Stephan Heuer
Vererbung, Semantische Netzwerke und Wissensrepräsentation
1992

Stephan Heuer
Vererbung, Semantische Netzwerke und Wissensrepräsentation
1992

Stephan Heuer
Vererbung, Semantische Netzwerke und Wissensrepräsentation
1992

Stephan Heuer
Vererbung, Semantische Netzwerke und Wissensrepräsentation
1992

Stephan Heuer
Vererbung, Semantische Netzwerke und Wissensrepräsentation
1992

Angewandte Mathematik

Vererbung, Semantische Netzwerke und Wissensrepräsentation

Inaugural-Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften im Fachbereich Mathematik
der Westfälischen Wilhelms-Universität Münster

vorgelegt von
Stephan Heuer
aus Münster

1992

Dekan:	Prof. Dr. Elisabeth Peveling
Erster Referent:	Prof. Dr. Colin W. Cryer
Zweiter Referent:	Prof. Dr. Wolfram-M. Lippe

Mündliche Prüfung:
Tag der Promotion:

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Einführung 1

1.2 Künstliche Intelligenz 2

1.2.1 Historie 5

1.3 Expertensysteme 6

1.4 Wissensrepräsentation 10

1.5 Expertensystem-Shells 11

1.5.1 EMYCIN 12

1.5.2 KEE 13

1.5.3 PlanPower, GABI und EVA 14

1.6 Kurze Inhaltsbeschreibung 15

1.6.1 Teil I: Die Expertensystemshell **DONALD** 15

1.6.2 Teil II: Vererbungstheorie 19

1.6.2.1 Theoretische Ansätze 20

1.6.2.2 Direkte Semantik der Vererbung 21

1.6.2.3 Erweiterungen des direkten Ansatzes 23

1.6.2.4 Komplexität der Vererbung 25

1.6.3 Ausblick 26

2 Formalismen zur Wissensrepräsentation **28**

2.1 Einführung 28

2.2 Graphen und Netzwerke: Grundbegriffe 29

2.3 Aussagen- und Prädikatenlogik: Grundbegriffe 33

2.3.1 Syntax der Aussagenlogik 33

2.3.2 Semantik der Aussagenlogik 34

2.3.3 Aussagenlogische Theorien 37

2.3.4 Deduktionssysteme 39

2.3.5 Syntax der Prädikatenlogik 41

2.3.6 Semantik der Prädikatenlogik 42

2.3.7 Prädikatenlogische Theorien 45

2.3.8 Deduktionssysteme 46

2.3.9 Klauseln 47

2.3.10 Prädikatenlogik zweiter Stufe 48

2.3.11 Modallogik 49

2.4 Wissenrepräsentation in LISP 53

2.5 Regelbasierte Systeme 55

2.5.1 Interpretierer für Produktionsregeln 55

2.5.2 Metaregeln 59

2.5.3 Effizienz 59

2.5.4 Regeln in Expertensystemen 60

2.6 Wissenrepräsentation in Prolog 61

2.6.1 Das Resolutionsprinzip 62

2.7 Strukturierte Objekte 64
2.7.1 Semantische Netze 66
2.7.2 Frames 68
2.8 Objektorientierte Programmierung 72
2.9 Logik 75
2.9.1 Das Frame Problem 76
2.9.2 Nichtmonotone Logik 77
2.10 Vergleich der Formalismen 78

3 Implementation semantischer Netzwerke 80

3.1 Einführung 81
3.2 FRL 81
3.3 KRL 85
3.3.1 Überblick 85
3.3.2 Syntax 88
3.3.3 Deskriptoren in KRL 88
3.3.4 Unit Kategorien 91
3.3.5 Matching 93
3.4 NETL 94
3.4.1 Übersicht 94
3.4.2 Individuen 95
3.4.3 Typen 98
3.4.4 Rollen 99
3.4.5 Aussagen 101
3.5 Bewertung 103
3.5.1 FRL 103
3.5.2 KRL 104
3.5.3 NETL 105

