algorithmus* **DPLL-Prozedur**

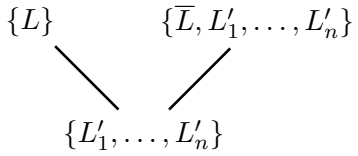
**Eingabe:** Klauselmenge $C$, die keine tautologischen Klauseln enthält

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

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

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

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



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

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

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

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

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

1. Isolierte Literale werden als wahr angenommen.

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

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

**Beispiel 3.7.1.** *Betrachte folgende Klauselmenge, wobei jede Zeile einer Klausel entspricht.*

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

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

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

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

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

*Weitere Schritte für $Q$ ergeben*

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

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

**Beispiel 3.7.2.** *Wir nehmen uns ein Rätsel von Raymond Smullyan vor: Die Frage nach dem Pfefferdieb. Es gibt drei Verdächtige: Den Hutmacher, den Schnapphasen und die (Hasel-)Maus. Folgendes ist bekannt:*

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

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

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

*Dies ergibt eine Klauselmenge (doppelte sind schon eliminiert):*

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

*Wir können verschiedene Verfahren zur Lösung verwenden.*

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

**Beispiel 3.7.3.** *Eine Logelei aus der Zeit: Abianer sagen die Wahrheit, Bebianer Lügen. Es gibt folgende Aussagen:*

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

*Eine offensichtliche Kodierung ergibt*

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

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

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

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


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

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

**Beispiel 3.7.4.** *Ein weiteres Rätsel von Raymond Smullyan:*

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

*Die Aussagen und die bekannten Tatsachen sind:*

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

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

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

*Die Formulierung ist:*

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

*höchstens einer sagt nicht die Wahrheit:*

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

*genau einer ist der Dieb:*

$$FID \vee FRD \vee HBD$$

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

*Die Aussagen:*

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

*Eingabe in den DPLL-Algorithmus:*

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

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

*Der Aufruf*

```
dpalle fischfroschexpr
```

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

**Beispiel 3.7.5.** *Anwendung auf ein Suchproblem: Das n-Damen Problem. s sollen Königinnen auf einem quadratischen Schachbrett der Seitenlänge $n$ so platziert werden, dass diese sich nicht schlagen können. Damit die Formulierung einfacher wird, erwarten wir, dass sich in jeder Zeile und Spalte eine Königin befindet. Man kann leicht ein Programm schreiben, dass die entsprechende Klauselmenge direkt erzeugt. Wir demonstrieren dies in Haskell, wobei wir Literale durch Zahlen darstellen: Negative Zahlen entsprechen dabei negativen Literalen.*

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

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

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

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

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

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

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

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

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

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

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

*Das Ergebnis der DPLL-Prozedur sind zwei Interpretationen:*

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

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

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

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

**Beispiel 3.7.6.** *Das n-Damen Problem gibt es noch in weiteren Varianten: Die Torus-Variante: In dem Fall setzt sich die Bedrohung über der Rand an der Seite und oben und unten fort. Die die Bedrohung entlang Diagonallinien des* $8 \times 8$ *Feldes ist so dasss die Dame in der ersten Zeile im unteren Beispiel die Dame in der vierten Zeile bedroht, wenn man die Diagonale nach links unten verfolgt.*

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

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

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

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

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

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

## 3.8 Modellierung von Problemen als Aussagenlogische Erfüllbarkeitsprobleme

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

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

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

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

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

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

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

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

Die Kodierung der Bedingung, dass genau eine Formel wahr ist, ist nun die Konjunktion der $at\_least\_one$ und $at\_most\_one$ Formeln:

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

**Beispiel 3.8.1.** *Wir betrachten das folgende Problem: $s$ Studenten schreiben Klausuren. Der Dozent hat $k$ verschiedene Klausurtypen entworfen und er weiß bereits, wie die Studenten im Klausurraum sitzen (er kennt die Paarungen aller benachbart sitzenden Studenten). Der Dozent möchte verhindern, dass benachbart sitzende Studenten den gleichen Klausurtyp erhalten. Das Problem besteht daher darin, welcher Student welchen Klausurtyp erhält.*

*Wir verwenden die Aussagenlogischen Variablen:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Die Formel, die zusichert, dass genau $K$ Formeln aus $S$ wahr sind, ist die Konjunktion der $at\_least$ und $at\_most$ Formeln:

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

**Beispiel 3.8.2.** *Wir betrachten eine Logelei aus der Zeit:*
*Tom, ein Biologiestudent, sitzt verzweifelt in der Klausur, denn er hat vergessen, das Kapitel über die Wolfswürmer zu lernen. Das Einzige, was er weiß, ist, dass es bei den Multiple-Choice-Aufgaben immer* genau 3 korrekte *Antworten gibt. Und dies sind die angebotenen Antworten:*

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

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

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

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

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

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

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

Z.B. kann man das Sudoku

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

folgendermaßen Vervollständigen (Lösen):

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

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

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

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

Die Klauselmenge zur Lösung des Sudokus besteht aus mehreren Teilen:

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

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

Die restlichen Klauseln sind statisch für jeden 9x9 Sudokus. Zunächst müssen wir zusichern, dass in jedem Feld genau eine Zahl steht. Hier für lässt sich die $exactly\_one$-Formel verwenden:

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

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

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

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

Schließlich brauchen wir noch die Bedingungen für die 3x3 Quadrate

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

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

# 4 Prädikatenlogik

## 4.1 Syntax und Semantik der Prädikatenlogik ($PL_1$)

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

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

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

### 4.1.1 Syntax der Prädikatenlogik erster Stufe

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

- Signatur
- Bildungsregeln für Terme
- Bildungsregeln für Formeln

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

**Beispiel 4.1.1.** *In einer Formel „$\forall x : \exists y : P(x, f(y))$“ sind $x$ und $y$ Variablensymbole, $f$ ist ein einstelliges Funktionssymbol und $P$ ein zweistelliges Prädikatensymbol.*

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

werte.
Die formale Definition ist:

**Definition 4.1.2** (Syntax von $PL_1$)**.** *Die Syntax von $PL_1$ besteht zunächst aus einer* Signatur $\Sigma$*, darauf aufbauend werden* Terme *erzeugt, die schließlich innerhalb von* Formeln *verwendet werden:*

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

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

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

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

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

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

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

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

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

**Beispiel 4.1.3.** *Signatur $\Sigma := (\{a, b, f, g\}, \{P, Q, R\})$ mit $arity(a) = arity(b) = arity(P) = 0$, $arity(f) = arity(Q) = 1$ $arity(g) = arity(R) = 2$. $V = \{x, y, z, \ldots\}$*

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

Mit der Forderung, dass mindestens ein Konstantensymbol vorhanden sein muss, ist implizit verbunden, dass die Trägermengen, über die in einer jeweiligen Interpretation quantifiziert wird, nicht leer sein kann – es muss mindestens ein Element vorhanden sein, auf das dieses Konstantensymbol abgebildet wird. Damit verhindert man, dass Quantifizierungen der Art „$\forall x : (P \wedge \neg P)$“ wahr gemacht werden können; indem die Menge, über die quantifiziert wird, leer ist. Einige der logischen Verknüpfungen, wie z.B. $\Rightarrow$ und $\Leftrightarrow$ sind redundant. Sie können durch die anderen dargestellt werden. Bei der Herstellung der Klauselnormalform (Abschnitt 4.1.4) werden sie dann auch konsequenterweise wieder eliminiert. Nichtsdestotrotz erleichtern sie die Lesbarkeit von Formeln beträchtlich und sind daher mit eingeführt worden.

**Definition 4.1.4** (Konventionen)**.** *Wir verwenden die folgenden Konventionen:*

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

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

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

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

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

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

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

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

**Beispiel 4.1.5.** *Seien* $x, y, z$ *Variablensymbole, dann gilt:*

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

**Definition 4.1.6** (Sprechweisen)**.** *Wir führen einige übliche Sprechweisen für spezielle Formeln und Terme ein:*

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

**Beispiel 4.1.7.** *Für eine geschlossene Formel:* $\forall x : \exists y : \ P(x, y)$*. Nicht geschlossen ist:* $\exists y : P(x, y)$*, da* $FV(\exists y : \ P(x, y)) = \{y\})$ .

### 4.1.2 Semantik

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

**Definition 4.1.8. Interpretation** *Gegeben eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und Variablensymbole $V$. Eine* Interpretation $I = (S, I_V)$ *mit $S = (D_S, \mathcal{F}_S, \mathcal{P}_S)$ besteht aus einer nichtleeren Menge $D_S$ (die Trägermenge), einer Interpretation der Funktionssymbole $\mathcal{F}_S$, die jedem Funktionssysmbol $f \in \mathcal{F}$ eine totale Funktionen $f_S \in \mathcal{F}_S$ über $D$ zuordnet und der Interpretation der Prädikatensymbole $\mathcal{P}_S$, die jedem Prädikat $P \in \mathcal{P}$ ein Prädikat $P_S \in \mathcal{P}_S$ (als Relation über $D$) zuordnet. Schließlich ist $I_V : V \to D_S$ eine Variablenbelegung, die jedem Variablensymbol einen Wert in der Trägermenge von $D_S$ zuordnet.*

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

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

**Definition 4.1.9.** (*Termalgebra*) *Für eine Signatur $\Sigma = (\mathcal{F}, \mathcal{P})$ und eine Menge von Variablen $V$ sei $(T(\Sigma, V)$ die Menge der Terme über der Signatur $\Sigma$ und den Variablen $V$. Sei $F_T$ die Menge der Funktionen $\{f_\Sigma \mid f \in \mathcal{F}, arity(f) = n, f_\Sigma(t_1, \ldots, t_n) := f(t_1, \ldots, t_n)\})$. Dann nennt man $(T(\Sigma, V), F_T)$ die* Termalgebra *über der Signatur $\Sigma$.*

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

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

**Definition 4.1.10** (Auswertung von Formeln)**.** *Sei $I$ eine Interpretation. Die Basisfälle sind:*

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

*Rekursionsfälle:*

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

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

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

**Definition 4.1.11** (Modelle, Tautologien etc.)**.** *Eine Interpretation $I$, die eine Formel $F$ wahr macht (erfüllt) heißt* Modell *von $F$. Man sagt auch: $F$ gilt in $I$ ($F$ ist wahr in $I$, $I$ erfüllt $F$). Bezeichnung: $I \models F$.*

*Eine Formel $F$ heißt:*

**allgemeingültig (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 4.1.12.** *Einige Beispiele für allgemeingültige, unerfüllbare, erfüllbare und falsifizierbare Formeln sind:*

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

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

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

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

Als weiteren, einfachen Testfall untersuchen wir Klauseln.

**Beispiel 4.1.13.** *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 $D_S$ und eine $\Sigma$-Algebra. Man startet mit einer Menge $A_0$, die mindestens soviele Elemente enthält, wie die Terme $s, t$ Konstanten und Variablen enthalten. Also $A_0 := \{a_1, \ldots, a_n, c_{x_1}, \ldots c_{x_m}\}$, wobei $a_i$ die Konstanten in $s, t$ sind und $x_i$ die Variablen.*

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

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

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

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

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

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

**Definition 4.1.14** (Semantische Folgerung)**.**
*$F \models G$ gdw. $G$ gilt (ist wahr) in allen Modellen von $F$.*

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

**Bemerkung 4.1.15.** *Das Beispiel der natürlichen Zahlen und der darin geltenden Sätze ist nicht vollständig mit $PL_1$ zu erfassen. Der Grund ist, dass man nur von einer einzigen festen $\Sigma$-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 4.1.16.** *Ein Beispiel für eine in $PL_1$ modellierbare Theorie sind die Gruppen. Man benötigt nur endlich viele Axiome. Und man kann dann danach fragen, welche Sätze in allen Gruppen gelten.*

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

**Theorem 4.1.17** (Deduktionstheorem)**.** *Für alle Formeln $F$ und $G$ gilt: $F \models G$ gdw. $F \Rightarrow G$ ist allgemeingültig (Tautologie).*

**Bemerkung 4.1.18.** *Will man wissen, ob eine Formel $F$ aus einer Menge von Axiomen $A_1, \ldots, A_n$ folgt, so kann man dies zunächst auf die äquivalente Frage zurückführen, ob $F$ aus der Konjunktion der Axiome $A_1 \wedge \ldots \wedge A_n$ folgt und dann auf die äquivalente Frage, ob die Implikation $(A_1 \wedge \ldots \wedge A_n) \Rightarrow F$ eine Tautologie ist.*

Das Deduktionstheorem gilt in anderen Logiken i.A. nicht mehr. 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 $PL_1$ definert 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.

### 4.1.3 Berechenbarkeitseigenschaften der Prädikatenlogik

Es gilt die Unentscheidbarkeit der Prädikatenlogik:

**Theorem 4.1.19.** *Es ist unentscheidbar, ob eine geschlossene Formel ein Satz der Prädikatenlogik ist.*

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

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

**Theorem 4.1.20.** *Die Menge der Sätze der Prädikatenlogik ist rekursiv aufzählbar.*

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

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

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

### 4.1.4 Normalformen von $PL_1$-Formeln

Ziel diese Abschnitts ist es, einen Algorithmus zu entwickeln, der beliebige Formeln in eine Klauselnormalform (conjunctive normal form, CNF), d.h. eine Konjunktion ($\wedge$) von Disjunktionen ($\vee$ ) von Literalen, transformiert. Diese Klauselnormalform ist nur „ganz außen“ allquantifiziert, es gibt keine inneren Quantoren. Folgendes Lemma erlaubt die Transformation von Formeln, wobei die Regeln in Tautologien entsprechen, aber als Transformationen benutzt werden. Diese sind Erweiterungen der Tautologien der Aussagenlogik.

**Lemma 4.1.21. Elementare Rechenregeln**

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

**Bemerkung 4.1.22.** *Mithilfe der obigen Tautologien kann man die sogenannte Pränexform und auch die Negationsnormalform einer Formel herstellen. Die* Pränex-form *ist dadurch gekennzeichnet, dass in der Formel zuerst alle Quantoren kommen (Quantorpräfix), und dann eine quantorenfreie Formel. Die* Negations-normalform *ist dadurch gekennzeichnet, dass alle Negationszeichen nur vor Atomen vorkommen und dass die Junktoren $\Rightarrow, \Leftrightarrow$ eliminiert sind.*

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

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

Die Elimination von Existenzquantoren ist die sogenannte Skolemisierung (Nach Thoralf Skolem).
Idee: Ersetze in $\exists x : P(x)$ das $x$, „das existiert“ durch ein Konstantensymbol $a$, d.h. $\exists x : P(x) \rightarrow P(a)$ Ersetze in $\forall x_1 \ldots x_n : \exists y : P(x_1, \ldots, x_n, y)$ das $y$ durch eine Funktion von $x_1, \ldots, x_n$, d.h. $\forall x_1 \ldots x_n : \exists y : P(x_1, \ldots, x_n, y) \rightarrow \forall x_1 \ldots x_n : P(x_1, \ldots, x_n, f(x_1, \ldots, x_n))$.
Im nächsten Theorem sei $G[x_1, \ldots, x_n, y]$ eine beliebige Formel, die die Variablensymbole $x_1, \ldots, x_n, y$ frei enthält und $G[x_1, \ldots, x_n, t]$ eine Variante von $F$, in der alle Vorkommnisse von $y$ durch $t$ ersetzt sind.

**Theorem 4.1.23. Skolemisierung**
*Eine Formel $F = \forall x_1 \ldots x_n : \exists y : G[x_1, \ldots, x_n, y]$ ist (un-)erfüllbar gdw. $F' = \forall x_1 \ldots x_n : G[x_1, \ldots, x_n, f(x_1, \ldots, x_n)]$ (un-)erfüllbar ist, wobei $f$ ein $n$-stelliges Funktionssymbol ist mit $n \geq 0$, das nicht in $G$ vorkommt.*

**Beispiel 4.1.24. Skolemisierung**

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

**Beispiel 4.1.25.** *Skolemisierung erhält i.a. nicht die Allgemeingültigkeit (Falsifizierbarkeit):*

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

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

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

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

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

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

**Theorem 4.1.26** (Allgemeinere Skolemisierung)**.** *Sei $F$ eine geschlossene Formel, $G$ eine existentiell quantifizierte Unterformel in $F$ an einer Position $p$, Weiterhin sei $G$ nur unter All-quantoren, Konjunktionen, und Disjunktionen. Die All-quantoren über $G$ binden die Variablen $x_1, \ldots, x_n$ mit $n \geq 0$. D.h. $F$ ist von der Form $F[\exists y : G'[x_1, \ldots, x_n, y]]$.*

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

**Definition 4.1.27** (Transformation in Klauselnormalform)**.** *Folgende Prozedur wandelt jede prädikatenlogische Formel in Klauselform (CNF) unter Erhaltung der Erfüllbarkeit um:*

1. *Elimination von $\Leftrightarrow$ und $\Rightarrow$: $F \Leftrightarrow G \to F \Rightarrow G \wedge G \Rightarrow F$ (Lemma 4.1.21) und*

   $F \Rightarrow G \to \neg F \vee G$

2. *Negation ganz nach innen schieben:*

$$\begin{array}{lcl} \neg\neg F & \to & F \\ \neg(F \wedge G) & \to & \neg F \vee \neg G \\ \neg(F \vee G) & \to & \neg F \wedge \neg G \\ \neg\forall x : F & \to & \exists x : \neg F \\ \neg\exists x : F & \to & \forall x : \neg F \end{array}$$

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

(*kann auch nur teilweise geschehen*)

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

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

5. *Existenzquantoren werden durch Skolemisierung eliminiert*

6. *Allquantoren nach außen schieben*

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

8. *Allquantoren vor die Klauseln schieben, anschließend umbenennen und die Quantoren löschen (alle Variablen werden als allquantifiziert angenommen).*

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

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

*oder in Mengenschreibweise:*

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

**Beispiel 4.1.28.** *Analog zur Aussagenlogik kann das CNF-Verfahren so optimiert werden, dass es statt exponentiell großen Formeln nur linear große Klauselmengen generiert.*

**Beispiel 4.1.29.**

$$\begin{array}{ll}
A1: & \mathit{Dieb}(\mathit{Anton}) \vee \mathit{Dieb}(\mathit{Ede}) \vee \mathit{Dieb}(\mathit{Karl}) \\
A2: & \mathit{Dieb}(\mathit{Anton}) \Rightarrow (\mathit{Dieb}(\mathit{Ede}) \vee \mathit{Dieb}(\mathit{Karl})) \\
A3: & \mathit{Dieb}(\mathit{Karl}) \Rightarrow (\mathit{Dieb}(\mathit{Ede}) \vee \mathit{Dieb}(\mathit{Anton})) \\
A4: & \mathit{Dieb}(\mathit{Ede}) \Rightarrow (\neg\, \mathit{Dieb}(\mathit{Anton}) \wedge \neg\, \mathit{Dieb}(\mathit{Karl})) \\
A5: & \neg\, \mathit{Dieb}(\mathit{Anton}) \vee \neg\, \mathit{Dieb}(\mathit{Karl})
\end{array}$$

*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 4.1.30.** *verschiedene Typen von Normalformen:*

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

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

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

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

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

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

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

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

## 4.2 Resolution

Ein Kalkül soll die semantische Folgerungsbeziehung durch syntaktische Manipulation nachbilden, d.h. genau dann wenn $F \models G$, soll es möglich sein, entweder $G$ aus $F$ durch syntaktische Manipulation abzuleiten ($F \vdash G$, positiver Beweis) oder $F \wedge \neg G$ durch syntaktische Manipulation zu widerlegen ($F \wedge \neg G \vdash false$, Widerlegungsbeweis). Für jeden Kalkül muss die Korrektheit gezeigt werden, d.h. wann immer $F \vdash G$, dann $F \models G$. Die Vollständigkeit, d.h. wann immer $F \models G$, dann $F \vdash G$ ist nicht notwendig. Was möglichst gelten sollte (aber bei manchen Logiken nicht möglich ist), ist die Widerlegungsvollständigkeit, d.h. wann immer $F \models G$ dann $F \wedge \neg G \vdash false$. Für $PL_1$ gibt es eine ganze Reihe unterschiedlicher Kalküle. Ein wichtiger und gut automatisierbarer ist der 1963 von John Alan Robinson entwickelte Resolutionskalkül (Robinson, 1965). Er arbeitet in erster Linie auf Klauseln.

### 4.2.1 Grundresolution: Resolution ohne Unifikation

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

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

**Definition 4.2.1.** *Resolution im Fall direkt komplementärer Resolutionsliterale.*

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

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

Beispiel 3.2.14 weiter fortgesetzt:

**Beispiel 4.2.2.** *siehe Beispiel 4.1.29*

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

*Resolutionsableitung:*

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

*Also, Ede wars.*

**Satz 4.2.3.** *Die Grund-Resolution ist korrekt:*

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

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

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

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

**Bemerkung 4.2.4.** *Im Sinne der Herleitbarkeit ist Resolution unvollständig: Nicht jede Formel, die semantisch folgt, läßt sich durch Anwenden der Resolution ableiten: Denn $P \models P \vee Q$. aber auf $P$ alleine kann man keine Resolution anwenden.*

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

**Theorem 4.2.5** (Widerlegungsvollständigkeit der Grundresolution)**.** *Jede endliche unerfüllbare Grundklauselmenge läßt sich durch Resolution widerlegen.*

### 4.2.2 Resolution im allgemeinen Fall

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

**Definition 4.2.6** (Substitution)**.** *Eine Substitution $\sigma$ ist eine Abbildung endlich vieler Variablen auf Terme. Die Erweiterung von $\sigma$ auf Terme ist definiert durch:*

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

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

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

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

**Beispiel 4.2.7.**

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

**Definition 4.2.8.** *Für eine Substitution* $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *heißt*

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

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

*Eine* Instantiierungrelation $\leq [W]$ *zwischen Substitutionen* $\sigma$ *und* $\tau$ *(für eine Menge von Variablen* $W$*) ist folgendermaßen definiert:*

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

**Beispiel 4.2.9.**

**Komposition:**

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

**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)*

**Definition 4.2.10.** *(Resolution mit Unifikation)*

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

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

**Beispiel 4.2.11.** *Dieses Beispiel (Eine Variante der Russelschen Antinomie) zeigt, dass noch eine Erweiterung der Resolution, die Faktorisierung, notwendig ist. Die Aussage ist: Der Friseur rasiert alle, die sich nicht selbst rasieren:*

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

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

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

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

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

**Definition 4.2.12. (Faktorisierung)**

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

Damit besteht der Resolutionskalkül jetzt aus Resolution und Faktorisierung.

**Definition 4.2.13.** *Der Resolutionskalkül transformiert Klauselmengen $S$ wie folgt:*

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

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

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

**Beispiel 4.2.14.** *Wir wollen die Transitivität der Teilmengenrelation mit Resolution beweisen. Wir starten mit der Definition von $\subseteq$ unter Benutzung von $\in$:*

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

*Das zu beweisende Theorem ist:*

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

*Umwandlung in Klauselform ergibt:*

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

*Resolutionswiderlegung:*

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

### 4.2.3 Unifikation

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

**Beispiel 4.2.15.** *Unifikatoren und allgemeinste Unifikatoren.*

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

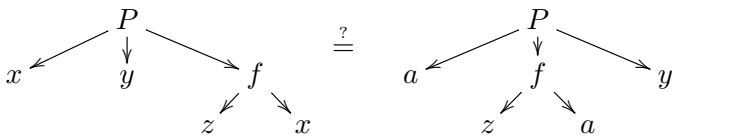
Fragen:

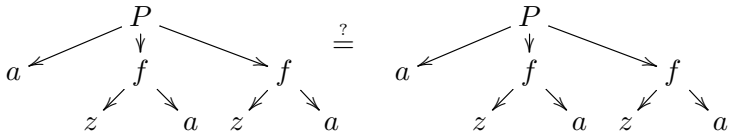
- Was heißt „allgemeinster“ Unifikator?
- Wieviele gibt es davon? ($\sigma' = \{y \mapsto x\}$ im obigen Beispiel ist offensichtlich auch einer.)
- Wie berechnet man sie?

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.

**Beispiel 4.2.16.** *Ein Beispiel zur Unifikation.*

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




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

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

**Definition 4.2.17.** *Unifikatoren und allgemeinste Unifikatoren.*

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

*Eine Substitution $\sigma$ heißt* allgemeinster Unifikator *für zwei Terme $s$ und $t$ wenn*

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

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

**Definition 4.2.18.** *Unifikationsalgorithmus $U1$:*

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

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

**Zustände:** *auf denen der Algorithmus operiert: Eine Menge $\Gamma$ von Gleichungen.*

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

*Unifikationsregeln:*

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

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

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

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

*Abbruchbedingungen:*

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

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

**Steuerung:**

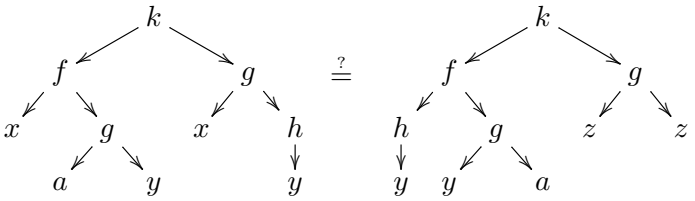
*Starte mit $\Gamma = \Gamma_0$, und transformiere $\Gamma$ solange durch (nichtdeterministische, aber nicht verzweigende) Anwendung der Regeln, bis entweder eine Abbruchbedingung erfüllt ist oder keine Regel mehr anwendbar ist. Falls eine Abbruchbedingung erfüllt ist, terminiere mit „nicht unifizierbar".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

**Beispiel 4.2.19.** *Unifikation von zwei Termen durch Anwendung der obigen Regeln:*

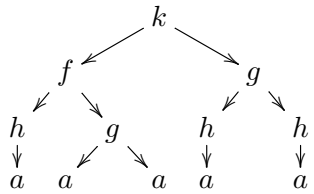
$$\begin{array}{lll}
 & \{k(f(x,g(a,y)),g(x,h(y)) \stackrel{?}{=} k(f(h(y),g(y,a)),g(z,z))\} & \\
\to & \{f(x,g(a,y)) \stackrel{?}{=} f(h(y),g(y,a)),g(x,h(y)) \stackrel{?}{=} g(z,z)\} & (\textit{Dekomposition}) \\
\to & x \stackrel{?}{=} h(y),g(a,y) \stackrel{?}{=} g(y,a),g(x,h(y)) = g(z,z) & (\textit{Dekomposition})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

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

*In der Baumdarstellung sieht es so aus:*




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




**Theorem 4.2.20.** *Der Unifikationsalgorithmus terminiert, ist korrekt und vollständig Erliefert, falls er nicht abbricht, genau einen allgemeinsten Unifikator.*

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

## 4.3 Vollständigkeit

**Theorem 4.3.1.** (*Gödel-Herbrand-Skolem Theorem*) *Zu jeder unerfüllbaren Menge $C$ von Klauseln gibt es eine endliche unerfüllbare Menge von Grundinstanzen* (*Grundklauseln*) *von $C$.*

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

## 4.4 Löschregeln: Subsumtion, Tautologie und Isoliertheit

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

[3] Das sind Klauseln mit nur einem Literal

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

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

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

**Definition 4.4.1. Isoliertes Literal**
*Sei $\mathcal{C}$ eine Klauselmenge, $D$ eine Klausel in $\mathcal{C}$ und $L$ ein Literal in $D$. $L$ heißt* isoliert, *wenn es keine Klausel $D' \neq D$ mit einem Literal $L'$ in $\mathcal{C}$ gibt, so dass $L$ und $L'$ verschiedenes Vorzeichen haben und $L$ und $L'$ unifizierbar sind.*

Die entsprechende Reduktionsregel ist:

**Definition 4.4.2. ISOL: Löschregel für isolierte Literale**
*Wenn $D$ eine Klausel aus $\mathcal{C}$ ist mit einem isolierten Literal, dann lösche die Klausel $D$ aus $\mathcal{C}$.*

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

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

**Theorem 4.4.3.** *Die Löschregel für isolierte Literale kann zum Resolutionskalkül hinzugenommen werden, ohne die Widerlegungsvollständigkeit zu verlieren.*

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

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

**Beispiel 4.4.4.** *Betrachte die Klauselmenge*

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

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

Eine weitere mögliche Redundanz ist die Subsumtion

**Definition 4.4.5.** *Seien $D$ und $E$ Klauseln. Wir sagen dass $D$ die Klausel $E$* subsumiert, *wenn es eine Substitution $\sigma$ gibt, so dass $\sigma(D) \subseteq E$*

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

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

Die entsprechende Reduktionsregel ist:

**Definition 4.4.6. SUBS: Löschregel für subsumierte Klauseln**
*Wenn $D$ und $E$ Klauseln aus $\mathcal{C}$ sind, $D$ subsumiert $E$ und $E$ hat nicht weniger Literale als $D$, dann lösche die Klausel $E$ aus $\mathcal{C}$ .*

**Beispiel 4.4.7.**

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

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

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

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

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

**Lemma 4.4.8.** *Seien $D, E$ Klauseln in der Klauselmenge $\mathcal{C}$, so dass $D$ die Klausel $E$ subsumiert. Dann wird jede Resolvente und jeder Faktor von $E$ von einer Klausel subsumiert, die ableitbar ist, ohne $E$ zu verwenden, wobei statt $E$ die Klausel $D$ oder Faktoren von $D$ verwendet werden.*

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

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

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

2. $D = \{L\} \cup D_R$, $\sigma D_R \subseteq E_R$ und $\sigma(L) = K$
   Die Resolvente von $D$ mit $F$ auf den Literalen $L, M$ und dem allgemeinsten Unifikator $\mu$ ist dann $\mu D_R \cup \mu F_R$. Sei $\tau'$ die Substitution, die auf $E$ wie $\tau\sigma$ wirkt und auf $F$ wie $\tau$. Diese Definition ist möglich durch Abänderung auf Variablen, da wir stets annehmen, dass verschiedene Klauseln variablendisjunkt sind. Da $\mu$ allgemeinst ist, gibt es eine Substitution $\lambda$ 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 $D_R = \{L_1, \ldots, L_m\} \cup D'_R$ und $\sigma L_i = \sigma L = K$. Mit einer Argumentation ähnlich zu der in Fall 1 sieht man, dass es einen Faktor $D'$ von $D$ gibt, der $L$ und

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

□

**Theorem 4.4.9.** *Der Resolutionskalkül zusammen mit der Löschung subsumierter Klauseln ist widerlegungsvollständig.*

*Beweis.* Induktion nach der Länge einer Resolutionsableitung mit Lemma 4.4.8 als Induktionsschritt und der Tatsache, dass die einzige Klausel, die die leere Klausel subsumiert, selbst nur die leere Klausel sein kann, liefert die Behauptung. □

**Definition 4.4.10.** *Sei $D$ eine Klausel. Wir sagen dass $D$ eine* Tautologie *ist, wenn $D$ in allen Interpretationen wahr ist.*

Beispiele für Tautologien sind $\{Pa, \neg Pa\}$, $\{Qa, P(f(x)), \neg P(f(x), Qb\}$ oder $\{Px, \neg Px\}$. Keine Tautologien sind $\{Px, \neg Pf(y)\}$ und $\{\neg P(x, y), P(y, x)\}$.
Ein syntaktisches Kriterium zur Erkennung von tautologischen Klauseln ist der Test, ob zwei komplementäre Literale $L, L'$ enthalten sind mit gleichen Atomen. (siehe Beispiel 4.1.13). Dieser Test ist für die ganze Klauselmenge in Zeit $O(n^3)$ durchführbar.

Die entsprechende Reduktionsregel ist:

**Definition 4.4.11. TAUT: Löschregel für tautologische Klauseln**
*Wenn $D$ eine tautologische Klausel aus der Klauselmenge $\mathcal{C}$ ist, dann lösche die Klausel $D$ aus $\mathcal{C}$.*

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

**Theorem 4.4.12.** *Die Löschregel für tautologische Klauseln ist widerlegungsvollständig.*

*Beweis.* Hierzu zeigt man, dass ein Beweis, der eine tautologische Klausel benutzt, verkürzt werden kann:

Sei $C = C_1 \vee L \vee \neg L$ eine tautologische Klausel, die im Beweis benutzt wird. Sei $D = D_1 \vee L'$ die Klausel, mit der als nächstes resolviert wird. Wir können annehmen, dass $L'$ und $\neg L$ komplementär sind mit allgemeinstem Unifikator $\sigma$. Das Resultat ist $\sigma(C_1 \vee L \vee D_1)$. Diese Resolvente wird von $D$ subsumiert. Mit Lemma 4.4.8 können wir den Resolutionsbeweis verkürzen, indem $D$ statt dieser Resolvente genommen wird. Mit Induktion können so alle Tautologien aus einer Resolutionsherleitung der leeren Klausel eliminiert werden. □

Es gilt:

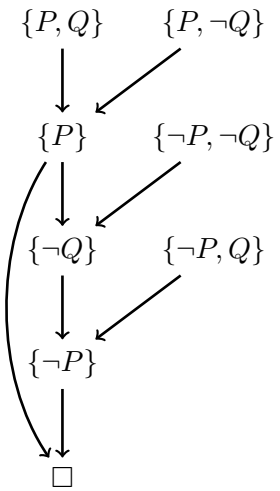
**Theorem 4.4.13.** *Der Resolutionskalkül zusammen mit Löschung subsumierter Klauseln, Löschung von Klauseln mit isolierten Literalen und Löschung von Tautologien ist widerlegungsvollständig.*

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

## 4.5 Lineare Resolution

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

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




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

### 4.5.1 Hornklauseln und SLD-Resolution

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

**Definition 4.5.1** (Hornklausel)**.** *Eine Hornklausel ist eine Klausel, die* höchstens *ein positives Literal enthält. Eine Klauselmenge, die nur aus Hornklauseln besteht, nennt man* Hornklauselmenge*.*

**Beispiel 4.5.2.** $\{\neq R(x), P(a), Q(f(y))\}$ *ist keine Hornklausel, da sie zwei positive Literale enthält. Hingegen sind die Klauseln* $\{\neg R(f(x)), \neg P(g(x,a),)Q(y)\}$ *und* $\{\neg R(g(y)), \neg P(h(b))\}$ *beide Hornklauseln, da die erste Klausel genau ein positives Literal (das Literal* $Q(y)$*) und die zweite Klausel gar keine positiven Literale enthält.*

Hornklauseln finden Verwendung in logischen Programmiersprache, wie z.B. Prolog, was wir noch behandeln werden. Hornklauseln lassen sich weiter unterteilen:

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

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

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

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

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

Als weitere Besonderheit der SLD-Resolution ist zu beachten, dass bei einem definiten Ziel als Zentralklausel und definitem Programm als Eingabeklauseln, jede lineare Resolution stets die Resolvente und *eine Seitenklausel* verwendet, d.h. es werden nie zwei Re-

solventen miteinander resolviert (beachte: bei allgemeiner linearer Resolution ist dieser Fall möglich). Der Grund liegt darin, dass alle Resolventen stets nur aus negativen Literalen bestehen, zur Resolution muss daher eine definite Klausel (mit positivem Literal) als zweite Elternklausel gewählt werden.

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

# 5

# Logisches Programmieren

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

## 5.1 Von der Resolution zum Logischen Programmieren

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

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

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

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

Die Klauseln:

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

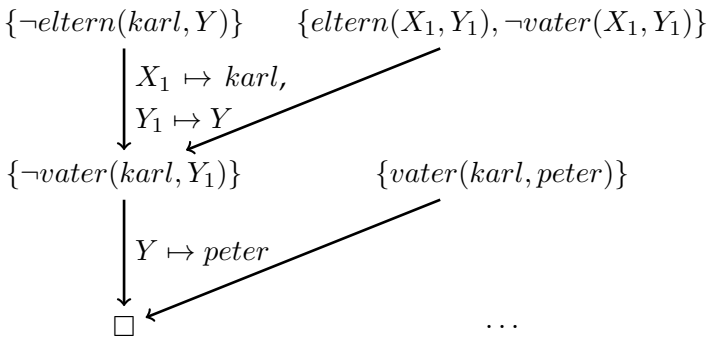
wobei $vater$ und $mutter$ zweistellige Prädikate und alle Vornamen Konstanten sind, kann man als Wissenbasis oder auch als Definition / Fakten auffassen.

Die $PL_1$-Formel

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

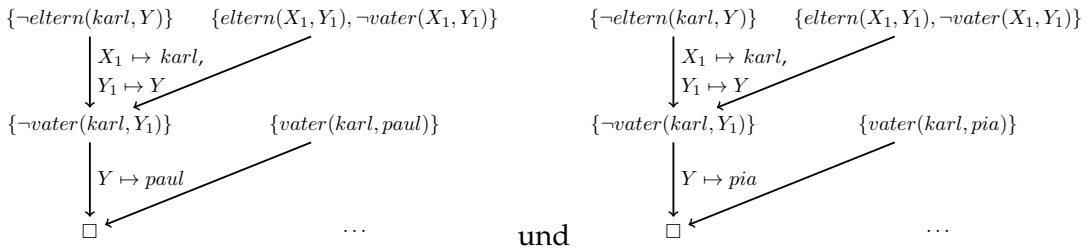
kann man als Definition der Relation $eltern$ auffassen. Beachte, dass wir sowohl $X$ als Eingabe und $Y$ als Ausgabe auffassen können, aber auch umgekehrt. In CNF ergibt dies gerade die beiden Klauseln $\{eltern(X, Y), \neg vater(X, Y)\}$ und $\{eltern(X, Y), \neg mutter(X, Y)\}$.

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




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

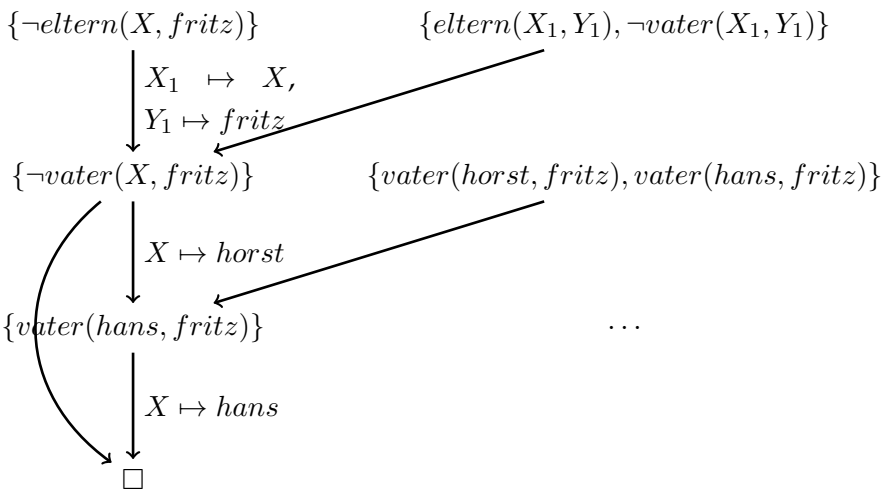
frage $\{\neg eltern(karl, Y)\}$. Wendet man alle in der Herleitung entstanden Substitutionen nacheinander auf diese Klausel an, so erhält man gerade $\{\neg eltern(karl, peter)\}$. Weil wir die leere Klausel hergeleitet haben, können wir daher nicht nur schließen, dass Karl ein Kind hat, sondern dass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




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

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

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




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

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

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

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

**Definition 5.1.1.**

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

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

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

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

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

  *geschrieben.*

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

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

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

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

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

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

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

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

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

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

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

?- eltern(karl,x).
false.

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

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

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

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

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

## 5.2 Semantik von Hornklauselprogrammen

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

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

Wir definieren einen solchen Resolutionsschritt formal:

**Definition 5.2.1.** *Sei $G = \Leftarrow A_1, \ldots, A_m, \ldots, A_k$ ein definites Ziel und $C = A \Leftarrow B_1, \ldots, B_q$ eine definite Klausel, wobei $C$ frisch umbenannt ist. Dann kann man aus $G$ und $C$ ein neues Ziel $G'$ wie folgt mit Resolution* herleiten*:*

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

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

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

**Definition 5.2.2.** *Sei $P$ ein definites Programm und $G$ ein definites Ziel. Eine* SLD-Ableitung *von $P \cup \{G\}$ ist eine Folge*

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

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

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

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

Weitere Sprechweisen:

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

**fehlgeschlagene SLD-Ableitung:** Wenn diese nicht fortsetzbar ist.

**unendliche SLD-Ableitung**

**Definition 5.2.3.** *Sei $P$ ein definites Programm und $G$ ein definites Ziel.*

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

**Theorem 5.2.4** (Soundness der SLD-Resolution)**.** *Sei $P$ ein definites Programm und $G$ ein definites Ziel. Dann ist jede berechnete Antwort $\theta$ auch korrekt. D.h. $P \models \theta(\neg G)$*

### 5.2.1 Vollständigkeit der SLD-Resolution

Wir werden verschiedene Vollständigkeitsaussagen für die SLD-Resolution formulieren.

**Theorem 5.2.5.** (*Widerlegungsvollständigkeit*)
*Sei $P$ ein definites Programm und $G$ ein definites Ziel. Wenn $P \cup \{G\}$ unerfüllbar ist, dann gibt es eine SLD-Widerlegung von $P \cup \{G\}$.*

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

**Theorem 5.2.6** (Vollständigkeit der SLD-Resolution)**.** *Sei $P$ ein definites Programm und $G$ ein definites Ziel. Zu jeder korrekten Antwort $\theta$ gibt es eine berechnete Antwort $\sigma$ für $P \cup \{G\}$ und eine Substitution $\gamma$, so dass für alle Variablen $x \in FV(G)$: $\gamma\sigma(x) = \theta(x)$.*

### 5.2.2 Strategien zur Berechnung von Antworten

**Definition 5.2.7.** *Der folgende Algorithmus berechnet alle Antworten mit einer Breitensuche:*

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

   *Sonst: mache weiter mit 1 mit dem Ziel $B$.*

Dieser Algorithmus hat zwei Verzweigungspunkte pro Resolutionsschritt:

- Die Auswahl eines Atoms
- Die Auswahl einer Klausel

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

**Satz 5.2.8.** *Vertauscht man in einer SLD-Widerlegung die Abarbeitung zweier Atome in einem Ziel, so sind die zugehörigen Substitutionen bis auf Variablenumbenennung gleich und die Widerlegung hat die gleiche Länge.*

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

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

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

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

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

**Definition 5.2.9.** *Gegeben sei ein definites Programm $P$ und ein definites Ziel $G$: Ein* SLD-Baum *für $P \cup \{G\}$ ist ein Baum der folgendes erfüllt:*

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

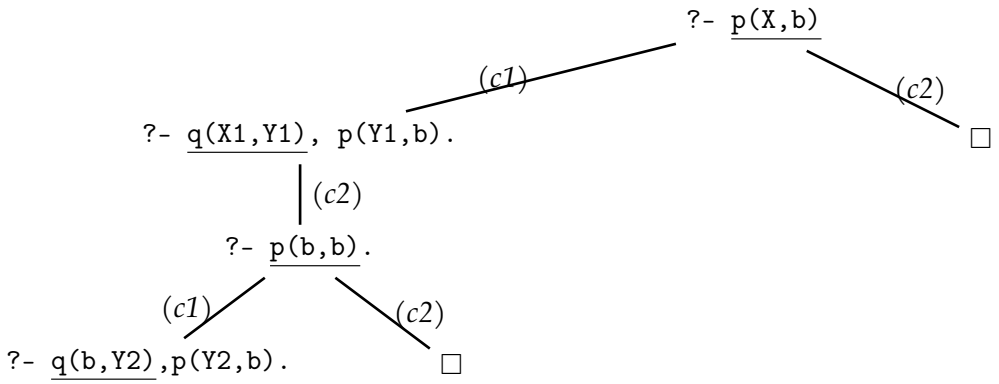
Die Äste (Kanten) entsprechen dabei SLD-Ableitungen. Erfolgreiche Widerlegungen sind Erfolgspfade, unendliche SLD-Ableitungen entsprechen unendlichen Pfaden. Fehlschläge entsprechen Pfaden zu Blättern, die mit einem nichtleeren Ziel markiert sind.

**Beispiel 5.2.10.** *Programm:*

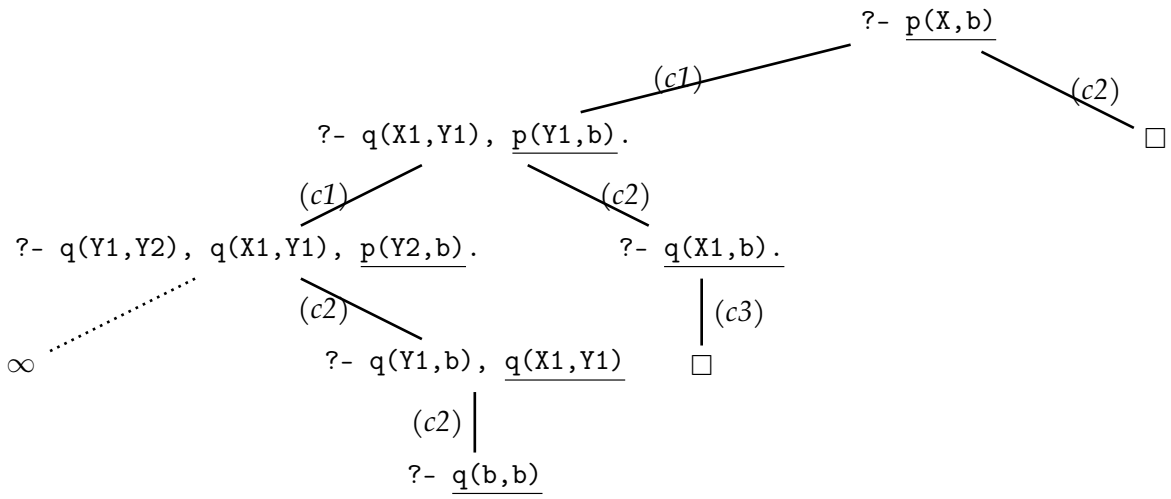
```
(c1)        p(X,Z) :-  q(X,Y), p(Y,Z).
(c2)        p(X,X).
(c3)        q(a,b).
```

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

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



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




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

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

## 5.3 Implementierung logischer Programmiersprachen: Prolog

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

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

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

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

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

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

### 5.3.1 Syntaxkonventionen von Prolog

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

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

Zusammengesetzte Terme (komplexe Terme) haben die Form

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

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

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

### 5.3.2 Beispiele zu Prologs Tiefensuche

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

**Beispiel 5.3.1.** *Nehmen wir an, wir wollen das Prädikat $vorfahr(X, Y)$ implementieren, das wahr ist, wenn $X$ ein Vorfahre von $Y$ ist (über beliebig viele Generationen hinweg). Mit dem bestehenden Prädikat $eltern$ kann man dies in Prolog implementieren als:*

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

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

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

*Dreht man die Klauseln um, so erhält man:*

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

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

*Eine bessere Variante ist:*

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

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

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

**Beispiel 5.3.2.** *Sei $p$ ein zweistelliges Prädikat, das symmetrisch und transitiv ist. Wir geben einige Fakten an und die beiden Eigenschaften:*

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

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

*Eine mögliche Abhilfe ist das Einführen eines zusätzlichen Parameters, der als Tiefenschranke verwendet wird (beachte den Abschnitt 5.3.3 zu Zahlen und arithmetischen Operationen).*

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

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

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

Gibt man z.B.

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

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

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

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

Weitere Veränderungen in Prolog gegenüber der theoretischen Fundierung sind:

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

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

### 5.3.3 Arithmetische Operationen

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

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

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

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

```
X = X.
```

Z.B.

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

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

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

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

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

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

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

Wir testen weitere Beispiele im Prolog-Interpreter:

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

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

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

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

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

Variable `is` arithmetischer Ausdruck

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

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

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

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

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

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

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

### 5.3.4 Listen und Listenprogrammierung

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

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

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

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

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

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

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

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

Wir testet das gleich:

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

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

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

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

Einige Aufrufe:

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

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

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

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

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

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

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

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

Einige Beispielaufrufe:

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

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

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

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

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

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

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

  Die neue Anfrage ist:

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

  Die nächste Unifikation ebenfalls mit zweiter Klausel

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

usw. Man sieht: das terminiert nicht.

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

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

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

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

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

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

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

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

Stellt man die Klauseln um, erhält man:

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

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

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

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

Damit kann man folgende Beispiele rechnen:

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

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

Beachte: Die Anfrage

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

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

**Beispiel 5.3.3.** *Sortieren von Listen von Zahlen:*

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

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

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

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

% ins_sortiere sortiert die Liste

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

*Einige Testaufrufe:*

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

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


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

**Beispiel 5.3.4.** *Programmierung von* `append` *zum Zusammenhängen von Listen:*

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

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

*Beispiele:*

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

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

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

**Beispiel 5.3.5.** *Mischen von zwei sortierten Listen:*

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

*Beispiele:*

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

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

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

**Beispiel 5.3.6.** *Einige weitere Beispiele für Listenfunktionen*

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

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

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

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

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

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

### 5.3.5 Differenzlisten

Wir führen an dieser Stelle eine weitere Darstellung von Listen ein, die sogenannten *Differenzlisten*. Diese werden später im Abschnitt 5.4 zum Parsen und Definite Clause Grammars eine Rolle spielen und verwendet. Diese Darstellung besitzt den Vorteil, dass das append-Prädikat wesentlich effizienter implementiert werden kann.

Betrachten wir zunächst die bisherige Implementierung von append erneut:

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

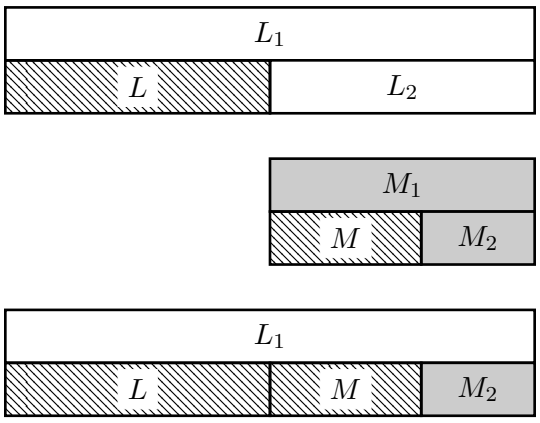
Der Nachteil dieser Implementierung liegt darin, dass die Auswertung der Ergebnisliste die gesamte erste Liste abarbeiten muss, d.h. die Laufzeit ist linear in der Länge der ersten Liste. Dies kann man sich klar machen, wenn man die Abarbeitung beispielhaft durchgeht:

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

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

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

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




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

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

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

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

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

Ein Beispielaufruf zeigt dies:

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

### 5.3.6 Der Cut-Operator: Steuerung der Suche

Prolog verfügt über den sogenannten Cut-Operator (der durch das Ausrufezeichen `!` dargestellt wird). Die Wirkung des Operators lässt sich grob dadurch beschreiben, dass die Suche an diesem Operator abgeschnitten wird (daher der Name).

Formaler: Betrachte die Programmklausel $B \Leftarrow A_1, \ldots, A_m, !, A_{m+1}, \ldots, A_n$. Die Wirkung von `!` ist die folgende: Wenn die SLD-Resolutionen, die $A_1, \ldots, A_m$ wegresolviert haben erfolgreich waren, dann wird bei Fehlschlagen der weiteren Suche in $A_{m+1}, \ldots, A_n$ kein Backtracking in $A_1, \ldots, A_m$ durchgeführt. Im SLD-Baum werden daher gerade die anderen Möglichkeiten, $A_1, \ldots, A_m$ wegzuresolvieren, nicht mehr betrachtet.

Wir betrachten ein Beispiel, um die Wirkung und die Nützlichkeit des Cut-Operators zu erläutern.

Betrachte das folgende Beispiel: Ein Mitarbeiter des Wetteramts muss je nach Windstärke $X$ eine der drei Mitteilungen $Y =$ „normal“, „windig“ oder „stürmisch“ weitergeben. Die genauen Regeln dafür sind:

1. Wenn $X < 4$, dann $Y = normal$.
2. Wenn $4 \leq 8$ und $X < 40$ dann $Y = windig$.
3. Wenn $8 \leq X$ dann $Y = stuermisch$.

Da sich der Mitarbeiter die Regeln nicht merken kann, hat er sich ein Prolog-Programm geschrieben:

```
mitteilung(X,normal)     :- X < 4.
mitteilung(X,windig)     :- 4 =< X, X < 8.
mitteilung(X,stuermisch) :- 8 =< X.
```

Angenommen die aktuelle Messung hat 3 ergeben, und der Mitarbeiter nimmt an, er muss „windig“ mitteilen, überprüft dies aber in Prolog:

```
?- mitteilung(3,Y), Y=windig.
false.
```

Wir betrachten den Ablauf der SLD-Resolution: Das erste Ziel ergibt zunächst die Instanziierung $Y \mapsto normal$. Anschließend führt die Unifikation von $windig = normal$ zum Fail. Prolog wird nun Backtracken und die beiden anderen Programmklauseln für mitteilung ausprobieren, obwohl dies eigentlich unnötig ist. Der trace in SWI-Prolog zeigt dies:

```
[trace] 3 ?- mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 4=<3 ? creep
   Fail: (8) 4=<3 ? creep
   Redo: (7) mitteilung(3, _G2797) ? creep
   Call: (8) 8=<3 ? creep
   Fail: (8) 8=<3 ? creep
   Fail: (7) mitteilung(3, _G2797) ? creep
false.
```

Mithilfe des Cut-Operators können wir diese weitere Suche abschneiden, und genauso auch für den Fall, dass die zweite Programmklausel schon instanziiert wurde (dann muss die dritte nicht mehr versucht werden). Als effizientere Version des Programms erhalten wir daher:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- 4 =< X, X < 8,!.
mitteilung(X,stuermisch) :- 8 =< X.
```

Schauen wir uns den trace erneut an, so sehen wir, dass die Suche früher abgebrochen wird:

```
[trace] 4 ?-
|    mitteilung(3,Y), Y=windig.
   Call: (7) mitteilung(3, _G2803) ? creep
   Call: (8) 3<4 ? creep
   Exit: (8) 3<4 ? creep
   Exit: (7) mitteilung(3, normal) ? creep
   Call: (7) normal=windig ? creep
   Fail: (7) normal=windig ? creep
false.
```

Die Verwendung des Cut hat hier also nur die Ausführung verändert (optimiert), aber die logische Semantik ist die selbe geblieben. Insbesondere gilt: Das Programm mit Cut verhält sich logisch genauso, wie wenn man die Cuts weglässt.

Eine weitere Optimierung im Beispiel ist, die Abfragen `4 =< X` in der zweiten Programmklauseln und `8 =< X` in der dritten Programmklausel wegzulassen, da die Suche dort erst hinkommt, wenn die entsprechenden Literale sowieso wahr sind. Das ergibt als dritte Variante:

```
mitteilung(X,normal) :- X < 4,!.
mitteilung(X,windig) :- X < 8,!.
mitteilung(X,stuermisch).
```

Beachte: Diese Implementierung sollte man nicht mit direkten Werten für die Ausgabe aufrufen. Da z.B `?- mitteilung(3,windig)` wahr wird. Im Gegensatz dazu ergibt `?- mitteilung(3,Y), Y=windig.` falsch.

Es stimmt nun auch nicht mehr: Das Programm verhält sich mit Cuts genauso wie ohne Cuts. Denn: Wenn wir die Cuts weglassen, erhalten wir

```
mitteilung(X,normal) :- X < 4.
mitteilung(X,windig) :- X < 8.
mitteilung(X,stuermisch).
```

Nun ergibt die Anfrage `?- mitteilung(3,Y), Y=windig.` als Ergebnis `true`!.

Das Beispiel demonstriert, dass das Hantieren mit Cuts zu semantischen Problemen führt, d.h. die Bedeutung der Programme kann sich ändern. Eine gute Programmierregel ist es, Cuts nur dann einzusetzen, wenn das Weglassen der Cuts die logische Bedeutung nicht ändert, da ansonsten die Verständlichkeit und Wartbarkeit der Programme schwieriger wird.

Wir betrachten einige Beispiele zur Programmierung mit Cut:

**Beispiel 5.3.7.** *Die Berechnung des Maximums zweier Zahlen kann wiefolgt in Prolog implementiert werden:*

```
max(X,Y,X) :- X >= Y.
max(X,Y,Y) :- Y < X.
```

*Mit Cut kann man dies umschreiben zu:*

```
max(X,Y,X) :- X >= Y,!.
max(X,Y,Y).
```

*Allerdings sollte man dieses Prädikat nicht direkt mit dem Ergebniswert aufrufen, denn z.B.* `?- max(3,1,1).` *ergibt wahr. Man sollte es daher in der Form* `?- max(3,1,Z), Z=3.` *verwenden.*

**Beispiel 5.3.8.** *Wir haben bereits den Enthaltentest eines Elements in einer Liste programmiert:*

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

*Mit Cut können wir dies effizienter programmieren als:*

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

*Wenn die erste Klausel erfolgreich war, muss die zweite nicht mehr betrachtet werden.*

*Beachte: Die alte Version liefert bei Anfrage* `?- member(X,[1,2,3]).` *drei verschiedene Lösungen* `X = 1 ; X = 2 ; X = 3.` *Die Version mit Cut liefert nur eine* `X = 1.`

**Beispiel 5.3.9.** *Ein if-then-else ist eigentlich nicht darstellbar in purem Prolog. Mit dem Cut-Operator kann dies jedoch programmiert werden als*

```
ifthenelse(B,P,Q) :- B,!,P.
ifthenelse(B,P,Q) :- Q.
```

**Beispiel 5.3.10.** *Das folgende Beispiel soll nochmal verdeutlichen, dass der Cut-Operator semantikverändernd ist. Betrachte das Programm*

```
p :- a,b.
p :-c.
```

*Die logische Bedeutung ist* $(a \wedge b \implies p) \wedge (c \implies p)$*. Drehen wir die Reihenfolge der Programmklauseln um, so ändert sich an der Bedeutung nichts. Betrachten wir nun die Variante mit cut:*

```
p :- a,!,b.
p :-c.
```

*Die logische Bedeutung ist nun* $(a \wedge b \implies p) \wedge (\neg a \wedge c \implies p)$*, da die zweite Klausel nur bei* $\neg a$ *betrachtet wird.*

*Umdrehen der Klauseln ergibt:*

```
p :-c.
p :- a,!,b.
```

*Die logische Bedeutung ist nun anders:* $(c \implies p) \wedge (a \wedge b \implies p)$*.*

### 5.3.7 Negation: Die Closed World-Annahme

In Prolog kann man in Programmklauseln keine Negation verwenden, da die Programmklausel $A \Leftarrow \neg B$ der Klausel $\{A, B\}$ entspräche, die keine definite Klausel mehr ist, da sie zwei positive Literale enthält. Trotzallem gibt es in Prolog einen `not`-Operator. Dessen Semantik ist jedoch nicht die logische Negation, sondern wird als Fehlschlagen der Suche interpretiert. Wir betrachten als Beispiel die Aussage: Maria mag alle Tiere außer Schlangen. Den ersten Teil der Aussage können wir durch

```
mag(maria,Y) :- tier(Y).
```

ausdrücken. Wir müssen aber noch die Schlangen ausschließen. Es gibt in Prolog das vordefinierte Literal `fail`, dass zum sofortigen Fehlschlagen der Suche führt.

Mit diesem `fail` und dem Cut-Operator können wir obige Aussage ausdrücken:

```
mag(maria,Y) :- Y=schlange,!,fail.
mag(maria,Y) :- tier(Y).
```

Zunächst wird geprüft, ob es sich um eine Schlange handelt. Ist dies der Fall, so wird kein Backtracking durchgeführt und die Suche mit fail beendet. Im Prolog-Interpreter erhält man das richtige Ergebnis (nach Hinzufügen einiger Fakten):

```
?- mag(maria,schlange).
false.

?- mag(maria,hund).
true.

?- mag(maria,katze).
true.
```

Im Grunde ist das Ergebnis `false` aber kein logisches Falsch, sondern sagt nur aus: Der Interpreter hat keine erfolgreiche SLD-Widerlegung gefunden. Genau dies liegt der sogenannten *Closed-World-Assumption* (CWA) zu Grunde: Alles was nicht aus dem Programm folgt, wird als falsch angesehen.

Das `not`-Prädikat ist entsprechend definiert als:

```
not(P) :- P,!,fail.
not(P) :- true.
```

D.h. `not(P)` liefert true, wenn Prolog keine erfolgreiche SLD-Widerlegung für `P` finden kann. Das entspricht natürlich nicht der logischen Negation. Der Programmierer muss sich deshalb darüber bewusst sein, was er tut, wenn er `not` verwendet.

Unser Beispiel kann daher auch programmiert werden als:

```
mag(maria,Y) :- tier(Y), not(Y = schlange).
```

Die Closed-World-Assumption wird in Prolog verwendet. Dies sieht man z.B. auch an folgendem Beispiel

```
rund(ball).
```

Die Anfrage `?- rund(ball)` liefert true. Hingegen liefert die Anfrage `?- rund(erde)` als Ergebnis `false`, obwohl wir eigentlich gar nicht wissen, ob die Erde rund ist, da keinerlei Aussage darüber aus dem Programm folgt. Die Closed-World-Assumption erzwingt das `false`, da angenommen wird, dass alles falsch ist, was nicht herleitbar ist. Ebenso ergibt die Anfrage `?- not(rund(erde))` das Ergebnis `true`, obwohl dies eigentlich nicht aus dem Programm folgt, sondern nur aufgrund der speziellen Semantik von `not`.

Abschließend betrachten wir einige Beispiele

**Beispiel 5.3.11.** *Unser Programm zu Verwandtschaftsbeziehungen, etwas erweitert:*

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

*Mit* `not` *können wir einfach Prädikate, für Geschwister-Beziehungen definieren:*

```
bruder(X,Y) :- mann(X),vater(Z,X),vater(Z,Y),not(X == Y), mutter(M,X),mutter(M,Y).
schwester(X,Y) :- frau(X),vater(Z,X),vater(Z,Y),not(X == Y),mutter(M,X),mutter(M,Y).
geschwister(X,Y) :-  vater(Z,X),vater(Z,Y),not(X == Y),mutter(M,X),mutter(M,Y).
```

Ein anderes, aber im Prinzip ähnliches Beispiel ist ein ungerichteter Graph:

**Beispiel 5.3.12.** *Ein ungerichteter Graph kann als Kantenrelation dargestellt werden:*

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

*Das Prädikat* `verbunden` *ist wahr, wenn zwei Knoten (über mehrere Kanten) verbunden sind:*

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

*So terminiert es allerdings nicht immer. Eine terminierende Variante ist:*

```
verbunden2(X,Y) :- verb(X,Y,[]).
ungkante(X,Y) :-  kante(X,Y).
ungkante(X,Y) :-  kante(Y,X).
verb(X,Y,_) :- ungkante(X,Y).
verb(X,Z,L) :- ungkante(X,Y), not(member(Y,L)), verb(Y,Z,[Y|L]).
```

### 5.3.8 Vergleich: Theoretische Eigenschaften und reale Implementierungen

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

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

## 5.4 Sprachverarbeitung und Parsen in Prolog

Eine häufige Anwendung der speziellen Abarbeitungsstrategie von Prolog ist die Verwendung als Implementierungssprache für Parser zu gegebenen Grammatiken, insbesondere für natürliche Sprachen (Bratko, 1990; Gal et al., 1991; Pereira & Shieber, 1987). Dies ist auch historisch gesehen die erste Anwendung von Prolog gewesen. Prolog verwendet als Suchstrategie Tiefensuche mit Backtracking. Dies kann direkt für einen rekursiv absteigenden Parser verwendet werden. Tatsächlich stehen nicht nur kontextfreie Grammatiken zur Verfügung, sondern auch erweiterte Grammatiken, z.B. für (schriftliche) Eingabe in natürlicher Sprache.

Wir betrachten zunächst kontextfreie Grammatiken in Prolog, und anschließend die erweiterten Grammatiken und die Sprachverarbeitung. Eine Kontextfreie Grammatik (CFG) besteht aus Nichtterminalen, Terminalen und Produktionen, wobei eine Produktion auf der linken Seite genau ein Nichtterminal hat und auf der rechten Seite eine Folge von Nichtterminalen und Terminalen. Z.B. erzeugt die folgende CFG mit Startsymbol $S$ alle Worte der Form $a^n b^n$:

$$\begin{aligned} S &\rightarrow ab \\ S &\rightarrow aSb \end{aligned}$$

In Prolog kann diese Grammatik mehr oder weniger direkt eingegeben werden als

```
s --> [a,b].
s --> [a], s, [b].
```

D.h. anstelle von $\rightarrow$ wird `-->` benutzt, Terminale werden in Listenklammern geschrieben und Folgen werden durch Kommata getrennt. Tatsächlich wird hier nur syntaktischer Zucker verwendet, d.h. Prolog übersetzt diese sogenannte Definite Clause Grammar (DCG) – die in diesem Fall auch eine CFG ist – in Hornklauseln. Dabei werden die Nichtterminale in Prädikate übersetzt, die jedoch zwei weitere Argumente zur Verarbeitung der Eingabe erhalten. D.h. im Beispiel wird durch die DCG das zweistellige Prädikat `s` definiert. Die Argumente stellen dabei gerade die Eingabeliste als Differenzliste dar (wobei ein Paar $(L_1, L_2)$ verwendet wird, und nicht die Darstellung $L_1 - L_2$). Wir betrachten zunächst einige Beispielaufrufe für unser Programm.

```
?- s([a,a,b,b],[]).
true.
?- s([a,b,b,a],[]).
false.
?- s([a,a,a,b,b,b],[]).
true.
?- s([a,a,a,b,b,b|X],X).
true .
?- s([a,a,a,b,b,b,c,d],[c,d]).
true.
```

Die genaue Übersetzung ist: Sei die Produktion `n --> n1,...,nm`, wobei alle `ni` zunächst Nichtterminale seien. Dann wird für jede solche Produktion die Klausel

```
n(Eingabe,Rest) :- n1(Eingabe,Rest1), n2(Rest1,Rest2),..., nm(Restm,Rest)
```

erzeugt, d.h. die Differenzliste wird quasi sequentiell aufgeteilt: Jedes der Nichtterminale `ni` nimmt sich zur Verarbeitung gerade soviele Elemente aus der Liste wie es zur Verarbeitung benötigt.

Die Übersetzung der Terminale erfolgt dementsprechend: Man kann z.B. annehmen, dass ein Prädikat `terminal` zur Verarbeitung aller Terminale hinzugefügt wird, mit der Definition

```
terminal(Terminale,Eingabe,Rest) :- append(Terminale,Rest,Eingabe).
```

Wenn $ni$ gerade die Liste `[t1,...,tk]` von Terminalen ist, dann wird $ni$ ersetzt durch `terminal([t1,...,tk],Rest`$I-1$`,Rest`$I$`)` Man kann auch auf die Verwendung des Prädikats `terminal` verzichten und die Bedingungen direkt kodieren.

Kehren wir zurück zur Beispielgrammatik und übersetzen diese nach obiger Anleitung in Prolog-Hornklauseln:

```
s(Ein,Rest) :- terminal([a,b],Ein,Rest).
s(Ein,Rest) :- terminal([a],Ein,Rest1), s(Rest1,Rest2), terminal([b],Rest2,Rest).
terminal(Terminale,Eingabe,Rest) :- append(Terminale,Rest,Eingabe).
```

Wenn man auf `terminal` verzichtet, und die entsprechenden Bedingungen direkt reinkodiert, erhält man

```
s([a,b|Ein],Ein).
s([a|Ein],Rest) :-  s(Ein,[b|Rest]).
```

DCGs können noch erweitert verwendet werden, indem man zusätzliche Argumente als Parameter an den Nichtterminalsymbolen einfügt. Diese werden dann mitunifiziert und es können dadurch weitere Bedingungen in die Grammatik hineinkodiert werden. Damit gehen DCGs über die Klasse der kontextfreien Sprachen hinaus, denn es kann z.B. eine DCG für die Sprache aller Wörter $a^n b^n c^n d^n$ angegeben ist, die bekanntlich nicht kontextfrei ist.

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

```
s            --> aa(X),bb(X),cc(X),dd(X).
aa(0)        --> [a].
aa(succ(X))  --> [a], aa(X).
bb(0)        --> [b].
bb(succ(X))  --> [b], bb(X).
cc(0)        --> [c].
cc(succ(X))  --> [c], cc(X).
dd(0)        --> [d].
dd(succ(X))  --> [d], dd(X).
```

Die zusätzlichen Parameter an den Nichtterminalen sind gerade Peanozahlen (die aus `succ` und 0 aufgebaut sind). Die erste Produktion erwingt gerade, dass die Anzahl der erzeugten a's, b's, c's und d's genau gleich sein muss.

Wir testen dies:

```
?- s([a,a,b,b,c,c,d,d],[]).
true .

?- s([a,a,a,b,b,c,c,d,d],[]).
false.

?- s(Eingabe,[]).
Eingabe = [a, b, c, d] ;
Eingabe = [a, a, b, b, c, c, d, d] ;
Eingabe = [a, a, a, b, b, b, c, c, c|...]
...
```

Die Übersetzung in reines Prolog ist gerade

```
s(Ein,Rest) :- aa(X,Ein,Rest1),bb(X,Rest1,Rest2),cc(X,Rest2,Rest3),dd(X,Rest3,Rest).
aa(0,Ein,Rest) :- terminal([a],Ein,Rest).
aa(succ(X),Ein,Rest) :- terminal([a],Ein,Rest1),  aa(X,Rest1,Rest).
bb(0,Ein,Rest) :- terminal([b],Ein,Rest).
bb(succ(X),Ein,Rest) :- terminal([b],Ein,Rest1),  bb(X,Rest1,Rest).
cc(0,Ein,Rest) :- terminal([c],Ein,Rest).
cc(succ(X),Ein,Rest) :- terminal([c],Ein,Rest1),  cc(X,Rest1,Rest).
dd(0,Ein,Rest) :- terminal([d],Ein,Rest).
dd(succ(X),Ein,Rest) :- terminal([d],Ein,Rest1),  dd(X,Rest1,Rest).
```

oder ohne Verwendung von `terminal`:

```
s(Ein,Rest) :- aa(X,Ein,Rest1),bb(X,Rest1,Rest2),cc(X,Rest2,Rest3),dd(X,Rest3,Rest).
aa(0,[a|Ein],Ein).
aa(succ(X),[a|Ein],Rest) :- aa(X,Ein,Rest).
bb(0,[b|Ein],Ein).
bb(succ(X),[b|Ein],Rest) :- bb(X,Ein,Rest).
cc(0,[c|Ein],Ein).
cc(succ(X),[c|Ein],Rest) :- cc(X,Ein,Rest).
dd(0,[d|Ein],Ein).
dd(succ(X),[d|Ein],Rest) :- dd(X,Ein,Rest).
```

Die zusätzlich erlaubten Parameter innerhalb der DCGs sind insbesondere für die Verarbeitung natürlicher Sprache nützlich, da sie bestimmte Attribute der Terminale testen können, die bei bestimmten Satzgliedern übereinstimmen müssen, z.B. Geschlecht, Fall, Zeit, usw. und die Akzeptanz einer Phrase steuern können.

Deutsch (Latein auch) z.B. bauen auf Wortendungen zum Anzeigen der Fälle, Singular/Plural Geschlecht an und erlauben somit auch eine freiere Wahl der Satzstellung. Als Beispielsprache ist Englisch einfacher, da es fast keine Wortendungen gibt, und die Reihenfolge der Worte eines Satzes fast nicht umstellbar ist.

### 5.4.1 Kontextfreie Grammatiken für Englisch

Eine Tabelle der *syntaktischen Kategorien* ist:

| | | |
|---|---|---|
| Det | Determiner (Artikel) | (the, a, some) |
| N | Noun (Nomen) | (table, computer, John) |
| V | verb | (writes, eats, having) |
| ADJ | adjectives | (big, fast) |
| ADV | adverbs | (very, slowly, yesterday) |
| AUX | auxiliaries (Hilfsverben) | (is, do, has will) |
| CON | conjunctions | (and, or) |
| PREP | prepositions | (to, on, with) |
| PRON | Pronouns (Pronomen) | (he, who, which) |

Weitere Bezeichnungen (Subkategorisierung und Komponenten):

| | | |
|---|---|---|
| S | Sentence | (Satz) |
| PN | proper noun | ("a ...." ist verboten) |
| IV | intransitive verb | hat kein Objekt |
| TV | transitives verb | hat Objekt. |

Komponenten eines Satzes (Phrasen, Sätze):

| | | |
|---|---|---|
| S | (Sentence) Satz | |
| NP | Nounphrase Nominalphrase | das dicke Buch |
| VP | Verbphrase | schreibe ein Buch |
| PP | Präpositionalphrase | mit einem Fernglas |
| ADJP | adjective phrase | größer als man erwartet |
| ADVP | adverbial phrase | (yesterday evening, gestern abend) |
| OptRel | relative clause, optional | |

Die (erste Annäherung an eine) Beschreibung der Grammatik einer natürlichen Sprache erfolgt mit *Phrasenstrukturregeln*. Diese können mit CFGs zur Konstruktion sowie zur Analyse von Sätzen bzw. Phrasen verwendet werden. Diese Grammatik ist natürlich abhängig von der beschriebenen Sprache (Deutsch, Englisch, Französisch, Russisch, usw.).

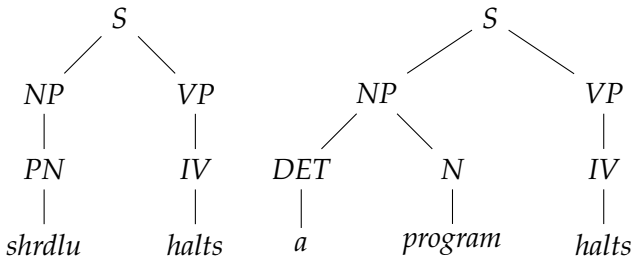
Eine kontextfreie Grammatik (CFG) zum Erzeugen einfacher englischer Sätze kann man wie folgt schreiben:

| | | |
|---|---|---|
| S | → | NP VP |
| NP | → | Det N |
| NP | → | Det N OptRel |
| NP | → | PN |
| Optrel | → | that VP |
| VP | → | TV NP |
| VP | → | IV |

Lexikoneinträge:

| | | |
|---|---|---|
| PN | → | terry |
| PN | → | shrdlu |
| Det | → | a |
| N | → | program |
| IV | → | halts |
| TV | → | writes |

**Beispiel 5.4.1.** *Herleitungsbäume für die Sätze „shrdlu halts“ und „a program halts“ sind:*




In Prolog kann man nun die Kontextfreie Grammatik als DCG eingeben, wobei wir gleich noch das Prädikat `istsatz` zur komfortableren Eingabe definieren:

```
istsatz(Ein) :-  s(Ein,[]).

s      --> np, vp.
np     --> pn.
np     --> det, n.
np     --> det, n, optrel.
optrel --> [that], vp.
vp     --> tv, np.
vp     --> iv.
% Lexikon:
pn     --> [terry].
pn     --> [shrdlu].
det    --> [a].
n      --> [program].
iv     --> [halts].
tv     --> [writes].
```

Einige Beispielaufrufe dazu sind:

```
?- istsatz([a,program,halts]).
true .
?- istsatz([shrdlu,halts]).
true .
?- istsatz([terry,writes,a,program,that,halts]).
true .
?- istsatz([terry,halts,a,program]).
false.
?- istsatz([terry,writes,a,shrdlu]).
false.
?- istsatz([terry,writes,a,program]).
true .
?- istsatz([terry,writes,shrdlu]).
true .
?- istsatz(X).
X = [terry, writes, terry] ;
X = [terry, writes, shrdlu] ;
X = [terry, writes, a, program] ;
X = [terry, writes, a, program, that, writes, terry] ;
X = [terry, writes, a, program, that, writes, shrdlu] ;
X = [terry, writes, a, program, that, writes, a, program] ;
...
```

Übersetzt man die DCG in reine Hornklauseln, so erhält man:

```
istsatz(Ein) :-  s(Ein,[]).

s(Ein,Rest)             :- np(Ein,Rest1), vp(Rest1,Rest).
np(Ein,Rest)            :- pn(Ein,Rest).
np(Ein,Rest)            :- det(Ein,Rest1), n(Rest1,Rest).
np(Ein,Rest)            :- det(Ein,Rest1),n(Rest1,Rest2), optrel(Rest2,Rest).
optrel([that|Ein],Rest) :- vp(Ein,Rest).
vp(Ein,Rest)            :- tv(Ein,Rest1), np(Rest1,Rest).
vp(Ein,Rest)            :- iv(Ein,Rest).
% Lexikon:
pn([terry|Ein],Ein).
pn([shrdlu|Ein],Ein).
det([a|Ein],Ein).
n([program|Ein],Ein).
iv([halts|Ein],Ein).
tv([writes|Ein],Ein).
```

Dieser Parser antwortet nur mit ja/nein bzw. terminiert nicht, insbesondere erhält man keinen Parsebaum.

Man teilt die Grammatik normalerweise ein in die eigentliche Grammatik und das *Lexikon*, das die Terminale beschreibt. Teilweise zählt man auch die *Präterminale* wie z.B. $N$ (Nomen) zum Lexikon.

### 5.4.2 Verwendung von Definite Clause Grammars

Wir verwenden nun Definite Clause Grammars (DCG's), d.h. wir Verwenden die Erweiterung der kontextfreien Grammatiken um Argumente (Attribute) von Nichtterminalen. Z.B. sollte der Numerus von NP und VP übereinstimmen, d.h. beides sollte entweder im Singular oder Plural sein: "sie gehen. er geht", aber nicht "er gehen".

Dies ergibt z.B. folgendes Grammatikfragment, in dem das Lexikon der Einfachheit halber in der DCG enthalten ist.

In der Syntax der DCGs sieht das folgendermaßen aus, wobei Attribute als Argumente geschrieben werden. Diese Syntax ist in den meisten Prologimplementierungen zulässig.

```
s             -->  np(Number), vp(Number).
np(Number)    -->  pn(Number).
vp(Number)    -->  tv(Number), np(_).
vp(Number)    -->  iv(Number).
pn(singular)  -->  [shrdlu].
pn(plural)    -->  [they].
iv(singular)  -->  [halts].
iv(plural)    -->  [halt].
```

Die Übersetzung in Prolog ergibt sich, durch Hinzufügen der Differenzlisten:

```
s(P0,P)            :- np(Number,P0,P1), vp(Number,P1,P).
np(Number,P0, P)   :- pn(Number,P0,P).
vp(Number,P0, P)   :- tv(Number, P0, P1), np(_, P1, P).
vp(Number,P0, P)   :- iv(Number, P0, P).
pn(singular,P0,P)  :- terminal([shrdlu], P0, P).
pn(plural, P0, P)  :- terminal([they],P0, P).
iv(singular, P0, P):- terminal([halts], P0, P).
iv(plural, P0, P)  :- terminal([halt], P0, P).
```

Die Anfrage `s([shrdlu, halts],[])` hat folgende Verarbeitung:

```
s([shrdlu, halts],[])
   np(N,[shrdlu, halts],P1), vp(N,P1,[])
   pn(N,[shrdlu, halts],P1), iv(N,P1,[])
      terminal(shrdlu,[shrdlu, halts],P1),  iv(singular,P1,[])
            %%%   (N = singular)
terminal(shrdlu,[shrdlu, halts],P1),  terminal([halts],P1,[])
            %%%  (Unifikation mit terminal ergibt P1 = [halts])
terminal([halts],[halts],[]).
Ergibt ‘‘yes‘‘.
```

Es gibt noch weitere dieser Attribute[2] z.B. Person (erste, zweite, dritte) muss übereinstimmen (ich bin; du bist; er, sie, es ist); bei NP und VP muss das Geschlecht übereinstimmen: das Haus (nicht „die Haus“), ebenso gibt es eine Übereinstimmung bei Det und N.

Im Deutschen kann man aus der Endung den Casus (teilweise) ablesen mittels morphologischer Verarbeitung. Die sogenannte morphologische Verarbeitung dient dazu, Worte zu analysieren: auf Endungen, Zerlegung zusammengesetzter Worte, .... In dieser Vorlesungen werden wir darauf nicht weiter eingehen.

Bestimmte Satzkonstruktionen erfordern Dativ, bzw. Akkusativ, ...die ebenfalls als erforderliche Attribute entweder direkt in die Grammatikregeln eingefügt werden oder als entsprechende Verbmarkierungen im Lexikon.
Auch die Zeit oder (Aktiv/Passiv) kann man hinzufügen.

Um weitere Prolog-Tests in eine DCG-Klausel einzufügen, muss man die Prolog-Literale mit geschweiften Klammern einklammern:

Ein Beispiel für eine kleine Micro-DCG des Deutschen ist, wobei folgende Eingabe alle gültigen Sätze dieser Grammatik ausgibt: `s(X,[])`. Z.B. `[ich,fahre, ein, auto]`.

[2] Attribute werden in der Computerlinguistik auch features genannt; Die zugehörigen Grammatiken Unifikationsgrammatiken

```
s     -->  np(Number,Sex,Person), vp(Number,Sex,Person).
np(Number,Sex,Person)    -->  pn(Number,Sex,Person,nom).
vp(Number,Sex,Person)    -->  tv(Number,Sex,Person,Semtv),
      npakk(akk,Semnp),
        {intersect(Semtv,Semnp,Semschnitt), not (Semschnitt  = [])}.
vp(Number,Sex,Person)    -->  iv(Number,Sex,Person).
pn(singular,no,1,nom)  -->  [ich].
pn(singular,no,2,nom)  -->  [du].
pn(singular,male,3,nom)  -->  [er].
pn(singular,female,3,nom)  -->  [sie].
pn(singular,neutrum,3,nom)  -->  [es].
pn(plural,no,1,nom)    -->  [wir].
pn(plural,no,2,nom)    -->  [ihr].
pn(plural,mfn,3,nom)    -->  [sie].
npakk(Fall,Sem)  --> det(Number,Sex,3,Fall),nom(Number,Sex,3,Fall,Sem).
iv(singular,male,3)  -->  [geht].
iv(singular,female,3)  -->  [geht].
iv(singular,neutrum,3)  -->  [geht].
iv(singular,no,1)  -->  [gehe].
iv(singular,no,2)  -->  [gehst].
iv(plural,mfn,3)    -->  [gehen].
iv(plural,no,1)    -->  [gehen].
iv(plural,no,2)    -->  [geht].
tv(singular,male,3,[transport])  -->  [f\"ahrt].
tv(singular,female,3,[transport])  -->  [f\"ahrt].
tv(singular,neutrum,3,[transport])  -->  [f\"ahrt].
tv(singular,no,1,[transport])  -->  [fahre].
tv(singular,no,2,[transport])  -->  [f\"ahrst].
tv(plural,mfn,3,[transport])    -->  [fahren].
tv(plural,no,1,[transport])    -->  [fahren].
tv(plural,no,2,[transport])    -->  [fahrt].
tv(singular,male,3,[information])  -->  [liest].
tv(singular,female,3,[information])  -->  [liest].
tv(singular,neutrum,3,[information])  -->  [liest].
tv(singular,no,1,[information])  -->  [lese].
tv(singular,no,2,[information])  -->  [liest].
tv(plural,mfn,3,[information])    -->  [lesen].
tv(plural,no,1,[information])    -->  [lesen].
tv(plural,no,2,[information])    -->  [lest].

det(singular,neutrum,3,akk) --> [ein].
det(singular,neutrum,3,akk) --> [kein].
det(plural,neutrum,3,akk) --> [zwei].
det(singular,male,3,akk) --> [einen].
det(singular,male,3,akk) --> [keinen].
det(plural,male,3,akk) --> [zwei].
det(singular,female,3,akk) --> [eine].
det(singular,female,3,akk) --> [keine].
det(plural,female,3,akk) --> [zwei].
nom(singular,neutrum,3,akk,[transport]) --> [auto].
nom(plural,neutrum,3,akk,[transport]) --> [autos].
nom(singular,male,3,akk,[transport]) --> [bus].
nom(plural,male,3,akk,[transport]) --> [busse].
nom(singular,female,3,akk,[information]) --> [zeitung].
nom(plural,female,3,akk,[information]) --> [zeitungen].
```

#### 5.4.2.1 Lexikon

Aus heutiger Sicht ist das Lexikon nicht nur eine Sammlung von Worten, sondern die *zentrale Struktur* der linguistischen Verarbeitung. Man kann sehr viel bereits im Lexikon kodieren und dann mit einfachen weiteren Regeln auskommen. D.h. dass z.B.viele Eigenschaften eines Wortes im Lexikoneintrag stecken, auch solche Angaben, die besagen in welchen Satzkonstruktionen bestimmte Verben erlaubt sind. Z.B. Passiv-verbot: („ich werde gelaufen"). z.B. dass ein Verb transitiv bzw intransitiv ist oder in beiden Versionen benutzt werden kann. Die Kontextinformation, die ein bestimmtes Wort benötigt, kann im Lexikon kodiert sein. Auch inhaltliche Informationen (Semantik) können dort enthalten sein.

**Beispiel 5.4.2.**

```
take: verb
 verbtype = transitive
 subject: role  = agent, semfeat= human
 object:  role  = instrument, semfeat=vehicle
 prep-obj: prep  = to, role=goal
 prep-obj: prep  = from, role=source, ...
```

#### 5.4.2.2 Berechnung von Parse-Bäumen

Man kann den Nichtterminalen und Terminalen ein Argument mitgeben, das den Parsebaum mitberechnet. (Dies geht auch in einer attributierten Grammatik.)

```
s         --->  np vp
np        --->  pn
optrel    --->  that vp
vp        --->  iv
```

Die Implementierung kann dafür z.B. jedes Prädikat um ein Argument erweitern, das den (berechneten) Syntaxbaum aufnimmt.