4 Die Expertensystem-Shell DONALD-III 107

4.1 Einführung 107

4.2 Strategien zur Problemlösung 108

4.2.1 Heuristische Klassifikation 110

4.2.2 Hierarchisches Hypothetisieren und Testen 113

4.2.3 Konstruktives Problemlösen 114

4.3 Motivation des Propose-and-Apply Ansatzes 116

4.4 Wahl der Implementierungssprache 120

4.4.1 Smalltalk 121

4.4.2 Prolog 122

4.4.3 LISP 122

4.4.4 Bewertung 123

4.5 Wissensrepräsentation mit Frames 124

4.5.1 Klassen, Teilklassen und Instanzen 124

4.5.2 Die Definition von Slots in Klassen und Teilklassen . . 126

4.5.3 Die Definition von Views in Teilklassen und Instanzen . 128

4.5.4 Die Syntax für Frames in **DONALD** 129

4.5.5 Implementation der Frames 131

4.5.6 Die interne Darstellung der Frames 132

4.6 Facetten und Demons in **DONALD** 134

4.6.1 Facetten 134

4.6.2 Demons 137

4.7 Produktionsregeln 139

4.8 Informationsvererbung in **DONALD** 142

4.8.1 Vererbungsstrategie in FRL 142

4.8.2 Vererbungsstrategie in **DONALD** 143

4.8.3 Korrektheit der Vererbung 146

4.9 Das Matchen von Frames 149

4.10 Prototypen und Hypothesen 151

4.10.1 Hypothesen 151

4.10.2 Prototypen 154

4.10.3 Der Datentyp HList 156

4.11 Die Synthese der Lösung 160

4.12 Nichtmonotones Schließen 163

5 Vererbungshierarchien **167**

5.1 Einführung 167

5.2 Taxonomische Hierarchien 168

5.3 Vorteile hierarchischer Strukturen 169

5.4 Die Notwendigkeit von Ausnahmen 170

5.5 Normative Inferenzen 171

5.6 Mehrfache Vererbung 172

5.7 Probleme mit Mehrfacher Vererbung 173

5.8 Vererbung und Prädikatenlogik 177

5.9 Nichtmonotone Logik 178

5.10 Default Logik 180

5.11 Bewertung 185

6 Direkte pfadbasierte Vererbungssysteme 186

6.1 Einführung 187

6.2 Die Idee der direkten Methode 189

6.3 Terminologie 190

6.4 Formale Definitionen 193

6.5 Eigenschaften 203

6.6 Theoretische Alternativen 206

6.6.1 Wissenshungrige und skeptische Methoden 206

6.6.2 Aufwärts- und Abwärtskonstruktion 207

6.6.3 On-path und Off-path intermediäre Token 208

6.6.4 Erweiterungen 210

6.7 Klassifikation von Netzwerken 211

6.8 Netzwerke mit Präferenzen 212

7 Komplexitätsbetrachtung 214

7.1 Probleme und Algorithmen 214

7.2 Entscheidungsprobleme 217

7.3 Die Klasse **P** von Problemen 218

7.4 Die Klasse **NP** von Problemen 219

7.5 Vergleich der Klassen **P** und **NP** 221

7.6 Die Komplexität der direkten Vererbung 223

7.7 Experimentelle Erfahrungen 227

7.8 Vererbung in **DONALD-III** 228

8 Schlußbetrachtung **232**

8.1 Zusammenfassung 232

8.2 Ausblick 234

A Ein Beispieldialog **236**

B Literaturverzeichnis **250**

Abbildungsverzeichnis

1.1 Mentale Fähigkeiten 2

1.2 Komponenten eines Wissensbasierten Systems 7

1.3 Das Hauptziel des Expertensystems MYCIN 13

2.1 Eine MYCIN Regel mit Erklärung 55

2.2 Ein typischer Interpretierer für Produktionsregeln 58

2.3 Eine MYCIN Regel in Prolog 64

2.4 Eine Regel aus dem Optionsgeschäft in Prolog 65

2.5 Semantisches Netzwerk zur Darstellung des Begriffs *Computer* 69

2.6 Die Klasse Anleger. 71

2.7 Die Instanz Options-Anleger. 71

2.8 Darstellung des Optionsanlegers mit FLAVORS 74

3.1 NETL: Der *INDV Knoten Clyde 97
3.2 NETL: der *TYPE Knoten Elephant 100
3.3 NETL: Der *INDV Knoten Clyde im Detail 103

4.1 Stufen bei dem Problemlösungsprozeß 109
4.2 Generische Operationen zur Analyse eines Systems 111
4.3 Generische Operationen zur Synthese eines Systems 112
4.4 Heuristische Klassifikation 112
4.5 Flußdiagramm einer Beratung mit **INVEST** 119
4.6 Die Klasse Aktie 125
4.7 Die Instanz BAG 126
4.8 Die Klasse Wertpapier 126
4.9 Die Teilklasse Kursneutrales Anlagepapier 127
4.10 Die Teilklasse Bundesschatzbrief 127
4.11 Die Teilklasse Bundesschatzbrief-A 127
4.12 Ein spezielles Wertpapier 128
4.13 Facetten und Demons eines Slots 135
4.14 Die Definiton der Klasse `Rule` 140
4.15 Regeln mit Variablen in **DONALD** 140
4.16 Eine Regel für Investment Anleger 141
4.17 Eine Frame-Hierarchie in **DONALD-III** 145
4.18 Ein Vererbungsbeispiel 147
4.19 Die Klasse Hypothesis 152

4.20 Die Instanz .RENTENF 153
4.21 Ein semantisches Netz von Hypothesen und Prototypen . . . 155
4.22 Die Instanz Kunde-1 156
4.23 Der Prototyp KURSN 157
4.24 Umgebungen der Prototypen OPT und KURSN 159
4.25 Hierarchie von Hypothesen 164

5.1 Das Tweety Triangle 170
5.2 Ein Netzwerk mit redundanter Information 174
5.3 Das Nixon Diamond 176

6.1 Gegenbeispiel 204
6.2 Das kaskadierte Nixon-Diamond. 207
6.3 Das Problem des Königselephanten 209

7.1 Konstruktion zu dem Beweis von Theorem 7.2 224
7.2 Der wesentliche Teil der Vererbung in **DONALD-III** 229

Tabellenverzeichnis

4.1 Framesyntax in **DONALD** 130
4.2 Die interne Form der Frames (als Struktur) 132
4.3 Die interne Form der Views (als Struktur) 133
4.4 Die interne Form der Slots (als Struktur) 133
4.5 Syntax für *Constraints* in **DONALD** 136

Kapitel 1

Einleitung

1.1 Einführung

Ziel dieser Arbeit ist es, sowohl praktische als auch mathematische Beiträge auf dem Gebiet der Wissensrepräsentation in Expertensystemen zu leisten, und zwar insbesondere im Bereich Wissensrepräsentation durch Semantische Netzwerke und Frames.

Da die Enwicklung von Expertensystemen ein Teilgebiet der Künstlichen Intelligenz *(Artificial Intelligence)* darstellt, soll als erstes das durch diesen Begriff umrissene Gebiet der Informatik kurz betrachtet werden.

Nach einem kurzen historischen Überblick folgt dann die Charakterisierung des speziellen Teilgebiets der *Wissensbasierten Systeme* oder *Expertensysteme*. Da die Wissensrepräsentation eine der beiden zentralen Komponenten jedes Expertensystems ist, wird ihr ein weiterer Abschnitt gewidmet. Den Abschluß des ersten Kapitels bildet eine kurze Beschreibung des Inhalts dieser Arbeit.

1.2 Künstliche Intelligenz

In [Charniak and McDermott, 1985] findet sich folgende sehr allgemeine Definition von Künstlicher Intelligenz (KI):

> Artificial Intelligence is the study of mental facilities through the use of computational models.[1]

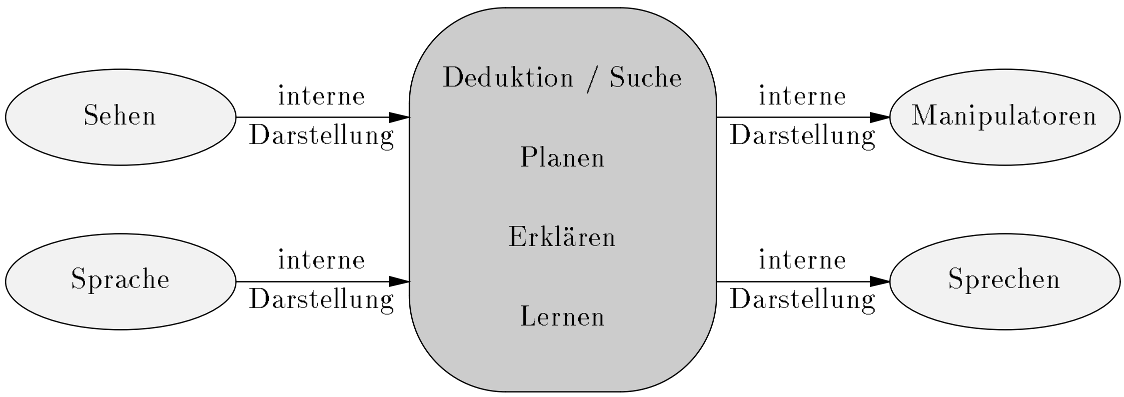


Abbildung 1.1: Mentale Fähigkeiten

In [Barr and Feigenbaum, 1981a] definiert Edward E. Feigenbaum die Künstliche Intelligenz etwas konkreter in folgender Weise:

> Artificial Intelligence (AI) is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit the characteristics we associate with intelligent human behavior — understanding language, reasoning, solving problems and so on.[2]

[1] Künstliche Intelligenz ist das Studium mentaler Fähigkeiten durch Verwendung von Berechnungsmodellen.