```
s(sent(TR_np, TR_vp), Ein, Rest) :- np(TR_np,...), vp(TR_vp,...).
np(nphrase(TR_pn),...) :- PN(TR_pn,...).
optrel(relativ_satz(TR_iv),...) :- iv(TR_iv,...)
.....
```

Hier sollte z.B. der Satz "Shrdlu halts" als Ausgabe den Syntaxbaum
`sent(nphrase(propernounn(shrdlu)), verbphrase(intransverb(halts)))`
erzeugen.

#### 5.4.2.3 DCG's erweitert um Prologaufrufe

Man kann in der Implementierung von DCGs Prolog-aufrufe einbetten, die bei der Übersetzung in Prolog mitübernommen werden. Syntaktisch macht man das durch Klammerung mit geschweiften Klammern:

```
s --> np,vp,{teste(X)}.
```

`s,np,vp` werden um die Ein- und Ausgabelisten ergänzt, das Literal in der Klammer wird übernommen.

#### 5.4.2.4 DCG's als formales System

DCG's (ohne eingebettete Literale) sind als formales System aufzufassen, das analog zu CFGs eine formale Sprache definiert. Zusätzlich zu CFGs sind an den Nichtterminalen Argumente erlaubt, die Variablen und Konstanten sein können und auf Gleichheit getestet werden dürfen.

Die formale Sprache die zu einer DCG gehört, entspricht genau den erfolgreichen Parses nach der Übersetzung in definite Klauseln, wenn man SLD-Resolution als operationale Semantik nimmt.

Allerdings ist das nicht dasselbe wie die Gleichsetzung mit der Prologimplementierung derselben, da nach der Übersetzung das Parsen als rekursiv absteigend festgelegt ist.

DCGs kann man zu attributierten Grammatiken dadurch abgrenzen, dass die letzteren kontextfreie Grammatiken sind, die in den Regeln Berechnungsalgorithmen für eine festgelegte Menge von Attributen enthalten, aber aufgrund der Berechnungen keine Eingabe ablehnen können.

Eine theoretische Klassifizierung zwischen DCGs und CFG bzgl der erkannten formalen Sprachen ist: Offenbar sind DCGs eine echte Erweiterung der CFGs.

# 6

# Qualitatives zeitliches Schließen

Um Repräsentation von Zeit und zeitlichen Zusammenhängen kommt man nicht herum. Es gibt viele Logiken hierfür, beispielsweise können Varianten der Modallogik Zeitpunkte als Zustände repräsentieren und erlauben Schlussfolgerungen aus gegebenen Aussagen. Wir betrachten in diesem Kapitel die Zeitlogik von James F. Allen (Allen, 1983), die auch *Allensche Intervall-Logik* genannt wird. In dieser Logik werden nicht konkrete Zeitpunkte oder Dauern repräsentiert, sondern es werden Aussagen über die *relative* Lage von Zeitintervallen getroffen. Dabei sind die Intervalle als Variablen (Namen) gegeben und es gibt Operatoren, um die relative Lage der Intervalle auszudrücken. Mithilfe üblicher aussgagenlogischer Junktoren können solche atomaren Aussagen zu komplexen Formeln zusammengesetzt werden. Nicht repräsentiert sind die genaue Dauer eines Intervalls oder dessen absolute zeitliche Lage. Sinnvoll ist der Einsatz der Allenschen Logik, wenn man nur die Information hat, wie verschiedene Aktionen, die eine gewisse Zeit dauern, zeitlich zueinander liegen können, und man daraus Schlussfolgerungen herleiten will. Sind die Intervalle genau bekannt, dann braucht man diese Logik nicht.

## 6.1 Allens Zeitintervall-Logik

Wir betrachten zunächst ein Beispiel, welches in Allenscher Intervall-Logik darstellbar ist.

**Beispiel 6.1.1.** *Wir nehmen ein Teil eines Rezeptes zum Apfelkuchenbacken.*

*A1:* *Hefeteig zubereiten*
*A2:* *Hefeteig gehen lassen*
*A3:* *Äpfel schälen und in Scheiben schneiden*
*A4:* *Blech einfetten*
*A5:* *Teig auf das Blech*
*A6:* *Äpfel auf den Kuchen setzen.*
…
*A10:* *Backofen heizt*
*A11:* *Kuchen im Backofen backen*

*Da wir annehmen, dass nur eine Person in der Küche ist, können wir folgende Beziehungen angeben:*

*A2 folgt direkt auf A1*
*A3 ist später als A1*
*A3 und A4 sind nicht gleichzeitig*
*A1 und A4 sind nicht gleichzeitig*
*A11 ist während A10,*
*A11 und A10 enden gleichzeitig*
*A11 ist nach A3*
*A11 ist nach A4*

*Hier kann man z.B. mit Transitivität schließen, dass A11 nach A1 stattfindet. Man kann auch ohne Widerspruch hinzufügen, dass A3 nach A4 stattfindet (d.h. man kann sequentialisieren).*

*Das hat Ähnlichkeiten zu Job Shop Scheduling Problemen (JSSP), bei denen aber feste Zeitdauern pro Aktion und Ausschlusskriterien mittels benutzter Ressourcen festgelegt werden. Auch Abhängigkeiten von Aktionen kann man in JSSPs formulieren.*

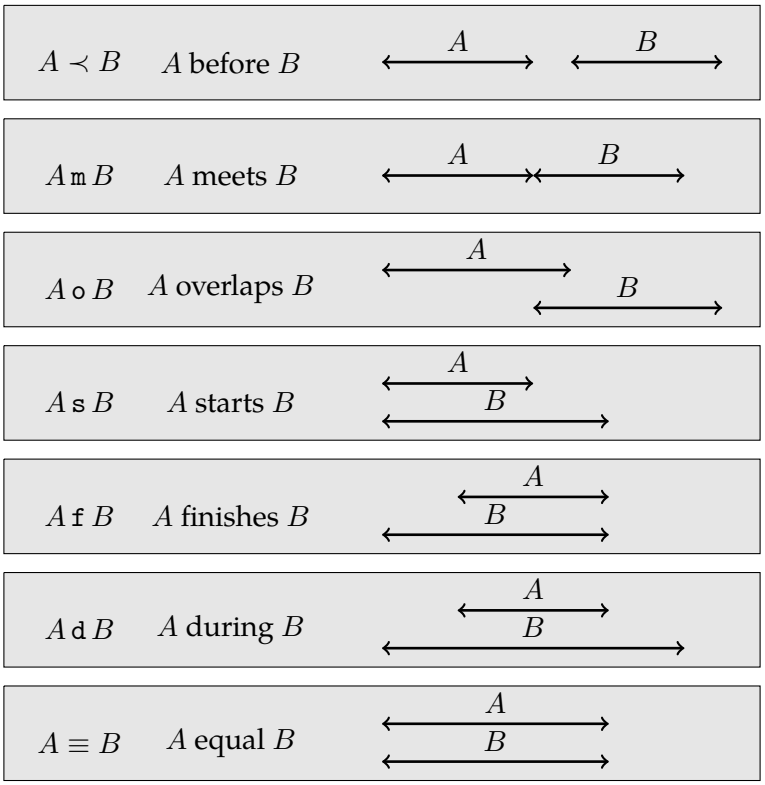
Der Allensche Intervall-Kalkül baut auf Zeitintervallen und binären Relationen zwischen den Intervallen auf, die deren zeitliche Lage beschreiben.

Die Interpretation der Intervalle kann man sich in den reellen Zahlen $\mathbb{R}$ vorstellen, aber in der Repräsentation der Allen-Logik kann man keine exakten Zeiten angeben und auch keine Zeitdauern formulieren.

Die Basis des Kalküls ist die Untersuchung der möglichen relativen Lagen zweier Intervalle $A = [AA, AZ]$, $B = [BA, BZ]$, wobei die Anfangs- und Enpunkte $AA, AZ, BA, BZ$ reelle Zahlen sind und die Intervalle positive Länge haben (d.h. $AA > BZ$ und $BZ > BA$). Eine vollständige und disjunkte Liste der Möglichkeiten ist in folgender Tabelle enthalten, wobei man noch die Fälle hinzunehmen muss, in denen $A, B$ vertauscht sind.

| *Bedingung* | *Abkürzung* | *Bezeichnung* |
|---|---|---|
| $AZ < BA$ | $\prec$ | $A$ before $B$ |
| $AZ = BA$ | `m` | $A$ meets B |
| $AA < BA < AZ < BZ$ | `o` | $A$ overlaps $B$ |
| $AA = BA < AZ < BZ$ | `s` | $A$ starts $B$ |
| $BA < AA < AZ = BZ$ | `f` | $A$ finishes $B$ |
| $BA < AA < AZ < BZ$ | `d` | $A$ during $B$ |
| $BA = AA, AZ = BZ$ | $\equiv$ | $A$ equal $B$ |

Als Bild kann man die Allenschen Relationen folgendermaßen illustrieren:



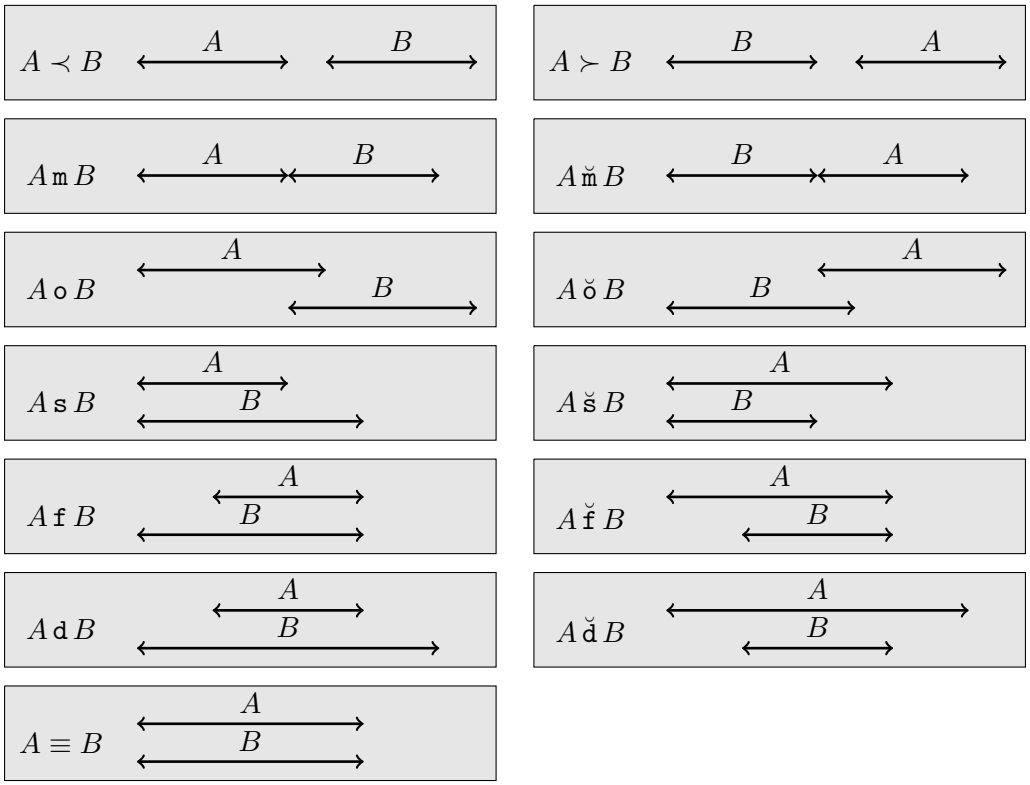

Nimmt man die (relationalen) Inversen dazu, dann hat man die 13 Allenschen Relationen zwischen Zeitintervallen. Die Inversen bezeichnet man mit $\breve{\mathtt{m}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}, \breve{\mathtt{f}}, \breve{\mathtt{d}}$, im Fall von $\prec$ mit $\succ$ und die relationale Inverse von $\equiv$ ist $\equiv$ selbst, da diese Relation symmetrisch ist.

**Definition 6.1.2** (Allensche Basisrelationen)**.** *Die 13 Allenschen Basis-Relationen sind:*

$$\{\equiv, \prec, \mathtt{m}, \mathtt{o}, \mathtt{s}, \mathtt{d}, \mathtt{f}, \succ, \breve{\mathtt{m}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}, \breve{\mathtt{d}}, \breve{\mathtt{f}}\}.$$

*Wir bezeichnen diese Menge mit $\mathcal{R}$.*

Wir stellen zur Verdeutlichung nochmal alle 13 Relationen als Bild dar:




Durch Vergleichen aller Paare von Basis-Relationen sieht man leicht, dass die Relationen alle verschieden sind:

**Satz 6.1.3.** *Die Allenschen Basis-Relationen sind paarweise disjunkt, d.h.*

$$A\, r_1\, B \wedge A\, r_2\, B \implies r_1 = r_2.$$

Formeln der Allenschen Zeitlogik werden durch Variablen für die Intervalle, verknüpft durch die Basisrelationen und durch aussagenlogische Junktoren gebildet. Wir definieren die Menge der Formeln formal:

**Definition 6.1.4** (Syntax der Allen-Formeln)**.** *Die Allen-Zeitlogik kann folgende* Allen-Formeln *bilden:*

- atomare Aussagen $A\ r\ B$, *wobei* $A, B$ *Intervallnamen sind und* $r$ *eine der Allenschen Basis-Relationen ist* (*d.h.* $r \in \mathcal{R}$)
- *aussagenlogische Kombinationen von atomaren Aussagen.*

Die Semantik dazu bildet Intervallnamen auf nichtleere, reelle Intervalle ab. Formeln sind wahr, wenn sie unabhängig von der exakten Zuordnung immer wahr sind. D.h. wenn für alle Abbildungen von Intervallnamen auf nichtleere, reelle Intervalle die Formel stets wahr ist.

**Definition 6.1.5** (Semantik der Allen-Formeln)**.** *Eine* Interpretation *$I$ bildet Intervallnamen auf nicht leere Intervalle $[a, b]$ ab, wobei $a, b \in \mathbb{R}$ und $a < b$. Die Erweiterung der Interpretation auf Allensche Formeln berechnet einen Wahrheitswert (0 oder 1) und ist wie folgt definiert:*

- *Wenn $A\ r\ B$ ein atomare Aussage ist und sei $I(A) = [AA, AZ]$ und $I(B) = [BA, BZ]$. Dann gilt:*
  - $r = \prec$: $I(A \prec B) = 1$, *gdw.* $AZ < BA$
  - $r = \mathtt{m}$: $I(A\ \mathtt{m}\ B) = 1$, *gdw.* $AZ = BA$
  - $r = \mathtt{o}$: $I(A\ \mathtt{o}\ B) = 1$, *gdw.* $AA < BA$, $BA < AZ$ *und* $AZ < BZ$
  - $r = \mathtt{s}$: $I(A\ \mathtt{s}\ B) = 1$, *gdw.* $AA = BA$ *und* $AZ < BZ$
  - $r = \mathtt{f}$: $I(A\ \mathtt{f}\ B) = 1$, *gdw.* $AA > BA$ *und* $AZ = BZ$
  - $r = \mathtt{d}$: $I(A\ \mathtt{d}\ B) = 1$, *gdw.* $AA > BA$ *und* $AZ < BZ$
  - $r = \equiv$: $I(A \equiv B) = 1$, *gdw.* $AA = BA$ *und* $AZ = BZ$
  - $r = \breve{r_0}$: $I(A\ \breve{r_0}\ B) = 1$, *gdw.* $I(B\ r_0\ A) = 1$
- *Für komplexe Formeln gilt wie üblich:*
  - $I(F \wedge G) = 1$ *gdw.* $I(F) = 1$ *und* $I(G) = 1$
  - $I(F \vee G) = 1$ *gdw.* $I(F) = 1$ *oder* $I(G) = 1$.
  - $I(\neg F) = 1$ *gdw.* $I(F) = 0$
  - $I(F \iff G) = 1$ *gdw.* $I(F) = I(G)$
  - $I(F \Rightarrow G) = 1$ *gdw.* $I(F) = 0$ *oder* $I(G) = 1$

*Eine Interpretation ist ein* Modell *für eine Allen-Formel $F$, gdw. $I(F) = 1$ gilt. Eine Allensche Formel $F$ ist:*

- *eine* Tautologie, *wenn jede Interpretation ein Modell für $F$ ist.*
- *ein* Widerspruch (*inkonsistent*), *wenn es kein Modell für $F$ gibt.*
- erfüllbar, *wenn es mindestens ein Modell für $F$ gibt.*

*Eine Allensche Formel $F$* folgt semantisch *aus der Allen-Formel $G$, geschrieben als $G \models F$, genau denn wenn alle Modelle für $G$ auch Modelle für $F$ sind (d.h. falls $I(G) = 1$, dann gilt auch $I(F) = 1$). Zwei Allensche Formeln $F, G$ sind* äquivalent, *gdw. $F \models G$ und $G \models F$ gilt.*

## 6.2 Darstellung Allenscher Formeln als Allensche Constraints

### 6.2.1 Abkürzende Schreibweise

Will man mehrere mögliche Lagebeziehungen (also vages Wissen) zwischen 2 Intervallen ausdrücken, so kann man dies disjunktiv machen. z.B.

$$A \prec B \vee A \,\mathtt{s}\, B \vee A \,\mathtt{f}\, B$$

Dies bedeutet: Aktion $A$ ist früher als $B$, oder $A, B$ starten gleichzeitig oder enden gleichzeitig, wobei in den letzten beiden Fällen $A$ kürzer als $B$ ist. Diese Disjunktion stellen wir verkürzt als $A \{\prec, \mathtt{s}, \mathtt{f}\} B$ dar .

Wir verwenden diese Schreibweise ab jetzt in der folgenden Form:

$$A \; S \; B, \text{ wobei } S \subseteq \mathcal{R}$$

und nennen eine solche Formel ein *atomares Allen-Constraint*. Damit kann man alle Disjunktionen von Einzel-Relationen zwischen zwei Intervallen $A, B$ darstellen.

Dies ergibt eine Anzahl von $2^{13}$ verschiedenen (unpräzisen) Relationsbeziehungen zwischen Zeitintervallen, wobei man diese Relationen einfach durch eine höchstens 13-elementige Disjunktion bzw. deren Abkürzung darstellen kann. Hier ist auch $A \emptyset B$ dabei, die unmögliche Relation (definiert als $I(A \emptyset B) = 0$ für jede Interpretation $I$), und $A \mathcal{R} B$, die Relation, die alles erlaubt.

### 6.2.2 Allensche Constraints

Wir zeigen in diesem Abschnitt, dass man Allen-Formeln viel einfacher darstellen kann: Ein *Allensches Constraint* ist eine Konjunktion von atomaren Allen-Constraints, d.h. eine Formel der Form $A_1 S_1 A_2 \wedge \ldots \wedge A_{n-1} S_n A_n$ wobei $S_i \subseteq \mathcal{R}$ und die $A_i$ nicht notwendigerweise verschieden sind. Jede Allensche Formel kann durch eine Disjunktion von Allenschen Constraints dargestellt werden. Hierfür genügt es zu zeigen, dass einige Vereinfachungen möglich sind, die diese „Normalform“ herstellen können. Im Wesentlichen reichen hierfür aussagenlogische Umformungen aus, jedoch müssen alle Negationen eliminiert werden.

**Definition 6.2.1** (Vereinfachungsregeln für Allensche Formeln)**.** *Die Vereinfachungsregeln für Allensche Formeln sind:*

- *Ein atomare Aussage der Form $A \; r \; A$ kann man immer vereinfachen zu $0, 1$:*
  - $A \; r \; A \; \rightarrow \; 0$, *wenn* $r \neq \equiv$ *und*
  - $A \; \equiv \; A \; \rightarrow \; 1$.
- *Negationszeichen kann man nach innen schieben.*
- *Eine Formel $\neg(A \; R \; B)$ kann man zu $A \; (\mathcal{R} \setminus R) \; B$ umformen.*
- *Unterformeln der Form $A \; R_1 \; B \wedge A \; R_2 \; B$ kann man durch $A \; (R_1 \cap R_2) \; B$ ersetzen.*
- *Unterformeln der Form $A \; R_1 \; B \vee A \; R_2 \; B$ kann man durch $A \; (R_1 \cup R_2) \; B$ ersetzen.*
- *atomare Allen-Constraints der Form $A \emptyset B$ kann man durch $0$ ersetzen.*

- *atomare Allen-Constraints der Form $A\,\mathcal{R}B$ kann man durch $1$ ersetzen.*
- *Alle aussagenlogischen Umformungen sind erlaubt.*

Durch Anwenden dieser Vereinfachungen kann man jede Allensche Formel solange transformieren, bis man eine Disjunktion von Konjunktionen von atomaren Constraints hat. Z.B. $A\ \{\prec, \mathtt{s}, \mathtt{o}\}\ B \wedge A\ \{\succ, \mathtt{s}, \mathtt{o}\}\ B$ entspricht $A\ \{\mathtt{s}, \mathtt{o}\}\ B$. Das Ergebnis ist eine i.A. kleinere Formel, die eine Disjunktion von Konjunktionen von atomaren Allen-Constraints ist. Eine Konjunktion von atomaren Allen-Constraints nennt man *Allen-Constraint*, und die Disjunktion von Allen-Constraints ein *disjunktives Allen-Constraint*.

Es gilt:

**Theorem 6.2.2.** *Jede Vereinfachungsregel für Allensche Formeln erhält die Äquivalenz, d.h. wenn $F \to F'$, dann sind $F$ und $F'$ äquivalente Formeln.*

*Beweis.* Dies lässt sich durch Verwendung der Semantik verifizieren. Für Vereinfachungen wie $\neg(A\ R\ B) \to A\ (\mathcal{R} \setminus R)\ B$ muss man im Wesentlichen zeigen, dass $\mathcal{R}$ alle möglichen Lagen von zwei Intervallen auf der reellen Achse abdeckt, und dass die Allenschen Basisrelationen alle disjunkt sind. □

## 6.3 Der Allensche Kalkül

Der Allensche Kalkül definiert Regeln, um aus gegebenen Allenschen Constraints weitere Beziehungen zwischen Intervallen zu folgern. Die wesentlichen Regeln des Allenschen Kalküls betreffen dabei Beziehungen zwischen drei Intervallen. Man kann beispielsweise aus $A \prec B \wedge B \prec C$ mittels Transitivität von $\prec$ die neue Beziehung $A \prec C$ schließen. Aus $A \prec B \wedge C \prec B$ kann man dagegen nichts neues über die relative Lage von $A$ zu $C$ schließen: Alles (im Sinne der Allenschen Relationen) ist noch möglich.

Da die Menge der Kombinationen endlich ist, kann man sich alle möglichen neuen Relationen als Verknüpfungen $r_1 \circ r_2$ bereits erstellen und damit weitere Relationen zwischen Intervallen herleiten. D.h. wir definieren $r_1 \circ r_2 \subseteq \mathcal{R}$ gerade als minimale Menge von Basisrelationen mit: $A\,r_1\,B \wedge B\,r_2\,C \models A\,(r_1 \circ r_2)\,C$. Wir verallgemeinern dies auch für disjunktive Mengen von Basirelationen: Seien $R_1, R_2 \subseteq \mathcal{R}$, dann ist $R_1 \circ R_2 \subseteq \mathcal{R}$ gerade die minimale Menge von Basisrelationen für die gilt: $A\ R_1\ B \wedge B\ R_2\ C \models A\ (R_1 \circ R_2)\ C$.

Beachte, dass für Basisrelationen $r_1, r_2$ die Komposition $r_1 \circ r_2$ nicht notwendigerweise eine Basisrelation ist, z.B. ist $\prec \circ \succ = \mathcal{R}$, da man aus $A \prec B \wedge B \succ C$ für die Beziehung zwischen $A$ und $C$ alles folgen kann (also $A\,\mathcal{R}\,C$).

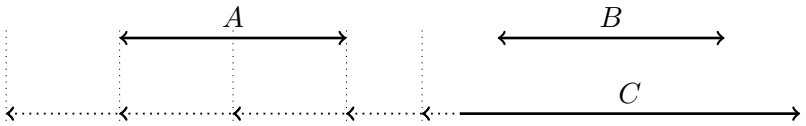
Die Werte für $r_1 \circ r_2$ kann man vor dem Start des Kalküls per Hand ausrechnen und in einer $13 \times 13$-Matrix abspeichern. Wir geben die Matrix als $12 \times 12$ (ohne die Einträge zu $\equiv$, denn es gilt stets $r \circ \equiv = \equiv \circ r = r$) in Abbildung 6.1 an. Dabei bedeuten die Einträge, die mit $\mathcal{R} \setminus$ beginnen, das Komplement der nachfolgenden Relationen.

Beispiele für Einträge in der $13 \times 13$-Matrix sind:

| | $\prec$ | $\succ$ | d | $\breve{\text{d}}$ | o | $\breve{\text{o}}$ | m | $\breve{\text{m}}$ | s | $\breve{\text{s}}$ | f | $\breve{\text{f}}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\prec$ | $\prec$ | $\mathcal{R}$ | $\prec$ o m d s | $\prec$ | $\prec$ | $\prec$ o m d s | $\prec$ | $\prec$ o m d s | $\prec$ | $\prec$ | $\prec$ o m d s | $\prec$ |
| $\succ$ | $\mathcal{R}$ | $\succ$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | $\succ$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | $\succ$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | $\succ$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | $\succ$ | $\succ$ | $\succ$ |
| d | $\prec$ | $\succ$ | d | $\mathcal{R}$ | $\prec$ o m d s | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | $\prec$ | $\succ$ | d | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ d f | d | $\prec$ o m d s |
| $\breve{\text{d}}$ | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | $\mathcal{R} \setminus \prec \succ$ m $\breve{\text{m}}$ | $\breve{\text{d}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\breve{\text{d}}$ | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | $\breve{\text{d}}$ |
| o | $\prec$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o d s | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\prec$ o m | $\mathcal{R} \setminus \prec \succ$ m $\breve{\text{m}}$ | $\prec$ | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o | $\breve{\text{d}}$ $\breve{\text{f}}$ o | d s o | $\prec$ o m |
| $\breve{\text{o}}$ | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\succ$ | $\breve{\text{o}}$ d f | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | $\mathcal{R} \setminus \prec \succ$ m $\breve{\text{m}}$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\succ$ | $\breve{\text{o}}$ d f | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ | $\breve{\text{o}}$ | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ |
| m | $\prec$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o d s | $\prec$ | $\prec$ | o d s | $\prec$ | $\equiv$ f $\breve{\text{f}}$ | m | m | d s o | $\prec$ |
| $\breve{\text{m}}$ | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\succ$ | $\breve{\text{o}}$ d f | $\succ$ | $\breve{\text{o}}$ d f | $\succ$ | $\equiv$ s $\breve{\text{s}}$ | $\succ$ | d f $\breve{\text{o}}$ | $\succ$ | $\breve{\text{m}}$ | $\breve{\text{m}}$ |
| s | $\prec$ | $\succ$ | d | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\prec$ o m | $\breve{\text{o}}$ d f | $\prec$ | $\breve{\text{m}}$ | s | $\equiv$ s $\breve{\text{s}}$ | d | $\prec$ o m |
| $\breve{\text{s}}$ | $\prec$ o m $\breve{\text{d}}$ $\breve{\text{f}}$ | $\succ$ | $\breve{\text{o}}$ d f | $\breve{\text{d}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\breve{\text{o}}$ | o $\breve{\text{d}}$ $\breve{\text{f}}$ | $\breve{\text{m}}$ | $\equiv$ s $\breve{\text{s}}$ | $\breve{\text{s}}$ | $\breve{\text{o}}$ | $\breve{\text{d}}$ |
| f | $\prec$ | $\succ$ | d | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o d s | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ | m | $\succ$ | d | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ | f | $\equiv$ f $\breve{\text{f}}$ |
| $\breve{\text{f}}$ | $\prec$ | $\succ$ $\breve{\text{o}}$ $\breve{\text{m}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o d s | $\breve{\text{d}}$ | o | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | m | $\breve{\text{o}}$ $\breve{\text{d}}$ $\breve{\text{s}}$ | o | $\breve{\text{d}}$ | $\equiv$ f $\breve{\text{f}}$ | $\breve{\text{f}}$ |

*Abbildung 6.1: Die Kompositionsmatrix der Allenschen Relationen*

**Beispiel 6.3.1.** *Wenn $A \prec B \wedge B \,\mathtt{d}\, C$, dann kann man sich alle Möglichkeiten für $A\ (\prec \circ \mathtt{d})\ C$ an folgendem Bild verdeutlichen, indem man alle möglichen linken Anfänge von $C$ betrachtet:*




*Man sieht, dass es genau die Möglichkeiten $\prec, \mathtt{o}, \mathtt{m}, \mathtt{s}, \mathtt{d}$ zwischen $A$ und $C$ gibt, d.h. $A\ \{\prec, \mathtt{o}, \mathtt{m}, \mathtt{s}, \mathtt{d}\}\ C$.*

Hat man mehrere Elementarrelationen, dann kann man die Kombination der Elementarrelationen bilden, das entsprechende Kompositions-Resultat in einer Tabelle ablesen, und dann die Vereinigung bilden. Z.B. $A\ \{\mathtt{m}, \mathtt{d}\}\ B \wedge B\ \{\mathtt{f}, \mathtt{d}\}\ C$ erlaubt auf folgende Relation zu schließen: $A\ (\mathtt{m} \circ \mathtt{f} \cup \mathtt{m} \circ \mathtt{d} \cup \mathtt{d} \circ \mathtt{f} \cup \mathtt{d} \circ \mathtt{d})\ C$.

Hier ergibt sich $A\ \{\mathtt{d}, \mathtt{s}, \mathtt{o}\} \cup \{\mathtt{d}, \mathtt{s}, \mathtt{o}\} \cup \{\mathtt{d}\} \cup \{\mathtt{d}\}\ C$. Das bedeutet: $A\ \{\mathtt{d}, \mathtt{s}, \mathtt{o}\}\ C$.

Allgemein gilt:

**Satz 6.3.2.** *Seien $r_1, \ldots, r_k, r'_1, \ldots, r'_k$ Allensche Basisrelationen. Dann gilt*

$$\{r_1, \ldots, r_k\} \circ \{r'_1, \ldots, r'_k\} = \bigcup \{r_i \circ r'_j \mid i = 1, \ldots, k, j = 1, \ldots, k'\}$$

D.h. man kann die Komposition von Teilmengen von Basisrelationen wieder auf die „punktweise“ Komposition der Basisrelationen zurückführen, die man aus obiger Tabelle ablesen kann.

Eine weitere Vereinfachung ist für die Inversion der Relationen möglich. Zunächst definieren wir:

**Definition 6.3.3** (Inversion für Mengen von Basisrelationen)**.** *Sei $S = \{r_1, \ldots, r_k\} \subseteq \mathcal{R}$. Dann sei*

$$\breve{S} = \{\breve{r_1}, \ldots, \breve{r_k}\}.$$

*Desweiteren definieren wir für eine Basisrelation $r$: $\breve{\breve{r}} = r$*

Damit gilt:

**Satz 6.3.4.** *Für $S \subseteq \mathcal{R}$ gilt: $A\ S\ B$ und $B\ \breve{S}\ A$ sind äquivalente Allensche Formeln.*

*Beweis.* Für die Basisrelationen ist das klar. Sei $S = \{r_1, \ldots, r_k\}$. Dann ist $A\ S\ B$ gerade $A\, r_1\ B \vee \ldots \vee A\, r_k\ B$, und $A\, r_1\ B \vee \ldots \vee A\, r_k\ B$ ist äquivalent zu $B\, \breve{r_1}\ A \vee \ldots \vee B\, \breve{r_k}\ A$, was wiederum per Definition gerade $B\ \breve{S}\ A$ ist. □

Ebenso gilt:

**Satz 6.3.5.** $\breve{}(r_1 \circ r_2) = \breve{r_2} \circ \breve{r_1}$.

Man kann sich auf die Konjunktion von Relationsbeziehungen beschränken, d.h. auf (konjunktive) Allen-Constraints, da man die Disjunktionen unabhängig bearbeiten kann.

### 6.3.1 Berechnung des Allenschen Abschlusses eines Constraints

Die folgenden Regeln stellen den Allenschen Kalkül dar. Sie dienen dazu den sogenannten Allenschen Abschluss eines Allenschen Constraints zu berechnen.

**Definition 6.3.6** (Regeln des Allenschen Kalküls)**.** *Die folgenden Regeln werden auf (Subformeln) von Allenschen Constraints angewendet:*

- *Aussagenlogische Umformungen dürfen verwendet werden.*
- $A\ R_1\ B \wedge A\ R_2\ B \to A\ (R_1 \cap R_2)\ B$
- $A\ \emptyset\ B \to 0$
- $A\ \mathcal{R}\ B \to 1$

- $A\ R\ A \to 0$, *wenn* $\equiv \notin R$.
- $A\ R\ A \to 1$, *wenn* $\equiv \in R$.
- $A\ R\ B \to A\ R\ B\ \wedge\ B\ \breve{R}\ A$
- $A\ R_1\ B \wedge B\ R_2\ C \to A\ R_1\ B \wedge B\ R_2\ C \wedge A\ (R_1 \circ R_2)\ C$.

Den Allenschen Abschluss eines (u.U. auch disjunktiven) Allenschen Constraints berechnet man, indem man obige Regel solange wie möglich anwendet:

**Definition 6.3.7** (Allenscher Abschluss)**.** *Für konjunktive Formeln (d.h. Allensche Constraints) werden die Regeln des Allenschen Kalküls solange angewendet, bis sich keine neuen Beziehungen mehr herleiten lassen.*

*Hat man einen disjunktiven Allenschen Constraint, so wendet man die Fixpunktiteration auf jede Komponente einzeln an, anschließend kann man u.U. vereinfachen: Erhält man für eine Komponente 1, so ist der disjunktive Allen-Constraint äquivalent zu 1. 0en kann man wie üblich streichen. Sind alle Disjunktionen falsch, dann hat mann eine Inkonsitenz entdeckt (die Eingabe ist ein widersprüchliches Allen-Constraint).*

**Beispiel 6.3.8.** *Wir betrachten nochmal das Beispiel des Kuchenbackens:*

*A1: Hefeteig zubereiten*
*A2: Hefeteig gehen lassen*
*A3: Äpfel schälen und in Scheiben schneiden*
*A4: Blech einfetten*
*A5: Teig auf das Blech*
*A6: Äpfel auf den Kuchen setzen.*

*Als Eingabe kann man die folgenden Relationen zwischen* $A1, \ldots, A6$ *nehmen, wenn mehrere Personen eine parallelisierte Verarbeitung ermöglichen:*

$A1\ \mathtt{m}\ A2$
$A2\ \{\mathtt{m}, \prec\}\ A5$
$A4\ \{\mathtt{m}, \prec\}\ A5$
$A3\ \{\mathtt{m}, \prec\}\ A6$
$A5\ \{\mathtt{m}, \prec\}\ A6$

*Das ergibt das Allensche Constraint*

$$A1\ \mathtt{m}\ A2 \wedge A2\ \{\mathtt{m}, \prec\}\ A5 \wedge A4\ \{\mathtt{m}, \prec\}\ A5 \wedge A3\ \{\mathtt{m}, \prec\}\ A6 \wedge A5\ \{\mathtt{m}, \prec\}\ A6.$$

*Berechnet man den Allenschen Abschluss, muss man im Wesentlichen nur prüfen, ob man mit der Transitivitätsregel neue Beziehungen findet.*

- *Aus* $A1\ \mathtt{m}\ A2 \wedge A2\ \{\mathtt{m}, \prec\}\ A5$ *erhält man* $A1 \prec A5$

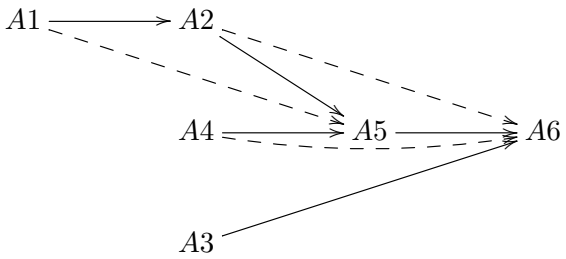
- *Aus* $A1 \prec A5 \wedge A5 \; \{\mathtt{m}, \prec\} \; A6$ *erhält man* $A1 \prec A6$
- *Aus* $A2 \; \{\mathtt{m}, \prec\} \; A5 \wedge A5 \; \{\mathtt{m}, \prec\} \; A6$ *erhält man* $A2 \prec A6$
- *Aus* $A4 \; \{\mathtt{m}, \prec\} \; A5 \wedge A5 \; \{\mathtt{m}, \prec\} \; A6$ *erhält man* $A4 \prec A6$

*Mehr kann man nicht herleiten, d.h. der Allensche Abschluss ist gerade das Constraint*

$A1 \; \mathtt{m} \; A2 \wedge A1 \prec A5 \wedge A1 \prec A6 \wedge A2 \; \{\mathtt{m}, \prec\} \; A5 \wedge A2 \prec A6 \wedge A4 \; \{\mathtt{m}, \prec\} \; A5 \wedge A4 \prec A6 \wedge A3 \; \{\mathtt{m}, \prec\} \; A6 \wedge A5 \; \{\mathtt{m}, \prec\} \; A6$.

*Beachte: Hinzufügen von Beziehungen* $Ai \; \mathcal{R} \; Aj$ *macht keinen Sinn, da diese direkt wieder gelöscht werden.*

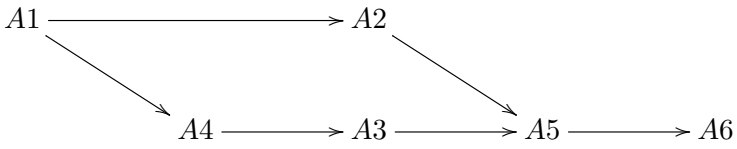
*Ein Bild zu den Relationen ist das folgende, wobei Pfeile* $A1 \rightarrow A2$ *bedeuten:* $A1$ *findet vor* $A2$ *statt (also* $m$ *oder* $\prec$*). Die gestrichelten Pfeile, deuten die Beziehungen an, die der Allensche Kalkül herleitet.*




**Beispiel 6.3.9.** *Wir betrachten weiterhin die sechs Aktionen zum Kuchenbacken und nehmen an, dass nur eine Person alleine backt, d.h. es gibt keine parallelen Aktionen. Dann wären die entsprechenden Relationen:*

$A1 \; \mathtt{m} \; A2$
$A2 \; \{\mathtt{m}, \prec\} \; A5$
$A4 \; \{\mathtt{m}, \prec\} \; A5$
$A3 \; \{\mathtt{m}, \prec\} \; A6$
$A5 \; \{\mathtt{m}, \prec\} \; A6$
$A1 \; \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \; A3$
$A1 \; \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \; A4$
$A1 \; \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \; A5$
$A3 \; \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \; A4$
$A3 \; \{\prec, \mathtt{m}, \breve{\mathtt{m}}, \succ\} \; A4$

*Eine Möglichkeit, die man durch Hinzufügen von Sequentialisierung bekommt, ist im Bild dargestellt:*



## 6.4 Untersuchungen zum Kalkül

Wir sagen, der Kalkül ist *korrekt*, wenn bei Transformation von $F$ nach $F'$ gilt, dass $F$ und $F'$ äquivalente Formeln sind.

Wir sagen, der Kalkül ist *herleitungs-vollständig*, wenn er für jedes konjunktive Constraint alle semantisch folgerbaren Einzel-Relationen herleiten kann.

Wir sagen, der Kalkül ist widerspruchs-*vollständig*, wenn er für jedes konjunktive Constraint herausfinden kann, ob es widersprüchlich ist, indem der Kalkül irgendwann die atomare Formel 0 herleitet.

Es stellen sich die folgenden Fragen:

- Wie aufwändig ist die Berechnung des Abschlusses der Allenschen Relationen?
- Ist die Berechnung herleitungs- bzw- widerspruchs-vollständig? D.h. gibt es nicht doch noch nicht erkannte Relationsbeziehungen, die aus der Semantik der linearen, reellen Zeitachse aus einer vorgegebenen Formel folgen?
- Was ist die Komplexität der Logik und der Herleitungsbeziehung, evtl. für eingeschränkte Eingabeformeln?
- Wie kann man den Allenschen Kalkül für aussagenlogische Kombinationen von Intervallformeln verwenden?

### 6.4.1 Komplexität der Berechnung des Allenschen Abschlusses

Ein Vervollständigungsalgorithmus kann analog zur Berechnung des transitiven Abschlusses einer Relation implementiert werden. Methoden des dynamischen Programmierens kann man dazu verwenden. Das ergibt einen polynomiellen Aufwand zur Vervollständigung.

Wir geben zwei Algorithmen an. Beide Algorithmen nehmen ein konjunktives Allensches Constraint als Eingabe, wobei dieses bereits in Form eines zweidimensionalen Arrays $R$ der Größe $n \times n$ vorliegt, dass über die $n$ Intervallnamen indiziert ist, d.h. die Constraints sind alle von der Form $A_i \; R_{i,j} \; A_j$. Nicht bekannte Relationen werden dementsprechend auf $\mathcal{R}$ gesetzt (für $A_i \; R_{i,i} \; A_i$ kann $R_{i,i}$ auch initial auf $\equiv$ gesetzt werden). Sobald in einem Eintrag die leere Menge hergeleitet wurde, kann der Algorithmus gestoppt werden, da der Constraint unerfüllbar ist. Wir erwähnen diesen möglichen Abbruch nicht explizit in den folgenden Algorithmen.

Der erste Algorithmus in Abbildung 6.2 wendet die Transitivitätsregel solange an, bis sich nichts mehr ändert (d.h. ein Fixpunkt erreicht ist). Der Algorithmus ist ähnlich zum Floyd-Warshall-Algorithmus zur Berechnung der transitiven Hülle einer Relation, hat jedoch (den notwendigen) zusätzlichen Fixpunkttest:

Offensichtlich ist der Algorithmus korrekt, da er solange rechnet, bis sich nichts mehr ändert (also der Fixpunkt erreicht ist). Die Laufzeit des Algorithmus ist im Worst-Case

*Algorithmus* **Allenscher Abschluss, Variante 1**

**Eingabe:** $(n \times n)$-Array $R$, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$

**Algorithmus**:

```
repeat
  change := False;
  for i := 1 to n do
    for j := 1 to n do
      for k := 1 to n do
        R' := R_{i,j} ∩ (R_{i,k} ∘ R_{k,j});
        if R_{i,j} ≠ R' then
          R_{i,j} := R;
          change := True;
        endif
      endfor
    endfor
  endfor
until change=False
```

*Abbildung 6.2: Algorithmus 1 zur Berechnung des Allenschen Abschlusses*

$O(n^5)$: Im schlimmsten Fall wird in den drei for-Schleifen nur ein einziger Eintrag $R_{i,j}$ geändert. Jeder Eintrag $R_{i,j}$ kann höchstens 13 Mal geändert werden (Verkleinerung der Relation $R_{i,j}$ für den Constraint $A_i\ R_{i,j}\ A_j$ jedesmal um ein Element). Das ergibt $O(n^2)$ Durchläufe der repeat-Schleife. Ein Durchlauf der repeat-Schleife verbraucht aufgrund der drei for-Schleifen $O(n^3)$ Zeit, da alle anderen Operationen als konstant angenommen werden können.

Ein besserer Algorithmus ist in Abbildung 6.3 dargestellt. Die Korrekheit dieses Algorithmus ergibt sich daraus, dass bei einer Änderung des Werts $R_{i,j}$ wieder alle Nachbarn, deren Wert evtl. neu berechnet werden muss, in die Queue eingefügt werden.

Die Laufzeit des Algorithmus ist im Worst-Case $O(n^3)$: Am Anfang enthält die Menge queue $O(n^3)$ Tripel. Jedes $R_{i,j}$ kann maximal 13 mal verändert werden. Bei einer Änderung an $R_{i,j}$ werden $2 * n$ Tripel in queue eingefügt. Da es nur $n^2$ viele $R_{i,j}$ gibt, werden insgesamt maximal $13 * 2 * n * n^2 = O(n^3)$ Tripel zu queue hinzugefügt. Also gibt es nicht mehr als $O(n^3)$ Durchläufe der while-Schleife, von denen maximal $O(n^2)$ Durchläufe $O(n)$ Laufzeit verbrauchen und die restlichen $O(n)$ in konstanter Laufzeit laufen. Das ergibt in der Summe eine Laufzeit von $O(n^3)$.

### 6.4.2 Korrektheit

Es gilt:

**Satz 6.4.1.** *Der Allensche Kalkül ist korrekt.*

*Algorithmus* **Allenscher Abschluss, Variante 2**

**Eingabe:** $(n \times n)$-Array $R$, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$

**Algorithmus**:
queue := $\{(i,k,j) \mid 1 \leq i \leq n, 1 \leq k \leq n, 1 \leq j \leq n\}$;

**while** queue $\neq \emptyset$ **do**

  Wähle und entferne Tripel $(i,k,j)$ aus queue;

  $R' := R_{i,j} \cap (R_{i,k} \circ R_{k,j})$;

  **if** $R_{i,j} \neq R'$ **then**

    $R_{i,j} := R'$;

    queue := queue ++ $\{(i,j,m) \mid 1 \leq m \leq n\}$ ++ $\{(m,i,j) \mid 1 \leq m \leq n\}$

  **endif**

**endwhile**

*Abbildung 6.3: Algorithmus 2 zur Berechnung des Allenschen Abschlusses*

*Beweis.* Die Korrekheit muss man mittels der Semantik nachweisen. Für die aussagenlogischen Umformungen ist dies offensichtlich.

- $A\ R_1\ B \wedge A\ R_2\ B$ ist äquivalent zu $A\ (R_1 \cap R_2)\ B$:
  Sei $R_1 = \{r_1, \ldots, r_k\}$, $R_2 = \{r'_1, \ldots, r'_{k'}\}$. Dann ist $A\ R_1\ B \wedge A\ R_2\ B$ äquivalent zu $\bigvee\{(A\ r_i\ B) \wedge (A\ r'_{i'} B) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Wenn man nun die Eigenschaft hinzunimmt, dass alle Basisrelationen echt disjunkt sind, so sieht man, dass die Formel äquivalent zu $\bigvee\{(A\ r_i\ B) \wedge (A\ r'_{i'} B) \mid 1 \leq i \leq k, 1 \leq i' \leq k', r_i = r'_{i'}\}$ ist, was gerade genau $A\ (R_1 \cap R_2)\ B$ entspricht.

- $A\ \emptyset\ B$ und $0$ sind äquivalent, da für jede Interpretation $I$ per Definition $I(A \emptyset B) = 0$ gilt.

- $A\ \mathcal{R}\ B$ ist äquivalent zu $1$, da jede Interpretation $I$, die Intervalle $I(A)$ und $I(B)$ interpretiert, und $\mathcal{R}$ alle möglichen Lagen abdeckt.

- $A\ R\ A$ ist äquivalent zu $0$, wenn $\equiv \notin R$: Jede Interpretation bildet $I(A)$ eindeutig auf ein Intervall ab.

- $A\ R\ A$ ist äquivalent zu $1$, wenn $\equiv \in R$: Jede Interpretation bildet $I(A)$ eindeutig auf ein Intervall ab.

- $A\ R\ B$ ist äquivalent zu $B\ \breve{R}\ A$: Das haben wir bereits gezeigt (Satz 6.3.4).

- Für die Transitivitätsregel, die $A\ R_1\ B \wedge B\ R_2\ C$ durch $A\ R_1\ B \wedge B\ R_2\ C \wedge A\ R_1 \circ R_2\ C$ ersetzt, kann man zunächst die Fälle untersuchen, in denen $R_1$ und $R_2$ genau eine Basisrelation enthalten, d.h. man muss die Korrekheit der Einträge in

der $13 \times 13$-Matrix überprüfen (anhand der durch die Interpretation gegebenen Lagen der Intervalle auf der reellen Achse). Für mehrelementige Mengen: Sei $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C$ gegeben. Dies ist eine Abkürzung für $(A\ r_1\ B \vee \ldots \vee A\ r_k\ B) \wedge (B\ r'_1\ C \vee \ldots \vee B\ r'_{k'}\ C)$. Ausmultiplizieren ergibt gerade $\bigvee\{(A\ r_i\ B \wedge B\ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Nun können wir für jede Konjunktion $A\ r_i\ B \wedge B\ r'_{i'}\ C$ die bereits als korrekt gezeigte Ersetzung durchführen. Das ergibt $\bigvee\{(A\ r_i\ B \wedge B\ r'_{i'}\ C \wedge A\ r_i \circ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Nach Umklammern und dem Rückgängigmachen des Ausmultiplizierens erhalten wir: $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C \wedge \bigvee\{(A\ r_i \circ r'_{i'}\ C) \mid 1 \leq i \leq k, 1 \leq i' \leq k'\}$. Die letzte Veroderung entspricht genau der Definition von $\circ$ auf Mengen, d.h. wir dürfen umformen zu $A\ \{r_1, \ldots, r_k\}\ B \wedge B\ \{r'_1, \ldots, r'_{k'}\}\ C \wedge A\ \{r_1, \ldots, r_k\} \circ \{r'_1, \ldots, r'_{k'}\}\ C$.

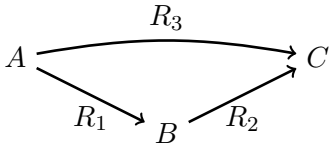
□

### 6.4.3 Partielle Vollständigkeit

Der Allensche Kalkül ist vollständig in eingeschränktem Sinn:

**Satz 6.4.2.** *Sind zwei Relationen $A_1\ R_1\ B_1\ \wedge\ A_2\ R_2\ B_2$ gegeben, dann ermittelt der Allensche Kalkül alle Relationsbeziehungen, die daraus folgen.*

Man kann einen Allenschen Constraint als sogenanntes Constraint-Netzwerk darstellen: Dabei werden die Intervallnamen als Knoten verwendet (pro Intervallname gibt es genau einen Knoten mit dem Intervallnamen als Markierung). Die Beziehungen zwischen den Intervallen werden als gerichtete Kanten mit Markierung der entsprechenden Basisrelationen dargestellt. Satz 6.4.2 sagt dann gerade aus, dass das Constraint-Netzwerk zum Allenschen Abschluss eines Constraints *Pfad-konsistent* ist, was gerade bedeutet, dass für je drei Knoten $A, B, C$ mit Kanten $A\ R_1\ B$, $B\ R_2\ C$ und $A\ R_3\ C$ des Netzwerks stets gilt: Wenn es eine Interpretation $I$ gibt, die $A\ R_3\ C$ erfüllt, dann können wir stets auch die anderen Kanten $A\ R_1\ B$ und $B\ R_2\ C$ erfüllen, ohne die Interpretation für $A$ und $C$ zu verändern, oder formaler: Wenn $I(A) = [AA, AZ], I(C) = [CA, CZ]$, so dass $I(A\ R_3\ C) = 1$, dann gibt es eine Interpretation $I'$ mit $I'(A) = I(A), I'(C) = I(C)$ und $I'(A\ R_1\ B) = 1, I'(B\ R_2\ C) = 1$. Als Teilausschnitt des Constraint-Netzwerks dargestellt:




Das der Allensche Abschluss die Pfadkonsistenz erfüllt ist klar, denn die Berechnung sichert gerade $R_3 \subseteq R_1 \circ R_2$ zu.

## 6.5 Unvollständigkeiten des Allenschen Kalküls.

Leider gilt:

**Theorem 6.5.1.** *Der Allensche Kalkül ist nicht herleitungs-vollständig.*

*Beweis.* Es genügt ein Gegenbeispiel anzugeben. Man benötigt vier Intervallkonstanten $A, B, C, D$. Die Beziehungen sind:

$$D \{\mathtt{o}\} B, D \{\mathtt{s},\mathtt{m}\} C, D \{\mathtt{s},\mathtt{m}\} A, A \{\mathtt{d},\breve{\mathtt{d}}\} B, C \{\mathtt{d},\breve{\mathtt{d}}\} B$$

Mit der kompositionellen Vervollständigung (nach Umdrehen) kann man aus $C \{\breve{\mathtt{s}},\breve{\mathtt{m}}\} D, D \{\mathtt{s},\mathtt{m}\} A$ die Relation $C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$ schließen. Aus $A \{\mathtt{d},\breve{\mathtt{d}}\} B, B \{\mathtt{d},\breve{\mathtt{d}}\} C$ kann man nur schließen, dass alles möglich ist. Der Schnitt ergibt somit $C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$. Oder anders ausgedrückt: Für das Allensche Constraint:

$$D \{\mathtt{o}\} B \wedge D \{\mathtt{s},\mathtt{m}\} C \wedge D \{\mathtt{s},\mathtt{m}\} A \wedge A \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{d},\breve{\mathtt{d}}\} B$$

ist der Allensche Abschluss:

$$D \{\mathtt{o}\} B \wedge D \{\mathtt{s},\mathtt{m}\} C \wedge D \{\mathtt{s},\mathtt{m}\} A \wedge A \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{d},\breve{\mathtt{d}}\} B \wedge C \{\mathtt{s},\breve{\mathtt{s}},\equiv,\mathtt{o},\breve{\mathtt{o}},\mathtt{d},\breve{\mathtt{d}},\mathtt{f},\breve{\mathtt{f}}\} A$$

Betrachtet man aber genauer die Möglichkeiten auf der reellen Achse, dann sieht man, dass in diesem speziellen Fall die Relation $C \{\mathtt{f},\breve{\mathtt{f}},\mathtt{o},\breve{\mathtt{o}}\} A$ nicht möglich ist. Die Lage von $B$ zu $D$ ist eindeutig. Wir betrachten daher die Möglichkeiten wie $A$ zu $D$ und $C$ zu $D$ liegen können. Da ergibt vier Fälle: (1) $D\ \mathtt{s}\ C$ und $D\ \mathtt{s}\ A$; (2) $D\ \mathtt{m}\ C$ und $D\ \mathtt{s}\ A$; (3) $D\ \mathtt{s}\ C$ und $D\ \mathtt{m}\ A$; (4) $D\ \mathtt{m}\ C$ und $D\ \mathtt{m}\ A$. Dabei muss man noch die Bedingungen zwischen $A$ zu $B$ und $C$ zu $B$ beachten (diese werden jedesmal eindeutig).

Die vier Fälle sind in Abbildung 6.4 als Bilder dargestellt sind:

Die Bilder zeigen: In Fall (1) ist $C \{\mathtt{s},\breve{\mathtt{s}}\ \equiv\} A$ möglich, in Fall (2) nur $C\ \mathtt{d}\ A$, in Fall (3) $C\ \breve{\mathtt{d}}\ A$ und in Fall 4 ist nur $C \{\mathtt{s},\breve{\mathtt{s}},\equiv\} A$ möglich. D.h. die Relation $C \{\mathtt{f},\breve{\mathtt{f}},\mathtt{o},\breve{\mathtt{o}}\} A$ ist niemals möglich, obwohl der Allensche Abschluss diese als Möglichkeiten herleitet.

Damit ist die Allensche Vervollständigung bezüglich der Semantik einer reellen Zeitachse nicht vollständig. Es können nicht alle Relationen exakt hergeleitet werden. □

Der Allensche Kalkül kann damit auch nicht immer erkennen, ob eine eingegebene Menge von Relationen inkonsistent ist.

**Satz 6.5.2.** *Der Allensche Kalkül ist nicht widerlegungsvollständig.*

*Beweis.* Ein Gegenbeispiel kann man durch Abwandlung des Gegenbeispiels zur Vollständigkeit gewinnen, man fügt die Relation $A \{\mathtt{f},\breve{\mathtt{f}}\} C$ hinzu, d.h. man erhält das Constraintnetzwerk:

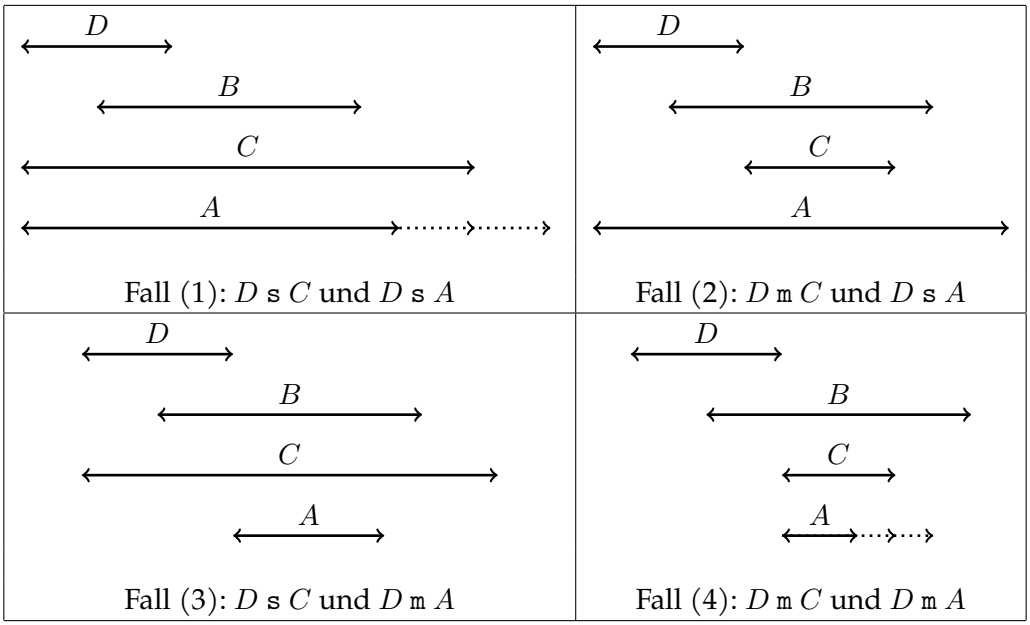
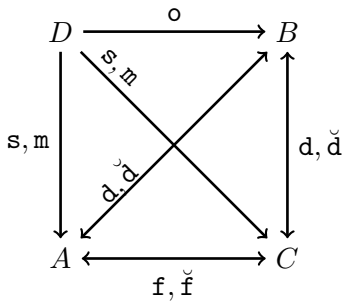



*Abbildung 6.4: Vier Fälle zur Herleitungs-Unvollständigkeit*




Man sieht, dass die Allensche Vervollständigung hier keine weiteren Erkenntnisse bringt: Man muss 12 Möglichkeiten der Komposition prüfen. Jede ergibt nur allgemeinere Bedingungen als die schon vorhandenen. Somit kann der Kalkül nichts Neues schließen. Insbesondere findet er keinen Widerspruch, obwohl die Menge der Constraints widersprüchlich ist. □

**Bemerkung 6.5.3.** *Man kann sich bei Anfragen an den Allen-Kalkül nur darauf verlassen, dass die Vervollständigung richtig ist, aber evtl. nicht optimal. Wenn man die Frage stellt: Ist das Constraint C widersprüchlich, so kann man sich nur bei „JA“ (d.h. Herleitung der 0) auf die Antwort verlassen, aber micht bei „NEIN“. Umgekehrt heisst das auch, dass man sich bei der Frage nach der Erfüllbarkeit dementsprechend nur auf die Antwort „NEIN“ verlassen kann.*

## 6.6 Einge Analysen zur Implementierungen

Ein Allensches Constraint kann man versuchen vollständig auf Erfüllbarkeit bzw Unerfülllbarkeit zu testen, indem man Fallunterscheidungen macht.

Wir schauen genauer auf Allensche Constraints, wobei wir annehmen, dass diese normalisiert sind, d.h. zwischen zwei Intervallkonstanten gibt es nur ein Constraint.

**Definition 6.6.1.** *Ein Allensches Constraint nennt man* eindeutig, *wenn für alle Paare $A, B$ von Intervallkonstanten gilt: Das Constraint enthält genau eine Beziehung $A\ r\ B$, wobei $r$ eine der dreizehn Basisrelationen ist.*

Es gilt:

**Satz 6.6.2.** *Der Allensche Abschluss eines eindeutigen Allenschen Constraints $F$ ist entweder $0$, oder wiederum $F$.*

*Beweis.* Es reicht zu zeigen, dass die Anwendung der Transitivitätsregel nur eindeutige Relationen erzeugen kann (nach Anwendung weiterer Regeln): Seien $A\ r_1\ B$, $B\ r_2\ C$ Teilformeln des eindeutigen Allenschen Constraint, dann muss es bereits eine Beziehung $A\ r_3\ C$ geben (sonst wäre das Constraint nicht eindeutig). Die Transitivitätsregel ergibt:

$$A\ r_1\ B \wedge B\ r_2\ C \wedge A\ r_3\ C \rightarrow A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2)\ C \wedge A\ r_3\ C.$$

Anschließend können wir eine weitere Regel anwenden:

$$A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2)\ C \wedge A\ r_3\ C \rightarrow A\ r_1\ B \wedge B\ r_2\ C \wedge A\ (r_1\ \circ\ r_2) \cap \{r_3\}\ C$$

. Da der Schnitt $(r_1\ \circ\ r_2) \cap \{r_3\}$ entweder $r_3$ oder $\emptyset$ (Unerfüllbarkeit) ergeben muss, folgt die Aussage. □

**Definition 6.6.3.** *Zu jedem Allenschen Constraint $C$ kann man die Menge aller zugehörigen eindeutigen Allenschen Constraints $D$ definieren, wobei gelten muss: Wenn $A\ r\ B$ in $D$ vorkommt und $A\ R\ B$ in $C$, dann gilt $r \in R$.*

**Lemma 6.6.4.** *Ein Allensches Constraint ist erfüllbar, gdw., es ein zugehöriges eindeutiges Constraint gibt, das erfüllbar ist.*

*Beweis.* Das ist klar, da die Einzelconstraints ja Disjunktionen sind, und bei einer erfüllenden Belegung genau eine der Relationen in der Relationsmenge gilt. □

**Satz 6.6.5.** *Ein eindeutiges Allensches Constraint ist erfüllbar, gdw. der Allensche Kalkül bei Vervollständigung das Constraint nicht verändert, d.h. wenn es ein Fixpunkt ist.*

*Beweis.* Wenn der Kalkül einen Widerspruch entdeckt, dann ist das Constraint natürlich widersprüchlich. Für den Fall, dass der Kalkül keinen Widerspruch entdeckt, kann man

*Algorithmus* **Erfüllbarkeitstest für konjunktive Allensche Constraints**

**Eingabe:** $(n \times n)$-Array $R$, mit Einträgen $R_{i,j} \subseteq \mathcal{R}$
**Ausgabe:** `True` (Widerspruch) oder `False` (erfüllbar)

**function** AllenSAT($R$):
  $R' :=$ AllenAbschluss($R$);
  **if** $\exists R'_{i,j}$ mit $R'_{i,j} = \emptyset$ **then return** `True` **endif**;// Widerspruch
  **if** $\forall R'_{i,j}$ gilt: $|R'_{i,j}| = 1$ **then return** `False` // eindeutig und erfüllbar
  **else**
    wähle $R'_{i,j}$ mit $R'_{i,j} = \{r_1, r_2, \ldots\}$;
    $R^l := R'$; $R^l_{i,j} := \{r_1\}$;// kopiere $R'$ und setze $(i,j)$ auf $r_1$
    $R^r := R'$; $R^r_{i,j} := R'_{i,j} \setminus \{r_1\}$;// kopiere $R'$ und setze $(i,j)$ auf $r_2, \ldots$
    **return** (AllenSAT($R^l$) $\wedge$ AllenSAT($R^r$));
  **endif**

*Abbildung 6.5: Vollständiger Erfüllbarkeitstest für konjunktive Allen-Constraints*

zeigen, dass eine totale Ordnung der Intervallenden möglich ist. Einen entsprechenden Beweis findet man bspw. in (Valdés-Pérez, 1987) □

Daraus ergibt sich ein (im worst-case exponentieller) Algorithmus, der die Erfüllbarkeit eines Allenschen Constraints testet:

- Sei $C$ ein Allensches Constraint.
- Sei $D$ die Menge aller eindeutigen Allenschen Constraints zu $C$.
- Berechne den Allenschen Abschluss jedes Constraints $C' \in D$.
- Wenn dabei kein Widerspruch auftritt, ist $C$ erfüllbar, anderenfalls $C$ widersprüchlich.

Hierbei kann man noch etwas geschickter vorgehen: Sobald ein Modell gefunden wurde, kann man komplett aufhören (die Erfüllbarkeit gilt) und sobald ein Widerspruch gefunden wurde braucht man die dazugehörigen eindeutigen Allen-Constraints nicht weiter zu betrachten (da der Allensche Abschluss auch auf mehrdeutigen Constraints korrekt ist). Abbildung 6.5 zeigt den so optimierten Algorithmus.

Die durchschnittliche Verzweigungsrate dieses Algorithmus ist 6.5, da man im schlimmsten Fall pro Beziehung $A\ R\ B$ dreizehn Fallunterscheidungen machen muss, und im besten Fall $R$ leer ist.

## 6.7 Komplexität

Komplexität des Problems und des Algorithmus sind zu unterscheiden:

Die Komplexität des Problems ist schlechter, denn es gilt:

**Satz 6.7.1.** *Die Erfüllbarkeit einer Konjunktion von Allenschen Relationen ist $\mathcal{NP}$-vollständig, bzw. die Erfüllbarkeit eines konjunktiven Allenschen Constraints ist $\mathcal{NP}$-vollständig.*

*Beweis.* Es ist leicht einzusehen, dass die Erfüllbarkeit eines Allenschen Constraints in $\mathcal{NP}$ ist. Man muss nur ein lineare Reihenfolge aller Werte $X_a, X_e$ angeben (bzw. raten), wobei $X$ eine Intervallkonstante ist und $X_a$ der Anfang und $X_e$ das Ende. Der Test, ob diese lineare Reihenfolge das Constraint erfüllt, ist dann polynomiell.

Zum Beweis der $\mathcal{NP}$-härte nehme das $\mathcal{NP}$-vollständige Problem der Drei-Färbbarkeit eines Graphen:

> Gegeben ein Graph mit Knotenmenge $N$ und Kantenmenge $K$. Gibt es eine Färbung der Knoten mit drei Farben, so dass benachbarte Knoten verschiedene Farbe haben?

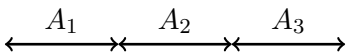
Sei eine Instanz gegeben, d.h. ein Graph $(N, K)$. Dann erzeuge ein Allensches Constraint:

Seien $A_1, A_2, A_3$ Intervallkonstanten. Für jeden Knoten $v_i \in N$ erzeuge eine Intervallkonstante. $B_i$. Erzeuge die folgenden Allenschen Relationen (d.h. verunde sie zu einem konjunktiven Allenschen Constraint):

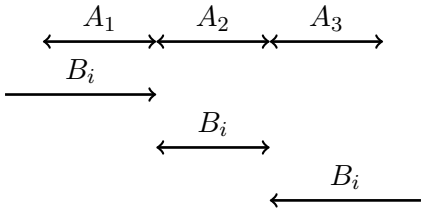
$$\begin{array}{lllll} & A_1 & \mathtt{m} & A_2 & \\ & A_2 & \mathtt{m} & A_3 & \\ \forall v_i \in N: & B_i & \{\mathtt{m}, \equiv, \breve{\mathtt{m}}\} & A_2 & \\ \forall (v_i, v_j) \in K: & B_i & \{\mathtt{m}, \breve{\mathtt{m}}, \prec, \succ\} & B_j & \end{array}$$

Die Idee dabei ist gerade: Die Lage jedes $B_i$ zu $A_2$ bestimmt die Farbe des Knotens $v_i$. Das kann man sich klar machen, indem man die Constraints als Bild veranschaulicht:

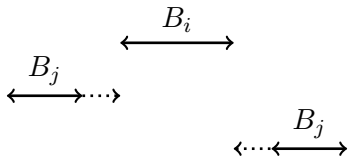
Die ersten beiden Constraints ergeben gerade




Die Constraints für die Knoten ergeben, dass $B_i$ entweder mit $A_2$ übereinstimmt ($B_i \equiv A_2$), oder direkt links davon ($B_i \ \mathtt{m} \ A_2$, überschneidend mit $A_1$) oder direkt rechts davon ($B_i \ \breve{\mathtt{m}} \ A_2$, überschneidend mit $A_3$) liegt:




Die Constraints für die Kanten ergeben, dass die Intervalle $B_i, B_j$ keine Überschneidungen haben:




Nun ist zu zeigen: Der Graph ist dreifärbbar, gdw. die Allenschen Relationen erfüllbar sind. Hierfür reicht die Zuordnung:

- $v_i$ hat Farbe 2 gdw. $B_i \equiv A_2$
- $v_i$ hat Farbe 1 gdw. $B_i \; \mathtt{m} \; A_2$
- $v_i$ hat Farbe 3 gdw. $B_i \; \breve{\mathtt{m}} \; A_2$

Diese Übersetzung kann in polynomieller Zeit abhängig von der Größe des Graphen gemacht werden. Diese Übersetzung zeigt die $\mathcal{NP}$-Härte der Allenschen Constraints.

Damit ist die Erfüllbarkeit von Allenschen Constraints $\mathcal{NP}$-vollständig. □

Als Schlussfolgerung kann man sagen, dass ein vollständiger Algorithmus nach aktuellem Wissenstand mindestens exponentiell sein muss, während der polynomiell Allensche Vervollständigungs-Algorithmus unvollständig sein muss.

## 6.8 Eine polynomielle und vollständige Variante

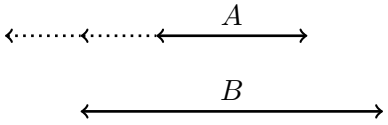
Man kann Varianten und Einschränkungen der Allenschen Constraints suchen, die einen sowohl vollständigen als auch polynomiellen Erfüllbarkeitstest erlauben. Eine solche Variante haben wir bereits gesehen: Für eindeutige Allensche Constraints ist der Allensche Kalkül vollständig.

Eine Variante ist die Menge der Relationen, die man durch eine Konjunktion von atomaren Ungleichungen der Form $x < y$ oder $x = y$ repräsentieren kann, wobei $x, y$ Endpunkte von Intervallen sind. Hier kommt es darauf an, dass man keine Disjunktionen hat, die eine Fallunterscheidung erzwingen.

Einen passenden Satz von Relationen gibt es:

Alle Basisrelationen, $\{\mathtt{d}, \mathtt{o}, \mathtt{s}\}$, und $\{\breve{\mathtt{o}}, \mathtt{f}, \mathtt{d}\}$ und deren Konverse. d.h. $\{\breve{\mathtt{d}}, \breve{\mathtt{o}}, \breve{\mathtt{s}}\}$, und $\{\mathtt{o}, \breve{\mathtt{f}}, \breve{\mathtt{d}}\}$. Beachte, dass man noch weitere dazu nehmen kann.

Die Relation $A\{\mathtt{d}, \mathtt{o}, \mathtt{s}\}B$ z.B. kann man als Ungleichung über den Endpunkten ausdrücken:




Wenn $A = [AA, AZ], B = [BA, BZ]$, dann entspricht obige Relation der Konjunktion der Ungleichungen: $AA < AZ, BA < BZ, AZ < BZ, BA < AZ$

Eine Konjunktion von solchen Relationsbeziehungen ergibt in der Summe ein Constraint, das eine Konjunktion von Ungleichungen der Endpunkte von Intervallen ist. Diese Konjunktion kann man mittels transitivem Abschluss über die Endpunkte von Intervallen vervollständigen. Dies geht in polynomieller Zeit. Wenn man eine Relation der Form $X < X$ erzeugt hat, dann ist das Constraint unerfüllbar. Ansonsten ist er erfüllbar, denn die Endpunkte kann man mit topologischem Sortieren in eine lineare Reihenfolge bringen.

Insgesamt hat man gezeigt, dass der so definierte Kalkül auf den eingeschränkten Constraints vollständig und polynomiell ist. Es gilt sogar, dass der Allensche Kalkül selbst auf diesen Constraints vollständig ist (siehe z.B. (Nebel & Bürckert, 1995)).

Der Hintergrund der speziellen Klasse von Allenschen Constraints ist, dass sich diese Klasse als Grund-Hornklauseln darstellen lassen. Hornklauseln sind Klauseln, die maximal ein positives Literal haben. Hierbei ist ein Literal positiv, wenn es ein unnegiertes Atom ist.

Für aussagenlogische Hornklauselmengen und auch für Grund-Hornklauselmengen gilt, dass deren Erfüllbarkeit in polynomieller Zeit testbar ist.

Die notwendige Menge von Axiomen und Fakten ist in dem eingeschränkten Fall eine Hornklauselmenge,:
Man hat Fakten in der Form $a < b$ und $c = d$, wobei $a, b, c, d$ unbekannte Konstanten sind. Es gibt auch Hornklauseln, die von der Symmetrie und Transitivität stammen:

$$\begin{array}{lcl} x < y \wedge y < z & \Rightarrow & x < z \\ x = y \wedge y = z & \Rightarrow & x = z \\ x = y & \Rightarrow & y = x \\ x < y \wedge y = z & \Rightarrow & x < z \\ x = y \wedge y < z & \Rightarrow & x < z \end{array}$$

Tatsächlich kann man weitere Allensche Constraints zulassen, und behält die Vollständigkeit des Allen-Kalküls: Dies ist genau die Klasse der Constraints deren Übersetzung in Constraints über Endpunkten Hornklauseln ausschließlich mit Literalen $a \leq b$, $a = b$ und $\neg(a = b)$ erzeugt (siehe (Nebel & Bürckert, 1995)). Ein interessanter Aspekt dabei ist, dass von den $2^{13} = 8192$ möglichen Relationen, 868 Relationen diese Eigenschaft aufweisen.

Für den vollständigen (aber exponentiellen) Algorithmus für beliebige Allensche Constraints, kann man dies ausnutzen, indem man nicht in eindeutige Relationen, sondern in Relationen entsprechend dieser Klasse Fallunterscheidungen durchführt, die durchschnittliche Verzweigungsrate kann dadurch von 6,5 auf 2,533 gesenkt werden (Nebel, 1997).

# 7 Maschinelles Lernen

## 7.1 Einführung: Maschinelles Lernen

Da die direkte Programmierung eines intelligenten Agenten sich als nicht möglich herausgestellt hat, ist es klar, dass man zum Erreichen des Fernziels der Künstlichen Intelligenz eine Entsprechung eines Lernprozesses benötigt, d.h. man benötigt *Maschinelles Lernen*.

Es gibt viele verschiedene Ansichten darüber, was Maschinelles Lernen ist, was mit Lernen erreicht werden soll usw. Hier sind erst Anfänge in der Forschung gemacht worden. Die praktisch erfolgreichsten Methoden sind solche, die auf statistisch/stochastischen Methoden basieren und mit der Adaption von Werten (Gewichten) arbeiten:

- Adaption von Gewichten einer Bewertungsfunktion aufgrund von Rückmeldungen. Z.B. Verarbeitung natürlicher Sprachen, Strategie-Spiele mit und ohne zufällige Ereignisse: Dame, Backgammon.

- Künstliche neuronale Netze: Lernen durch gezielte Veränderung von internen Parametern. Deren praktischer Nutzen und Anwendbarkeit ist im Wesentlichen auf praktikable automatische Lernverfahren zurückzuführen.

Das Lernen von neuen Konzepten, Verfahren, logischen Zusammenhängen, usw. hat bisher nur ansatzweise Erfolg gehabt.

Für den Agenten-basierten Ansatz, soll das Lernen eine Verbesserung der Performanz des Agenten bewirken:

- Verbesserung der internen Repräsentation.

- Optimierung bzw. Beschleunigung der Erledigung von Aufgaben.

- Erweiterung des Spektrums oder der Qualität der Aufgaben, die erledigt werden können.

**Beispiel 7.1.1.** *Drei beispielhafte Ansätze sind:*

- *Erweiterung und Anpassung des Lexikons eines computerlinguistischen Systems durch automatische Verarbeitung von geschrieben Sätzen, wobei der Inhalt dieser Sätze gleichzeitig automatisch erfasst werden sollte.*

- *Adaption von Gewichten einer Bewertungsfunktion in einem Zweipersonenspiel, wobei man abhängig von Gewinn/Verlust Gewichte verändert. Das wurde für Dame und Backgammon mit Erfolg durchgeführt.*

- *Lernen einer Klassifikation durch Vorgabe von Trainingsbeispielen, die als positiv/negativ klassifiziert sind.*

### 7.1.1 Einordnung von Lernverfahren

Die Struktur eines lernenden Systems kann man wie folgt beschreiben:

**Agent** (ausführende Einheit, performance element). Dieser soll verbessert werden anhand von Erfahrung; d.h. etwas lernen.

**Lerneinheit** (learning element). Hier wird der Lernvorgang gesteuert und bewertet. Insbesondere wird hier vorgegeben, was gut und was schlecht ist. Hier kann man auch die Bewertungseinheit (critic) und den Problemgenerator einordnen.

**Umwelt** In der Umwelt soll agiert werden. Die Rückmeldung über den Ausgang bzw. den Effekt von Aktionen kommt aus dieser Umwelt. Das kann eine künstliche, modellhafte Umwelt oder auch die reale Umwelt sein.

Zum Teil wird Agent und Lerneinheit zusammen in einen erweiterten Agent verlagert. Prinzipiell sollte man diese Einheiten unterscheiden, denn die Bewertung muss außerhalb des Agenten sein, sonst wäre die Möglichkeit gegeben, die Bewertung an die schlechten Aktionen anzupassen, statt die Aktionen zu verbessern.

Folgende *Lernmethoden* werden unterschieden:

**Überwachtes Lernen** (supervised learning). Diese Methode geht von der Situation aus in der es einen allwissenden Lehrer gibt. Die Lerneinheit kann dem Agenten bei jeder Aktion sagen, ob diese richtig war und was die richtige Aktion gewesen wäre. Das entspricht einem unmittelbaren Feedback über die exakt richtige Aktion. Alternativ kann man eine feste Menge von richtigen und falschen Beispielen vorgeben und damit dann ein Lernverfahren starten.

**Unüberwachtes Lernen** (unsupervised learning). Dies ist der Gegensatz zum überwachten Lernen. Es gibt keine Hinweise, was richtig sein könnte. Damit Lernen möglich ist, braucht man in diesem Fall eine Bewertung der Güte der Aktion.

**Lernen durch Belohnung/Bestrafung** (reinforcment learning). Dieser Ansatz verfolgt das Prinzip „mit Zuckerbrot und Peitsche“. Hiermit sind Lernverfahren gemeint, die gute Aktionen belohnen, schlechte bestrafen, d.h. Aktionen bewerten, aber die richtige Aktion bzw. den richtigen Parameterbereich nicht kennen.

Man kann man die Lernverfahren noch unterscheiden nach der Vorgehensweise. Ein *inkrementelles* Lernverfahren lernt ständig dazu. Ein anderer Ansatz ist ein *Lernverfahren mit Trainingsphase*: Alle Beispiele werden auf einmal gelernt.

Man kann die Lernverfahren auch entsprechend der Rahmenbedigungen einordnen:

- Beispielwerte sind exakt oder ungefähr bekannt bzw. mit Fehlern behaftet
- Es gibt nur positive bzw. positive und negative Beispiele

### 7.1.2 Einige Maßzahlen zur Bewertung von Lern- und Klassifikationsverfahren

Wir beschreiben kurz Vergleichsparameter, die man zur Abschätzung der Güte von Klassifikatorprogrammen bzw. Lernverfahren verwendet.

**Beispiel 7.1.2.** *Beispiele, um sich besser zu orientieren:*

- *Klassifikation von „Vogel“ anhand bekannter Attribute, wie kann-fliegen, hat-Federn, usw.*
- *Vorhersage, dass ein Auto noch ein Jahr keinen Defekt hat aufgrund der Parameter wie Alter, gefahrene Kilometer, Marke, Kosten der letzten Reparatur, usw.*
- *Medizinischer Test auf HIV: Antikörper*
- *Vorhersage der Interessen bzw. Kaufentscheidung eines Kunden aufgrund der bisherigen Käufe und anderer Informationen (online-Buchhandel).*
- *Kreditwürdigkeit eines Kunden einer Bank, aufgrund seines Einkommens, Alters, Eigentumsverhältnisse, usw (Adresse?).*

Ein *Klassifikator* ist ein Programm, das nur binäre Antworten auf Anfragen gibt: ja / nein. Die Aufgabe ist dabei, Objekte, beschrieben durch ihre Attribute, bzgl. einer anderen Eigenschaft zu klassifizieren, bzw. eine zukünftiges Ereignis vorherzusagen. Typische Beispiele sind die Bestimmung von Tier- Pflanzenarten anhand eines Exemplars, oder die Diagnose einer Krankheit anhand der Symptome.

Wir beschreiben die *abstrakte Situation* genauer. Es gibt:

- eine Menge $M$ von Objekten (mit innerer Strukur)
- das Programm $P : M \to \{0, 1\}$, welches Objekte klassifiziert
- die wahre Klassifikation $K : M \to \{0, 1\}$

Bei Eingabe eines Objekts $x \in M$ gilt:

- Im Fall $K(x) = P(x)$ liegt das Programm richtig. Man unterscheidet in

  **richtig-positiv** Wenn $P(x) = 1$ und $K(x) = 1$.

  **richtig-negativ** Wenn $P(x) = 0$ und $K(x) = 0$.

- Im Fall $K(x) \neq P(x)$ liegt das Programm falsch. Hier wird noch unterschieden zwischen

  **falsch-positiv** Wenn $P(x) = 1$, aber $K(x) = 0$.

  **falsch-negativ** Wenn $P(x) = 0$, aber $K(x) = 1$.

Die folgenden Werte entsprechen der Wahrscheinlichkeit mit der das Programm $P$ eine richtige positive (bzw. negative) Klassifikation macht. Es entspricht der Wahrscheinlichkeit, mit der eine Diagnose auch zutrifft. Hierbei wird angenommen, dass es eine Gesamtmenge $M$ aller Objekte gibt, die untersucht werden.

**Recall** (Richtig-Positiv-Rate, Sensitivität, Empfindlichkeit, Trefferquote; sensitivity, true positive rate, hit rate):

Der Anteil der richtig klassifizierten Objekte bezogen auf alle tatsächlich richtigen.

$$\frac{|\{x \in M \mid P(x) = 1 \wedge K(x) = 1\}|}{|\{x \in M \mid K(x) = 1\}|}$$

**Richtig-Negativ-Rate** (true negative rate oder correct rejection rate, Spezifität)

Der Anteil der als falsch erkannten bezogen auf alle tatsächlich falschen:

$$\frac{|\{x \in M \mid P(x) = 0 \wedge K(x) = 0\}|}{|\{x \in M \mid K(x) = 0\}|}$$

Die folgenden Werte entsprechen der Wahrscheinlichkeit mit der ein als positiv klassifiziertes Objekt auch tatsächlich richtig klassifiziert ist, bzw. die Wahrscheinlichkeit mit der eine positive Diagnose sich als richtig erweist. Oder anders herum: eine negativ Diagnose die Krankheit ausschließt.

Der Wert der Präzision ist ein praktisch relevanterer Wert als der recall, da diese aussagt, wie weit man den Aussagen eines Programms in Bezug auf eine Klassifikation trauen kann.

**Precision** (Präzision, positiver Vorhersagewert, Relevanz, Wirksamkeit, Genauigkeit, positiver prädiktiver Wert, positive predictive value)

Der Anteil der richtigen unten den als scheinbar richtig erkannten

$$\frac{|\{x \in M \mid P(x) = 1 \wedge K(x) = 1\}|}{|\{x \in M \mid P(x) = 1\}|}$$

**Negative-Vorhersage-Rate** Der Anteil richtig als falsch klassifizierten unter allen als falsch klassifizierten

$$\frac{|\{x \in M \mid P(x) = 0 \wedge K(x) = 0\}|}{|\{x \in M \mid P(x) = 0\}|}$$

Im medizinischen Bereich sind alle diese Werte wichtig. Bei seltenen Krankheiten kann ein guter Recall, d.h. der Anteil der Kranken, die erkannt wurden, mit einer sehr schlechten Präzision verbunden sein. Zum Beispiel, könnte ein Klassifikator für Gelbfieber wie folgt vorgehen: Wenn Körpertemperatur über 38,5 C, dann liegt Gelbfieber vor. In Deutschland haben 10.000 Menschen Fieber mit 38,5 C aber nur 1 Mensch hat Gelbfieber, der dann auch Fieber hat. Dann ist der Recall 1, aber die Precision ist $0.0001$, also sehr schlecht. Hier muss man also möglichst beide Größen ermitteln, und den Test genauer machen (precision erhöhen).

## 7.2 Wahrscheinlichkeit und Entropie

In diesem Abschnitt führen wir den Begriff der *Entropie* ein, wobei wir zunächst eine kurze Wiederholung zu diskreten Wahrscheinlichkeiten geben.

### 7.2.1 Wahrscheinlichkeit

Sei $X$ ein Orakel, das bei jeder Anfrage einen Wert aus der Menge $\{a_1, \ldots, a_n\}$ ausgibt, d.h. $X$ ist analog zu einer Zufallsvariablen. Man interessiert sich für die *Wahrscheinlichkeit* $p_i$, dass das Orakel den Wert $a_i$ ausgibt. Macht man (sehr) viele Versuche, so kommt in der Folge der Ergebnisse $b_1, \ldots, b_m$, für ein festes $i$ der Anteil der $a_i$ in der Folge dem Wert $p_i$ immer näher. Man nennt die Zahlen $p_i, i = 1, \ldots, n$ auch *diskrete Wahrscheinlichkeitsverteilung* (der Menge $a_i$), bzw. des Orakels $X$.

Zum Beispiel ist beim Münzwurf mit den Ausgängen $Kopf$ und $Zahl$ in einer ausreichend langen Folge in etwa die Hälfte $Kopf$, die andere Hälfte $Zahl$, d.h. man würde hier Wahrscheinlichkeiten $0,5$ und $0,5$ zuordnen.

Es gilt immer $0 \leq p_i \leq 1$ und $\sum_i p_i = 1$. Sind die $a_i$ Zahlen, dann kann man auch den *Erwartungswert* ausrechnen: $E(X) = \sum_i p_i a_i$. Das ist der Wert, dem die Mittelwerte der (Zahlen-)Folgen der Versuche immer näher kommen.

Wenn man die Arbeitsweise von $X$ kennt, dann kann man mehr Angaben machen. Z.B. das sogenannte *Urnenmodell*:

> $X$ benutzt einen Eimer in dem sich Kugeln befinden, rote, blaue und grüne. Bei jeder Anfrage wird „zufällig“ eine Kugel gezogen, deren Farbe ist das Ergebnis, und danach wird die Kugel wieder in den Eimer gelegt.

In diesem Fall sind die Wahrscheinlichkeiten $p_{\text{rot}}, p_{\text{blau}}, p_{\text{grün}}$ jeweils genau die *relativen Häufigkeiten* der roten, blauen, bzw. grünen Kugeln unter den Kugeln, die sich in der Urne jeweils vor dem Ziehen befinden.

### 7.2.2 Entropie

Zunächst führen wir den Begriff des Informationsgehalts ein, der von einigen Lernverfahren benötigt wird.

Wenn man eine diskrete Wahrscheinlichkeitsverteilung $p_i, i = 1, \ldots, n$ hat, z.B. von Symbolen $a_i, i = 1, \ldots, n$, dann nennt man

$$I(a_k) := \log_2(\frac{1}{p_k}) = -\log_2(p_k) \geq 0$$

den *Informationsgehalt* des Zeichens $a_k$. Das kann man interpretieren als Grad der Überraschung beim Ziehen des Symbols $a_k$ aus einer entsprechenden Urne, bzw. bei der Übermittlung von Zeichen durch einen Kommunikationskanal. D.h. das Auftreten eines seltenen Symbols hat einen hohen Informationsgehalt. Wenn man nur ein einziges Symbol hat, dann ist $p_1 = 1$, und der Informationsgehalt ist $I(a_1) = 0$. Eine intuitive Erklärung des Informationsgehalts ist die mittlere Anzahl der Ja/Nein-Fragen, die man stellen muss, um die gleiche Information zu bekommen.

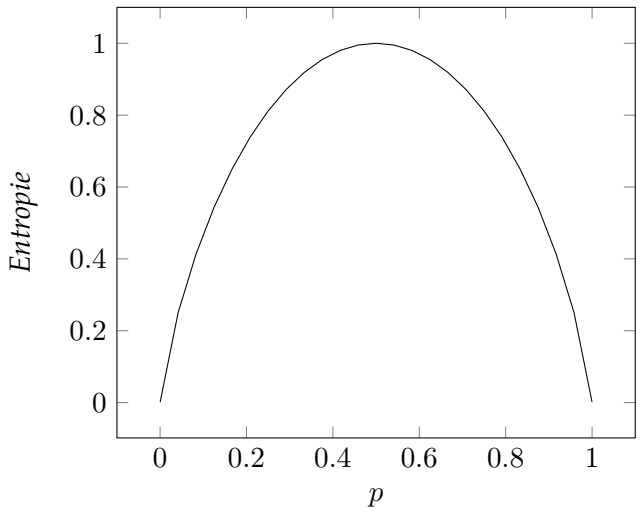
**Beispiel 7.2.1.** *Zum Beispiel im Falle von 8 Objekten, die gleich oft vorkommen, ergibt sich* $\log(0.125) = -3$ *für jedes Objekt, d.h. der Informationsgehalt jedes Zeichens ist* 3 *und auch der* mittlere Informationsgehalt*, ermittelt aus der gewichteten Summe ist* 3*. Kommen zwei davon, sagen wir mal* $a_1, a_2$*, sehr häufig vor und die anderen praktisch nie, dann ergibt sich als mittlerer Informationsgehalt in etwa* $0.5 * \log_2(0.5) + 0.5 * \log_2(0.5) + 6 * 0.001 * \log_2(0.001) \approx 1$.

Die *Entropie* oder der *mittlere Informationsgehalt* der Symbole in der Wahrscheinlichkeitsverteilung wie oben kann dann berechnet werden als

$$I(X) = \sum_{i=1}^{n} p_i * \log_2(\frac{1}{p_i}) = -\sum_{i=1}^{n} p_i * \log_2(p_i) \geq 0.$$

Bei Kompressionen eines Files oder bei Kodierung von Nachrichten über einem Kanal ist das eine untere Schranke für die mittlere Anzahl von Bits pro Symbol, die man bei bester Kompression bzw binärer Kodierung erreichen kann.

**Beispiel 7.2.2.** *Nimmt man ein Bernoulli-Experiment, d.h. zwei Zeichen,* $Kopf$ *und* $Zahl$ *wobei* $Kopf$ *mit der Wahrscheinlichkeit* $p$ *und* $Zahl$ *mit Wahrscheinlichkeit* $1 - p$ *auftritt, dann ergibt sich in etwa die Kurve:*




*D.h. die Entropie (der mittlere Informationsgehalt eines Münzwurfs) ist maximal, wenn man das Zeichen nicht vorhersagen kann. Bei einer Wahrscheinlichkeit von $p = 0,9$ kann man vorhersagen, dass $Kopf$ sehr oft auftritt. Das ist symmetrisch zu $p = 0,1$. Die Entropie ist in beiden Fällen $0,469$.*

## 7.3 Lernen mit Entscheidungsbäumen

Wir betrachten nun das Lernen mit sogenannten *Entscheidungsbäume*. Diese Bäume repräsentieren das Klassifikator-Programm. Hierbei wird zunächst mit einer Menge von Beispielen, der Entscheidungsbaum aufgebaut, der im Anschluss als Klassifikator verwendet werden kann. Unser Ziel wird es sein, Algorithmen vorzustellen, die einen *guten* Entscheidungsbaum erstellen. Güte meint hierbei, dass der Baum möglichst flach ist. Man kann sich dies auch anders verdeutlichen: Hat man Objekte mit vielen Attributen, so möchte man die Attribute herausfinden, die markant sind, für die Klassifikation, und diese zuerst testen. Unter guten Umständen, braucht man dann viele Attribute gar nicht zu überprüfen. Als wesentliches Hilfsmittel der vorgestellten Algorithmen wird sich das Maß der Entropie erweisen.

### 7.3.1 Das Szenario und Entscheidungsbäume

Wir fangen allgemein mit dem zu betrachtenden Szenario an. Gegeben ist eine Menge von Objekten, von denen man einige Eigenschaften (Attribute) kennt. Diese Eigenschaften kann man darstellen mit einer fest vorgegebenen Menge von $n$ Attributen. D.h. man kann jedes Objekt durch ein $n$-Tupel der Attributwerte darstellen.

**Definition 7.3.1.** *Wir definieren die wesentlichen Komponten des Szenarios:*

- *Es gibt eine endliche Menge $A$ von Attributen.*

- *Zu jedem Attribut $a \in A$ gibt es eine Menge von möglichen Werten $W_a$. Die Wertemengen seien entweder endlich, oder die reellen Zahlen: $\mathbb{R}$.*
- *Ein Objekt wird beschrieben durch eine Funktion $A \to \times_{a \in A} W_a$. Eine alternative Darstellung ist: Ein Objekt ist ein Tupel mit $|A|$ Einträgen, bzw. ein Record, in dem zu jedem Attribut $a \in A$ der Wert notiert wird.*
- *Ein* Konzept *$K$ ist repräsentiert durch eine Prädikat $P_K$ auf der Menge der Objekte. D.h. ein Konzept entspricht einer Teilmenge aller Objekte, nämlich der Objekte $o$, für die $P_K(o) =$ `True` ergibt.*

**Beispiel 7.3.2.** *Bücher könnte man beschreiben durch die Attribute: (Autor, Titel, Seitenzahl, Preis, Erscheinungsjahr). Das Konzept „billiges Buch" könnte man durch Preis $\leq 10$ beschreiben. Das Konzept „umfangreiches Buch" durch Seitenzahl $\geq 500$.*

Für die Lernverfahren nimmt man im Allgemeinen an, dass jedes Objekt zu jedem Attribut einen Wert hat, und daher der Wert „unbekannt" nicht vorkommt. Im Fall unbekannter Attributwerte muss man diese Verfahren adaptieren.

**Definition 7.3.3** (Entscheidungsbaum)**.** *Ein* Entscheidungsbaum *zu einem Konzept $K$ ist ein endlicher Baum, der an inneren Knoten zum Wert eines Attributes folgende Abfragen machen kann:*

- *bei reellwertigen Attributen gibt es die Alternativen $a \leq v$ oder $a > v$ für einen Wert $v \in \mathbb{R}$, Es gibt einen Teilbaum für Ja und einen für Nein.*
- *bei diskreten Attributen wird der exakte Wert abgefragt. Es gibt pro möglichem Attributwert einen Teilbaum*

*Die Blätter des Baumes sind mit Ja oder Nein markiert. Das entspricht der Antwort auf die Frage, ob das eingegebene Objekte zum Konzept gehört oder nicht.*

*Diskrete Attribute sollten pro Pfad im Baum nur einmal vorkommen, stetige Attribute können im Pfad mehrmals geprüft werden.*

*D.h. ein Entscheidungsbaum $B_K$ ist die Darstellung eines Algorithmus zum Erkennen, ob ein vorgelegtes Objekt $O$ zum Konzept $K$ gehört.*

Jeder Entscheidungsbaum definiert ein Konzept auf den Objekten. Die Entscheidungsbäume sind so definiert, dass für jedes Objekt nach Durchlauf des Entscheidungsbaumes ein Blatt mit Ja oder Nein erreicht wird.

Die Mengen der Objekte, bei denen der Pfad mit einem Ja endet, sind in diesem Konzept, die anderen Objekte nicht.

- Wenn es nur diskrete Attribute gibt, dann entsprechen die Konzepte genau den Entscheidungsbäumen: Zu jedem Konzept kann man offenbar eine (aussagenlogische) Formel in DNF angeben: die $a_1 = v_1 \wedge \ldots a_n = v_n$ als Konjunktion enthält, wenn

das Tupel $(v_1, \ldots, v_n)$ im Konzept enthalten ist. Diese kann man leicht in einen Entscheidungsbaum überführen.

- Bei Verwendung von reellwertigen Attributen kann nicht jedes Konzept durch einen endlichen Entscheidungsbaum beschrieben werden: z.B. alle geraden Zahlen. Auch in einfachen Fällen, in denen das Konzept durch $\bigcup I_i$, d.h. als Vereinigung von unendlich vielen reellen Intervallen, dargestellt ist, gilt das.

**Beispiel 7.3.4.** *Als praktische Anwendung kann man reale Konzepte mittels einer endlichen Menge von Attributwerten bezüglich einer vorher gewählten Menge von Attributen beschreiben. Das ist i.A. eine Approximation des realen Konzepts.*

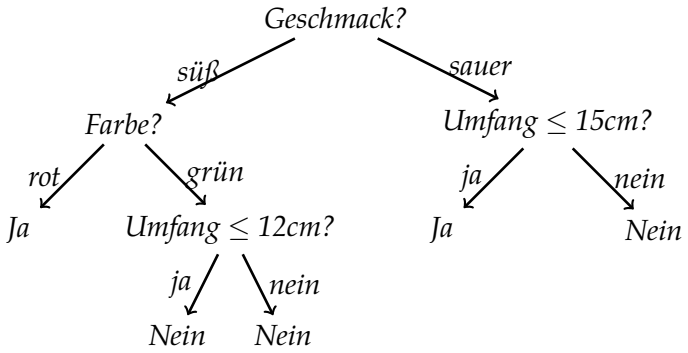
*Tiere könnte man z.B. durch folgende Attribute beschreiben:*

| | |
|---|---|
| *Größe* | *reell* |
| *Gewicht* | *reell* |
| *Kann fliegen* | *Boolesch* |
| *Nahrung* | *pflanzlich / tierisch / Allesfresser* |
| *Körpertemperatur* | *reell* |

*Für die Menge der Insekten könnte man aufgrund dieser Attribute einen Entscheidungsbaum hinschreiben, allerdings würden dann auch Nichtinsekten mit Ja klassifiziert.*