[2] Künstliche Intelligenz (KI) ist der Teil der Computerwissenschaft, der sich mit dem Design intelligenter Computersysteme beschäftigt, das heißt mit Systemen, die Charakteristiken aufweisen, die wir mit intelligentem menschlichem Verhalten assoziieren — Sprachverständis, Schlußfolgern, Problemlösen und so weiter.

Das Wort Intelligenz in diesen Definitionen ist ein wenig irreführend. Mit *intelligent* wird im allgemeinen Sprachgebrauch nur besonders kreatives Vorgehen von *Menschen* bezeichnet. Man könnte also glauben, es handele sich darum, eine Technik zur Entwicklung besonders tiefgründiger Einsichten zu erarbeiten.

Ganz im Gegensatz dazu geht es bei den interessantesten Problemen der Künstlichen Intelligenz um die Untersuchung von Fähigkeiten, die jeder ganz normale Mensch besitzt. Diesem fällt es z.B. außerordentlich leicht, Gegenstände zu erkennen und sich so im dreidimensionalen Raum zurechtzufinden. Auch das Verstehen natürlicher gesprochener Sprache findet oft ohne erkennbare Anstrengung statt. Bis heute sind Computer der menschlichen Informationsverarbeitung nur bei der Bearbeitung so stupider Aufgaben wie dem Multiplizieren vielstelliger Zahlen (oder ähnlich wohldefinierter Probleme) überlegen.

Leider hat sich bis heute nicht allzuviel an dem Mißverständis der Rolle des Computers im allgemeinen und der Aufgabenstellung der Künstlichen Intelligenz im besonderen geändert. Dieses Mißverständis ist wohl nach wie vor in unserer Gesellschaft verbreitet und wurde schon von Marvin Minsky in seinem Artikel „Steps Toward Artificial Intelligence“ [Minsky, 1963] beschrieben:

> A visitor to our planet might be puzzled about the role of computers in our technology. On the one hand, he would read and hear all about wonderful "mechanical brains" baffling their creators with prodigious intellectual performance. And he (or it) would be warned that these machines must be restrained, lest they overwhelm us by might, persuasion, or even by the relevation of truths too terrible to be born. On the other hand, our visitor would find the machines being denounced, on all sides, for their slavish obedience, unimaginative literal interpretations, and incapacity for innovation or initiative; in short, for their inhuman dullness.

Seit der Erfindung des Computers in den vierziger Jahren ist es nun möglich geworden, sich direkt ein Bild von dem Schwierigkeitsgrad der von Menschen zu bewältigenden Aufgaben zu machen, indem man versucht, dieselbe Aufgabe mit Hilfe des Computers zu lösen. Es stellte sich heraus, daß zur Nachbildung der grundlegenden Bildverarbeitungsschritte bereits Millionen von arithmetischen Operationen nötig sind (siehe z.B. [Marr, 1982] oder [Ballard and Brown, 1982]).

Da Computer und Gehirne aus völlig unterschiedlichen Materialien bestehen, ist diese Diskrepanz eigentlich nicht weiter verwunderlich. Der Künstlichen Intelligenz liegt aber eine zentrale Arbeitshypothese zu Grunde:

> Die Funktion des menschlichen Gehirns kann auf einer gewissen Ebene als eine Art Berechnung verstanden werden.

Um welche Ebene genau es sich dabei handelt, ist eben genau die Frage, deren Klärung man näher kommen will. Das obige Dogma ist natürlich nicht unumstritten (siehe z.B. [Dreyfus, 1985a] und [Dreyfus, 1985b]). Je näher diese Hypothese der Wirklichkeit kommt, um so mehr wird sich die Anwendung von Berechnungsmodellen zur Studie von mentalen Fähigkeiten auszahlen.

Es liegt also der Schluß nahe, die Künstliche Intelligenz sei als ein Teilgebiet der Psychologie anzusehen. Die meisten Wissenschaftler würden dem jedoch widersprechen. Erstens beschäftigt sich die KI mit ablauffähigen Programmen, während sich Psychologen im allgemeinen mehr zu experimenteller Verifikation von Hypothesen hingezogen fühlen. Entscheidend ist jedoch die Tatsache, daß die KI sich zwar mit dem mit Intelligenz einhergehenden Verhalten beschäftigt, aber in keinster Weise auf eine bestimmte Methode zur Erzielung dieser Ergebnisse festgelegt ist. Ein KI-Programm kann also durchaus anders vorgehen als ein Mensch.

Die Entwicklungsphase von KI-Programmen ist oft von einer Vielzahl von Designentscheidungen geprägt. Meistens stehen keine objektiven Kriterien oder auch nur Erfahrungswerte zur Verfügung. In diesem Fall kann es dann durchaus sinnvoll sein, sich mit der Methode menschlicher Problemlösung zu beschäftigen, die ja auf jeden Fall *eine* funktionierende Lösung liefert. Schwierig ist hierbei noch die genaue Analyse und Implementation der verwendeten Methoden.

Das ultimative Ziel der Künstlichen Intelligenz (das allerdings in weiter Ferne liegt und wahrscheinlich auch niemals erreicht werden wird) ist die Schaffung eines künstlichen Wesens. Dieses Wesen besteht aus den in Abbildung 1.1 auf Seite 2 dargestellten Teilmodulen. Die Grobstruktur dieses Diagramms ist `Eingaben` → `Interna` → `Ausgaben`. Die Verbindungen zu den `Interna` sind mit *Interne Darstellung* bezeichnet. Genau diese internen Darstellungen werden uns im Verlaufe dieser Arbeit beschäftigen. Auch auf diesem Spezialgebiet gibt es eine Reihe vollkommen unterschiedlicher Paradigmen, auf die in Kapitel 2 noch eingegangen wird.

1.2.1 Historie

Die Entwicklung der Künstlichen Intelligenz begann vor ca. 40 Jahren mit einer von euphorischer Zuversichtlichkeit und Experimentierfreudigkeit geprägten Phase[McCarthy, 1985].

Damals hatten McCarthy und seine Mitarbeiter [McCarthy *et al.*, 1962] mit der Umsetzung des zunächst nur rein theoretischen Konzepts des λ-Kalküls in eine Programmiersprache für Computer mit Namen LISP (LISt Processing), die das damals als einzige Alternative zur Verfügung stehende FORTRAN (FORmula TRANslator) bei der Verarbeitung symbolischer Information weit in den Schatten stellte, die Möglichkeit geschaffen, nicht nur Probleme der Ingenieurswissenschaften zu lösen oder numerische Algorithmen zu implementieren, sondern auch den Versuch zu unternehmen, den Computer zur Lösung von solchen Problemen einzusetzen, wie sie auch Menschen tagtäglich bearbeiten müssen.

Mit Hilfe dieser neuen Programmiersprache wurde es möglich, Programme zu erarbeiten, die nicht nur auf ein ganz spezielles Problem anwendbar sind. Der von Newell und Simon entwickelte General Problem Solver (GPS) [Newell and Simon, 1963] basiert auf einer Technik, die als *means-ends* Analyse bezeichnet wird. Es geht hierbei um die Darstellung des Zustandes der problemrelevanten Welt in symbolischer Form durch die Angabe aller wichtigen Relationen zwischen den vorhandenen Objekten und ihren Eigenschaften, sowie um die Charakterisierung der möglichen Aktionen durch Angabe von Voraussetzungen und Auswirkungen auf den zukünftigen Zustand des Systems. Wenn dies gelänge, ergäbe sich die Aussicht auf die Verwendbarkeit eines allgemeinen Problemlösungsalgorithmus. Voraussetzung ist aber die Repräsentation der

Zustände und Aktionen in symbolischer Form. Genau dieser Punkt stellte sich als interessant und äußerst problematisch heraus.

Means-ends geht von einem gegebenen Startzustand aus. Dieser Startzustand wird durch Anwendung der gegebenen möglichen Operatoren sukzessive in den Zielzustand überführt. Der Fortschritt im Problemlösungsprozeß wird dabei durch die Berechnung und Bewertung der zwischen Zuständen vorliegenden Differenzen verfolgt. Jeder Operator stellt dabei Voraussetzungen für seine Anwendbarkeit und gibt an, welche Differenzen zwischen Zuständen er reduzieren kann. Falls die Voraussetzungen eines Operators im aktuellen Zustand nicht gegeben sein sollten, so betrachtet GPS die Erfüllung dieser Voraussetzungen als Teilziel und löst dieses rekursiv als eigenständiges Problem.

Zum Überblick über den aktuellen Stand der Forschung empfiehlt sich z.B. die Konsultation des vierbändigen Standardwerks von Barr und Feigenbaum [Barr and Feigenbaum, 1981a] oder des sehr umfassenden und auch technischeren Werks von Charniak und McDermott [Charniak and McDermott, 1985].

Nilsson gibt in [Nilsson, 1980] eine theoretisch sehr gut fundierte Darstellung. Besonders gelungen ist darin das Kapitel über Frame-strukturierte Repräsentationen, das sowohl ihre logischen Grundlagen als auch die speziellen Verarbeitungs- und Organisationsformen von Frames betrachtet. Auch in das Resolutionsprinzip zum mechanischen Beweisen von Theoremen wird hier gut eingeführt.

Eine gute Einführung bieten ebenso [Rich, 1983] oder [Winston, 1984]. Shapiros zweibändige *Encyclopedia of Artificial Intelligance* ([Shapiro, 1987a] und [Shapiro, 1987b]) bietet auf über 1200 Seiten Definitionen und weiterführende Literaturhinweise für alle relevanten Begriffe der gesamten KI.

1.3 Expertensysteme

In der vorliegenden Arbeit wird insbesondere auf den Teilbereich **Wissensbasierte Expertensysteme** der KI eingegangen. Gerade die bereits erwähnten Probleme mit dem Begriff der KI werden in diesem recht klar umrissenen

Aufgabenfeld vermieden. Hier geht es um die Simulation menschlichen Problemlösens durch einen Experten des jeweiligen Gebiets. Ein Experte sticht aber nicht notwendigerweise durch seinen hohen Intelligenzquotienten hervor, sondern vielmehr durch spezialisiertes Know-How.

Folgende Definition soll die wesentlichen Aspekte darstellen:

> Wissensbasierte Expertensysteme enthalten menschliches Wissen in einer Form, das sie zur Lösung von Problemen befähigt, die ansonsten *menschliche* Intelligenz erfordern. Wissensbasierte Systeme speichern Wissen elektronisch und sind auch in der Lage dieses anzuwenden.

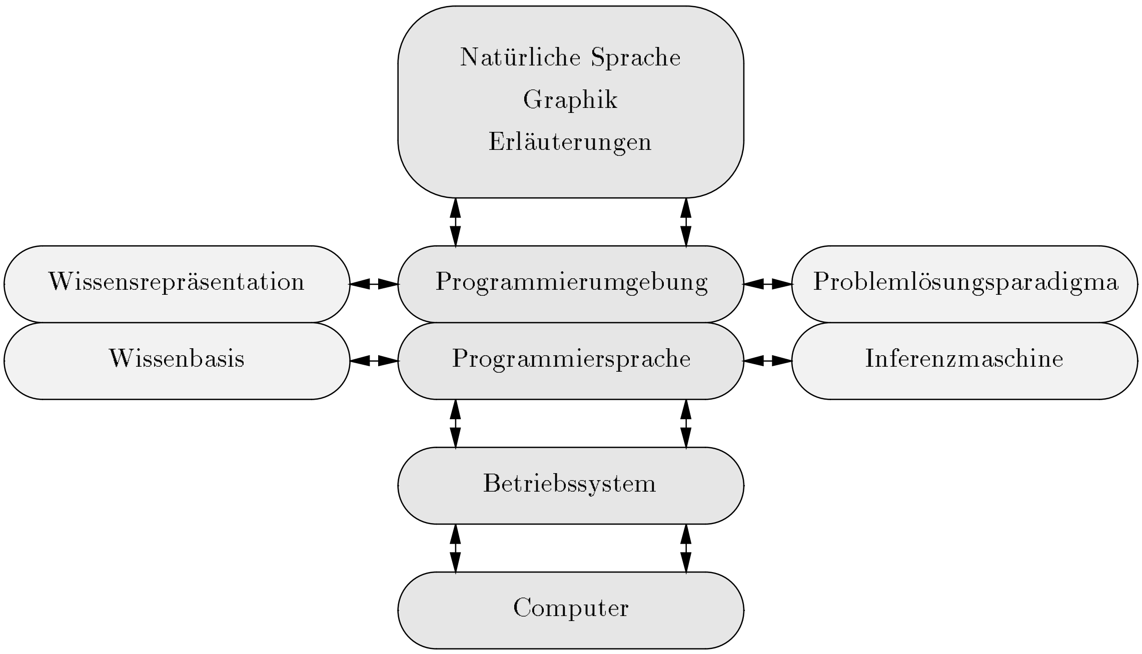


Abbildung 1.2: Komponenten eines Wissensbasierten Systems

Im Moment wird noch das meiste menschliche Wissen einzig und allein in Buchform angeboten. Dieses im wesentlichen passive Wissen muß noch vom Leser gefunden, interpretiert und dann zur Anwendung auf die Lösung eines konkreten Problems angepaßt werden.