**Beispiel 7.3.5.** *Seien die Objekte Äpfel mit Attributen: Geschmack (süß/sauer), Farbe (rot/grün), Umfang (in cm).*

*Ein Entscheidungsbaum zum Konzept: „guter Apfel“ könnte sein:*




*Man sieht schon, dass dieser Entscheidungsbaum überflüssige Abfragen enthält: Da die Frage nach „Umfang ≤ 12cm?“ nur in „Nein“-Blättern endet, kann die Abfrage direkt durch ein „Nein“-Blatt ersetzt werden.*

Es gibt verschiedene Algorithmen, die die Aufgabe lösen sollen, einen Entscheidungsbaum für ein Konzept zu lernen, wobei man beispielsweise eine Menge von positiven Beispielen und eine Menge von negativen Beispielen vorgibt.

Ein *guter Entscheidungsbaum* ist z.B. ein möglichst kleiner, d.h. mit wenigen Fragen.

Der im Folgenden verwendetet Entropie-Ansatz bewirkt, dass das Verfahren einen Entscheidungsbaum erzeugt der eine *möglichst kleine mittlere Anzahl* von Anfragen bis zur Entscheidung benötigt. Einen Beweis dazu lassen wir weg. Das Verfahren ist verwandt zur Konstruktion von Huffman-Bäumen bei Kodierungen.

### 7.3.2 Lernverfahren ID3 und C4.5

Es wird angenommen, dass alle Objekte vollständige Attributwerte haben, und dass es eine Menge von positiven Beispielen und eine Menge von negativen Beispielen für ein zu lernendes Konzept gibt, die möglichst gut die echte Verteilung abbilden. Für rein positive Beispielmengen funktionieren diese Verfahren nicht.

Wichtig für die Lernverfahren ist es, herauszufinden, welche Attribute für das Konzept irrelevant bzw. relevant sind. Nachdem ein Teil des Entscheidungsbaumes aufgebaut ist, prüfen die Lernverfahren die Relevanz weiterer Attribute bzw. Attributintervalle.

Das Lernverfahren ID3 (Iterative Dichotomiser 3) iverwendet den Informationsgehalt der Attribute bezogen auf die Beispielmenge. Der Informationsgehalt entspricht der mittleren Anzahl der Ja/Nein-Fragen, um ein einzelnes Objekt einer Klasse zuzuordnen. Das Lernverfahren versucht herauszufinden, welche Frage den größten Informationsgewinn bringt, wobei man sich genau auf die in einem Entscheidungsbaum erlaubten Fragen beschränkt. Das Ziel ist daher die mittlere Anzahl der Fragen möglichst klein zu halten.

Sei $M$ eine Menge von Objekten mit Attirbuten. Wir berechnen den Informationsgehalt der Frage, ob ein Beispiel positiv/negativ ist in der Menge aller positiven / negativen Beispiele. Sei $p$ die Anzahl der positiven und $n$ die Anzahl der negativen Beispiele für das Konzept. Wir nehmen eine Gleichverteilung unter den Beispielen an, d.h. wir nehmen an, dass die relative Häufigkeit die reale Verteilung in den Beispielen widerspiegelt. Die Entropie bzw. der Informationsgehalt ist dann:

$$I(M) = \frac{p}{p+n} * \log_2(\frac{p+n}{p}) + \frac{n}{p+n} * \log_2(\frac{p+n}{n})$$

Sei $a$ ein Attribut. Wir schreiben $m(a)$ für den Wert des Attributs $a$ eines Objekts $m \in M$. Hat man ein mehrwertiges Attribut $a$ mit den Werten $w_1, \ldots, w_n$ abgefragt, dann zerlegt sich die Menge $M$ der Beispiele in die Mengen $M_i := \{m \in M \mid m(a) = w_i\}$, wobei $w_i, i = 1, \ldots, k$ die möglichen Werte des Attributes sind. Seien $p_i, n_i$ für $i = 1, \ldots, k$ die jeweilige Anzahl positiver (negativer) Beispiele in $M_i$, dann ergibt sich nach Abfragen des Attributs an Informationsgehalt (bzgl. positiv/negativ), wobei $I(M_i)$ der Informationsgehalt (bzgl. positiv/negativ) der jeweiligen Menge $M_i$ ist.

$$I(M|a) = \sum_{i=1}^{k} P(a = w_i) * I(M_i)$$

D.h. es wird der nach relativer Häufigkeit gewichtete Mittelwert der enstandenen Infor-

mationsgehalte der Kinder berechnet (was analog zu einer bedingten Wahrscheinlichkeit nun ein bedingter Informationsgehalt ist). Für jedes solches Kind mit Menge $M_i$ gilt wiederum:

$$I(M_i) = \frac{p_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{p_i}) + \frac{n_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{n_i})$$

Insgesamt können wir daher $I(M|a)$ berechnen als:

$$I(M|a) = \sum_{i=1}^{k} \frac{p_i + n_i}{p + n} * \Big(\frac{p_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{p_i}) + \frac{n_i}{p_i + n_i} * \log_2(\frac{p_i + n_i}{n_i})\Big)$$

Im Falle, dass $M_i = \emptyset$ ist, nehmen wir an, dass der Beitrag zur Summe $0$ ist. Um Fallunterscheidungen zu vermeiden, nehmen wir an, dass Produkte der Form $\frac{0}{a} * \log_2(\frac{a}{0})$ als $0$ zählen. Das ist berechtigt, da der Grenzwert von $\lim_{x \to 0} x * \log_2(x) = 0$ ist.

**Definition 7.3.6** (ID3: Entscheidungsbaum Lernen)**.** *ID3 startet mit einem leeren Baum und als Eingabe einer Menge $M$ von positiven und negativen Beispielen. Die Wurzel ist am Anfang der einzige* offene Knoten *mit Menge $M$.*

*Die Abbruchbedingung für einen offenen Knoten ist: Die Menge $M$ besteht nur noch aus positiven oder nur noch aus negativen Beispielen. In diesem Fall wird der Knoten geschlossen, indem er mit „Ja“ bzw. „Nein“ markiert wird. Ein unschöner Fall ist, dass die Menge $M$ leer ist: In diesem Fall muss man auch abbrechen, kennt aber die Markierung des Blattes nicht, man hat beide Möglichkeiten Ja bzw. Nein.*

*Für jeden offenen Knoten mit mit Menge $M$ wird jeweils das Attribut ausgewählt, das den größten Informationsgewinn bietet. D.h. dasjenige $a$, für das der*

$$\text{Informationsgewinn: } I(M) - I(M|a)$$

*maximal ist. Der offene Knoten wird nun geschlossen, indem das entsprechende Attribut am Knoten notiert wird und für jede Attributsausprägung wird ein Kind samt der entsprechenden Menge von Objekten berechnet. Die enstandenen Knoten sind offene Knoten und werden nun im Verfahren rekursiv (mit angepassten Menge $M$) bearbeitet.*

Beachte, wenn die Abbruchbedingung erfüllt ist, ist der Informationsgehalt an diesem Blatt 0. Normalerweise gibt es eine weitere praktische Verbesserung: Es gibt eine Abbruchschranke: wenn der Informationsgewinn zu klein ist für alle Attribute, dann wird der weitere Aufbau des Entscheidungsbaum an diesem Knoten abgebrochen, und es wird „Ja“ oder „Nein“ für das Blatt gewählt.

Anmerkungen zum Verfahren:

- Durch diese Vorgehensweise wird in keinem Ast ein diskretes Attribut zweimal abgefragt, da der Informationsgewinn 0 ist.

- Der Algorithmus basiert auf der Annahme, dass die vorgegebenen Beispiele repräsentativ sind. Wenn dies nicht der Fall ist, dann weicht das durch den Entscheidungsbaum definierte Konzept evtl. vom intendierten Konzept ab.

- Wenn man eine Beispielmenge hat, die den ganzen Tupelraum abdeckt, dann wird genau das Konzept gelernt.

**Beispiel 7.3.7.** *Wir nehmen als einfaches überschaubares Beispiel Äpfel und die Attribute Geschmack* $\in \{süß, sauer\}$ *und Farbe* $\in \{rot, grün\}$*. Das Konzept sei „guter Apfel".*

*Es gibt vier Varianten von Äpfeln,* $\{(süß, rot), (süß, grün), (sauer, rot), (sauer, grün)\}$*. Wir geben als Beispiel vor, dass die guten Äpfel genau* $\{(süß, rot), (süß, grün)\}$ *sind. Wir nehmen mal an, dass pro Apfelvariante genau ein Apfel vorhanden ist.*
*Es ist offensichtlich, dass die guten genau die süßen Äpfel sind, und die Farbe egal ist. Das kann man auch nachrechnen, indem man den Informationsgewinn bei beiden Attributen berechnet:*

*Der Informationsgehalt* $I(M)$ *vor dem Testen eines Attributes ist:*

$$0.5log_2(2) + 0.5log_2(2) = 1$$

*Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt* $0,5 * (log_2(1) + 0) + 0,5 * (0 + log_2(1)) = 0$*, d.h. Der Informationsgewinn ist maximal.*

*Nach dem Testen des Attributes „Farbe" ergibt sich als Informationsgehalt* $0,5 * (0,5log_2(2) + 0,5log_2(2)) + 0,5 * (0,5log_2(2) + 0,5log_2(2)) = 0,5 * 1 + 0,5 * 1 = 1$*, d.h. man gewinnt nichts.*

*Als Variation des Beispiels nehmen wir irgendeine Anzahl der Äpfeln in jeder Kategorie an:*

| *süß,rot* | *süß,grün* | *sauer,rot* | *sauer,grün* |
|---|---|---|---|
| *10* | *20* | *4* | *6* |

*D.h. es gibt 30 gute und 10 schlechte Äpfel.*
*Der Informationsgehalt ist vor dem Testen:*

$$0.75log_2(1,333) + 0.25log_2(4) \approx 0,311 + 0,5 = 0,811$$

*Nach dem Testen des Attributs „Geschmack" ergibt sich:*

$$\begin{aligned} & \frac{30}{40} * \left(\frac{30}{30}\log_2(1) + \frac{0}{30}\log_2(0)\right) \\ + \quad & \frac{10}{40} * \left(\frac{10}{10}\log_2(1) + \frac{0}{10}\log_2(0)\right) \\ = \quad & 0 \end{aligned}$$

*d.h. Der Informationsgewinn ist maximal.*
*Im Falle, dass die Farbe getestet wird, ergibt sich:*

$$\frac{14}{40} * 0,8631 + \frac{26}{40} * 0,7793) \approx 0,80867\ldots$$

*D.h. ein minimaler Informationsgewinn ist vorhanden. Der kommt nur aus der leicht unterschiedlichen Verteilung der guten Äpfel innerhalb der roten und grünen Äpfel und innerhalb aller Äpfel. Genauer gesagt: der Gewinn kommt daher, dass die Beispielmenge der 40 Äpfel nicht genau die Wahrheit abbildet.*

*Wird die Wahrheit richtig abgebildet, d.h. sind die Verteilungen gleich, dann:*

| *süß,rot* | *süß,grün* | *sauer,rot* | *sauer,grün* |
|---|---|---|---|
| *10* | *20* | *3* | *6* |

*Dann ergibt sich 0.7783 als Entropie $I(M)$ und auch für die Auswahl der Farbe danach, d.h. es gibt keinen Informationsgewinn für das Attribut Farbe.*

**Beispiel 7.3.8.** *Wir erweitern das Beispiel der einfachen Äpfel um eine Apfelnummer. Der Einfachheit halber gehen die Nummern gehen von 1 bis 4. Zu beachten ist, dass dieses Attribut eine Besonderheit hat: es kann nicht der ganze Tupelraum ausgeschöpft werden, da es ja zu jeder Nummer nur einen Apfel geben soll. Das spiegelt sich auch in den prototypischen Beispielen:*

*Es gibt vier Äpfel, $\{(1, süß, rot), (2, süß, grün), (3, sauer, rot), (4, sauer, grün)\}$. Wir geben als Beispiel vor, dass die guten Äpfel gerade $\{(1, süß, rot), (2, süß, grün)\}$ sind. Wir rechnen den Informationsgewinn der drei Attribute aus.*

*Der Informationsgehalt $I(M)$ vor dem Testen eines Attributes ist:*

$$0.5log_2(2) + 0.5log_2(2) = 1$$

*Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt $0,5 * (log_2(1) + 0) + 0,5 * (0 + log_2(1)) = 0$, d.h. Der Informationsgewinn ist maximal.*

*Nach dem Testen des Attributes „Farbe" ergibt sich als Informationsgehalt $0,5 * (0,5log_2(2) + 0,5log_2(2)) + 0,5 * (0,5log_2(2) + 0,5log_2(2)) = 0,5 * 1 + 0,5 * 1 = 1$, d.h. man gewinnt nichts.*

*Nach dem Testen des Attributs „Nummer" ergibt sich $\sum 1 * log(1)$, somit insgesamt $0$. Der Informationsgewinn ist ebenfalls maximal. Weiter unten werden wie sehen, dass der Informationsgewinn in diesen sinnlosen Fällen durch Normierung kleiner wird.*

**Beispiel 7.3.9.** *Wir nehmen als erweitertes Beispiel für Äpfel und die Attribute Geschmack $\in \{süß, sauer\}$ und Farbe $\in \{rot, gelb, grün\}$. Das Konzept sei „schmeckt-wie-er-aussieht". Es gibt sechs Kombinationen der Attribute: $\{(süß, rot), (süß, grün), (süß, gelb), (sauer, rot), (sauer, grün), (sauer, gelb)\}$. Wir geben als Beispiel die Menge $\{(süß, rot), (sauer, grün), (süß, gelb), (sauer, gelb)\}$ vor. Wir berechnen den Informationsgewinn bei beiden Attributen:*

*Der Informationsgehalt $I(M)$ vor dem Testen irgendeines Attributs ist:*

$$4/6 * log_2(6/4) + 1/3log_2(3) = 0.9183$$

$$4/6 * log_2(6/4) + 1/3log_2(3) = 0.9183$$

*Nach dem Testen des Attributes „Geschmack" ergibt sich als Informationsgehalt $I(süss) = I(sauer) \approx 0.9183$. Die Gesamtinformation nach Abfrage des Attributs „Geschmack" ist:* $0.5 * 0.9183 + 0.5 * 0.9183 = 0.9183$ .

*d.h. Der Informationsgewinn ist null.*

*Nach dem Testen des Attributes „Farbe" ergibt sich als I(grün) = I(rot) = 1, I(gelb) = 0. Die Gesamtinformation nach Abfrage der Farbe ist:* $1/3 * 1 + 1/3 * 1 = 2/3 \approx 0.667$. *D.h. man hat Information gewonnen. Im Endeffekt muss man bei diesem Beispiel doch beide Attribute abfragen*

Man kann das Verfahren auch für reellwertige Attribute verwenden, wobei man als Grenzabfrage „$> w?$" nur endlich viele Werte ausprobieren muss, die sich aus den Werten der entsprechenden Attribute in den Beispielen ergeben. Es ist klar, dass ein Konzept wie „Fieber" aus den aktuell gegebenen Temperaturen und der Klassifizierung Fieber j/n nur annähernd gelernt werden kann.

Die Methode ID3 funktioniert recht gut, aber wenn ein Attribut zuviele Ausprägungen hat, wird die Frage nach diesem Attribut bevorzugt, da es im Extremfall (Personalnummer. o.ä.) dazu kommen kann, dass die Mengen $\{m \in M \mid m(a) = v\}$ einelementig werden, und somit der Informationsgewinn maximal ist.

#### 7.3.2.1 C4.5 als verbesserte Variante von ID3

Das von Quinlan vorgeschlagene System C4.5 benutzt statt des Informationsgewinns einen normierten Informationsgewinn, wobei der obige Wert durch die Entropie des Attributs (d.h. der Verteilung bzgl. der Attributwerte) dividiert wird. Somit vergleicht man Attribute anhand

$$\text{Informationsgewinn} * \text{Normierungsfaktor} \qquad D.h.$$

$$\big(I(M) - I(M \mid a)\big) * \text{Normierungsfaktor}$$

Das bewirkt, dass Attribute mit mehreren Werten nicht mehr bevorzugt werden, sondern fair mit den zweiwertigen Attributen verglichen werden. Ohne diese Normierung werden mehrwertige Attribute bevorzugt, da diese implizit mehrere Ja/Nein-Fragen stellen dürfen, während ein zweiwertiges Attribut nur einer Ja/Nein-Frage entspricht. Dieser Vorteil wird durch den Normierungsfaktor ausgeglichen, der den Informationsgewinn auf binäre Fragestellung normiert, d.h. dass man den Informationsgewinn durch ein Attribut mit 4 Werten durch 2 dividiert, da man 2 binäre Fragen dazu braucht.

Der Normierungsfaktor für ein Attribut $a$ mit den Werten $w_i, i = 1, \ldots, k$ ist:

$$\frac{1}{\sum_{i=1}^{k} P(a = w_i) * \log_2(\frac{1}{P(a = w_i)})}$$

Bei einem Booleschen Attribut, das gleichverteilt ist, ergibt sich als Normierungsfaktor $0,5 * 1 + 0,5 * 1 = 1$, während sich bei einem Attribut mit $n$ Werten, die alle gleichverteilt sind, der Wert

$$\frac{1}{n * \frac{1}{n} * \log_2(n)} = \frac{1}{\log_2(n)}$$

ergibt.

Durch diese Vorgehensweise wird die Personalnummer und auch die Apfelnummer als irrelevantes Attribut erkannt. Allerdings ist es besser, diese Attribute von vorneherein als irrelevant zu kennzeichnen, bzw. erst gar nicht in die Methode einfließen zu lassen.

**Beispiel 7.3.10.** *Im Apfelbeispiel s.o. ergibt sich bei Hinzufügen eines Attributes Apfelnummer mit den Ausprägungen* $1, 2, 3, 4$, *als Normierungsfaktor für Apfelnummer:*

$$\frac{1}{\frac{1}{4} * 2 + \ldots + \frac{1}{4} * 2} = 0.5$$

*Damit wird die Abfrage nach dem Geschmack vor der Apfelnummer bevorzugt.*

#### 7.3.2.2 Übergeneralisierung (Overfitting)

Tritt auf, wenn die Beispiele nicht repräsentativ sind, oder nicht ausreichend. Der Effekt ist, dass zwar die Beispiele richtig eingeordnet werden, aber der Entscheidungsbaum zu fein unterscheidet, nur weil die Beispiele (zufällig) bestimmte Regelmäßigkeiten aufweisen.

**Beispiel 7.3.11.** *Angenommen, man will eine Krankheit als Konzept definieren und beschreibt dazu die Symptome als Attribute:*
*Fieber: Temperatur, Flecken: j/n, Erbrechen: j/n, Durchfall: j/n, Dauer der krankheit: Zeit, Alter des Patienten, Geschlecht des Patienten, ....*
*Es kann dabei passieren, dass das Lernverfahren ein Konzept findet, dass beinhaltet, dass Frauen zwischen 25 und 30 Jahren diese Krankheit nicht haben, nur weil es keine Beispiele dafür gibt. Auch das ist ein Fall von overfitting.*

*Besser wäre es in diesem Fall, ein Datenbank aller Fälle zu haben. Die Erfahrung zeigt aber, dass selbst diese Datenbank aller Krankheiten für zukünftige Fragen oft nicht ausreicht, da nicht jede Frage geklärt werden kann: z.B. Einfluss des Gendefektes XXXXX auf Fettsucht.*

**Abschneiden des Entscheidungsbaumes: Pruning**

Beheben kann man das dadurch, dass man ab einer gewissen Schranke den Entscheidungsbaum nicht weiter aufbaut, und den weiteren Aufbau an diesem Knoten stoppt: **Abschneiden des Entscheidungsbaumes (Pruning)**.

Wenn kein Attribut mehr einen guten Informationsgewinn bringt, dann besteht der Verdacht, dass alle weiteren Attribute eigentlich irrelevant sind, und man das Verfahren an dem Blatt stoppen sollte. Dies kann man bei bekannter Verteilung mittels eines statistischen Test abschätzen.

Hierbei ist es i.a. so, dass es an dem Blatt, an dem abgebrochen wird, noch positive und negative Beispiele gibt. Die Markierung des Knoten wählt man als Ja, wenn es signifikant mehr positive als negative Beispiel gibt, und als Nein, wenn es signifikant mehr negative als positive Beispiel gibt. Das ist natürlich nur sinnvoll, wenn man weiß, das es falsche Beispiele geben kann.

Hat man verrauschte Daten, z.B. mit Messfehler behaftete Beispiele, dann ist Lernen von Entscheidungsbäumen mit Pruning die Methode der Wahl.

## 7.4 Versionenraum-Lernverfahren

Wir betrachten ein weiteres Verfahren zum Lernen eines Konzepts über Objekten mit mehrwertigen Attributen, wobei wir davon ausgehen, dass Objekte durch diskrete Attribute und deren Werte beschrieben werden, aber das die Konzeptsprache noch Subsumtionsalgorithmen auf den Konzeptbeschreibungen zur Verfügungstellt. D.h. es gibt Algorithmen, die die Teilmengenbeziehung von Konzepten entscheiden können.

Wir stellen Objekte in Tupelschreibweise dar $(w_1, \ldots, w_n)$, wobei $w_i$ der Wert des $i$. Attributs $A_i$ ist. Betrachte z.B. Äpfel mit den Attributen Farbe (mit den Ausprägungen rot, grün, gelb), Geschmack (mit den Ausprägungen süß, sauer), Herkunft (mit den Ausprägungen Deutschland, Spanien, Italien) und Größe (mit den Ausprägunen S,M,L). Dann stellen wir einen roten, süßen Apfel aus Spanien mittlerer Größe als ⟨ rot, süß, Spanien, M ⟩ dar.

Hypothesen (und auch Konzepte) sind Mengen von Objekten repräsentiert durch die folgende Syntax: ⟨ rot, süß, Spanien, M ⟩ ist das Konzept, dass genau das Objekt ⟨ rot, süß, Spanien, M ⟩ enthält (also die einelementige Menge {⟨ rot, süß, Spanien, M ⟩}).

Wir erlauben anstelle eines Attributwerts auch ein Fragezeichen ?. So repräsentiert z.B. ⟨ rot, süß, ?, M ⟩ die Menge aller roten und süßen Äpfel mittlerer Größe (unabhängig von ihrer Herkunft). Sei $h$ eine Hypothese und $e$ ein Objekt, dann schreiben wir $e \in h$ falls $e$ in der Menge der Objekte liegt, die durch $h$ repräsentiert werden. Zusätzlich erlauben wir noch $\emptyset$ als Zeichen für einen Attributwert. Sobald ein solcher Wert im Tupel enthalten ist, repräsentiert es das leere Konzept (Hypothese) welches keine Objekte enthält[1].

Auf den Konzepten / Hypothesen ist eine Ordnung $\leq$ definiert, die Spezialisierung

[1] Es wäre hier auch möglich ein Symbol für das leere Konzept zu verwenden, anstatt die Kennzeichnung am Attributwert vorzunehmen

bzw. Generalisierung ausdrückt. z.B. gilt $\langle \emptyset, \text{sauer} \rangle < \langle ?, \text{sauer} \rangle < \langle ?, ? \rangle$ Diese Ordnung ist i.a. nicht total.

Sei $B$ eine Menge von Beispielen (Objekten) und $h$ eine Hypothese. Dann nennen wir $h$ konsistent für $B$ wenn für alle $b \in B$ gilt:

- Wenn $b$ positives Beispiel ist, dann gilt $b \in h$.
- Wenn $b$ negatives Beispiel ist, dann gilt $b \notin h$.

Das Versionenraum-Lernverfahren ist ein inkrementelles Lernverfahren, d.h. die positiven und negativen Beispiele werden einzeln abgearbeitet und die Hypothese jeweils verfeinert.

Die Grundidee dabei ist recht einfach:

Aus der Menge aller Hypothesen, merke als Zustand, die maximale Menge der Hypothesen, die den bisherig gesehenen Beispielen nicht widerspricht. D.h. die Menge enthält die Hypothesen, die konsistent mit den bisherigen Beispielen sind und maximal ist: Alle bisher gesehenen *positiven* Beispiele müssen in jeder Hypothese der Menge enthalten sein, alle bisher gesehenen *negativen* Beispiele sind in keiner Hypothese der Menge enthalten.

Die so beschriebene Menge nennt man *Versionenraum*. Das zugehörige Lernverfahren verwaltet den Versionenraum und nimmt Anpassungen für jedes neue Beispiel vor. Das Verfahren ist inkrementell, da alte Beispiele nie wieder betrachtet werden müssen.

Zur kompakten Repräsentation des Versionenraums werden zwei Mengen von Hypothesen verwaltet:

- die untere Grenze des Versionenraums $S$ (die speziellsten Hypothesen): Für jede Hypothese $h \in S$ gilt: $h$ ist konsistent für die Menge der bisher gesehenen Beispiele und es gibt kein $h'$ mit $h' < h$ das konsistent für die Menge der bisher gesehenen Beispiele ist.
- die obere Grenze des Versionenraums $G$ (die allgemeinsten Hypothesen): Für jede Hypothese $h \in G$ gilt: $h$ ist konsistent für die Menge der bisher gesehenen Beispiele und es gibt kein $h'$ mit $h' > h$ das konsistent für die Menge der bisher gesehenen Beispiele ist.

Wenn $S = \{s_1, \ldots, s_n\}$ und $G = \{g_1, \ldots, g_m\}$, dann ist der Versionenraum genau: $\{h \mid \exists i, j : s_i \leq h \leq g_j\}$.

Bei Eingabe eines neuen Beispiels $e$ werden die Mengen $S$ und $G$ neu berechnet:

Wenn $e$ ein positives Beispiel ist:

- Die Hypothesen $h \in S \cup G$ mit $e \in h$ müssen nicht verändert werden.
- Für jedes $h \in S$ mit $e \notin h$: $h$ ist zu speziell und muss verallgemeinert werden. Ersetze $h$ durch alle $h'$ wobei

  - $h'$ ist eine minimale Verallgemeinerung von $h$ bzgl. $e$ , d.h. $h' > h$, $e \in h'$ und es gibt keine Verallgemeinerung $h''$ von $h$ bzgl. $e$ mit $h' > h''$.
  - Zudem werden nur solche $h'$ eingefügt, die nicht zu allgemein sind, d.h. es muss eine Hypothese $g \in G$ geben mit $g \geq h'$.

  Schließlich (um $S$ kompakt zu halten): Entferne alle Hypothesen aus $S$ die echte allgemeiner sind als andere Hypothesen aus $S$

- Für jedes $h \in G$ mit $e \notin h$: $h$ ist zu speziell aber $h$ kann nicht verallgemeinert werden (da $h \in G$). Daher: Lösche $h$ aus $G$.

Wenn $e$ ein negatives Beispiel ist:

- Die Hypothesen $h \in S \cup G$ mit $e \notin h$ müssen nicht verändert werden.
- Für jedes $h \in S$ mit $e \in h$: $h$ ist zu allgemein aber $h$ kann nicht spezialisiert werden (da $h \in S$). Daher: Lösche $h$ aus $S$.
- Für jedes $h \in G$ mit $e \in h$: $h$ ist zu allgemein und muss spezialisiert werden. Ersetze $h$ durch alle $h'$ wobei
  - $h'$ ist eine minimale Spezialisierung von $h$ bzgl. $e$, d.h. $h' < h$, $e \notin h'$ und es gibt keine Spezialisierung $h''$ von $h$ bzgl. $e$ mit $h'' > h'$.
  - Zudem werden nur solche $h'$ eingefügt, die nicht zu speziell sind, d.h. es muss eine Hypothese $s \in S$ geben mit $s \leq h'$.

  Schließlich (um $G$ kompakt zu halten): Entferne alle Hypothesen aus $G$ die echte spezieller sind als andere Hypothesen aus $G$

Das Versionenraum-Lernverfahren startet mit der größtmöglichen Menge uns setzt daher:

- $S$ enthält die speziellste Hypothese: $S = \{\langle \emptyset ?, \ldots, ? \emptyset \rangle\}$
- $G$ enthält die allgemeinste Hypothese: $G = \{\langle ?, \ldots, ? \rangle\}$

Mögliche Ausgänge des Algorithmus sind:

- Wenn $S = G$ und $S, G$ einelementig, d.h. $S = G = \{h\}$ dann ist $h$ das gelernte Konzept
- Wenn $S$ oder $G$ ist die leere Menge: In diesem Fall ist der Versionenraum kollabiert. D.h. es gibt keine konsistente Hypothese für die vorgegebenen Trainingsbeispiele.
- $S$ und $G$ sind nicht leer aber auch nicht einelementig und gleich: Der aufgespannte Versionenraum enthält das Konzept aber dieses ist nicht eindeutig identifizierbar. D.h. alle Hypothesen im Versionenraum sind konsistent.

**Beispiel 7.4.1.** *Wir betrachten die Apfel-Objekte und die folgende Beispielmenge:*
(*süß,rot,Italien,L*)$^+$
(*süß,gelb,Spanien,S*)$^-$
(*süß,gelb,Spanien,L*)$^+$
(*sauer,rot,Deutschland,L*)$^-$
*Hierbei sind die* $^+$ *markierten Beispiele positiv und die mit* $^-$ *markierten Beispiele negativ.*
*Das Versionenraum-Lernverfahren startet mit*

$$S = \{\langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\}$$
$$G = \{\langle ?, ?, ?, ? \rangle\}$$

1. *Einfügen des Beispiels* (*süß,rot,Italien,L*)$^+$*:*
   - *Da* $(süß,rot,Italien,L) \in \langle ?, ?, ?, ? \rangle$*, wird die Menge* $G$ *nicht verändert.*
   - *Da* $(süß,rot,Italien,L) \notin \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle$ *muss diese Hypothese der Menge* $S$ *verallgemeinert werden. Die minimale Verallgemeinerung ist* $\langle süß,rot,Italien,L \rangle$ *und daher wird* $S$ *neu gesetzt als* $S = \{\langle süß,rot,Italien,L \rangle\}$*.*

2. *Einfügen des Beispiels* (*süß,gelb,Spanien,S*)$^-$*:*
   - *Da* $(süß,gelb,Spanien,S) \notin \langle süß,rot,Italien,L \rangle$*, wird* $S$ *nicht verändert.*
   - *Da* $(süß,gelb,Spanien,S) \in \langle ?, ?, ?, ? \rangle$*, muss diese Hypothese aus* $G$ *spezialisiert werden. Minimale Spezialisierungen sind:*
     $\langle sauer, ?, ?, ? \rangle$ $\langle ?,rot,?, ? \rangle$ $\langle ?,gruen,?, ? \rangle$ $\langle ?, ?,Italien,? \rangle$
     $\langle ?, ?,Deutschland,? \rangle$ $\langle ?, ?, ?,L \rangle$ $\langle ?, ?, ?,M \rangle$

     *Davon sind jedoch einige zu speziell, da es keine Hypothese* $s \in S$ *gibt, die spezieller oder gleich ist. Daher verbleiben nur* $\langle ?,rot,?, ? \rangle$*,* $\langle ?, ?,Italien,? \rangle$ *und* $\langle ?, ?, ?,L \rangle$*. Wir setzen daher* $G = \{\langle ?,rot,?, ? \rangle, \langle ?, ?,Italien,? \rangle, \langle ?, ?, ?,L \rangle\}$*.*

3. *Einfügen des Beispiels* (*süß,gelb,Spanien,L*)$^+$*:*
   - *Da* $(süß,gelb,Spanien,L) \notin \langle ?,rot,?, ? \rangle \in G$ *muss die Hypothese gelöscht werden.*
   - *Da* $(süß,gelb,Spanien,L) \notin \langle ?, ?,Italien,? \rangle \in G$ *muss die Hypothese gelöscht werden.*
   - *Da* $(süß,gelb,Spanien,L) \in \langle ?, ?, ?,L \rangle \in G$ *bleibt Hypothese in* $G$*.*
   - *Ergibt* $G = \{\langle ?, ?, ?,L \rangle\}$
   - *Da* $(süß,gelb,Spanien,L) \notin \langle süß,rot,Italien,L \rangle \in S$ *muss die Hypothese verallgemeinert werden. Die minimale Verallgemeinerung ist:* $\langle süß,?,?,L \rangle$ *und daher* $S = \{\langle süß,?,?,L \rangle\}$

4. *Einfügen des Beispiels* (*sauer,rot,Deutschland,L*)$^-$*:*
   - *Da* $(sauer,rot,Deutschland,L) \notin \langle süß,?,?,L \rangle$*, bleibt* $S$ *unverändert.*

- *Da (sauer,rot,Deutschland,L) $\in \langle ?,?,?,L\rangle$, muss die Hypothese spezialisiert werden.Minimale Spezialiserungen sind:*
  $\langle$*süß,?,?,L*$\rangle$ $\langle$*?,gelb,?,L*$\rangle$ $\langle$*?,grün,?,L*$\rangle$
  $\langle$*?,?,Italien,L*$\rangle$ $\langle$*?,?,Spanien,L*$\rangle$
  *jedoch ist nur* $\langle$*süß,?,?,L*$\rangle$ *nicht zu speziell. Daher* $G = \langle$*süß,?,?,L*$\rangle$

*Alle Beispiele sind abgearbeitet und es gilt* $S = G = \{\langle$*süß,?,?,L*$\rangle\}$. *Daher ist* $\langle$*süß,?,?,L*$\rangle$ *das gelernte Konzept.*

**Beispiel 7.4.2.** *Betrachte als weiteres Beispiel die Menge:*
(*süß,rot,Italien,L*)$^+$
(*süß,gelb,Italien,L*)$^-$
*Das Versionenraum-Lernverfahren startet mit*

$$S = \{\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle\}$$
$$G = \{\langle ?, ?, ?, ?\rangle\}$$

1. *Einfügen des Beispiels (süß,rot,Italien,L)$^+$:*
   - *Da (süß,rot,Italien,L)* $\in \langle ?, ?, ?, ?\rangle$, *wird die Menge G nicht verändert.*
   - *Da (süß,rot,Italien,L)* $\notin \langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$ *muss diese Hypothese der Menge S verallgemeinert werden. Die minimale Verallgemeinerung ist* $\langle$*süß,rot,Italien,L*$\rangle$ *und daher wird S neu gesetzt als* $S = \{\langle$*süß,rot,Italien,L*$\rangle\}$.
2. *Einfügen des Beispiels (süß,gelb,Italien,L)$^-$:*
   - *Da (süß,gelb,Italien,L)* $\notin \langle$*süß,rot,Italien,L*$\rangle$ *wird S nicht verändert.*
   - *Da (süß,gelb,Italien,L)* $\in \langle ?, ?, ?, ?\rangle$, *muss die Hypothese spezialisiert werden. Minimale Spezialisierungen sind:*
     $\langle$*sauer*$, ?, ?, ?\rangle$ $\langle$*?,grün*$, ?, ?\rangle$ $\langle ?,$*rot*$, ?, ?\rangle$ $\langle ?, ?,$*Deutschland*$, ?\rangle$
     $\langle ?, ?,$*Spanien*$, ?\rangle$ $\langle ?, ?, ?,$*S*$\rangle$ $\langle ?, ?, ?,$*M*$\rangle$
     *Davon verbleibt nur* $\langle ?,$*rot*$, ?, ?\rangle$, *da die anderen zu speziell sind. Daher setzen wir* $G = \{\langle ?,$*rot*$, ?, ?\rangle\}$.

*Alle Beispiele wurden abgearbeitet und*
$S = \{\langle$*süß,rot,Italien,L*$\rangle\}$.
$G = \{\langle ?,$*rot*$, ?, ?\rangle\}$
*Das bedeutet, dass alle Hypothesen h mit*

$$\langle \text{süß,rot,Italien,L}\rangle \leq h \leq \langle ?,\text{rot}, ?, ?\rangle$$

*konsistent mit der Beispielmenge sind.*

**Beispiel 7.4.3.** *Betrachte das vorherige Beispiel, erweitert um ein weiteres Beispiel:*

$(süß,rot,Italien,L)^+$
$(süß,gelb,Italien,L)^-$
$(süß,grün,Italien,L)^+$
*Die Abarbeitung der ersten beiden Beispiele resultiert in:*
$S = \{\langle süß,rot,Italien,L\rangle\}$.
$G = \{\langle ?,rot,?,?\rangle\}$
*Wir betrachten das Einfügen von* $(süß,grün,Italien,L)^+$*:*

- *Da* $(süß,grün,Italien,L) \notin \langle ?,rot,?,?\rangle$ *und die Hypothesen in* $G$ *nicht verallgemeinert werden dürfen, wird die Hypothese gelöscht. D.h.* $G = \emptyset$
- *Da* $(süß,grün,Italien,L) \notin \langle süß,rot,Italien,L\rangle$*, muss das Konzept verallgemeinert werden zu* $\langle süß,?,Italien,L\rangle$*. Da es in* $G$ *jedoch keine Konzepte gibt, ist das Konzept (wie jegliches Konzept in diesem Fall) zu allgemein und wird gelöscht. Daher setze* $S = \emptyset$*.*

*Da* $G = S = \emptyset$ *gibt es keine konsistente Hypothese für die Beispielmenge. Der Grund liegt darin, dass die beiden Beispiele* $(süß,rot,Italien,L)^+$ *und* $(süß,grün,Italien,L)^+$ *als minimales Konzept das Konzept* $\langle süß,?,Italien,L\rangle$ *erfordern. Aber dieses minimale Konzept bereits das negative Beispiel* $(süß,gelb,Italien,L)^-$ *beinhaltet.*
*Abbhilfe kann hier nur eine mächtigere Konzeptsprache schaffen.*

Ein weiteres Problem der Versionenraum-Methode ist, dass sie nicht mit verrauschten Daten umgehen kann. In diesem Fall kann sie keine konsistente Hypothese finden.

Man kann Varianten der Konzeptsprachen verwenden die ausdruckstärker sind. Z.B. ist eine ausdrucksstärkere Konzeptsprache:

- Erlaube „$a = M_a$“ für Attribute a, wobei $M_a$ eine Teilmenge der möglichen Ausprägungen von $a$ ist ist. Damit kann man Quader im Objektraum erzeugen. Diese Sprache nennen wir *Quader-Konzepte*.
- Erlaube Disjunktionen der Quader-Konzepte. Damit kann man bereits alle Konzepte darstellen, wenn die Menge der Attribute und Ausprägungen endlich ist.

# 8

# Konzeptbeschreibungssprachen

In diesem Kapitel behandeln wir Ansätze zur Wissensrepräsentation und Wissensverarbeitung. Der Schwerpunkt liegt dabei auf sogenannten Beschreibungslogiken (Description Logics). Diese haben insbesondere den Vorteil, dass sie eine eindeutig definierte Semantik (innerhalb der Prädikatenlogik) besitzen und viele Inferenzmechanismen und -algorithmen existieren. Zunächst werden wir jedoch kurz auf die historische Entwicklung eingehen, und vorhergehende Ansätze der Semantischen Netze und sogenannter Frames betrachten. Im Anschluss beschäftigen wir uns eingehend mit den Beschreibungslogiken.

## 8.1 Ursprünge

In diesem Abschnitt stellen wir kurz die (älteren) Formalismen der Semantischen Netze und Frames vor, aus denen (und deren Nachteile) die Beschreibungslogiken entwickelt wurden. Wir geben hier keinen umfassenden Überblick über diese Ansätze (dafür sei auf entsprechende Literatur verwiesen), sondern der Abschnitt ist eher als kurzer Einblick zu sehen.

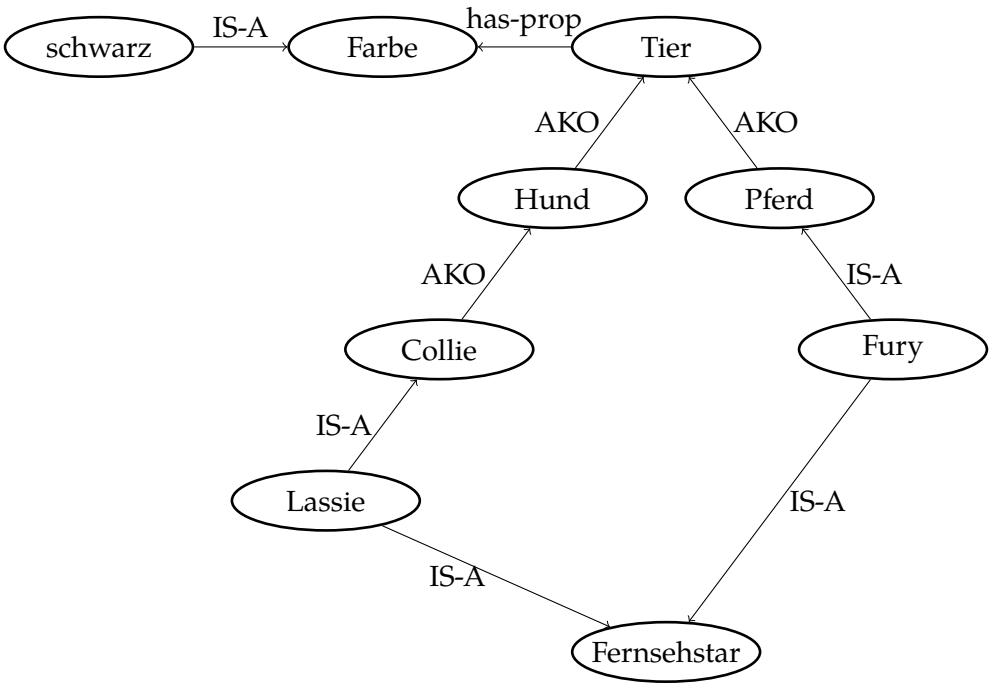
### 8.1.1 Semantische Netze

Semantische Netze sind ein Formalismus zur Darstellung von Unterbeziehungen und Enthaltensseinsbeziehungen mittels (gerichteten) Graphen. Dabei sind

**Knoten** markiert mit *Konzepten*, Eigenschaften, (tlw. auch Individuen)

**Gerichtete Kanten (Links)** geben Beziehungen (Relationen) zwischen Konzepten an. Wesentliche Kanten sind Instanzbeziehungen (IS-A), Unterkonzeptbeziehungen (A-KIND-OF, AKO), Eigenschaften (Prop), Part-of, connected-to, ...

Ein Beispiel ist das folgende Netz zu Tieren:




Allerdings gibt es viele verschiedene Möglichkeiten und Notationen, je nach festgelegtem Formalismen.

Besonderheiten der Semantischen Netze sind:

- Repräsentieren von Beziehungen und Eigenschaften ist möglich.
- Vererbung von Eigenschaften über AKO-links und über IS-A links, je nach Eigenschaft.

Man kann mit semantischen Netzen auch logische Formeln darstellen: UND-Verzweigungen, ODER-Verzweigungen, Quantifizierte Ausdrückebei letzteren entsteht ein Problem der Eindeutigkeit und des Bindungsbereichs.

#### 8.1.1.1 Operationen auf semantischen Netzen:

- Anfragen der Form „Was kann fliegen?“können mittels Matching von Teilgraphen und Beschreibungen bzw. Angaben von Teilnetzen beantwortet werden. D.h. die Suche orientiert sich an bestimmten Knoten oder Kanten oder an ganzen Unterstrukturen.
- Veränderung: Eintragung, Entfernen, Ändern von Kanten und Knoten.
- Operation: Suche nach Verbindungswegen im Netz.

**Bemerkung 8.1.1.** *Probleme der semantischen Netze:*

- *Die Semantik war nur ungenau definiert.*

- *Jedes Programm arbeitete auf einer anderen semantischen Basis. z.B. was bedeuten jeweils zirkuläre Links ?*
- *Darstellung als Graph wird sehr unübersichtlich für große Netze.*

### 8.1.2 Frames

*Frames* sind ein Konzept innerhalb von Repräsentationssprachen, zunächst ohne prozedurale Komponente. Die Grundlagen dazu stammen aus der Kognitionsforschung (Minsky, 1975) und den Scripts (Schenk & Abelson, 1975). Die Frame-Sprachen haben eine Analogie zu Klassen in Objektorientierten Programmiersprachen, allerdings ist die Absicht von Frames nicht ein Programm zu strukturieren, sondern einen (Wissens-)Bereich strukturiert darzustellen.

- Frames beschreiben Klassen oder Instanzen
  - Namen
  - Oberklasse(e)
  - Eigenschaften (Slots)
- Vererbung von Eigenschaften (Slot-Werten) von Oberklassen auf Unterklassen
- Slot:
  - Klasse (Wertebereich)
  - Defaultwerte
  - generische Werte (gilt für alle Instanzen)
  - Bedingungen (z.B. Wertebereichseinschränkungen)
  - Prozedurale Zusätze (z.B. Dämonen, die bei Eintragung eines Slotwertes aktiv werden)

Dies ergibt eine implizite Klassenhierarchie (sog. Prototypen). Zum Beispiel:

| | |
|---|---|
| Vogel | (Oberklasse: Wirbeltiere) |
| | (Farbe: Farbe , Gewicht: Zahl, kann-fliegen: Bool, … |
| | #Beine: 2) |
| grüne-Vögel | (Oberklasse: Vogel) |
| | (Farbe:grün) |

**Bemerkung 8.1.2.** *Diese Darstellung hat einige inhärente Probleme zu lösen:*

- *multiple Vererbung*
- *semantische Unterscheidung: Prototyp / individuelles Objekt*
- *logische Operatoren*
- *Semantik von Defaults und überschriebenen Defaults*

## 8.2 Attributive Konzeptbeschreibungssprachen (Description Logic)

Wir betrachten im folgenden eine Sprachfamilie, die man als Nachfolger von KL-ONE sehen kann. Empfehlenswert als sehr guter Überblick ist das Handbuch (Baader et al., 2010) und zur Komplexität der Artikel (Donini et al., 1997).

Man nennt die Konzeptbeschreibungssprachen auch *Terminologische Sprachen*. Sie haben sich entwickelt aus einer Sprache KL-ONE (Brachman, ca. 1980), die zur Repräsentation und Verarbeitung der Semantik von natürlichsprachlichen Ausdrücken entwickelt wurde. Es gibt ähnliche Strukturen bei der automatischen Verarbeitung der Syntax von natürlichsprachlichen Ausdrücken: Features (Attribute, bzw. Merkmalsstrukturen) (siehe Shieber: unification grammars).

Man kann die Konzeptbeschreibungssprachen als Weiterentwicklung von semantischen Netzen und Frames sehen, wobei sie den Vorteil einer eindeutigen und deklarativ definierten Semantik haben und nicht zu mächtige Konstrukte (wie z.B. Prozeduren) zulassen.

Neuere Entwicklungen sind (relativ schnelle) und vollständige Schlussfolgerungsmechanismen, Erweiterung in viele Richtungen wie Zeitrepräsentation, Kombination mit anderen Formalismen.

Ein auf Beschreibungslogik basierendes Wissensverarbeitungssystem muss Möglichkeiten bieten, die Wissensbasis darzustellen und zu verändern, sowie neue Schlüsse mithilfe von Inferenzmechanismen aufgrund der Wissenbasis zu ziehen.

Die Wissensbasis besteht dabei aus den beiden Komponenten T-Box und A-Box. Die T-Box legt *Terminologie* des Anwendungsbereichs fest, d.h. das Vokabular, das verwendet werden soll. Die A-Box legt *Annahmen* (Assertions) über die Individuen (unter Verwendung der in der T-Box gegebenen Terminologie) fest. Verglichen mit Datenbanksystemen, legt die T-Box das Datenbankschema fest, und die A-Box die eigentlichen Daten.

Das Vokabular der T-Box besteht aus *Konzepten* und *Rollen*. Konzepte repräsentieren dabei Menge von Individuen, Rollen stehen für binäre Relationen zwischen Individuen. Atomare Konzepte und atomare Rollen sind nur Bezeichner, d.h. sie werden durch ihren Namen repräsentiert und später semantisch als Mengen bzw. Relationen interpretiert. Neben atomaren Konzepten und Rollen gibt es in allen Beschreibungslogiken Konstrukte, um *komplexe* Beschreibungen von Konzepten und Rollen auszudrücken (dies sind Formeln, die als Atome dann gerade atomare Konzepte und Rollen verwenden). Innerhalb der T-Box werden diese komplexen Konzepte und Rollen definiert und mit neuen Namen versehen. D.h. im Grunde besteht die T-Box aus *atomaren* Bezeichnern und *definierten* Namen.

Je nach verfügbaren Konstrukten zur Konstruktion von komplexen Konzepten und Rollen werden die verschiedenen Beschreibungslogiken unterschieden. Im Wesentlich gibt es hierbei zwei Abwägungen: Je mehr Konstrukte verfügbar sind, desto einfacher (oder gar mehr) lässt sich Wissen ausdrücken, andererseits gilt jedoch auch: Je mehr Konstrukte

vorhanden sind, umso komplexer sind die Inferenzmechanismen. Schon in relativ ausdrucksschwachen Beschreibungslogiken können Probleme untentscheidbar oder nichtpolynomiell sein.

Typische Fragestellungen, die ein Wissensverarbeitungssystem beantworten sollte, sind: Sind die Beschreibungen innerhalb der T-Box erfüllbar (also nicht widersprüchlich), gibt es Beschreibungen die in anderen Beschreibungen enthalten sind (üblicherweise als Subsumption bezeichnet).

Bezüglich der A-Box ist eine wichtige Fragestellung, ob die Daten konsistent sind, d.h. gibt es ein Modell, dass die Annahmen über die Individuen einhält.

Bevor wir uns wieder mit T-Box und A-Box im speziellen beschäftigen, führen wir verschiedene Konzeptbeschreibungssprachen ein. Hierfür konzentrieren wir uns auf die Konstruktion und Semantik von Konzept- und Rollenbeschreibungen.

### 8.2.1 Die Basis-Sprache $\mathcal{AL}$

Atomare Beschreibungen sind atomare Konzepte und atomare Rollen. Komplexe Beschreibungen werden durch Konzeptkonstruktoren erstellt. Wir verwenden $A, B$ für atomare Konzepte, $R$ für atomare Rollen und $C, D$ für komplexe Konzepte. Die Sprache $\mathcal{AL}$ gilt als eine minimale Sprache von praktischem Interesse. Komplexe Konzeptbeschreibungen werden durch *Konzept-Terme* gebildet.

**Definition 8.2.1** (Syntax von Konzept-Termen in $\mathcal{AL}$)**.** *Die Syntax von Konzepttermen von $\mathcal{AL}$ ist durch die folgende Grammatik definiert, wobei $A$ atomares Konzept, $C, D$ Konzepte und $R$ atomare Rolle:*

$$
\begin{array}{rcll}
C, D \in \mathcal{AL} & ::= & A & \textit{atomares Konzept} \\
& | & \top & \textit{universelles Konzept} \\
& | & \bot & \textit{leeres Konzept} \\
& | & \neg A & \textit{atomares Komplement} \\
& | & C \sqcap D & \textit{Schnitt} \\
& | & \forall R.C & \textit{Wert-Einschränkung} \\
& | & \exists R.\top & \textit{beschränkte existentielle Einschränkung}
\end{array}
$$

Beachte das in $\mathcal{AL}$ die Komplementbildung nur für atomare Konzepte erlaubt ist, und dass für die existentielle Einschränkung nur das universelle Konzept erlaubt ist (d.h. $\exists R.C$ für beliebiges Konzept $C$ ist nicht erlaubt).

**Beispiel 8.2.2.** *Nehmen wir an* Person *und* Weiblich *seinen atomare Konzepte. Dann drückt das Konzept* Person $\sqcap$ Weiblich *gerade weibliche Personen aus, während das Konzept* Person $\sqcap \neg$Weiblich *alle nicht weiblichen Personen ausdrückt.*

*Sei* hatKind *eine atomare Rolle. Dann können wird die Konzepte* Person $\sqcap \exists$hatKind.$\top$ *und* Person $\sqcap \forall$hatKind.Weiblich *konstruieren. Das erste Konzept beschreibt Personen, die Kinder ha-*

*ben. Evtl. sollten wir das Konzept verfeinern zu* Person $\sqcap \exists$hatKind.$\top \sqcap \forall$hatKind.Person, *um sicherzustellen, dass alle Kinder auch Personen sind.*

*Das zweite Konzept beschreibt Personen, deren Kinder alle weiblich sind. Anstelle des ersten Konzepts können kinderlose Personen durch das Konzept* Person$\sqcap \forall$hatKind.$\bot$ *beschrieben werden.*

Eigentlich können wir die Beispiele jedoch noch gar nicht interpretieren, da uns noch die *Semantik* der Beschreibungen fehlt. Das holen wir jetzt nach. Wie üblich definieren wir eine *Interpretation*:

**Definition 8.2.3** (Semantik von $\mathcal{AL}$)**.** *Eine* Interpretation $I$ *einer $\mathcal{AL}$-Formel legt folgendes fest:*

- *Eine nichtleere Menge $\Delta$ von* Objekten.
- *Für jedes atomare Konzept $A$: $I(A)$ als Teilmenge von $\Delta$, d.h. $I(A) \subseteq \Delta$.*
- *Für jede atomare Rolle $R$: $I(R)$ als binäre Relation über $\Delta$, d.h. $I(R) \subseteq \Delta \times \Delta$.*

*Die Erweiterung einer Interpretation $I$ auf komplexe Konzeptbeschreibungen ist induktiv durch die folgenden Fälle definiert.*

$$\begin{array}{lcl} I(C_1 \sqcap C_2) & = & I(C_1) \cap I(C_2) \\ I(\bot) & = & \emptyset \\ I(\top) & = & \Delta \\ I(\neg A) & = & \Delta \setminus I(A) \\ I(\exists R.\top) & = & \{x \in \Delta \mid \exists y.(x,y) \in I(R)\} \\ I(\forall R.C) & = & \{x \in \Delta \mid \forall y.(x,y) \in I(R) \Rightarrow y \in I(C)\} \end{array}$$

*Zwei Konzepte $C, D$ sind* äquivalent, *geschrieben als $C \equiv D$, genau dann wenn $I(C) = I(D)$ für alle Interpretationen $I$ gilt*

**Beispiel 8.2.4.** *Sei $I$ die Interpretation mit $\Delta = \{Marie, Horst, Susi, Fritz, Lassie\}$, $I$*(Person) $= \Delta \setminus \{Lassie\}$, $I$(Weiblich) $= \{Marie, Susi, Lassie\}$ *und $I$*(hatKind) $=$

$\{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\}$. *Dann können wir ausrechnen:*

$$I(\mathsf{Person} \sqcap \mathsf{Weiblich}) = I(\mathsf{Person}) \cap I(\mathsf{Weiblich}) = \{Marie, Susi\}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \exists \mathsf{hatKind}.\top) \\
&= I(\mathsf{Person}) \cap I(\exists \mathsf{hatKind}.\top) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \exists y.(x,y) \in I(\mathsf{hatKind})\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \exists y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\}\} \\
&= I(\mathsf{Person}) \cap \{Fritz, Marie\} \\
&= \{Fritz, Marie\}
\end{aligned}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \forall \mathsf{hatKind}.\mathsf{Weiblich}) \\
&= I(\mathsf{Person}) \cap I(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Weiblich})\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\} \\
&\qquad\qquad \Rightarrow y \in \{Marie, Susi, Lassie\}\} \\
&= I(\mathsf{Person}) \cap \{Marie, Horst, Susi, Lassie\} \\
&= \{Marie, Horst, Susi\}
\end{aligned}$$

$$\begin{aligned}
& I(\mathsf{Person} \sqcap \forall \mathsf{hatKind}.\bot) \\
&= I(\mathsf{Person}) \cap I(\forall \mathsf{hatKind}.\bot) \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\bot)\} \\
&= I(\mathsf{Person}) \cap \{x \in \Delta \mid \forall y.(x,y) \in \{(Fritz, Susi), (Marie, Susi), (Fritz, Horst)\} \Rightarrow y \in \emptyset\} \\
&= I(\mathsf{Person}) \cap \{Horst, Susi, Lassie\} \\
&= \{Horst, Susi\}
\end{aligned}$$

**Beispiel 8.2.5.** *Es gilt* $(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \sqcap (\forall \mathsf{hatKind}.\mathsf{Student})$ *und* $\forall \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Student})$ *sind äquivalente Konzepte. Das lässt sich mit der Semantik nachweisen:*

$$\begin{aligned}
& I(\forall \mathsf{hatKind}.\mathsf{Weiblich} \sqcap \forall \mathsf{hatKind}.\mathsf{Student}) \\
= \; & I(\forall \mathsf{hatKind}.\mathsf{Weiblich}) \cap I(\forall \mathsf{hatKind}.\mathsf{Student}) \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Weiblich})\} \\
& \cap \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in I(\mathsf{Student})\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow (y \in I(\mathsf{Weiblich}) \wedge y \in I(\mathsf{Student}))\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in (I(\mathsf{Weiblich}) \cap I(\mathsf{Student}))\} \\
= \; & \{x \in \Delta \mid \forall y.(x,y) \in I(\mathsf{hatKind}) \Rightarrow y \in (I(\mathsf{Weiblich} \sqcap \mathsf{Student}))\} \\
= \; & I(\forall \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Student}))
\end{aligned}$$

*Beachte, der Beweis ist völlig unabhängig von den atomaren Konzepten und Rollen, d.h. es gilt allgemein:*

$$(\forall R.C) \sqcap (\forall R.D) \;\equiv\; \forall R.(C \sqcap D)$$

### 8.2.2 Allgemeinere Konzept-Definitionen

Wir betrachten nun mögliche Erweiterungen der Basissprache $\mathcal{AL}$. Eine erste Erweiterung ist das Verwenden von *atomaren Funktionen*, die auch als *Attribute*, *Features*, *Attributsbezeichner* oder *funktionale Rollen* bezeichnet werden. Ihre Verwendung entspricht der Verwendung von Rollen, d.h. sie dürfen überall dort verwendet werden, wo auch Rollen erlaubt sind. Der Unterschied liegt in der Semantik: Jede Interpretation darf eine atomare Funktion nur als (partielle) *einstellige Funktion* auf $\Delta$, also als Funktion $f : \Delta \to \Delta$ definieren, was äquivalent dazu ist, dass es $f$ eine *rechtseindeutige* binäre Relation ist[1]. Eine echte Änderung der Syntax beim Einführen der atomaren Funktionen ist nicht notwendig, lediglich ist eine disjunkte Aufteilung der Namen in: Namen für Konzepte, Namen für Rollen und Namen für atomare Funktionen notwendig.

**Beispiel 8.2.6.** *Beispiele für atomare Funktionen sind* istMutterVon (*was wohl einer totalen Funktion entsprechen sollte*), *und* istEhepartner (*was i.A. einer partiellen Funktion entspricht*).

Wir betrachten weitere Konstrukte, die in Konzeptbeschreibungssprachen (neben den bereits eingeführten) verwendet werden, um komplexe Konzeptbeschreibungen $C$ zu bilden.

**Definition 8.2.7.** *Die Syntax zur Bildung von komplexen Konzeptbeschreibungen aus Definition 8.2.1 kann erweitert werden:*

| | | |
|---|---|---|
| $C ::=$ | $\ldots$ | |
| $\mid$ | $\neg C$ | *Komplement* |
| $\mid$ | $(C_1 \sqcup C_2)$ | *Vereinigung* |
| $\mid$ | $(\exists R.C)$ | *existenzielle Einschränkung* |
| $\mid$ | $(\leq\ n\ R)$, $(\geq\ n\ R)$ | *Anzahlbeschränkungen* (*number restrictions*) |
| $\mid$ | $(\leq n\ R.C)$, $(\geq\ n\ R.C)$ | *qualifizierte Anzahlbeschränkung.* |
| $\mid$ | $(R_1; R_2; \ldots; R_n = R'_1; R'_2; \ldots; R'_m)$ | *Pfadgleichungen, Attributsübereinstimmung* |

Es gibt auch Konstrukte, die es erlauben *komplexe Rollen* zu konstruieren, ein solches Konstrukt ist

$$(\text{RESTRICT } R\ C) \quad \text{Rolleneinschränkung}$$

Wir werden später weitere solcher Konstrukte behandeln.

In der Literatur werden teilweise andere Symbole verwendet, z.B.:

[1] D.h. für alle $x, y, z \in \Delta$: Wenn $(x, y) \in f$ und $(x, z) \in f$ muss gelten $y = z$

| | | |
|---|---|---|
| AND | entspricht | $\sqcap$ |
| OR | entspricht | $\sqcup$ |
| NOT | entspricht | $\neg$ |
| SOME | entspricht | $\exists$ |
| ALL | entspricht | $\forall$ |
| ... | | |

**Beispiel 8.2.8.** *Wir geben einige Beispiele, was man in diesen Sprachen ausdrücken kann. Wenn man die Konzeptnamen* Mensch, Frau, Mann *und keine weiteren Axiome hat, sind die Beziehungen zwischen* Mensch, Frau *und* Mann *ziemlich frei. Als Axiome würde man sich wünschen* $\mathsf{Frau} \sqcap \mathsf{Mann} = \bot$, *d.h. die beiden Konzepte sollen disjunkt sein; und* $\mathsf{Mensch} \equiv \mathsf{Frau} \sqcup \mathsf{Mann}$, *d.h. Menschen sind entweder Mann oder Frau.*

*Mittels der Relationen kann man weitere Konzepte definieren:*

$$\mathsf{Eltern} := \mathsf{Mensch} \sqcap (\exists \mathsf{hatKind}.\mathsf{Mensch})$$

*Da das noch Freiheiten lässt, ist folgende Definition genauer*

$$\mathsf{Eltern} := \mathsf{Mensch} \sqcap (\exists \mathsf{hatKind}.\mathsf{Mensch}) \sqcap (\forall \mathsf{hatKind}.\mathsf{Mensch})$$

*Desweiteren kann man definieren:*

$$\mathsf{Mutter} := \mathsf{Frau} \sqcap \mathsf{Eltern}$$

*Hier kann das Wissensrepräsentationssystem in Prinzip herausfinden, dass gilt* $I(\mathsf{Mutter}) \subseteq I(\mathsf{Frau})$. *Eine Studentin könnte man definieren durch*

$$\mathsf{Studentin} := \mathsf{Frau} \sqcap (\exists \mathsf{studiertFach}.\top)$$

*Anzahlbeschränkungen kann man verwenden z.B. in*

$$\begin{aligned} \mathsf{Bigamist} \quad := \quad & \mathsf{Mann} \sqcap (\geq\ 2\ \mathsf{verheiratetMit}) \\ & \sqcap (\leq\ 2\ \mathsf{verheiratetMit}) \sqcap (\forall \mathsf{verheiratetMit}.\mathsf{Frau}) \end{aligned}$$

*wenn* verheiratetMit *eine Rolle ist. Ist* verheiratetMit *sowieso eine atomare Funktion, dann sollte die Semantik (die wir gleich definieren) liefern:* $I(\mathsf{Bigamist}) = \emptyset$ *für jede Interpretation* $I$.

*Sei* isstGerne *eine Rolle, dann beschreibt das Konzept*

$$\mathsf{Mensch} \sqcap (\mathsf{verheiratetMit}; \mathsf{isstGerne} = \mathsf{isstGerne})$$

*gerade alle Menschen deren Ehepartner das gleiche Essen mögen, wie sie selbst.*

*Verwendet man atomare Rollen und Pfadgleichungen, so beschreibt das Konzept*

$$\mathsf{Mann} \sqcap (\mathsf{hatNachnamen} = \mathsf{hatMutter}; \mathsf{hatNachnamen})$$

*alle Männer, die den selben Nachnamen wie ihre Mutter haben* (*wobei* hatNachnamen *und* hatMutter *atomare Funktionen sind*).

Man erkennt, dass der Formalismus die Konzeptbildung mittels Vereinigung, Schnitt, Komplement, und über Eigenschaften bzw. Relationen erlaubt.

### 8.2.3 Semantik von Konzepttermen

Für Konzeptterme in der erweiterten Syntax erweitern wir die modulare, deklarative Semantik:

**Definition 8.2.9** (Semantik von Konzepttermen)**.** *Eine Interpretation ist analog zu Definition 8.2.3 definiert, wobei $I$ neben atomaren Konzepten und atomaren Rollen, die Bedeutung der atomaren Funktionen als* partielle Funktionen $f : \Delta \to \Delta$ *festlegt.*

*Die Interpretation wird auf die neuen Konstrukte wie folgt erweitert:*

$$
\begin{aligned}
I(C_1 \sqcup C_2) &= I(C_1) \cup I(C_2) \\
I(\exists R.C) &= \{x \in \Delta \mid \exists y.(x,y) \in I(R) \text{ und } y \in I(C)\} \\
I(\leq\ n\ R) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R)\}| \leq n\} \\
I(\geq\ n\ R) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R)\}| \geq n\} \\
I(\leq\ n\ R.C) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R) \wedge y \in I(C)\}| \leq n\} \\
I(\geq\ n\ R.C) &= \{x \in \Delta \mid |\{y \in \Delta \mid (x,y) \in I(R) \wedge y \in I(C)\}| \geq n\} \\
I(R_1; R_2; \ldots; R_n = R'_1; R'_2; \ldots; R'_m) &= \left\{ x \in \Delta \mid \begin{array}{r} \{y \in \Delta \mid (x,y) \in (I(R_1) \circ \ldots \circ I(R_n))\} \\ = \{y \in \Delta \mid (x,y) \in (I(R'_1) \circ \ldots \circ I(R'_m))\} \end{array} \right\} \\
&\quad \text{wobei } \circ \text{ die Komposition von Relationen ist}^1 \\
I(\mathit{RESTRICT}\ R\ C) &= \{(x,y) \in \Delta \times \Delta \mid (x,y) \in I(R) \text{ und } y \in I(C)\}
\end{aligned}
$$

Diese Semantik erlaubt es, Gleichheiten bzw. Untermengenbeziehungen von Konzepten zu zeigen.

**Beispiel 8.2.10.** *Wir zeigen* $(\exists R.C) \equiv (\exists(\mathit{RESTRICT}\ R\ C).\top)$. *Sei $I$ eine Interpretation. Dann gilt für die linke Seite:*

$$I(\exists R.C) = \{x \mid \exists y.(x,y) \in I(R) \wedge y \in I(C)\}$$

*Die rechte Seite hat als Interpretation:*

$$
\begin{aligned}
& \{x \mid \exists y.(x,y) \in I(\mathit{RESTRICT}\ R\ C)\} \\
= \ & \{x \mid \exists y.(x,y) \in \{(a,b) \mid (a,b) \in I(R) \wedge b \in I(C)\} \\
= \ & \{x \mid \exists y.(x,y) \in \{(a,b) \in I(R) \mid b \in I(C)\} \\
= \ & \{x \mid \exists y.(x,y) \in I(R) \wedge y \in I(C)\}
\end{aligned}
$$

[1] D.h. $R_1 \circ R_2 = \{(x,z) \mid (x,y) \in R_1 \wedge (y,z) \in R_2\}$

Das Beispiel zeigt auch, dass man voll existentielle Einschränkung nicht braucht, wenn RESTRICT und die beschränkte existentielle Einschränkung vorhanden sind.

### 8.2.4 Namensgebung der Familie der $\mathcal{AL}$-Sprachen

Die Namensgebung vieler Erweiterung der Basissprache $\mathcal{AL}$ erfolgt durch Anhängen weiterer Buchstaben, d.h. man verwendet als mögliche Sprachbezeichner

$$\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}]$$

wobei die Buchstaben für die entsprechenden zu $\mathcal{AL}$ hinzugefügten Konstrukte stehen:

| | |
|---|---|
| $\mathcal{U}$ | Union: Vereinigung ($\sqcup$) |
| $\mathcal{E}$ | Volle existenzielle Beschränkung ($\exists R.C$) |
| $\mathcal{N}$ | number restriction: Anzahlbeschränkung. $((\leq\ n\ R), (\geq\ n\ R))$ |
| $\mathcal{C}$ | Volles Komplement ($\neg C$) |

Z.B. ist $\mathcal{ALEN}$ die Sprache, die auf $\mathcal{AL}$ aufbaut und noch allgemeine existenzielle Beschränkung und Anzahlbeschränkung hat.

Nicht alle diese Namen bezeichnen verschiedene Sprachen aus semantischer Sicht. Denn z.B. ist semantisch $\mathcal{ALUE} = \mathcal{ALC}$, denn es gilt

$$C \sqcup D \equiv \neg(\neg C \sqcap \neg D)$$

und

$$\exists R.C \equiv \neg(\forall R.\neg C).$$

D.h. alles was in $\mathcal{ALUE}$ ausgedrückt werden kann, kann auch in $\mathcal{ALC}$ ausgedrückt werden und umgekehrt.

Wir beweisen die letzte Gleichung:

$$\begin{aligned}
I(\neg(\forall R.\neg C)) &= \Delta \setminus \{a \in \Delta \mid \forall b.(a,b) \in I(R) \Rightarrow y \in I(\neg C)\} \\
&= \Delta \setminus \{a \in \Delta \mid \forall b.(a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C))\} \\
&= \{a \in \Delta \mid \neg(\forall b.(a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.\neg((a,b) \in I(R) \Rightarrow b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.(a,b) \in I(R) \wedge \neg(b \in (\Delta \setminus I(C)))\} \\
&= \{a \in \Delta \mid \exists b.(a,b) \in I(R) \wedge (b \in I(C))\} \\
&= I(\exists R.C)
\end{aligned}$$

Deshalb kann man Vereinigung $\mathcal{U}$ und volle existenzielle Beschränkung durch $\mathcal{AL}$ mit Komplementen ausdrücken. Umgekehrt kann man volle Negation durch Vereinigung $\mathcal{U}$ und volle existenzielle Beschränkung ausdrücken, da man die Normalform darstellen kann.

Insgesamt erhält man 8 Sprachen: $\mathcal{AL}, \mathcal{ALC}, \mathcal{ALCN}, \mathcal{ALN}, \mathcal{ALU}, \mathcal{ALE}, \mathcal{ALUN}, \mathcal{ALEN}$ . Wir geben eine Tabelle der Konstrukte an, die in diesen Sprachen erlaubt ist:

| Name | Konstrukte | implizite Konstrukte |
|---|---|---|
| $\mathcal{AL}$ | $\sqcap, \bot, \top, \neg A, \forall R.C, \exists R.\top$ | |
| $\mathcal{ALC}$ | $\mathcal{AL}$, $\neg C$ | $\sqcup$,$\exists R.C$ |
| $\mathcal{ALCN}$ | $\mathcal{AL}$, $\neg C$, $(\leq\ n\ R)$ | $\sqcup$, $\exists R.C$, $(\geq\ n\ R)$ |
| $\mathcal{ALN}$ | $\mathcal{AL}$, $(\leq\ n\ R)$ $(\geq\ n\ R)$ | |
| $\mathcal{ALU}$ | $\mathcal{AL}$, $\sqcup$ | |
| $\mathcal{ALE}$ | $\mathcal{AL}$, $\exists R.C$ | |
| $\mathcal{ALUN}$ | $\mathcal{AL}$, $\sqcup$, $(\leq\ n\ R)$, $(\geq\ n\ R)$ | |
| $\mathcal{ALEN}$ | $\mathcal{AL}$, $\exists R.C$, $(\leq\ n\ R)$, $(\geq\ n\ R)$ | |

Die Sprachen $\mathcal{ALC}$ und $\mathcal{ALCN}$ spielen eine besondere Rolle, da sie sehr allgemein sind. Man kann z.B. so viel repräsentieren wie Aussagenlogik, denn Schnitte, Vereinigungen und Komplemente sind erlaubt (was gerade zum Kodieren von Konjunktion, Disjunktion und Negation in der Aussagenlogik verwendet werden kann).

Es gibt noch weitere Konzeptsprachen, die aufgrund der Historie andere Namen (ungleich vom Schema $\mathcal{AL}$+Erweiterung) haben. Diese werden mit $\mathcal{FL}$ (für frame-based description language) abgekürzt. Es gibt drei Varianten:

| Name | Konstrukte |
|---|---|
| $\mathcal{FL}_0$ | $\sqcap, \forall R.C$ |
| $\mathcal{FL}^-$ | $\sqcap, \forall R.C, \exists R.\top$ |
| $\mathcal{FL}$ | $\sqcap, \forall R.C, \exists R.C$ |

Es gelten folgende Beziehungen zwischen den $\mathcal{FL}$-Sprachen und den $\mathcal{AL}$-Sprachen:

$$\begin{aligned} \mathcal{AL} &\equiv \mathcal{FL}^- \cup \{\neg A, \top\} \\ \mathcal{ALC} &\equiv \mathcal{FL}^- \cup \{\neg C\} \\ \mathcal{ALC} &\equiv \mathcal{FL} \cup \{\neg C\} \end{aligned}$$

Ebenso gibt es noch eine Erweiterung der $\mathcal{ALUEN}$-Sprachen um den Schnitt von Rollen, d.h. die Syntax ist erweitert, so dass man dort, wo sonst nur elementare Rollen verwendet werden, den Schnitt von mehreren elementaren Rollen verwenden darf.

$$\mathcal{R}: \quad R_1 \sqcap R_2 \quad \text{Schnitt von Rollen}$$

Die Namemsgebung ist dann entsprechend:

$$\mathcal{AL}[\mathcal{U}][\mathcal{E}][\mathcal{N}][\mathcal{C}][\mathcal{R}]$$

Dies ergibt 8 weitere Sprachen in Erweiterung der $\mathcal{ALUEN}$-Sprachen. Es gelten weitere Sprach-Äquivalenzen: Da $\mathcal{C}$ immer $\mathcal{U}$ und $\mathcal{E}$ impliziert,
ist z.B. $\mathcal{ALUCR} = \mathcal{ALCR} = \mathcal{ALECR}$

### 8.2.5 Inferenzen und Eigenschaften

Wir definieren die interessanten Eigenschaften für Konzepte und zwischen Konzepte etwas formaler. Diese Eigenschaften möchte man mit Inferenzverfahren nachweisen.

**Definition 8.2.11.** *Seien $C, D$ (evtl. komplexe) Konzepte*

- *Ein Konzept $D$* subsumiert *ein Konzept $C$ (geschrieben als $C \sqsubseteq D$), gdw. für alle Interpretationen $I$ gilt: $I(C) \subseteq I(D)$.*
- *Ein Konzept $C$ ist* konsistent *gdw. es eine Interpretation $I$ gibt, so dass gilt: $I(C) \neq \emptyset$. Gibt es keine solche Interpretation, so nennt man $C$* inkonsistent.
- *Zwei Konzepte $C$ und $D$ sind* disjunkt, *gdw. für alle Interpretationen $I$ gilt: $I(C) \cap I(D) = \emptyset$.*
- *Zwei Konzepte $C$ und $D$ sind* äquivalent *($C \equiv D$) gdw. für alle Interpretationen $I$ gilt: $I(C) = I(D)$.*

Beachte, dass $C$ konsistent nicht bedeutet, dass $C$ in allen Interpretationen nicht leer ist, sondern dass eine Interpretation $I$ mit $I(C) \neq \emptyset$ ausreichend ist.

Man kann je nach Sprache die verschiedenen Fragestellungen ineinander überführen:

- $C$ ist inkonsistent, gdw. $C \equiv \bot$.
- $C \equiv D$ gdw. $C \sqsubseteq D$ und $D \sqsubseteq C$.
- $C$ ist disjunkt zu $D$ gdw. $C \sqcap D$ inkonsistent.
- $C$ inkonsistent, gdw. $C$ wird von $\bot$ subsumiert ($C \sqsubseteq \bot$)
- Wenn allgemeine Komplemente erlaubt, dann gilt:
  $C \sqsubseteq D$ gdw. $C \sqcap \neg D$ inkonsistent ist.
- Wenn allgemeine Komplemente erlaubt, dann gilt:
  $C \sqsubseteq D$ gdw. $C$ und $\neg D$ disjunkt sind.

Wenn die Sprache allgemeine Komplemente erlaubt, dann sind Subsumtion, Disjunktheitstest und Konsistenztest äquivalent und haben auch gleiche Komplexität. Dies gilt z.B. in der Sprache $\mathcal{ALC}$.

In der Sprache $\mathcal{FL}$ sind das aber verschiedene Fragestellungen.

### 8.2.6 Anwendungen der Beschreibungslogiken

In Ontologien der Life-Sciences (Medizin, Biologie) gibt es Anwendungsmöglichkeiten von Beschreibungslogiken zur Konsistenzprüfung. Einige Beispiele sind:

- SNOMED CT (ein Akronym für Systematized Nomenclature of Medicine – Clinical Terms) ist eine medizinische Nomenklatur, die ca. 500.000 Konzepte umfasst
- The National Cancer Institute Thesaurus , 45.000 Konzepte
- GO: gene ontology: ca. 20.000 Konzepte
- „ GALEN was concerned with the computerisation of clinical terminologies."

Insbesondere gibt es folgende Anwendungen einer ausdrucksschwachen Beschreibungslogik $\mathcal{EL}$:

- Überprüfung der Konsistenz einer Menge von Konzepten
- Überprüfung der Erweiterbarkeit einer Menge von Konzepten.

Die Beschreibungslogik $\mathcal{EL}$ hat als Syntax der Konzepte:

$$C ::= \top \mid A \mid C \sqcap D \mid \exists R.C$$

Die Subsumption in $\mathcal{EL}$ ist polynomiell und ist damit anwendungsgeeignet.
Ein Beispiel für eine Subsumtionsbeziehung ist:

$$\exists \mathsf{child}.\mathsf{Rich} \sqcap \exists \mathsf{child}.(\mathsf{Woman} \sqcap \mathsf{Rich}) \equiv \exists \mathsf{child}.(\mathsf{Woman} \sqcap \mathsf{Rich})$$

Als weiteres (eigentlich falsches) Beispiel wurde mittels Subsumption erkannt:

$$\text{„}\mathsf{Finger\text{-}Amputation} \sqsubseteq \text{Arm-Amputation"},$$

was von den anderen Konzepten impliziert wurde, aber so nicht gemeint war und zu Korrekturen führte.

## 8.3 T-Box und A-Box

### 8.3.1 Terminologien: Die T-Box

Eine Terminologie ist normalerweise eine Vereinbarung von Namen für Konzepte. In der allgemeinsten Form in $\mathcal{AL}$ ist es eine Menge von terminologischen Axiomen der Formen

$$C \sqsubseteq D \text{ oder } C \equiv D$$

wobei $C, D$ Konzepte sind.
Wenn es Rollenterme gibt, dann können die Axiome auch die Form

$$R \sqsubseteq S \text{ oder } R \equiv S$$

haben, wobei $R, S$ Rollen sind. Es handelt sich um Inklusionen und um Gleichheiten.

**Definition 8.3.1.** *Gegeben eine Menge $T$ von terminologischen Axiomen. Dann* erfüllt *eine Interpretation $I$ diese Axiome, wenn für alle Axiome*

- *$C \sqsubseteq D$ (bzw. $R \sqsubseteq S$) $\in T$ die Gleichung $I(C) \subseteq I(D)$ (bzw. $I(R) \subseteq I(S)$) gilt und*
- *für alle Axiome $C \equiv D$ (bzw. $R \equiv S$) die Gleichung $I(C) = I(D)$ (bzw. $I(R) = I(S)$ ) gilt.*

*Wenn $I$ die Menge der Axiome $T$ erfüllt, dann sagen wir $I$ ist ein* Modell *für $T$.*

Die einfachste Variante einer Menge von terminologischen Axiomen ist eine Menge von *Definitionen*. D.h. Gleichungen der Form $A = C$ (bzw. (R = S), wobei $A$ (bzw. $R$) ein (symbolischer) Name ist und $C$ ein Konzeptterm (bzw. $S$ ein Rollenterm. Wenn zusätzlich jeder definierte Name höchstens einmal auf der linken Seite einer Definition auftaucht, dann spricht man von einer **T-Box**. Dies entspricht dem Sprachgebrauch von KL-ONE[2].

Eine T-Box nennt man *zyklisch*, wenn es einen Namen gibt, der (evtl. implizit) durch sich selbst definiert ist. Ansonsten nennt man die T-Box azyklisch.

Eine azyklische T-Box ist dadurch gekennzeichnet, dass man alle definierten Namen in rechten Seiten von Axiomen durch Definitionseinsetzung eliminieren kann. D.h. man kann die T-Box selbst eliminieren und nur mit Konzepttermen arbeiten. Der Nachteil, den man sich erkauft, ist die mögliche exponentielle Vergrößerung der Konzeptterme durch die Einsetzung.

Wenn keine zyklischen Definitionen vorliegen, kann man den symbolischen Namen eine eindeutige Interpretation $I$ geben, wenn man bereits eine Interpretation $I_0$ der (nicht-definierten) atomaren Symbole gegeben hat.

Wenn zyklische Definitionen vorliegen, geht das nur noch unter einschränkenden Bedingungen.

**Beispiel 8.3.2.** *Beispiel für eine Terminologie (T-Box):*

$$
\begin{array}{lcl}
\mathsf{Frau} & \equiv & \mathsf{Person} \sqcap \mathsf{Weiblich} \\
\mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\
\mathsf{Mutter} & \equiv & \mathsf{Frau} \sqcap \exists \mathsf{hatKind.Person} \\
\mathsf{Vater} & \equiv & \mathsf{Mann} \sqcap \exists \mathsf{hatKind.Person} \\
\mathsf{Eltern} & \equiv & \mathsf{Vater} \sqcup \mathsf{Mutter} \\
\mathsf{Grossmutter} & \equiv & \mathsf{Mutter} \sqcap \exists \mathsf{hatKind.Eltern} \\
\mathsf{MutterMitVielenKindern} & \equiv & \mathsf{Mutter} \sqcap (\geq\ 3\ \mathsf{hatKind}) \\
\mathsf{MutterMitTochter} & \equiv & \mathsf{Mutter} \sqcap (\forall \mathsf{hatKind.Frau}) \\
\mathsf{Ehefrau} & \equiv & \mathsf{Frau} \sqcap (\exists \mathsf{hatEhemann.Mann})
\end{array}
$$

[2] Ein von Brachman beschriebenes Wissensrepräsentationssystem für die Verarbeitung natürlichsprachlicher Ausdrücke

*Da diese T-Box azyklisch ist und nur aus Definitionen besteht, kann man stets aus einer Interpretation $I_0$, die nur die Konzeptnamen* Person, Weiblich, hatKind *und* hatEhemann *interpretiert, ein Modell $I$ für die T-Box erzeugen, indem man setzt $I(\mathsf{Person}) := I_0(\mathsf{Person})$, $I(\mathsf{Weiblich}) := I_0(\mathsf{Weiblich}$, $I(\mathsf{hatKind}) := I_0(\mathsf{hatKind})$ und $I(\mathsf{hatEhemann}) := I_0(\mathsf{hatEhemann})$ und für alle anderen Konzeptnamen (d.h. den definierten Namen), die Interpretation einfach „ausrechnet", z.B. $I(\mathsf{Frau}) = I_0(\mathsf{Person}) \cap I_0(\mathsf{Weiblich})$.*

**Beispiel 8.3.3.** *Ein Beispiel für eine sinnvolle zyklische T-Box ist:*

$$\mathsf{Mensch'} \equiv \mathsf{Tier} \sqcap \forall \mathsf{HatEltern.Mensch'}$$

*Menschen sind alle Tiere, die deren Elteren nur Menschen sind.*

*Das Finden eines Modells ist in diesem Fall nicht so einfach, denn selbst wenn man eine Interpretation $I_0$ hat die* Tier *und* HatEltern *festlegt, können wir nicht ohne weiteres ein Modell finden, denn $I(\mathsf{Mensch'}) := I_0(\mathsf{Tier}) \cap \{x \mid \forall y.(x,y) \in I_0(\mathsf{HatEltern}) \implies y \in I(\mathsf{Mensch'})\}$ ist eine rekursiv definierte Menge. Man benötigt eine Interpretation (als Menge) $I(\mathsf{Mensch'}) \subseteq \Delta$, die ein Fixpunkt der rekursiven Gleichung ist.*

### 8.3.2 Fixpunktsemantik für zyklische T-Boxen

Zwei weitere Beispiele für sinnvolle zyklische T-Box-Definitionen sind:

$$\mathsf{MnurS} \equiv \mathsf{Mann} \sqcap \forall \mathsf{hatKind.MnurS}$$

das entspricht dem Konzept: „Mann, der nur Söhne hat, und für dessen Söhne das gleiche gilt". Zyklen können auch bei Datenstrukturen auftreten, z.B. für binäre Bäume:

$$\mathsf{BinBaum} \equiv \mathsf{Baum} \sqcap (\leq\ 2\ \mathsf{hatAst}) \sqcap \forall \mathsf{hatAst.BinBaum}$$

Eine Fixpunktsemantik hilft hier weiter. Im letzten Beispiel muss man auf jeden Fall einen kleinsten Fixpunkt nehmen (sonst wären unendlich tiefe Bäume auch enthalten!) Zunächst ist klar, dass eine Modell $I$ einer T-Box, alle Axiome erfüllen muss. Für jedes Axiom $A \equiv C_A$ muss $I(A) = I(C_A)$ gelten. Wenn $A$ in $C_A$ als Name vorkommt, dann ist diese Bedingung nichttrivial.

**Beispiel 8.3.4.** *Eine solche Interpretation muss nicht immer existieren, wenn die T-Box zyklisch ist:*

$$A \equiv \neg A$$

*kann man zwar hinschreiben, aber keine Interpretation dafür angeben, denn $\Delta \neq \emptyset$ war vorausgesetzt, und $I(A) = \Delta \setminus I(A)$ ist daher nie erfüllt.*

Man kann eine **Fixpunktsemantik** für eine zyklische T-Box mit Gleichungen als Grenzwert unter gewissen Bedingungen konstruieren: Zunächst hat man eine Basisinterpretation $I_B$ der nicht-definierten Namen.
Man definiert eine Folge von Interpretationen $I_i, i = 0, 1, 2, \ldots$, und

1. erweitert $I_B$: sei $I_0(A) = \emptyset$ für alle definierten Namen $A$.
2. Danach definiert man $I_{i+1}(A) := I_i(C_A)$ für alle $i$ und jede Definition $A \equiv C_A$.
3. Wenn die Folge monoton steigend ist, d.h. für alle Namen stets $I_i(A) \subseteq I_{i+1}(A)$ gilt, dann kann man $I_\infty(A) = \bigcup_i I_i(A)$ definieren.
   Das ergibt einen kleinsten Fixpunkt.

Einen größten Fixpunkt erhält man mit folgendem Vorgehen:

1. Man startet mit $I_B$ und erweitert diese so: $I_0(A) = \Delta$ für alle definierten Namen $A$.
2. Danach definiert man $I_{i+1}(A) := I_i(C_A)$ für jede Definition $A \equiv C_A$.
3. Wenn die Folge monoton fallend ist, d.h. für alle Namen stets $I_{i+1}(A) \subseteq I_i(A)$ gilt, dann kann man $I_\infty(A) = \bigcap_i I_i(A)$ definieren.
   Das ergibt einen größten Fixpunkt.

**Beispiel 8.3.5.** *Betrachte erneut die T-Box $A \equiv \neg A$. Der Versuch einen Fixpunkt zu erzeugen, scheitert, sei $\Delta$ beliebig als nicht-leere Menge festgelegt. Dann gibt es keinen kleinsten Fixpunkt:*

$$\begin{aligned} I_0(A) &= \emptyset \\ I_1(A) &= I_0(\neg A) = \Delta \setminus I_0(A) = \Delta \\ I_2(A) &= I_1(\neg A) = \Delta \setminus I_1(A) = \emptyset \end{aligned}$$

*Jetzt sieht man schon die Nichtmonotonie, da $\Delta = I_1(A) \not\subseteq I_2(A) = \emptyset$.*
*Für den größten Fixpunkt geht es analog:*

$$\begin{aligned} I_0(A) &= \Delta \\ I_1(A) &= I_0(\neg A) = \Delta \setminus I_0(A) = \emptyset \\ I_2(A) &= I_0(\neg A) = \Delta \setminus I_1(A) = \Delta \end{aligned}$$

*Da $I_2(A) \not\subseteq I_1(A)$ ist die Monotonie verletzt.*

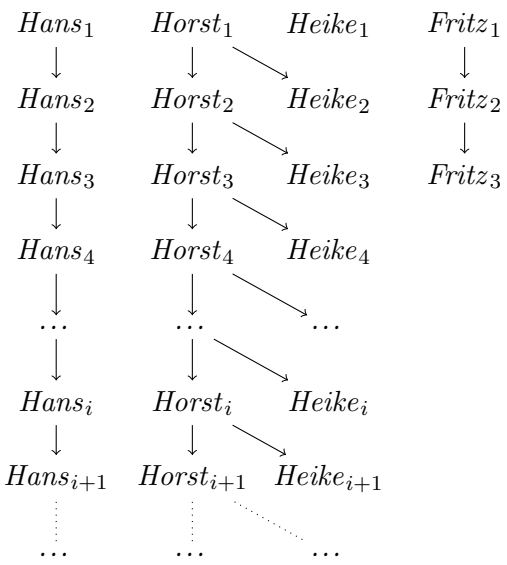
**Beispiel 8.3.6.**

$$\mathsf{MnurS} \equiv \mathsf{Mann} \sqcap \forall \mathsf{hatKind}.\mathsf{MnurS}$$

*Sei $I_B$ eine Interpretation, die $\Delta$, $I(\mathsf{Mann})$ und $I(\mathsf{hatKind})$ festlegt als*

$$\begin{aligned} \Delta &= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Horst_i \mid i \in \mathbb{N}\} \cup \{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\ I_B(\mathsf{Mann}) &= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Horst_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\ I_B(\mathsf{hatKind}) &= \{(Hans_i, Hans_{i+1}) \mid i \in \mathbb{N}\} \cup \{(Horst_i, Horst_{i+1}) \mid i \in \mathbb{N}\} \\ &\quad \cup \{(Horst_i, Heike_{i+1}) \mid i \in \mathbb{N}\} \cup \{(Fritz_i, Fritz_{i+1}) \mid i \in \{1,2\}\} \end{aligned}$$

*Als Graph wobei eine gerichtete Kante für „hatKind“ steht*




*Intuitiv erwartet man, dass die Interpretation von* MnurS *alle drei $Fritz_i$ und alle $Hans_i$ enthält. Wir zeigen, dass man dafür die größte Fixpunkt-Semantik nehmen muss, während die kleinste Fixpunkt-Semantik zuwenig liefert.*

*Wir berechnen den kleinsten Fixpunkt:*

$$
\begin{array}{lcl}
I_0(\mathsf{MnurS}) & = & \emptyset \\
I_1(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_0(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_3\}) = \{Fritz_3\} \\
I_2(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_1(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_2, Fritz_3\}) = \{Fritz_2, Fritz_3\} \\
I_3(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_2(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_1, Fritz_2, Fritz_3\}) = \{Fritz_1, Fritz_2, Fritz_3\} \\
I_4(\mathsf{MnurS}) & = & I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_3(\mathsf{MnurS})\} \\
 & = & I_B(\mathsf{Mann}) \cap (\{Heike_i \mid i \in \mathbb{N}\} \cup \{Fritz_1, Fritz_2, Fritz_3\}) = \{Fritz_1, Fritz_2, Fritz_3\} \\
I_j(\mathsf{MnurS}) & = & \{Fritz_1, Fritz_2, Fritz_3\} \textit{ für alle weiteren } i
\end{array}
$$

*Das ergibt* $\bigcup_i I_i(\mathsf{MnurS}) = \{Fritz_1, Fritz_2, Fritz_3\}$.

*Nimmt man den größten Fixpunkt, werden auch die „unendlichen Pfade“ beachtet:*

$$\begin{aligned}
I_0(\mathsf{MnurS}) &= \Delta \\
I_1(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_0(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\Delta) \\
&= I_B(\mathsf{Mann}) \\
I_2(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_1(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_B(\mathsf{Mann})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i, Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}) \\
&= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_3(\mathsf{MnurS}) &= I_B(\mathsf{Mann}) \cap \{x \in \Delta \mid \forall y.(x,y) \in I_B(hatKind) \Rightarrow y \in I_2(\mathsf{MnurS})\} \\
&= I_B(\mathsf{Mann}) \cap (\{Heike_i, Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}) \\
&= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \\
I_j(\mathsf{MnurS}) &= \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\} \textit{ für alle weiteren } i
\end{aligned}$$

*Das ergibt* $\bigcap_i I_i(\mathsf{MnurS}) = \{Hans_i \mid i \in \mathbb{N}\} \cup \{Fritz_i \mid i \in \{1,2,3\}\}$.

**Theorem 8.3.7.** *Ist die Terminologie ohne Komplemente definiert, dann kann man sowohl einen kleinsten als auch einen größten Fixpunkt der Interpretationen als Erweiterung einer Basisinterpretation definieren.*

Der Grund dafür, dass dieses Verfahren funktioniert, ist, dass man folgende Monotonie in diesem Fall hat:

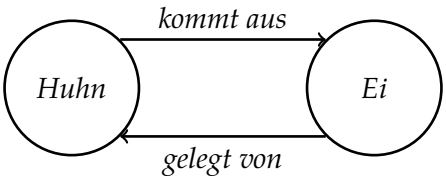
$$I \subseteq I' \Rightarrow I(C) \subseteq I'(C)$$

wobei $I \subseteq I'$ definiert ist als: $\forall$atomare Konzepte $A : I(A) \subseteq I'(A)$. Das wiederum folgt daraus, dass $\sqcap, \sqcup, \forall R.C, \exists R.C, (\geq\ n\ R)$ alle monoton im Konzept-Argument sind (falls es eines gibt). Wenn man über $\mathcal{ALCN}$ hinausgeht, ist das Konstrukt $(\geq\ n\ R.C)$ monoton, während $(\leq\ n\ R.C)$ nicht monoton ist.

**Theorem 8.3.8.** *Ist die Terminologie so definiert, dass jeder zyklische Pfad durch die Terme durch eine gerade Anzahl Negationen geht, denn kann man sowohl einen kleinsten als auch einen größten Fixpunkt der Interpretationen als Erweiterung einer Basisinterpretation definieren.*

Der Grund für das Funktionieren ist in diesem Fall ebenfalls die Monotonie bzgl. Interpretationen, wobei jedes Komplement die Monotonie in eine Antimonotonie verwandelt und nach zwei solchen Übergängen das Verhalten wieder monoton ist.

**Beispiel 8.3.9.** *Wir betrachten ein Beispiel für eine zyklische T-Box, die nicht nur aus Definition besteht. „Jedes Huhn kommt aus einem Ei. Jedes Ei wurde von einem Huhn gelegt.“ Die graphische Darstellung kann man so notieren:*




*Die zugehörigen Axiome in der T-Box sind:*

$$\begin{array}{lll} \mathsf{Huhn} & \sqsubseteq & (\exists \mathsf{kommtAus}.\mathsf{Ei}) \\ \mathsf{Ei} & \sqsubseteq & (\exists \mathsf{gelegtVon}.\mathsf{Huhn}) \end{array}$$

*Versuche ein Modell zu finden, so dass* $Clarissa \in I(\mathsf{Huhn})$*:*

1. *Dann muss gelten: Es gibt ein Objekt* $ClarissaEi$ *mit* $(Clarissa, ClarissaEi) \in I(\mathsf{kommtAus})$*. Das reicht jedoch nicht, denn es muss noch sichergestellt werden, dass* $ClarissaEi \in I(\mathsf{Ei})$*.*

2. *Also wird eine Mutter von Clarissa benötigt, die das Ei gelegt hat, d.h. es gibt* $ClarissaEi$ *mit* $(ClarissaEi, ClarissaMutter) \in I(\mathsf{gelegtVon})$*. Jetzt muss aber sichergestellt werden, dass* $ClarissaMutter$ *in* $I(\mathsf{Huhn})$ *ist, usw.*

*D.h. in allen endlichen Modellen mit* $I(\mathsf{Huhn}) \neq \emptyset$ *gibt es nur nur Hühner, die ihre eigene Vorfahren sind. Wenn das Modell unendlich sein darf, dann kann es auch Hühner geben, die nicht ihre eigenen Vorfahren sind.*

### 8.3.3 Inklusionen in T-Boxen

Als terminologische Axiome sind auch Inklusionen $A \sqsubseteq C$ erlaubt (wie wir sie gerade beim Huhn- / Ei-Beispiel gesehen haben). Dabei ist $A$ ein Name. Wenn man diese Namen einführt und nur durch eine Inklusion spezifiziert, dann kann man diese Axiome auf eine normale T-Box (die nur Äquivalenzen enthält) zurückführen, wobei man den Freiheitsgrad in einen neuen Namen kodiert:

Enthalten die terminologischen Axiome Inklusionen und kommen alle Namen auf linken Seite von Definitionen und Inklusionen nur jeweils einmal auf linken Seiten vor, dann kann man daraus eine äquivalente T-Box machen durch folgendes Verfahren:

- Zu jeder Inklusion $A \sqsubseteq C_A$ erfinde einen neuen Namen $\hat{A}$.
- Ersetze die Inklusion $A \sqsubseteq C_A$ durch die Definition $A \equiv \hat{A} \sqcap C_A$.

Damit hat man aus Inklusionen Gleichungen gemacht. Als Preis muss man neue Namen einführen. Die Modelle vorher und nachher sind die gleichen, wenn man sich beim Vergleich der Interpretationen auf die gemeinsamen Namen beschränkt.

D.h. Inklusionen sind nur dann kritisch, wenn man von bereits definierten Namen noch eine extra Inklusion verlangt.

**Beispiel 8.3.10.** *Hat man die T-Box*

$$\mathsf{Mann} \sqsubseteq \mathsf{Person}$$

*, so kann man mit dem eben beschrieben Verfahren, daraus die T-Box*

$$\mathsf{Mann} \equiv \widehat{\mathsf{Mann}} \sqcap \mathsf{Person}$$

*machen (dabei ist* $\widehat{\mathsf{Mann}}$ *der neu eingeführte Konzeptname).*
*Anders verhält es sich bei der T-Box:*

$$\begin{array}{lcl} \mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\ \mathsf{Mann} & \sqsubseteq & \mathsf{Tier} \end{array}$$

*Hier genügt es nicht, daraus die T-Box*

$$\begin{array}{lcl} \mathsf{Mann} & \equiv & \mathsf{Person} \sqcap \neg\mathsf{Frau} \\ \mathsf{Mann} & \equiv & \widehat{\mathsf{Mann}} \sqcap \mathsf{Tier} \end{array}$$

*zu erzeugen, da diese nicht einfach interpretiert werden kann, denn es gibt jetzt zwei Definition für* Mann.

### 8.3.4 Beschreibung von Modellen: Die A-Box

Von terminologischen Axiomen bzw. im Spezialfall von T-Boxen wird nur eine Struktur auf den Konzepten erzwungen. Was fehlt, ist eine konkretere Beschreibung von Modellen (Interpretationen). In diesem Sinn entspricht eine T-Box einem Datenbankschema, während eine A-Box einer Datenbank dazu entspricht. Der Name A-Box ist von Assertion hergeleitet.

**Definition 8.3.11.** *Gegeben eine T-Box* $\mathcal{T}$.
*Eine* A-Box $\mathcal{A}$ *zu* $\mathcal{T}$ *ist definiert als Menge von Annahmen über Individuen (Objekte) der Formen*

- $C(a)$ *wobei* $C$ *ein Konzeptterm ist und* $a$ *ein Individuenname.*
- $R(a, b)$ *wobei* $R$ *eine Rolle ist (evtl. ein Rollenterm) und* $a, b$ *sind Individuennamen.*

Man kann eine A-Box auch allgemeiner auf Basis einer Menge von terminologischen Axiomen definieren.

**Beispiel 8.3.12.** *Beispiele für Einträge in einer A-Box sind*

$$\begin{array}{l} \mathsf{MutterMitTochter}(\texttt{Maria}) \\ \mathsf{Vater}(\texttt{Peter}) \\ \mathsf{hatKind}(\texttt{Maria}, \texttt{Paul}) \\ \mathsf{hatKind}(\texttt{Maria}, \texttt{Peter}) \\ \mathsf{hatKind}(\texttt{Peter}, \texttt{Harry}) \end{array}$$

*Dabei sind* `Peter`, `Harry`, `Maria` *und* `Paul` *Konstanten.*

Das hat Ähnlichkeiten mit einer Datenbank in Prolog, bzw. mit Datalog und deduktiven Datenbanken. Allerdings sind in Konzeptbeschreibungssprachen allgemeinere Konzeptterme möglich. z.B. ist

$$(\exists \mathsf{hatKind}.\mathsf{Person})(\mathtt{Michael})$$

ebenfalls ein gültiger Eintrag.

**Definition 8.3.13** (Semantik der A-Box)**.** *Die Semantik $I$ einer A-Box $\mathcal{A}$ zur T-Box $\mathcal{T}$ erweitert die Semantik einer T-Box: Sei $I$ eine Interpretation zur T-Box $\mathcal{T}$. Die Interpretation kann auf die A-Box $\mathcal{A}$ erweitert werden durch:*

- *Jedem Individuennamen $a$ wird ein Objekt $I(a) \in \Delta$ zugeordnet. Hierbei wird die sogenannte* **unique names assumption** *beachtet: Verschiedenen Individuennamen werden verschiedenen Objekte zugeordnet, d.h. $I(a) = I(b)$ gdw. $a = b$.*
- *Für $C(a) \in \mathcal{A}$, ist $I(C(a)) = 1$ gdw. $I(a) \in I(C)$.*
- *Für $R(a, b) \in \mathcal{A}$ ist $I(R(a, b)) = 1$ gdw. $(I(a), I(b)) \in I(R)$ gilt.*

### 8.3.5 T-Box und A-Box: Terminologische Beschreibung

Eine *terminologische Beschreibung* (oder auch terminologische Datenbank) besteht aus

- Menge von terminologischen Axiomen.
- Mengen von Annahmen über Existenz und Eigenschaft von Objekten (A-Box)

Das kann als Spezialfall eine Kombination von T-Box und A-Box sein.

**Beispiel 8.3.14.** *Ein Beispiel für eine T-Box und eine zugehörige A-Box ist:*
*T-Box:*

$$\begin{array}{lcl}
\mathsf{Motor} & \sqsubseteq & \mathsf{Komponente} \\
\mathsf{Lampe} & \sqsubseteq & \mathsf{Komponente} \sqcap (\neg \mathsf{Motor}) \\
\mathsf{Stecker} & \sqsubseteq & \mathsf{Komponente} \sqcap (\neg \mathsf{Lampe}) \sqcap (\neg \mathsf{Motor}) \\
\mathsf{Gerät} & \sqsubseteq & (\forall \mathsf{hatTeil}.\mathsf{Komponente}) \sqcap (\neg \mathsf{Komponente}) \\
\mathsf{ElektroGerät} & \equiv & \mathsf{Gerät} \sqcap (\exists \mathsf{hatTeil}.\mathsf{Stecker})
\end{array}$$

*A-Box:*

$$\begin{array}{l}
\mathsf{Motor}(\mathtt{Motor1234}) \\
\mathsf{Komponente}(\mathtt{Lichtmaschine320}) \\
\mathsf{hatTeil}(\mathtt{Motor1234}, \mathtt{Lichtmaschine320}) \\
\ldots
\end{array}$$

**Definition 8.3.15.** *Gegeben eine T-Box $\mathcal{T}$ und eine A-Box $\mathcal{A}$.*

- *Dann ist $\mathcal{A}$* konsistent, *wenn es Modell $I$ von $\mathcal{T}$ gibt, das alle Einträge in der A-Box wahr macht.*

- *Wir schreiben $\mathcal{A} \models C(a)$ gdw. für alle Modelle $I$ von $\mathcal{T}$, die alle Einträge der A-Box wahr machen, auch $I(C(a))$ gilt.*

Folgende Inferenzen und Anfragen an T-Box und A-Box sind von praktischem Interesse:

- *Konsistenztest:* Prüfen, ob definierte Konzepte konsistent sind.
  $\mathsf{Motor} \sqcap (\neg\mathsf{Motor})$ ist inkonsistent, d.h. es gibt keine Objekte in diesem Konzept.

  Damit kann man auch erkennen, ob eine A-Box widersprüchliche Annahmen enthält. Z.B ist $(\mathsf{Motor} \sqcap \neg\mathsf{Motor})(\texttt{Motor123})$ nicht erfüllbar.

- *Subsumtionstest:* Prüfen, ob ein Konzept eine Untermenge eines anderen ist oder ob Konzepte disjunkt sind. Dadurch kann man z.B. die Struktur der Terminologie angeben.

- *Retrieval Problem* Berechne alle Instanzen eines Konzepts, wobei nur auf die Konstanten in der A-Box zugegriffen werden darf. z.B. „Welche Motoren gibt es ?“

  Das kann man formal schreiben als die Frage: Zu gegebenem Konzept $C$, finde alle $a$ mit $\mathcal{A} \models C(a)$.

- *Pinpointing (bzw. Realization Problem)* Das ist die Einordnung von Objekten in Konzepte. Z.B. die Frage: „Ist $\texttt{Staubsauger1}$ ein $\mathsf{ElektroGerät}$?“ Genauer kann man diese Anfrage spezifieren als: Gegeben ein Individuum $a$, finde das spezifischste Konzept $C$, so dass $\mathcal{A} \models C(a)$ gilt. D.h. finde das kleinste Konzept in der Subsumtionsordnung.

Es gibt folgende Zusammenhänge zwischen den Inferenzproblemen der A-Box und T-Box:

**Satz 8.3.16.** *Es gilt:*

- *$\mathcal{A} \models C(a)$ gdw. $\mathcal{A} \cup \{\neg C(a)\}$ ist inkonsistent.*

- *$C$ ist konsistent gdw. $C(a)$ konsistent ist für einen neuen Namen $a$.*

*Beweis.* Der erste Teil gilt, da ein Modell für die T-Box, das alle Axiome aus $\mathcal{A}$ wahr macht, auch $C(a)$ wahr machen muss, damit $\mathcal{A} \models C(a)$ gilt; jedes solche Modell muss daher $\neg C(a)$ falsch machen, und daher ist $\mathcal{A} \cup \{\neg C(a)\}$ inkonsistent.
Der zweite Teil gilt, da $C(a)$ nur dann wahr werden kann, wenn $I(C)$ nicht leer ist. □

### 8.3.6 Erweiterung der Terminologie um Individuen

Man kann in der T-Box auch Konzepte erlauben, die man als Aufzählungskonzepte ansehen kann: Die Syntax ist:

$$A \equiv \{a_1, \ldots . a_n\}$$

wobei $a_i$ Individuennamen sind. Die Semantik ist fixiert so dass $I(\{a_1, \ldots, a_n\}) = \{I(a_1), \ldots, I(a_n)\}$ gilt.