Die meisten heutigen Computerprogramme enthalten keine signifikante Menge von Wissen, sie bestehen aus Algorithmen und Daten. Algorithmen bestimmen, wie bestimmte Probleme anzupacken sind, und die Daten charakterisieren die Parameter einer speziellen Probleminstanz. Leider paßt menschliches Wissen oft nicht in dieses Modell, sondern besteht aus vielen kleinen Fragmenten von Know-How.

Zum Zweck der Modellierung von Sachverhalten und Lösungsmethoden stellen Wissensbasierte Systeme zweierlei Arten von Information dar:

- Information über die in der Anwendungsdomäne des Systems vorkommenden und relevanten Objekte, die im allgemeinen sowohl konkreter als auch gedanklicher Art ist. Die Darstellung der Information sollte neben einfacher Beschreibung von Eigenschaften auch die Kodierung von komplexeren strukturellen Zusammenhängen ermöglichen.
- Information über die Art und Weise, wie mit den obigen Objekten Probleme gelöst werden können.

Wissensbasierte Systeme stellen nun Organisationskonzepte zur Verfügung, die es erlauben, diese Wissensfragmente in einer sogenannten Wissensbasis[3] *(Knowledge Base)* zu speichern und bei Bedarf wieder aufzufinden. Daraus ergeben sich eine Reihe von Unterschieden zu konventionellen Programmen unter den Aspekten Programmorganisation, Darstellung des Wissens und Interpretation dieser Darstellung, die sich letztendlich auch in der mit dem Benutzer stattfindenden Interaktion niederschlagen.

Natürlich läßt sich mit Hilfe von Wissensbasierten Systemen nicht jedes beliebige Problem lösen, es gibt vielmehr eine Reihe von Anwendungen, die den Einsatz von Expertensystemen nahelegen. Es handelt sich hierbei insbesondere um medizinische Diagnose, Computerkonfiguration, Interpretation von chemischen Daten und Strukturanalyse, Sprach- und Bildverarbeitung, Unterstützung von finanziellen Entscheidungsprozessen, Signalinterpretation, Suche nach Mineralien, Erzlagerstätten und Ölvorkommen, Militärorganisation

[3] Manchmal wird auch der Begriff Wissensbank verwendet, der die Verwandtschaft zu einer herkömmlichen Datenbank unterstreicht.

und Planung, Hilfestellung beim Umgang mit Computern und Computerprogrammen[4] und auch Design von VLSI-Schaltkreisen[5].

Die Entwickler von Expertensystemen stellten sich dabei folgenden Anforderungen:

- Darstellung einer relativ großen Menge von unsicheren und heuristischen Wissensfragmenten.
- Automatische Lösung von Problemen mit einer variablen Menge von Strategien, die eventuell erst im Laufe des Lösungsprozesses dynamisch ausgesucht werden.
- Einfache Anpassung des Systems an neues Wissen in kürzester Zeit, d.h. gute Wartbarkeit.
- Das in den Systemen enthaltene Wissen soll auch zur Generierung von sinnvollen Erklärungen für sein Verhalten dienen können.

Damit stellen Expertensysteme ein Gebiet mit direktem Bezug zu der Anwendung dar, auf dem die Entwicklung seit Anfang der achziger Jahre sehr stürmisch verlaufen ist. Aber auch hier machte sich in den letzten Jahren eine gewisse Ernüchterung breit, als nach anfänglich überwiegend positivem Echo auf die ersten prototypischen Anwendungen eine mehr skeptische Stimmung nicht zu übersehen war. Dafür sind mehrere Faktoren verantwortlich zu machen:

- Durch den Erfolg kleiner prototypischer Systeme wurden übersteigerte Erwartungshaltungen geweckt. Dabei blieb unberücksichtigt, daß gerade auch die Menge der zu berücksichtigenden Informationen ein Problem darstellt. Was im Kleinen funktionierte, muß nicht unbedingt auf eine umfangreichere Anwendung übertragbar sein.

[4] Zu diesem Problemkreis gehört auch das von der Europäischen Gemeinschaft im Rahmen der *ESPRIT* Initiative gefördete Projekt **FOCUS** (Front Ends For Open And Closed User Systems), das in Zusammenarbeit mit mehreren Universitäten und Partnern aus dem kommerziellen Softwareentwicklungsbereich am Institut für Numerische und instrumentelle Mathematik verfolgt wird. Ein Aspekt dieser Forschungsarbeit ist die Hilfestellung bei der Anwendung einer umfangreichen Bibliothek mathematischer Software.

[5] Very Large Scale Integration

- Die kommerziell angebotenen Systeme stehen in einer eher verwirrenden Marktvielfalt zur Verfügung und bieten ein unklares Leistungsprofil. Viele Systeme stehen auch für Personal Computer zur Verfügung, vielfach lassen sich mit ihnen aber wirklich nur kleine Demonstrationsprobleme lösen, wie auch eigene Versuche leider zeigten.
- Auch das fehlende Verständnis der theoretischen Grundlagen der Wissensrepräsentation verhinderte die erfolgreiche Analyse von Fehlschlägen.

Einen guten Überblick über diese Problematik geben [Buchanan and Shortliffe, 1984], [Davis and Lenat, 1980], [Hayes-Roth *et al.*, 1983], [Mitchie, 1984], [Alty and Coombs, 1984] und [Jackson, 1990].

Im Lichte der obigen Bemerkungen läßt sich die Idee des Wissensbasierten Systems als eine Methode verstehen, komplexe Softwaresysteme entlang der Organisation von Abbildung 1.2 auf Seite 7 zu strukturieren, die die Trennung des System in eine Wissensbasis und eine darauf arbeitenden Problemlösungskomponente beinhaltet. Neben der praktischen Entwicklung eines vollständigen Expertensystems wird die Implementation von Wissensbasen und deren theoretische Eigenschaften im Vordergrund dieser Arbeit stehen.

1.4 Wissensrepräsentation

Wissensrepräsentation eröffnet die Frage danach, wie Information, die in symbolischer Form in einem Computer gespeichert ist, Bedeutung erlangen kann, denn die verwendeten Symbole haben ja keine Verbindung mit irgendetwas außerhalb des Computers.

Ganz allgemein erfordert die Lösung von Problemen mit Computern die Darstellung aller *relevanten* Informationen in der Maschine. Dieser Modellierungsprozeß stellt den wesentlichen Designprozeß bei der Erstellung eines komplexen Softwaresystems dar. Während man sich bei vielen theoretischen Problemen der Mathematik und Informatik mit der Kodierung der Sachverhalte durch relativ wenige einfache Konzepte beschränken kann, sind bei Beschäftigung mit Problemen der *realen* Welt weit komplexere Zusammenhänge zu berücksichtigen.