Damit kann man z.B. das Konzept Grundfarben $\equiv \{\texttt{rot}, \texttt{blau}, \texttt{gelb}\}$ definieren.

Man kann den Effekt allerdings auch kodieren, wenn man $\sqcap, \sqcup, \neg$ in der Beschreibungssprache hat.

### 8.3.7 Open-World und Closed-World Semantik

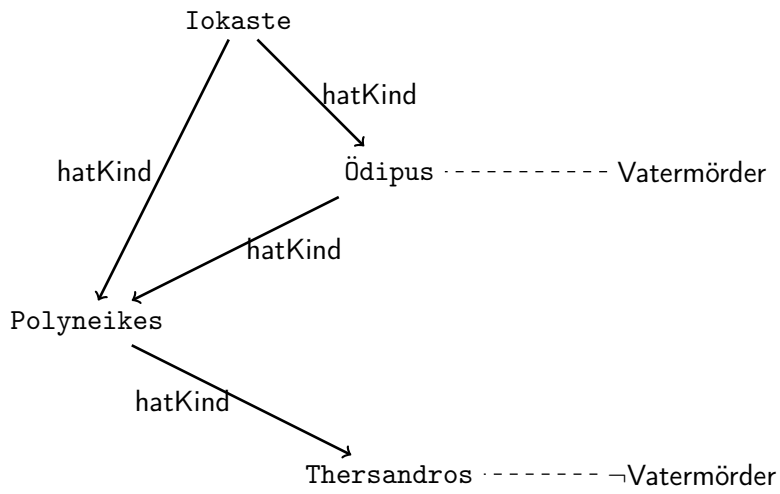
Im Gegensatz zu Datenbanken bei denen die Closed-World-Semantik angenommen wird, die eine eindeutige Semantik bei Datenbanken garantiert, benutzt man bei A-Boxen der Beschreibungssprachen die Open-World-Semantik. D.h., man geht davon aus, dass man unvollständiges Wissen hat. Der Schlussfolgerungsmechanismus wird somit fehlende Einträge nicht als negative Einträge werten. Da die Logik nicht auf Hornklauseln beruht, ist das auch die bessere Wahl, denn CWA + nicht-Hornklauseln können zu Inkonsistenzen führen.

In der Open-World Semantik kann man neue Fakten hinzufüge, ohne dass alte Schlüsse ungültig werden; d.h. Schlussfolgern ist monoton.

Den Unterschied kann man sich klarmachen am Beispiel: $\mathsf{hatKind}(\texttt{Maria}, \texttt{Peter})$. Wenn das der einzige Eintrag in einer Datenbank ist, dann hat Peter keine Geschwister. Wenn es der einzige Eintrag in einer A-Box ist, dann ist es durchaus möglich, dass Peter noch weitere Geschwister hat, nur hat die A-Box darüber keine Information. Man kann aber mit der Angabe $(\leq\ 1\ \mathsf{hatKind})(\texttt{Maria})$ in der A-Box die Information eintragen, dass es keine Geschwister gibt.

**Beispiel 8.3.17.** *Dieses Beispiel dient der Illustration des Verhaltens der Inferenzen durch die Open-World-Annahme.*
*Eine A-Box $\mathcal{A}_{oed}$ zum Problem des Ödipus:*

| | |
|---|---|
| $\mathsf{hatKind}(\texttt{Iokaste}, \texttt{Ödipus})$ | $\mathsf{hatKind}(\texttt{Iokaste}, \texttt{Polyneikes})$ |
| $\mathsf{hatKind}(\texttt{Ödipus}, \texttt{Polyneikes})$ | $\mathsf{hatKind}(\texttt{Polyneikes}, \texttt{Thersandros})$ |
| $\mathsf{Vatermörder}(\texttt{Ödipus})$ | $\neg\mathsf{Vatermörder}(\texttt{Thersandros})$ |




*Frage: kann man aus dieser A-Box folgendes schließen?*

$$\mathcal{A}_{oed} \models (\exists \mathsf{hatKind}.(\mathsf{Vatermörder} \sqcap (\exists \mathsf{hatKind}.\neg \mathsf{Vatermörder})))(\mathtt{Iokaste})$$

*Schaut man sich das an, dann könnte man folgendermaßen schließen:*
`Iokaste` *hat zwei Kinder,* `Ödipus` *und* `Polyneikes`*. Bei* `Ödipus` *ist der erste Teil erfüllt, aber man kann nichts über den zweiten Teil der Konjunktion sagen, da es kein Wissen über* Vatermörder(`Polyneikes`) *gibt. Nimmt man* `Polyneikes` *als Kandidat, so weiss man nicht, dass er* Vatermörder *ist, also scheint man nichts schließen zu können.*

*Korrekt ist aber, dass man eine Fallunterscheidung machen kann:* `Polyneikes` *ist entweder* Vatermörder *oder nicht. Im ersten Fall ist er der Kandidat, der die Formel als* hatKind *von* `Iokaste` *erfüllt. Im zweiten Fall ist der Kandidat* `Ödipus`*. Somit kann man obige Formel aus der A-Box schließen.*

Wie dieses Beispiel zeigt, kann die Open-World-Annahme Fallunterscheidungen erforderlich machen, um korrekt und vollständig schließen zu können.

## 8.4 Inferenzen in Beschreibungslogiken: Subsumtion

### 8.4.1 Struktureller Subsumtionstest für die einfache Sprache $\mathcal{FL}_0$

Wir wiederholen: Die Beschreibungssprache $\mathcal{FL}_0$ hat als Konstrukte nur Konjunktion und Wertbeschränkungen, d.h. nur atomare Konzepte, $C \sqcap D$ und $\forall R.C$ sind möglich. Diese Sprache ist eine Untersprache von $\mathcal{AL}$. Wir betrachten sie hier, um einen sogenannten strukturellen Subsumtionstest zu illustrieren.

Da Komplemente fehlen, gibt es in $\mathcal{FL}_0$ keinen direkten Zusammenhang zwischen Inkonsistenz von Konzepten und Subsumtion.

Es gilt:

**Lemma 8.4.1.** *Alle Konzepte in $\mathcal{FL}_0$ sind konsistent.*

*Beweis.* Das kann man ganz einfach dadurch zeigen, dass man eine Interpretation $I$ angibt, die folgendes erfüllt:

$$\begin{aligned} I(A) &= \Delta \quad \text{für alle atomaren Konzepte } A \\ I(R) &= \Delta \times \Delta \quad \text{für alle Rollen } R \end{aligned}$$

Dann werden sogar alle zusammengesetzten Konzepte $C$ immer als $I(C) = \Delta$ interpretiert. □

D.h. der Konsistenztest in $\mathcal{FL}_0$ ist trivial. Das kommt daher, dass man keine Möglichkeit hat, Konzepte mittels Negation einzuschränken.

Trotzdem ist der Subsumtionstest für $\mathcal{FL}_0$ nichttrivial. Z.B. gilt sicher $A \sqcap B \sqsubseteq A$, aber $A \not\sqsubseteq A \sqcap B$ für atomare Konzepte $A, B$.

#### 8.4.1.1 Struktureller Subsumtionstest

Allgemein (nicht nur in $\mathcal{FL}_0$) lässt sich der strukturelle Subsumtionstest, der $C \sqsubseteq D$ testet, durch die folgenden beiden Schritte beschreiben:

1. Bringe $C$ und $D$ in eine Normalform $C'$ bzw. $D'$.
2. Vergleiche $C'$ und $D'$ syntaktisch.

Beide Teile variieren von Sprache zu Sprache.

In $\mathcal{FL}_0$ ist eine Normalform folgendermaßen definiert:

$$A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

wobei die Kommutativität und Assoziativität von $\sqcap$ ausgenutzt wird, um Klammern wegzulassen, und $A_i$ verschiedene atomare Konzepte sind, die $R_i$ verschiedene Rollensymbole sind, und $C_i$ Konzepte in Normalform sind.

D.h. ein Normalformalgorithmus für $\mathcal{FL}_0$ wird folgendes tun:

- assoziativ Ausklammern, dann Umsortieren, und dann gleiche $A_i$ in Konjunktionen eliminieren.
- Falls es einen Unterausdruck $\forall R.C \sqcap \forall R.D$ gibt, nutzt man aus, dass dieser äquivalent zu $\forall R.C \sqcap D$ ist, da das Rollensymbol das gleiche ist.

  Als Begründung folgende Gleichungskette:

$$\begin{aligned} & \{x \mid \forall y.xI(R)y \Rightarrow y \in I(C_1) \cap I(C_2)\} \\ = \; & \{x \mid \forall y.\neg(x\ I(R)\ y \vee y \in I(C_1) \cap I(C_2)\} \\ = \; & \{x \mid \forall y.(\neg(x\ I(R)\ y) \vee y \in I(C_1) \wedge (\neg(x\ I(R)\ y) \vee y \in I(C_2)\} \\ = \; & \{x \mid \forall y.(\neg(x\ I(R)\ y) \vee y \in I(C_1)) \wedge \forall y.\neg(x\ I(R)\ y) \vee y \in I(C_2)\} \\ = \; & I((\forall\ R.C_1) \sqcap\ (\forall\ R.C_2)) \end{aligned}$$

- Es wird rekursiv in den rechten Seiten $C$ aller Ausdrücke $\forall R.C$ dasselbe Verfahren durchgeführt.

Der Vergleich zweier Normalformen wird folgendermaßen durchgeführt: Angenommen, wir haben zwei Ausdrücke $D, D'$:

$$D \equiv A_1 \sqcap \ldots \sqcap A_m \sqcap \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n$$

und

$$D' \equiv A'_1 \sqcap \ldots \sqcap A'_{m'} \sqcap \forall R'_1.C'_1 \sqcap \ldots \sqcap \forall R'_{n'}.C'_{n'}$$

Dann ist $D \sqsubseteq D'$ gdw:

- Jedes atomare Konzept $A'_i$ kommt unter den $A_j$ vor und
- zu jedem Ausdruck $\forall R'_i.C'_i$ gibt es einen Ausdruck $\forall R_j.C_j$, so dass die Rollennamen gleich sind, d.h. $R'_i = R_j$ und $C_j$ von $C'_i$ subsumiert wird. Dieser Test wird rekursiv durchgeführt.

Dieser Subsumtionstest ist korrekt und vollständig. Dabei bedeutet „korrekt“ , dass bei Antwort „ja“auch wirklich $I(D) \subseteq I(D')$ in allen Modellen $I$ gilt. „Vollständig“ bedeutet, dass die Antwort „nein“ impliziert, dass es ein Modell $I$ gibt, das ein Gegenbeispiel ist, d.h. $I(D) \not\subseteq I(D')$ für ein Modell $I$.

Will man einen Beweis führen, so ist es relativ einfach zu begründen, dass er korrekt ist, da man die Mengensemantik verwenden kann:

- Für die atomaren Konzepte: Da jedes atomare Konzept $A'_i$ unter den $A_j$ vorkommt, können wir $A_1 \sqcap \ldots \sqcap A_m$ auch umschreiben zu $A'_1 \sqcap \ldots \sqcap A'_{m'} \sqcap A''_1 \sqcap A''_k$ wobei $A''_i$ irgendwelche der $A_i$ sind. Nimmt man nun die Interpretation so sieht man leicht:

  $$I(A'_1) \cap \ldots \cap I(A'_{m'}) \cap I(A''_1) \cap I(A''_k) \subseteq I(A'_1) \cap \ldots \cap I(A'_{m'})$$

  da Schnitte die Mengen nur verkleinern.

- Für die Wertbeschränkungen reicht es zu zeigen, dass $I(\forall R_j.C_j) \subseteq I(\forall R'_i.C'_i)$ wenn $R_j$ und $R'_i$ die gleichen Rollennamen sind und $C_j \sqsubseteq C'_i$. Wir schreiben zur vereinfachten Darstellung $R$ anstelle von $R_j, R'_i$: Aus $C_j \sqsubseteq C'_i$ dürfen wir schließen $I(C_j) \subseteq I(C'_i)$ für jedes Modell $I$. Da

  $$I(\forall R.C_j) = \{x \in \Delta \mid \forall y.(x,y) \in I(R) \implies y \in I(C_j)\}$$

  und

  $$I(\forall R.C'_i) = \{x \in \Delta \mid \forall y.(x,y) \in I(R) \implies y \in I(C'_i)\}$$

  sieht man, dass die erste Menge kleiner (oder gleich) sein muss.

- Schließlich muss man sich noch klar machen, dass $S_1 \subseteq S_1'$ und $S_2 \subseteq S_2'$ impliziert $S_1 \cap S_2 \subseteq S_1' \cap S_2'$ (damit wir die beiden Teile: atomare Konzepte und Wertbeschränkungen getrennt voneinander betrachten durften).

Die Vollständigkeit kann man z.B. zeigen, indem man im Falle, dass der Algorithmus scheitert, ein Modell angibt, in dem die Subsumtionsbeziehung tatsächlich nicht gilt.

Die *Komplexität dieses Algorithmus* kann man wie folgt abschätzen:
Die Normalformherstellung arbeitet auf der Termstruktur und sortiert sinnvollerweise, d.h. man hat einen Anteil $O(n*log(n))$ für die Herstellung der Normalform. Das rekursive Vergleichen ist analog und könnte in einer sortierten Darstellung sogar linear gemacht werden. Insgesamt hat man $O(n * log(n))$ als Größenordnung des Zeitbedarfs.

**Beispiel 8.4.2.** *Betrachte die $\mathcal{FL}_0$-Konzeptdefinitionen:*

$$\begin{array}{lcl} C_1 & \equiv & (\forall R_1.A_1) \sqcap A_2 \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_2 \sqcap A_3 \sqcap A_4)) \\ C_2 & \equiv & ((\forall R_2.\forall R_1.A_4) \ \sqcap \ A_2 \ \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \end{array}$$

*Wir testen, ob $C_1 \sqsubseteq C_2$ und ob $C_2 \sqsubseteq C_1$ mithilfe des strukturellen Subsumtionstests. Zunächst berechnen wir die Normalformen $NF(C_1), NF(C_2)$:*
*Normalformberechnung für $C_1$:*

$$\begin{array}{lcl} C_1 & \equiv & (\forall R_1.A_1) \sqcap A_2 \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_2 \sqcap A_3 \sqcap A_4)) \\ & \to & A_2 \sqcap (\forall R_1.A_1) \sqcap (\forall R_2.A_5) \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \to & A_2 \sqcap (\forall R_1.A_1) \sqcap (\forall R_2.A_5 \sqcap (\forall R_1.(A_2 \sqcap A_3 \sqcap A_4))) = NF(C_1) \end{array}$$

*Normalformberechnung für $C_2$:*

$$\begin{array}{lcl} C_2 & \equiv & ((\forall R_2.\forall R_1.A_4) \ \sqcap \ A_2 \ \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \to & A_2 \sqcap ((\forall R_2.\forall R_1.A_4) \sqcap (\forall R_2.\forall R_1.(A_3 \sqcap A_4)) \\ & \to & A_2 \sqcap (\forall R_2.((\forall R_1.A_4) \sqcap \forall R_1.(A_3 \sqcap A_4))) \\ & \to & A_2 \sqcap (\forall R_2.(\forall R_1.(A_4 \sqcap A_3 \sqcap A_4)) \\ & \to & A_2 \sqcap (\forall R_2.(\forall R_1.(A_3 \sqcap A_4)) = NF(C_2) \end{array}$$

*Nun müssen wir die Normalformen vergleichen: Die atomaren Konzepte sind genau gleich, da beide Normalformen $A_2$ als atomares Konzept in der Konjuktion haben. Für die Wertbeschränkungen: Da $\forall R_1 \ldots$. nur in der $NF(C_1)$ vorkommt, können wir sofort schließen $C_2 \not\sqsubseteq C_1$. Für $C_1 \sqsubseteq C_2$ müssen wir für $\forall R_2 \ldots$. rekursiv zeigen Das erfordert als rekursiven Vergleich: $A_5 \sqcap (\forall R_1.(A_2 \sqcap A_3 \sqcap A_4))) \sqsubseteq (\forall R_1.(A_3 \sqcap A_4))$ Für die atomaren Konzepte ist der Vergleich erfolgreich, da $A_5$ nur links vorkommt. Für die Wertbeschränkungen müssen wir rekursiv vergleichen: $(A_2 \sqcap A_3 \sqcap A_4) \sqsubseteq (A_3 \sqcap A_4)$ was offensichtlich erfüllt ist. Daher dürfen wir schließen $C_1 \sqsubseteq C_2$.*

### 8.4.2 Struktureller Subsumtionstest für weitere DL-Sprachen

#### 8.4.2.1 Subsumtionstest für die Sprache $\mathcal{FL}^-$

Die Sprache $\mathcal{FL}^-$ hat $\sqcap, \forall R.C, (\exists R.\top)$ als Konstrukte. Der Subsumtionsalgorithmus für $\mathcal{FL}^-$ geht vor wie der Subsumtionstest für $\mathcal{FL}_0$:

1. Bringe die Konzeptterme in eine $\mathcal{FL}^-$-Normalform:

$$\begin{array}{ll} & A_1 \sqcap \ldots \sqcap A_m \\ \sqcap & \forall R_1.C_1 \sqcap \ldots \sqcap \forall R_n.C_n \\ \sqcap & \exists R'_1.\top \sqcap \ldots \sqcap \exists R'_k.\top \end{array}$$

   Analog wie in $\mathcal{FL}_0$ fasst man $\forall R.C$-Ausdrücke zusammen, wenn das gleiche Rollensymbol vorkommt.

2. Jetzt vergleicht man strukturell, wobei man genauso wie bei $\mathcal{FL}_0$ vorgeht, nur dass man $\exists R.\top$-Ausdrücke wie atomare Konzepte behandelt.

Auch der $\mathcal{FL}^-$-Subsumtions-Algorithmus ist korrekt und vollständig.
Der Zeitaufwand des Algorithmus ist $O(n * \log(n))$.

#### 8.4.2.2 Weitere Sprachen

Erweitert man $\mathcal{FL}_0$ um das konstante Konzept $\bot$, dann kann man ebenfalls einen Subsumtionstest durchführen, nur muss man beachten, dass man Konjunktionen, die $\bot$ enthalten zu $\bot$ vereinfacht, und dass $\bot$ von allen Konzepten subsumiert wird.

Erweitert man $\mathcal{FL}_0$ um $\bot$ und atomare Negation, d.h. zusätzlich ist noch $\neg A$ für atomare Konzepte $A$ erlaubt, dann kann man ebenfalls einen strukturellen Subsumtionsalgorithmus angeben.

Zusätzlich muss nur beachtet werden:

- kommt in einer Konjunktion $A$ und $\neg A$ vor, dann wird die gesamte Konjunktion durch $\bot$ ersetzt.
- $\bot$ wird von allen Konzepten subsumiert.
- Beim Vergleich von Normalformen werden negierte Atome wie neue Namen behandelt (das man ersetzt $\neg A$ durch $NOTA$ vor dem Vergleich.

#### 8.4.2.3 Subsumtions-Algorithmus für $\mathcal{AL}$

Wir können somit auch einen strukturellen Subsumtions-Algorithmus für $\mathcal{AL}$ angeben: Es fehlen an Konstrukten nur: $\top$ und $\exists R.\top$.

Zu beachten ist:

- In Konjunktionen, die $\top$ enthalten, kann man das $\top$ streichen.
- Das Konzept $\forall R.\top$ kann man durch $\top$ ersetzen. Ebenso kann man $\neg\top$ und $\neg\bot$ direkt vereinfachen. D.h. bei der Normalformherstellung spielt $\top$ keine Rolle. Es kann nur als Gesamtergebnis vorkommen (abgesehen von der Syntax $\exists R.\top$).
- Bei der Subsumtion ist nur zu beachten, dass $\top$ alles subsumiert.

#### 8.4.2.4 Konflikte bei Anzahlbeschränkungen

Wenn Anzahlbeschränkungen mit betrachtet werden, dann ergeben sich weitere Möglichkeiten für Konflikte, die in strukturellen Subsumtionsalgorithmen zu beachten sind:

Z.B. gibt es eine Interferenz zwischen $\forall R.\bot$ und $(\geq\ 1\ R)$. Das kann bei Schnitten ein Konflikt sein: $\forall R.\bot \sqcap (\geq\ 1\ R)$ ist äquivalent zu $\bot$. Es gibt auch neue Subsumtionen zu beachten: $(\geq\ n\ R) \sqsubseteq (\geq\ m\ R)$ gdw. $n \geq m$.

Es gibt weitere Sprachen, die einen strukturellen Subsumtionsalgorithmus haben:

| | |
|---|---|
| $\mathcal{ALE}$ | $\mathcal{AL}$ erweitert um $\exists R.C$. |
| $\mathcal{ALU}$ | $\mathcal{AL}$ erweitert um $\sqcup$. |
| $\mathcal{ALN}$ | $\mathcal{AL}$ erweitert um Anzahlbeschränkungen $(\leq\ n\ R)$. |

Die Sprache $\mathcal{ALN}$ hat einen polynomiellen und strukturellen Subsumtionsalgorithmus. Die Sprache $\mathcal{FL}$ hingehgen hat ein PSPACE-vollständiges Subsumtionsproblem.

Allerdings sind strukturelle Subsumtionsalgorithmen i.a. ungeeignet, wenn Disjunktion (d.h. Vereinigung) oder volle Negation vorkommt, z.B. in der Sprache $\mathcal{ALC}$.

### 8.4.3 Subsumtion und Äquivalenzen in $\mathcal{ALC}$

$\mathcal{ALC}$ ist die Konzeptbeschreibungssprache, die $\sqcap, \sqcup, \neg, \bot, \top, \forall R.C, \exists R.C$ erlaubt, aber nur atomare Rollen und keine Anzahlbeschränkungen.

In dieser Sprache kann man mittels der Semantik zeigen, dass folgende Äquivalenzen gelten:

$$\begin{array}{lcl}
\neg(C_1\ \sqcap\ C_2) & \equiv & (\neg\, C_1)\ \sqcup\ (\neg\, C_2) \\
\neg(C_1\ \sqcup\ C_2) & \equiv & (\neg\, C_1)\ \sqcap\ (\neg\, C_2) \\
\neg(\neg\, C) & \equiv & C \\
\neg(\forall R.C) & \equiv & (\exists R.(\neg\, C)) \\
\neg\,(\exists R.C) & \equiv & (\forall R.(\neg\, C)) \\
\neg\,\bot & \equiv & \top \\
\neg\,\top & \equiv & \bot
\end{array}$$

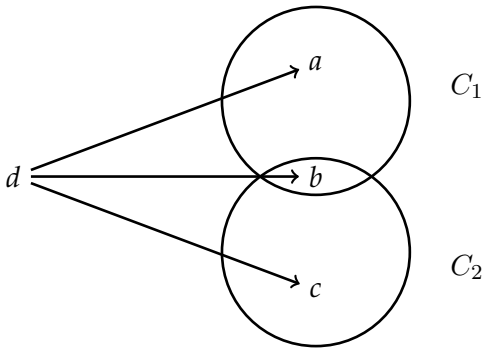
**Beispiel 8.4.3.** *Analog zur Prädikatenlogik gilt die folgende Gleichung* nicht*!*

$$(\forall R.(C_1\ \sqcup\ C_2)) = (\forall R.C_1)\ \sqcup\ (\forall R.C_2)$$

*Wir zeigen, dass die Gleichung nicht gilt, indem wir ein Gegenbeispiel angeben. Sei $I$ die Interpretation mit*

$$\begin{aligned} \Delta &= \{a, b, c, d\} \\ I(C_1) &= \{a, b\} \\ I(C_2) &= \{b, c\} \\ I(R) &= \{(d, b), (d, a), (d, c)\} \end{aligned}$$

*Als Bild:*




*Es gilt*

$$\begin{aligned} I(\forall R.(C_1 \sqcup C_2)) &= \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{a, b, c\}\} \\ &= \{a, b, c, d\} \end{aligned}$$

$$\begin{aligned} I((\forall R.C_1) \sqcup (\forall R.C_2)) &= \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{a, b\}\} \\ &\quad \cup \{x \in \{a, b, c, d\} \mid \forall y.(x, y) \in \{(d, b), (d, a), (d, c)\} \Rightarrow y \in \{b, c\}\} \\ &= \{a, b, c\} \end{aligned}$$

*Da $d \in I(\forall R.(C_1 \sqcup C_2))$ aber $d \notin I((\forall R.C_1) \sqcup (\forall R.C_2))$ sind die beiden Konzepte nicht äquivalent.*

**Lemma 8.4.4.** *Der Subsumtionstest in $\mathcal{ALC}$ lässt sich als Konsistenztest formulieren und umgekehrt: $C_1 \sqsubseteq C_2$ gilt gdw. $(C_1 \sqcap \neg C_2) \equiv \bot$*

Will man $C \not\equiv \bot$ testen, dann kann man auch $C \sqsubseteq \bot$ testen. Da ein Entscheidungsverfahren dann ja oder nein sagt, kann man daraus die Konsistenz bzw. Inkonsistenz von $C$ schließen.

### 8.4.4 Ein Subsumtionsalgorithmus für $\mathcal{ALC}$

Der Algorithmus zum Subsumtionstest von zwei Konzepten in $\mathcal{ALC}$ ist ein Tableauverfahren. Es verwendet den Konsistenztest für ein Konzept und prüft daher ob eine Interpretation mit $I(C) \neq \emptyset$ existiert. Die Idee dabei ist: Konstruiere eine solche Interpretation oder zeige, dass jede Konstruktion einer solchen Interpretation scheitern muss. Das Tableauverfahren arbeitet dabei mit einer neuen Struktur, sogenannten *Constraint-Systemen*. Der Algorithmus ist so allgemein, dass er leicht erweiterbar ist, um auch die Konsistenz

von A-Boxen zu prüfen. Er ist auch leicht erweiterbar auf allgemeinere Konzeptbeschreibungssprachen.

Die Idee des Algorithmus ist es, eine Interpretation aufzubauen, die $I(C)$ nicht leer macht. Dabei geht man vorsichtig vor, so dass der Algorithmus entweder bemerkt, dass ein Widerspruch aufgetreten ist, der anzeigt, dass das Konzept $C$ leer ist, oder es sind ausreichend viele Objekte und Beziehungen gefunden, die ein Modell darstellen.

Wir definieren zunächst die Struktur des Constraint-Systems:

**Definition 8.4.5.** *Ein* Constraint *ist eine Folge von Ausdrücken der Form:*

$$x : X \qquad xRy \qquad X \sqsubseteq C \qquad X \sqsubseteq Y \sqcup Z \qquad X(\forall R)Y \qquad X(\exists R)Y$$

*wobei $x, y, z$ Elemente (von $\Delta$), $X, Y, Z$ Konzeptnamen und $C$ (auch komplexe) Konzepte sind.* [3]

Wir beschreiben informell, was die einzelnen Constraints bedeuten: Diese entsprechen *Anforderungen* an das Modell $I$: $x : X$ entspricht der Bedingung $x \in I(X)$, $xRy$ entspricht der Bedingung $(x, y) \in I(R)$, $X \sqsubseteq C$ entpsricht der Bedingung $I(X) \subseteq I(C)$, $X \sqsubseteq Y \sqcup Z$ entspricht der Bedingung $I(X) \subseteq I(Y) \cup I(Z)$, $X(\forall R)Y$ entspricht der Bedingung $\forall a \in I(X) : \forall b : (a, b) \in I(R) : b \in I(Y)$, d.h. für alle Elemente $a$ in $I(X)$, die links in $I(R)$ vorkommen, muss jedes rechte Element $b$ auch in $I(Y)$ sein, $X(\exists R)Y$ entspricht der der Bedingung $\forall a \in I(X) : \exists b.(a, b) \in I(R) : b \in I(Y)$, d.h. für alle Element $a$ in $I(X)$ muss es mindestens ein Paar $(a, b) \in I(R)$ geben, wobei $b \in I(Y)$ ist.

Wie wir gleich sehen werden, wird im ersten Schritt (bevor das Tableau aufgebaut wird) das Constraint-System aufgefaltet. Hierfür werden die folgenden Regeln verwendet:

$$\begin{array}{lcl}
X \sqsubseteq (\forall R.C) & \rightarrow & X(\forall R)Y, Y \sqsubseteq C, \text{ wobei } Y \text{ ein neuer Name} \\
X \sqsubseteq (\exists R.C) & \rightarrow & X(\exists R)Y, Y \sqsubseteq C, \text{ wobei } Y \text{ ein neuer Name} \\
X \sqsubseteq C \sqcap D & \rightarrow & X \sqsubseteq C, X \sqsubseteq D \\
X \sqsubseteq C \sqcup D & \rightarrow & X \sqsubseteq Y \sqcup Z, Y \sqsubseteq C, Z \sqsubseteq D, \\
 & & \text{wenn und } C \text{ oder } D \text{ kein atomares Konzept ist,} \\
 & & \text{und } Y, Z \text{ sind neue Namen sind.} \\
X \sqsubseteq \top & \rightarrow & \text{nichts zu tun}
\end{array}$$

Es ist relativ leicht einzusehen, dass diese Regeln terminieren und als Normalform ein Constraint-System liefern, dass nur noch Konzeptnamen und negierte Konzeptnamen für $C$ in Constraints $X \sqsubseteq C$ enthält.

Der zweite Schritt des Verfahrens baut ein Tableau auf, indem folgende Regeln benutzt werden, um Individuen-Variablen korrekt einzufügen und das System zu vervollständigen:

[3] Bei Verallgemeinerung auf $\mathcal{ALCN}$ muss man darauf achten, dass Objektvariablen verschiedene Objekte bezeichnen

1. Wenn $x : X$ und $X(\exists R)Y$ da sind, aber keine Variable $y$ mit $xRy$ und $y : Y$, dann füge eine neue Variable $y$ mit den Constraints $xRy$ und $y : Y$ ein.

2. Wenn $x : X, X(\forall R)Y, xRy$ da ist, dann füge $y : Y$ hinzu.

3. Wenn $x : X, X \sqsubseteq Y \sqcup Z$ da sind, aber weder $x : Y$ noch $x : Z$, dann füge $x : Y$ oder $x : Z$ hinzu.

Beachte nur die letzte Regel stellt eine Verzweigung im Tableau dar. Beachte auch, dass Blätter im Tableau mit Constraint-System markiert sind, die vervollständigt sind, d.h. keine der drei Regeln ist mehr anwendbar.

Schließlich kann für jedes solche Blatt (teilweise auch schon früher, dann kann man die Vervollständigung auf dem entsprechenden Pfad stoppen) wird geprüft, ob das Constraintsystem *widersprüchlich* ist:

**Definition 8.4.6.** *Ein Constraint-System ist* widersprüchlich, *wenn die Konstellation* $x : X, X \sqsubseteq A, x : Y, Y \sqsubseteq \neg A$, *oder* $x : X, X \sqsubseteq \neg\top$, *oder* $x : X, X \sqsubseteq \bot$ *vorkommt.*

Jetzt haben wir alle Teilschritte beisammen, um den Algorithmus zur Konsistenzprüfung zu beschreiben:

**Definition 8.4.7.** *Der Algorithmus zur Konsistenzfeststellung von $C$ arbeitet folgendermaßen: Starte mit dem Constraint $x : X, X \sqsubseteq C$ (das entspricht gerade der Bedingung, dass es eine Interpretation $I$ gibt, die $C$ nicht leer interpretiert (denn $x \in I(X) \subseteq I(C)$)).*

*Als erster Schritt wird $x : X, X \sqsubseteq C$ aufgefaltet.*

*Anschließend wir das Tableau aufgebaut, indem das aufgefaltete Constraintsystem (nichtdeterministisch) vervollständigt wird (Verzweigungen ergeben sich aus Regel 3 der Vervollständigunsregeln). Wenn es möglich ist, ein Constraint-System zu erzeugen, das nicht widersprüchlich ist (d.h. es gibt einen Pfad dessen Blatt nicht widersprüchlich ist), gebe „konsistent" aus; ansonsten, wenn alle erzeugbaren Constraint-Systeme (d.h. alle Blätter des Tableaus) widersprüchlich sind, gebe „inkonsistent" aus.*

Beachte, dass der Algorithmus nicht mit einer T-Box als Eingabe arbeitet, sondern nur ein (evtl. komplexes) Konzept $C$ erhält. D.h. für den Fall einer T-Box muss diese zuvor entfaltet werden.

**Beispiel 8.4.8.** *Seien* Mann *und* hatKind *atomare Konzepte. Wir prüfen die Konsitenz des Konzepts:*

$$\mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann})$$

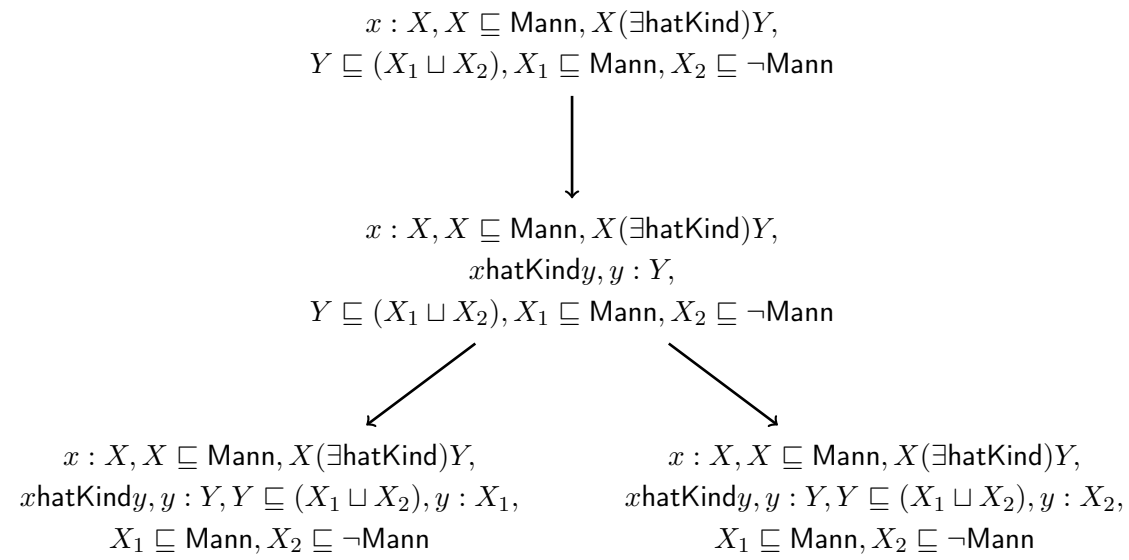
*Wir starten daher mit*

$$x : X, X \sqsubseteq \mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}).$$

*Entfalten des Constraintsystems:*

$$\begin{aligned} & x : X, X \sqsubseteq \mathsf{Mann} \sqcap \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\ \rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X \sqsubseteq \exists \mathsf{hatKind}.(\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\ \rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X(\exists \mathsf{hatKind})Y, Y \sqsubseteq (\mathsf{Mann} \sqcup \neg \mathsf{Mann}) \\ \rightarrow \quad & x : X, X \sqsubseteq \mathsf{Mann}, X(\exists \mathsf{hatKind})Y, Y \sqsubseteq (X_1 \sqcup X_2), X_1 \sqsubseteq \mathsf{Mann}, X_2 \sqsubseteq \neg \mathsf{Mann} \end{aligned}$$

*Vervollständigung ergibt das Tableau:*




*Beide Blätter sind vervollständigt, aber nicht widersprüchlich. Tatsächlich kann man die Modelle ablesen:*

- *Für das linke Blatt kann ablesen:* $\Delta = \{x, y\}$, $I(X) = \{x\}$, $I(Y) = \{y\}$, $I(X_1) = \{x, y\}$, $I(\mathsf{hatKind}) = \{(x, y)\}$ *und* $I(\mathsf{Mann}) = \{x, y\}$, $I(X_2) = \emptyset$.
- *Für das rechte Blatt kann man ablesen:* $\Delta = \{x, y\}$, $I(X) = \{x\}$, $I(Y) = \{y\}$, $I(X_2) = \{y\}$, $I(\mathsf{hatKind}) = \{(x, y)\}$, $I(\mathsf{Mann}) = \{x\}$, $I(X_1) = \{x\}$ .

*Das Konzept ist daher konsistent (einer der beiden Pfad hätte dafür gereicht).*

Da man zeigen kann, dass dieser Algorithmus terminiert und im Falle der Terminierung die richtige Antwort gibt, hat man ein Entscheidungsverfahren zur Feststellung der Konsistenz von $\mathcal{ALC}$-Konzepten.

**Theorem 8.4.9.** *Subsumtion und Konsistenz in $\mathcal{ALC}$ sind entscheidbar. Der Algorithmus kann in polynomiellem Platz durchgeführt werden.*

Aber es gilt:

**Theorem 8.4.10.** *Konsistenz in $\mathcal{ALC}$ ist PSPACE-hart. D.h Konsistenz in $\mathcal{ALC}$ ist PSPACE-vollständig.*

Der Nachweis kann direkt geführt werden, indem man zeigt, dass die Frage nach der Gültigkeit von quantifizierten Booleschen Formeln direkt als $\mathcal{ALC}$-Konsistenzproblem kodiert werden kann.

Dazu nimmt man quantifizierte Booleschen Formeln, die einen Quantorprefix haben und als Formel eine Klauselmenge (Konjunktion von Disjunktionen). Da die Kodierung recht technisch ist, geben wir als Beispiel die Kodierung der gültigen Formel $\forall x.\exists y.(x \vee \neg y) \wedge (\neg x \vee y)$ an:
Man benötigt nur ein Konzept $A$ und eine Rolle $R$: Die Kodierung ist wie folgt:

$$\begin{array}{rl|l} & \exists R.A \sqcap \exists R.\neg A & \forall x \\ \sqcap & \forall R.(\exists R.A \sqcup \exists R.\neg A) & \exists y \\ \sqcap & (\forall R.(\neg\, A \sqcap (\forall R.A))) & (\neg x \vee y) \\ \sqcap & (\forall R.(A \sqcap (\forall R.\neg\, A))) & (x \vee \neg y) \end{array}$$

### 8.4.5 Subsumtion mit Anzahlbeschränkungen

Auch für die Konzeptsprache $\mathcal{ALCN}$, die $\mathcal{ALC}$ um Anzahlbeschränkungen erweitert, kann man mit Constraintsystemen einen Algorithmus zum Testen der Konsistenz angeben.

Die Erweiterung betreffen die Konzeptterme der Form $(\geq\ n\ R.C)$ und $(\leq\ n\ R.C)$. In dem Fall erweitert man zunächst die Constraints um Terme $X(\leq\ n\ R)Y$ und $X(\geq\ n\ R)Y$. Das Auffalten erhält als zusätzliche Regeln

$$X \sqsubseteq (\geq\ n\ R.C) \rightarrow X(\geq\ n\ R)Y, Y \sqsubseteq C$$
$$X \sqsubseteq (\leq\ n\ R.C) \rightarrow X(\leq\ n\ R)Y, Y \sqsubseteq C$$

und für die Vervollständigung mit Elementen muss man zu $x : X, X(\leq\ n\ R)Y$ die Variablen $y$ zählen mit $xRy$ und entweder Variablen $y$ hinzufügen, oder zwei Variablen gleichsetzen. Diese Schritte müssen evtl. mehrfach durchgeführt werden.

Auch hier kann man zeigen, dass man das so durchführen kann, dass das Verfahren terminiert, aber es bleibt $PSPACE$-hart, so dass jeder Algorithmus exponentiell ist.

#### 8.4.5.1 Konsistenztest von A-Boxen

Wie schon erwähnt, kann man den Subsumtionstest bzw. den Konsistenztest von Konzepten auf Konsistenz von A-Boxen ausdehnen. Allerdings können A-Boxen etwas allgemeiner starten, so dass es tatsächlich passieren kann, dass der naive Algorithmus der Vervollständigung nicht terminiert. Es ist aber möglich, auf bestimmte Vervollständigungen zu verzichten, so dass er immer terminiert, ohne dass der Algorithmus unvollständig wird.

Es gilt:

**Theorem 8.4.11.** *Konsistenz von $\mathcal{ALCN}$-A-Boxen ist entscheidbar und PSPACE-complete.*

### 8.4.6 Komplexität der Subsumtions-Inferenzen in der T-Box/A-Box

Um die Konzeptsprachen zu bewerten, untersucht man die Ausdrucksstärke und die zugehörige Komplexität der Inferenzprobleme wie der Subsumtion. Hier gibt es, je nach Sprachumfang, eine Hierarchie von Sprachen und Komplexitäten.

- Die Beschreibungssprache $\mathcal{FL}^-$, die nur $\sqcap, \forall$ und $(\exists R)$ erlaubt, hat einen Subsumtionstest, der polynomiell ist.
- Die Beschreibungssprache $\mathcal{FL}$, die nur $\sqcap, \forall R.C$, $(\exists R.C)$ erlaubt, hat ein PSPACE-vollständiges Subsumtionproblem. Beachte, dass analog zur Sprache $\mathcal{FL}^-$ in der Konzeptsprache $\mathcal{FL}$ alle Konzepte konsistent sind.
- Erlaubt man nur $\sqcap, \sqcup, \neg$, dann ist der Subsumtionstest co-NP-vollständig, da dies gerade das Komplement von SAT ist.
- Die Sprache $\mathcal{ALC}$ erlaubt $\sqcap, \neg, \sqcup, \forall, (\exists R.C)$. Der Konsistenztest $\mathcal{ALC}$ ist genauso schwer wie der Subsumtionstest. Beide sind PSPACE-complete.
- erlaubt man zuviel, z.B. $\mathcal{ALC}$ und zusätzlich den Vergleich von Verknüpfungen von Relationen:
  Alle, deren Kinder die gleichen Fächer wie sie selbst studieren ( hatKind; studiertFach) = studiertFach) Dann wird Subsumtion sogar unentscheidbar.

## 8.5 Erweiterungen, weitere Fragestellungen und Anwendungen

In diesem letzten Abschnitt betrachten wir kurz einige weitere Erweiterungen und andere Fragestellungen, die im Zusammehang mit Beschreibungslogiken auftreten.

### 8.5.1 Konstrukte auf Rollen: Rollenterme

Es gibt noch weitere Konstrukte auf Rollen: Schnitt, Vereinigung, Komplement, transitiver Abschluss, die man verwenden kann um die Ausdruckskraft einer Beschreibungssprache zu erhöhen.

| | | |
|---|---|---|
| $R \sqcap S$ | Schnitt von Rollen | $I(R \sqcap S) = I(R) \cap I(S)$ |
| $R \sqcup S$ | Vereinigung von Rollen | $I(R \sqcup S) = I(R) \cup I(S)$ |
| $\neg R$ | Komplement einer Rolle | $I(\neg R) = \Delta \times \Delta \setminus I(R)$ |
| $(R^{-1})$ | Rolleninversion | $I(R^{-1}) = \{(b,a) \mid (a,b) \in I(R)\}$ |
| $(R \circ S)$ | Rollenkomposition | $I(R \circ S) = \{(a,c) \mid \exists b.(a,b) \in I(R), (b,c) \in I(S)\}$ |
| $(R^+)$ | transitiver Abschluss | $I(R^+) =$ transitiver Abschluss von $I(R)$. |

Die Ergebnisse, die es in der Literatur gibt, sind Entscheidungsverfahren und Komplexitäten von Subsumtionsalgorithmen.

z.B.: Die Sprache $\mathcal{ALCN}$ mit Rollenschnitt hat ein PSPACE-vollständiges Subsumtionsproblem, wenn man Zahlen in Strichcode schreibt.

Die Sprache $\mathcal{ALC}_{trans}$, die $\mathcal{ALC}$ um transitive Rollen erweitert, hat ein entscheidbares Subsumtionsproblem. Interessanterweise ergeben sich hier Brücken und Übergänge zu anderen Logiken, beispielsweise zur propositional dynamic logic, die eine erweiterte Modallogik ist

### 8.5.2 Unifikation in Konzeptbeschreibungssprachen

Das Unifikationsproblem für Konzeptbeschreibungssprachen lässt sich wiefolgt beschreiben:

| | |
|---|---|
| Problem: | Gegeben zwei Konzeptterme $C, D$. |
| Frage: | Gibt es eine Einsetzung $\sigma$ von Konzepttermen für die atomare Konzepte, so dass $\sigma(C) \equiv \sigma(D)$? |

Matching von Konzepttermen ist das eingeschränkte Unifikationsproblem, wenn Ersetzungen nur in einem Term erlaubt ist:

Eine mögliche Anwendung des Matching ist das Debugging einer Wissensbasis, die Description Logic verwendet. Dies eröffnet die Möglichkeiten zu erkennen ob man verschiedene Kodierungen des gleichen intendierten Konzepts eingegeben hat.

Zum Beispiel: Alice und Bob defnieren Konzepte für: Frauen, die Töchter haben:

| | |
|---|---|
| Alice: | $\mathsf{Frau} \sqcap \exists \mathsf{hatKind}.\mathrm{Frau}$ |
| Bob: | $\mathsf{Weiblich} \sqcap \mathsf{Mensch} \sqcap \exists \mathsf{hatKind}.(\mathsf{Weiblich} \sqcap \mathsf{Mensch})$ |

Die Einsetzung $\{\mathsf{Frau} \mapsto \mathsf{Weiblich} \sqcap \mathsf{Mensch}\}$ macht die beiden Konzepte gleich, und ist daher ein Unifikator.

#### 8.5.2.1 Unifikation in $\mathcal{EL}$

: Unifikation in $\mathcal{EL}$ ist entscheidbar, genauer: ist NP-vollständig (siehe (Baader & Morawska, 2009)).

Ein Algorithmus dazu hat Anwendungen in der Ontologiedatenbank Snomed. Eine unschöne Eigenschaft dieses Problems ist, dass es Unifikationsprobleme gibt, deren Lösungsmenge unendliche viele Substitutionen enthalten muss. z.B. $X \sqcap \exists R.Y \equiv^? \exists R.Y$. In der Anwendung braucht man evtl. nicht die volle Lösungsmenge.

#### 8.5.2.2 Unifikation in $\mathcal{ALC}$

Beachte, dass die Entscheidbarkeit der Unifikation in $\mathcal{ALC}$ im Moment ein offenes Problem ist. Das Problem ist äquivalent zur Entscheidbarkeit der Unifikation in der Basis-(Multi-)Modallogik $K$.

Da Aussagenlogik mit $\wedge, \vee, \neg$ und propositionalen Konstanten eine Untermenge der Sprache ist, ist die Boolesche Unifikation ein Subproblem, d.h. das Problem is NP-hart.

### 8.5.3 Beziehung zu Entscheidungsbäumen, Entscheidungslisten

Wir vergleichen die Ausdrucksmöglichkeiten der Beschreibungslogik mit den Möglichkeiten von attributierten Objekten wie z.B. Entscheidungsbäumen und -listen (siehe dazu auch das nächste Kapitel dieses Skripts).

Um die verschiedenen Attribut-Begriffe hier zu unterscheiden, sprechen wir von EB-Attributen, wenn die Darstellung der Objekte bei Entscheidungsbäumen gemeint ist.

Objekte mit nur diskreten EB-Attributen sind in DL direkt darstellbar, wenn man statt Rollen Attribute nimmt und die Attributwerte mittels Aufzählungsmengen darstellt:

Das Konzept `Farbe = rot` kann man dann als $\exists$`hatFarbe`.{`rot`} wobei man schon definiert haben muss, dass Farbe ein Attribut und keine Rolle ist.

Damit sind die EB-Konzepte alle darstellbar, wenn man Vereinigung, Schnitt und Negation erlaubt. Aber: man hat in DL noch etwas mehr, denn es ist auch darstellbar, dass die Farbe unbekannt ist bzw. noch nicht definiert ist. Aus den DL erhält man einen Subsumtionsalgorithmus auch für die etwas allgemeineren Konzepte.

### 8.5.4 Merkmals-strukturen (feature-structures)

Wenn keine Rollen, sondern nur Attribute erlaubt sind, d.h. statt binären Relationen hat man Attribute (partielle Funktionen), dann erhält man sogenannte *Merkmalsstrukturen*.

Erste Beobachtung ist folgende:
Da es keine echten Relationen mehr gibt, kann man $\forall R.C$ ersetzen durch einen $\exists$-Ausdruck, denn es gilt:

$$(\exists A.C) \sqcup \neg(\exists A.\top) = (\forall A.C)$$

da das Attribut $A$ funktional sein muss.

Weiterhin gilt jetzt:

$$(\exists A.(C_1 \sqcup C_2)) = (\exists A.C_1) \sqcup (\exists A.C_2)$$

denn:

$$
\begin{aligned}
& \{x \mid \exists y.(I(A)(x) = y \wedge (y \in I(C_1)) \wedge y \in I(C_2))\} \\
= \; & \{x \mid \exists y.(I(A)(x) = y \wedge y \in I(C_1)) \wedge I(A)(x) = y \wedge y \in I(C_2)\} \\
= \; & \{x \mid (\exists y.(I(A)(x) = y \wedge y \in I(C_1))) \wedge \exists y.I(A)(x) = y \wedge y \in I(C_2)\} \\
= \; & I((\exists A.K_1) \sqcup (\exists A.C_2))
\end{aligned}
$$

Entsprechende Aussagen gelten für $\sqcap$. Hierbei ist der Definitionsbereich zu beachten.

Es gilt: Die Subsumtion in $\mathcal{ALC}$ mit funktionalen Rollen, d.h. Attributen, ist entscheidbar, auch wenn agreements (role value maps) hinzugenommen werden.

### 8.5.5 Verarbeitung natürlicher (geschriebener) Sprache

Man kann Merkmalsstrukturen im Bereich (Unifikations-)Grammatiken zur Verarbeitung natürlicher Sprache in der Computerlinguistik verwenden. Man schreibt die Anwendung von Funktionen auch in der Dot-Notation. D.h. statt $f(g(x))$ schreibt man $x.f.g$

**Beispiel 8.5.1.** *Beispielsweise kann man das Wort „sie“im Deutschen wie folgt charakterisieren:*

$$
\begin{aligned}
sie \;\; := \;\; & (Syn.cat = PP) \sqcap (Syn.agr.person = 3) \sqcap (Syn.agr.sex = W) \\
& \sqcap (Syn.agr.num = sg) \\
\textit{oder} & \\
sie \;\; := \;\; & (Syn.cat = PP) \sqcap (Syn.agr.person = 3) \sqcap (Syn.agr.num = pl)
\end{aligned}
$$

*Grammatikregeln werden um Attributgleichungen (feature agreements, Pfadgleichungen) erweitert:*
*NP = Det Adj N* $\quad (Det.person) = (N.person) \sqcap (Det.casus) = (N.casus)$

Anwendungen für Merkmalsstrukturen sind:

- Lexikoneinträge von Wörtern, oder Wortklassen:
- Semantische Zuordnung von Wörtern aus dem Lexikon
  z.B. Bank: Gebäude oder Sitzmöbel
- Kontextfreie Grammatikregeln mit Bedingungen

Der Test, ob eine Regel anwendbar ist, erfordert dann einen Konsistenztest für eine Merkmalsstruktur

Dieser Test wird auch Unifikation von Merkmalsstrukturen genannt („feature unification“).

Die entsprechenden Grammatiken heißen auch Unifikationsgrammatiken (PATR-II, STUF, ....)

Die Ausdrucksstärke von Merkmalsstrukturen kann je nach erlaubten Konstrukten ebenfalls unterschiedlich sein.

## 8.6 OWL – Die Web Ontology Language

Die Web Ontology Language (kurz OWL in analogie zu owl (Engl. Eule)) ist eine durch das W3C standardisierte formale Beschreibungssprache zur Erstellung von Ontologien[4], d.h. der maschinell verarbeitbaren Erstellung von Konzepten und Beziehungen. OWL trägt dabei eine bedeutsame Rolle zum semantischen Web, der Weiterentwicklung des Internet, in der sämtliche Information auch semantisch dargestellt, erfasst und entsprechend verarbeitet werden kann. Als Syntax baut OWL auf der RDF-Syntax[5] auf und verwendet insbesondere XML als maschinell verarbeitbare Syntax.

Es gibt drei Varianten von OWL: OWL Lite, OWL DL, und OWL Full. Jede der Sprachen ist eine Erweiterung der vorhergehenden, d.h. OWL Lite lässt am wenigsten zu, OWL DL erweitert OWL Lite, und OWL Full erweitert OWL DL. Die beiden ersten Varianten OWL Lite und OWL DL entsprechen Beschreibungslogiken und lassen sich entsprechend durch Beschreibungslogiken formalisieren, hingegen passt OWL Full nicht mehr in den Rahmen der Beschreibungslogiken.

OWL Lite entspricht der Beschreibungslogik $\mathcal{SHIN}(\mathbf{D})$ und OWL DL der Beschreibungslogik $\mathcal{SHOIN}(\mathbf{D})$.

Wir erklären diese Namen:

- $\mathcal{S}$ steht für $\mathcal{ALC}$ erweitert um transitive Rollen, d.h. neben atomaren Konzepten und Rollen, Werteinschränkung, existentieller Werteinschränkung, Vereinigung, Schnitt und Negation stehen spezielle Rollen zur Verfügung, die stets als transitive Relation interpretiert werden müssen, d.h. für jede Interpetation $I$ und transitive Rolle $R$ muss stets gelten $(x, y) \in I(R), (y, z) \in I(R) \Rightarrow (x, z) \in I(R)$. Der Name $\mathcal{S}$ wurde verwendet, da diese Logik in enger Beziehung zur Modallogik $S_4$ steht.
- $\mathcal{H}$ steht für Rollenhierarchien, d.h. Axiome der Form $R \sqsubseteq S$ für Rollen $R, S$ sind erlaubt. und werden semantisch als $I(R) \subseteq I(S)$ interpretiert.
- $\mathcal{I}$ steht für inverse Rollen, d.h. $R^-$ ist als Rollenkonstruktor erlaubt (wenn $R$ eine Rolle ist) und wird als $I(R^-) = \{(y, x) \mid (x, y) \in I(R)\}$ interpretiert.
- $\mathcal{N}$ kennen wir bereits: number restrictions $(\leq\ n\ R)$, $(\geq\ n\ R)$
- $\mathcal{Q}$ steht für qualifizierte Anzahlbeschränkungen $(\leq n\ R.C)$ und $(\geq\ n\ R.C)$
- $\mathcal{O}$ steht für "nominal": Dies ist der Konstruktor $\{o\}$, wobei $o$ ein Individuenname ist. Er konstruiert einelementige Mengen von Individuen, die in der T-Box verwendet werden dürfen.
- $\mathbf{D}$ meint, dass konkrete Datentypen verwendet werden dürfen (dies ist eine Variante sogenannter *concrete domains*). In OWL dürfen die XML Schema Datentypen (Integer, Strings, Float, ...) verwendet werden.

[4] siehe http://www.w3.org/2004/OWL/
[5] RDF = Resource Description Framework

Eine OWL Lite- oder OWL DL- Ontologie entspricht einer TBox zusammen mit einer Rollenhierarchie, die den Wissenbereich mittels Konzepten und Rollen beschreiben. Allerdings wird in OWL statt Konzept der Begriff *Klasse* (Class) und statt Rolle der Begriff *Eigenschaft* (Property) verwendet. Eine Ontologie besteht aus einer Menge von Axiomen, die beispielsweise Subsumtionsbeziehungen zwischen Konzepten (oder Rollen) zusichern.

Genau wie in den bereits gesehenen Beschreibungslogiken werden OWL Klassen durch einfachere Klassen und Konstruktoren gebildet. Die in OWL verfügbaren Konstrukte und ihre Entsprechung in Beschreibungslogik zeigt die folgende Tabelle:

| Konstruktor | Syntax in DL |
|---|---|
| `owl:Thing` | $\top$ |
| `owl:Nothing` | $\bot$ |
| `intersectionOf` | $C_1 \sqcap \ldots \sqcap C_n$ |
| `unionOf` | $C_1 \sqcup \ldots \sqcup C_n$ |
| `complementOf` | $\neg C$ |
| `oneOf` | $\{a_1, \ldots, a_m\}$ |
| `allValuesFrom` | $\forall R.C$ |
| `someValuesFrom` | $\exists R.C$ |
| `hasValue` | $\exists R.\{a\}$ |
| `minCardinality` | $\geq nR$ |
| `maxCardinality` | $\leq nR$ |
| `inverseOf` | $R^-$ |

Die RDF/XML-Darstellung ist dabei noch nicht gezeigt, da sie ziemlich unübersichtlich ist. Z.B. kann das Konzept Mensch $\sqcap$ Weiblich in XML-Notation als