Auf dem Gebiet der Wissensbasierten Systeme bedeutet Wissensrepräsentation die systematische Kodierung des Spezialwissens eines Experten. Der Begriff Repräsentation impliziert im Gegensatz zur Kodierung im Sinne von Verschlüsselung die sinnvolle Organisation des Wissens, so daß dieses erreichbar und anwendbar ist. Konkret macht diese Forderung die Adressierung des Wissensspeichers durch Inhalte und nicht durch Ortsangabe oder Namensangabe (wie in den klassischen Programmiersprachen) unumgänglich.

Winston definiert in [Winston, 1984] eine Repräsentation als Menge von syntaktischen und semantischen Konventionen, die es ermöglichen, Objekte zu beschreiben. Die Semantik gibt dabei an, wie den syntaktischen Konstruktionen Bedeutung zugeordnet werden kann.

Aber nicht nur die Komplexität der auftretenden Strukturen stellt eine neue Herausforderung für das sogenannte Software-Engineering (siehe z.B. [Darnell and Margolis, 1988]) dar, auch die schiere Menge der Information und die stark ausgeprägten Interdependenzien erforderen eine effiziente und redundanzfreie Darstellung, um das Problem der Wartung und Aktualisierung beherrschbar werden zu lassen. Leider lassen sich viele bewährte Lösungsmethoden aus dem Bereich der Datenbanken aufgrund der nur in beschränktem Maße vorhandenen Uniformität der Daten nicht einfach übertragen, obwohl das in letzter Zeit immer häufiger eingesetzte *relationale* Datenbankmodell (siehe z.B. [Date, 1984] für eine Beschreibung einer aktuellen Implementation oder den grundlegenden Aufsatz von Codd [Codd, 1970]) einen Weg zu größerer Flexibilität weist, als sie das schon etwas ältere hierarchische Datenbankmodell zu bieten vermag.

Das zweite und dritte Kapitel wird sich eingehender mit der Problematik der verschiedenen Möglichkeiten zur Wissensrepräsentation befassen, wobei der Schwerpunkt nicht auf dem klassischen regelbasierten Formalismus liegt, der eine Fülle von „wenn ..., dann ...“–Regeln verwendet, sondern auf der Betrachtung Strukturierter Objekte und Semantischer Netzwerke.

1.5 Expertensystem-Shells

Die ersten Expertensysteme wurden *from scratch* erstellt, in dem Sinne, daß die primitiven Daten- und Kontrollsstrukturen einer existierenden Programmiersprache benutzt wurden, um das nötige Wissen darzustellen und die Anwendung zu kontrollieren.

Den nächsten Entwicklungsschritt stellten Implementationen spezieller Regel- oder Frame-Sprachen in einer bestehenden Programmiersprache dar. Wissen wurde dann anschließend in dieser speziellen Sprache dargestellt. Diese neuen Sprachen führten in der Regel dann zu folgenden Abstraktionen:

- *Module,* wie z.B. Produktionsregeln oder Frames zur Darstellung des Wissens.
- *Interpretierer,* um die Aktivierung der obigen Module zu kontrollieren.

Die Module ergeben zusammengefaßt die Wissensbasis des Expertensystems, während der Interpretierer die Inferenzmaschine implementiert. In einigen Fällen wurde klar, das diese Komponenten wiederverwendbar waren, so daß sie als Basis für andere Projekte dienen konnten. Solche Programme enstanden oft aus existierenden Expertensystemen und wurden deshalb unter dem Namen Expertensystem-Schalen *(shells)* bekannt.

1.5.1 EMYCIN

MYCIN [Shortliffe, 1976] wurde ab 1972 an der Stanford Universität in einer Zusammenarbeit zwischen Medizinern und KI-Forschern in LISP programmiert, um einen Arzt, der *kein* Experte bei der Behandlung von Blutinfektionen ist, beim Einsatz von Antibiotika auf diesem Gebiet zu unterstützen. Auf MYCIN wird noch im zweiten Kapitel genauer eingegangen, deshalb sollen hier einige kurze Bemerkungen genügen. MYCIN basiert auf Produktionsregeln, die auch unsicheres Wissen mit Hilfe von sogenannten *Certainty Factors* darstellen können. Diese Faktoren sind ein Maß für die Evidenz des durch Anwendung der Regel gefolgerten neuen Wissens. Als Kontrollstruktur benutzt MYCIN das *Backward Chaining*, das von dem in Abbildung 1.3 dargestellten Hauptziel *(Goal)* ausgeht, die beste Therapie zu finden.

Ausgehend von diesem Ziel, überprüft MYCIN nun die Voraussetzungen aller Regeln, die dieses Ziel erreichen könnten. Sind diese Voraussetzungen nicht

`IF:`	1) There is an organism which requires therapy, and
	2) Consideration has been given to any other organism requiring therapy
`THEN:`	Compile a list of possible therapies, and determine the best one in the list.

Abbildung 1.3: Das Hauptziel des Expertensystems MYCIN

direkt nachprüfbar, so wiederholt sich die Suche nach Regeln, die diese Voraussetzungen schaffen könnten. So entsteht durch Visualisierung ein sogenannter AND/OR Baum oder *Goal Tree*.[6]

Interessant war nun die Idee, MYCINs Komponenten in neuen Systemen wiederzuverwenden. Man entfernte aus dem Expertensystem MYCIN alle auf die Diagnose von Blutinfektionen bezogenen spezialisierten Teile, und erhielt so die Expertensystem-Shell **E**MYCIN[7]. Basierend auf dieser Shell konnte dann auch ein System PUFF [Kunz *et al.*, 1978] zur Interpretation von Lungenfunktionstests entwickelt werden.

Im Rahmen des INTERNIST Projektes [Miller *et al.*, 1982] wurde ein größeres und komplexeres Projekt begonnen. INTERNIST behandelt das gesamte Gebiet der Inneren Medizin. Es handelt sich hier einmal um eine Vielzahl möglicher Krankheitsursachen[8]. Als schwerwiegender stellte sich jedoch die Tatsache heraus, daß es gar nicht ungewöhnlich ist, daß ein Mensch an zehn oder mehr Krankheiten gleichzeitig leidet und sich die Symptome so vermischen und die korrekte Diagnose diffizil machen.

1.5.2 KEE

Das *Knowledge Engineering Environment* (KEE) [Fikes and Kehler, 1985] ist ein Beispiel für eine moderne Expertensystem-Shell. KEE bietet objektori-

[6] AND steht für die Konjunktion der Prämissen, OR für die Auswahl an Regeln zur Herleitung der Konklusion. Siehe z.B. [Charniak and McDermott, 1985, S. 270] für eine genauere Betrachtung.

[7] Das **E** steht für *empty* oder *essential*.

[8] Die Schätzungen variieren zwischen 2.000 und 10.000.

entierte Wissensrepräsentation mit mehrfacher Vererbung und dem typischen Nachrichtenaustausch *(Message Passing)* zwischen den Objekten als Ersatz für die üblichen Prozeduraufrufe. Ebenso enthalten ist eine Regelsprache und ein PROLOG ähnlicher Hornklauselinterpretierer, die beide gut in das Gesamtsystem eingebunden sind. Abgerundet wird das Bild durch eine Programmierumgebung in hochauflösender Graphik.

Der folgende Abschnitt behandelt noch kurz Systeme, die sich mit Anwendungsgebieten befassen, die der Wertpapieranlage ähneln.

1.5.3 PlanPower, GABI und EVA

PlanPower [Stansfield and Greenfield, 1987] assistiert bei der Finanzplanung für Einzelpersonen oder Firmen. Eine Besonderheit ist die Berücksichtigung des Zeitfaktors durch die Repräsentation von Fakten als Zeitreihen [Humpert and Holley, 1988]. PlanPower enthält Wissen über ca. 120 Anlageobjekte, die mit ca. 6000 Regeln verknüpft sind. Durch die Zeitreihenanalyse entsteht eine beträchtliche Verzweigung bei der Planung, so daß eine Planung durchaus eine Stunde Rechenzeit beansprucht.

GABI [Baltes, 1985] ist in LISP und PEARL (siehe [Deering *et al.*, 1981] und [Deering *et al.*, 1982]) implementiert und arbeitet mit Frames. 110 Regeln, die mit den von MYCIN bekannten *Certainty Factors* arbeiten, stellen das Wissen zur Anlageberatung dar. Die Strategie von GABI erfragt in einem Vordialog Daten über den Kunden und versucht dann, daraus Anlagehypothesen zu generieren. Die Hypothese mit der größten Evidenz wird dann anhand der Regeln verifiziert. GABI stellt die Hypothesen aber im Gegensatz zu **DONALD** *nicht* hierarchisch dar, sondern behandelt alle auf derselben Stufe. Für umfangreichere Wissensbasen erscheint dieses Vorgehen problematisch.

EVA [Bachem, 1986] wurde mit Hilfe der Expertensystem-Shell BABYLON [Christaller *et al.*, 1989] erstellt. Der Lösungsansatz ähnelt dem von GABI bekannten Verfahren. Die möglichen Empfehlungen sind bislang nur ein Ausschnitt aus der Angebotspalette einer Bank. Umfassend behandelt wurden bisher nur kursneutrale Anlagen wie Sparbriefe und Termineinlagen. So werden viele interessante Aspekte der Anlageberatung, wie z.B. Währungs- und Kursrisiko, ausgespart.

1.6 Kurze Inhaltsbeschreibung

Diese Arbeit besteht aus einem theoretischen und einem praktischen Teil. Ausgangspunkt war der Entwurf der Expertensystemshell **DONALD** und der mit ihr realisierten Anwendung **INVEST** zur Beratung bei der Wertpapieranlage.[9]

Hierdurch wurden praktische Erfahrungen gewonnen, so daß eine Reihe von Mängeln erkannt werden konnten. Es zeigte sich, daß obwohl die in **DONALD** verwendete Wissensrepräsentation mit Strukturierten Objekten viele Vorteile hat, ihre Theorie doch noch sehr lückenhaft ist.

Es gibt mehrere mathematisch interessante und praktisch relevante Aspekte der Theorie der Semantischen Netzwerke, die im zweiten Teil dieser Arbeit untersucht werden. Die Kapitel 2 und 3 bilden sowohl die Basis für die späteren theoretischen Untersuchungen als auch den Hintergrund für **DONALD**.

1.6.1 Teil I: Die Expertensystemshell DONALD

Mit Clarksons *A Model of the Trust Investment Process* [Clarkson, 1963] liegt eine sehr frühe Arbeit über ein Thema vor, das eine gewisse Verwandschaft zu der in **INVEST** bearbeiteten Problematik aufweist. Resultat des Vergleichs zwischen der Ausgabe des benutzten Trust Investment Modells mit den aufgezeichneten Entscheidungen der Investoren ergaben nach Clarksons Einschätzung starke Hinweise darauf, daß die Mechanismen des verwendeten Modells einen erheblichen Teil des Trust Investments nachbilden. Es handelt sich jedoch im Vergleich zu **INVEST** mehr um eine Leistungsbewertung von Aktien mit mathematischen Methoden. Clarksons Modell geht von einer Liste von möglichen Industriezweigen aus. Unter Berücksichtigung allgemeiner Wirtschaftsdaten und Informationen über einzelne Wirtschaftzweige und Unternehmen werden diejenigen Aktien gesucht und ausgewählt, die die geforderten Attribute haben. Es werden nicht nur algebraische Operationen durchgeführt, sondern auch die Faustregeln eines traditionellen Experten nachgebildet.

[9] In Zusammenarbeit mit Experten der Volksbank Münster und der Westdeutschen Genossenschaftszentralbank (WGZ) in Münster.

DONALD-III ist die Weiterentwicklung der in [Heuer, 1987], [Koch und Heuer, 1986] und [Koch und Heuer, 1987] dargestellten Expertensystem-Shell **DONALD-II** und entstand in enger Zusammenarbeit mit der Erstellung des Expertensystems **INVEST** zur Wertpapieranlageberatung ([Heuer *et al.*, 1988] und [Koch, 1989a]).

Wie in Abschnitt 4.3 ausführlicher diskutiert werden wird, handelt es sich bei der Wertpapieranlageberatung *nicht* um ein traditionelles Diagnoseproblem, sondern vielmehr um die echte *Synthese* einer Lösung. Das führte zum Versagen eines rein regelbasierten Ansatzes. Die Implementierung eines Interpretierers für Produktionsregeln in **DONALD-III** erschien nur sinnvoll, damit die prozedurale Komponente auch ohne Grundkenntnisse der Programmiersprache LISP [Tatar, 1987] für den Konstrukteur der Wissensbasis (den sogenannten *Knowledge Engineer*) beherrschbar ist.

In dem vierten Kapitel werden nun unter anderem auch die Komponenten von **DONALD-III** beschrieben. Dazu gehören die Erzeugung und Verwaltung der Frames und ihre Kompilation in eine interne, optimierte Form. Der Matcher für Frames, ein Regelinterpretierer und das Benutzerinterface vervollständigen das Bild. Den letzten Teil bilden die Algorithmen zur Verwaltung des Netzes von Hypothesen und zur nichtmonotonen Konstruktion der Anlagevorschläge *(Proposals)*.