```
<owl:Class>
 <owl:intersectionOf rdf:parseType="Collection">
 <owl:Class rdf:about="#Mensch"/>
 <owl:Class rdf:about="#Weiblich"/>
 </owl:intersectionOf>
</owl:Class>
```

geschrieben werden, und $\leq 2$hatKind$\top$ kann als

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasChild"/>
  <owl:minCardinality rdf:datatype="&xsd;NonNegativeInteger">
    2
  </owl:minCardinality>
</owl:Restriction>
```

geschrieben werden.

Konstrukte zum Formulieren von Axiomen zeigt die folgende Tabelle

| Konstruktor | Syntax in DL |
|---|---|
| `subClassOf` | $C_1 \sqsubseteq C_2$ |
| `equivalentClass` | $C_1 \equiv C_2$ |
| `subPropertyOf` | $R_1 \sqsubseteq R_2$ |
| `equivalentProperty` | $R_1 \equiv R_2$ |
| `disjointWith` | $C_1 \sqcap C_2 \equiv \bot$ bzw. $C_1 \sqcap \neg C_2$ |
| `sameAs` | $\{a_1\} \equiv \{a_2\}$ |
| `differentFrom` | $\{a_1\} \equiv \neg\{a_2\}$ |
| `TransitiveProperty` | definiert eine transitive Rolle |
| `FunctionalProperty` | definiert eine funktionale Rolle |
| `InverseFunctionalProperty` | definiert eine inverse funktionale Rolle |
| `SymmetricProperty` | definiert eine symmetrische Rolle |

Ein Grund für die beiden Varianten OWL Lite und OWL DL ist, dass OWL DL als zu umfangreich angesehen wird, so dass sich unerfahrene Benutzer schwer tun beim Verwenden. Auch aus Komplexitätssicht sind die Sprachen verschieden: Subsumption und Konsistenztest sind in beiden Sprachen entscheidbar, aber in $\mathcal{SHOIN}(\mathbf{D})$ sind die Tests NEXPTIME-vollständig, während in sie in $\mathcal{SHIN}(\mathbf{D})$ „nur“ EXPTIME-vollständig sind.

Es gibt einige Inferenzwerkzeuge für die OWL-Ontologien, eine Auswahl ist: Racer (http://www.racer-systems.com/), FaCT++ (http://owl.man.ac.uk/factplusplus/), Pellet (http://pellet.owldl.com/).

# Literatur


**Allen, J. F.** (**1983**). Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843.

**Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., & Patel-Schneider, P. F.** (**2010**). *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, New York, NY, USA, 2nd edition.

**Baader, F. & Morawska, B.** (**2009**). Unification in the description logic $\mathcal{EL}$. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, RTA '09, pages 350–364. Springer-Verlag, Berlin, Heidelberg.

**Bratko, I.** (**1990**). *Prolog – Programming for Artificial Intelligence*. Addison Wesley.

**Donini, F. M., Lenzerini, M., Nardi, D., & Nutt, W.** (**1997**). The complexity of concept languages. *Inf. Comput.*, 134(1):1–58.

**Ebbinghaus, H.-D., Flum, J., & Thomas, W.** (**1986**). *Einführung in die mathematische Logik*. Wissenschaftliche Buchgesellschaft Darmstadt.

**Eder, E.** (**1992**). *Relative Complexities of First order Calculi*. Vieweg, Braunschweig.

**Gal, A., Lapalme, G., Saint-Dizier, P., & Somers, H.** (**1991**). *Prolog for Natural Language Processing*. John Wiley & sons.

**Haken, A.** (**1985**). The intractability of resolution. *Theoretical Computer Science*, 39:297–308.

**Michalewicz, Z.** (**1992**). *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 3 edition.

**Minsky, M.** (**1975**). A framework for representing knowledge. In P. H. Winston, editor, *The Psychology of Computer Vision*, pages 211–277. McGraw-Hill, London.

**Nebel, B.** (**1997**). Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ord-horn class. *Constraints*, 1(3):175–190.

**Nebel, B. & Bürckert, H.-J.** (**1995**). Reasoning about temporal relations: a maximal tractable subclass of allen's interval algebra. *J. ACM*, 42(1):43–66.

**Pereira, F. C. & Shieber, S. M.** (**1987**). *Prolog and Natural-Language Analysis*. CSLI.

**Robinson, J. A.** (**1965**). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41.

**Russell, S. J. & Norvig, P.** (**2010**). *Artificial Intelligence - A Modern Approach* (*3. internat. ed.*). Pearson Education.

**Schenk, R. & Abelson, R. P.** (**1975**). Scripts, Plans and Knowledge. In *International Joint Conference on Artificial Intelligence*, pages 151–157.

**Smith, B.** (**1982**). *Prologue to Reflection and Semantics in a Procedural Language*. Readings in Knowledge Representation. Morgan Kaufmann, California.

**Valdés-Pérez, R. E.** (**1987**). The satisfiability of temporal constraint networks. In *Proceedings of the sixth National conference on Artificial intelligence - Volume 1*, AAAI'87, pages 256–260. AAAI Press.

**Wegener, I.**, editor (**1996**). *Highlights aus der Informatik*. Springer, Berlin.