Bei der Implementation des Systems wurden folgende Ziele verfolgt:

- Repräsentation einer respektablen Menge von Informationen über Wertpapiere. Diese ist stark baumartig strukturiert und wird möglichst redundanzfrei gespeichert.

- Darstellung von prototypischen Situationen und Anlagevorschlägen, die durch Vergleich zwischen von den Experten erwarteten und den beobachteten Situationen das Vorgehen des Systems bestimmen. Die Prototypen und Hypothesen sind untereinander mehrfach vernetzt und bilden komplexe Muster, die keine Darstellung als Baum mehr zulassen.

- *Keine* Verwendung von Produktionsregeln als Paradigma zur Lösung von Problemen. Stattdessen dominiert ein Ansatz zur Erkennung von gespeicherten Mustern in den Daten, der in mehreren Verfeinerungsstufen durchgeführt wird. Diese Muster sind ebenfalls in Form einer *Vererbungshierarchie* abgelegt.

- Integration prozeduraler Komponenten. Die automatische Bildung von Summen und Anteilen innerhalb der Datenfelder soll als einfaches Beispiel genügen. Durch diese Mechanismen kann ebenfalls die Anbindung an externe Datenbanken realisiert werden.

- Modularität. Durch Aufteilung der Funktionen in hierarchisch angeordnete Komponenten mit definierten Schnittstellen können einzelne Teile ausgetauscht werden, ohne andere Module ebenfalls neu designen zu müssen. Diese Eigenschaft ist bei dem inhärent experimentellen Charakter des Systems von großer Wichtigkeit.

- Effizienz. **DONALD-III** sollte auf einem Personal Computer oder einer Workstation ablaufen können. Das konnte durch eine sehr effiziente Wissensrepräsentation und Kompilationsalgorithmen gewährleistet werden.

- Portabilität. Identischer Quellcode kommt auf unterschiedlichen Computersystemen zum Einsatz. Die aktuelle Implementation orientiert sich zu diesem Zweck an Common Lisp [Steele Jr., 1990] in der neuesten Version. Leider steht noch kein ANSI[10] oder ISO Standard zur Verfügung.

- Aufgabenstellungen wie ein adäquates Benutzerinterface sollten dagegen ganz bewußt eine geringere Rolle spielen, da es nicht hauptsächlich um den tatsächlichen Einsatz von **INVEST** in einer Bankfiliale ging.

Trotz der Abstriche in Bezug auf die Benutzerfreundlichkeit waren die in **DONALD** verwendeten Ansätze auch in der Praxis erfolgreich. **INVEST** kann Beratungen unter Berücksichtigung der rund 100 wichtigsten Gruppen von Wertpapieren durchführen. Dabei reicht das Spektrum von Sparbriefen und Bundesschatzbriefen über Aktien bis Optionen.

Nach den anfänglichen Schwierigkeiten mit regelbasierten Ansätzen konnten die Experten das von ihnen gelieferte Wissen ohne Mühe in **INVEST** wiedererkennen. Auch weitere, später zur Beurteilung hinzugezogene Experten der

[10] American National Standard

WGZ konnten rasch mit den verwendeten Formalismen vertraut gemacht werden und sogar direkt Änderungsvorschläge vorbringen. So ergab sich eine recht effiziente Kommunikation zwischen Experten und Entwicklern bei der Pflege und Vervollständigung der Wissensbasis, die ständigen Änderungen unterworfen ist, wie es bei Wissensbasierten Systemen der Regelfall ist. Solche Systeme werden im eigentlichen Sinne niemals *fertig*.

Beim direkten Testeinsatz in einer Münsteraner Bank, der durch die Portierung auf einen Personal Computer möglich war, konnte **INVEST** zeigen, daß es bei der Beratung in der Lage ist, mit menschlichen Kollegen zu konkurrieren. Es soll jedoch betont werden, daß das System niemals als Ersatz für einen menschlichen Berater konzipiert war. Es geht vielmehr um die Unterstützung der Experten durch das gespeicherte Wissen von weiteren kompetenten Beratern. Das System kann so etwas wie eine zweite Meinung vermitteln, die durch die Begründungen des Systems für seine Entscheidungen eventuell neue relevante Aspekte aufzeigen kann. Zusammenfassend bietet die Beratung durch einen Computer folgende Vorteile[11]:

- Kompetente Anlageberatung wird möglich auch dort, wo sie bislang aus Kostengründen nicht machbar war.
- Die Beratung ist objektiv und gründlich.
- Es wird eine flächendeckend gleichmäßige Beratungsqualität erzielt.

Den Schluß der Übersicht über die Shell **DONALD** sollen einige Fakten bilden.

DONALD-III besteht aus circa 20 LISP Sourcecode-Dateien. Dazu kommen noch 10 Wissensbasen. Insgesamt umfaßt das System 10.000 Zeilen Sourcecode, von denen fast 4000 Zeilen auf die Wissensbasen entfallen. Aus diesem Sourcecode entstehen je nach Prozessorarchitektur ungefähr ein Megabyte Objectcode.

[11] Vergleiche [Koch, 1989b].

DONALD-III kann durch Anpassung an das verwendete LISP-System, die in genau einem für den speziellen LISP-Dialekt zugeschnittenen File vorgenommen wird, in verschiedenen Umgebungen ablaufen. Im Moment stehen Implementationen für das recht betagte Franz-LISP[12] auf Workstations der Firma PCS, für Golden Common LISP der Firma Gold Hill auf IBM-kompatiblen Personal Computern[13] und schließlich für die aktuelle Entwicklungsumgebung zur Verfügung. Momentan wird Austin Kyoto Common Lisp (AKCL) in der Version 1.600 unter SUN OS auf einer SUN Sparcstation IPX benutzt. AKCL basiert auf dem originalen KCL Systems [Yuasa and Hagiya, 1985] des Research Institute for Mathematical Sciences der Universität Kyoto, das nun durch die Universität von Austin (Texas) weiterentwickelt, gepflegt und auf neue Hardware- und Betriebssystemversionen portiert wird. Da AKCL auf allen gängigen Versionen von UNIX läuft, steht **DONALD-III** also auch z.B. unter dem für Personal Computer am weitesten verbreiteten SCO Unix v3.2 der Santa Cruz Operation zur Verfügung.

Der Ablauf einer typischen Beratung mit dem Expertensystem **INVEST** ist in Anhang A festgehalten. Die oben erwähnte Sparcstation IPX leistet 28.5 MIPS *(Million Instructions Per Second)* und 4.2 MFLOPS *(Million Floating Point Operations Per Second)*. Auf dieser Hardware benötigt eine typische Beratung ca. 5 Sekunden Rechenzeit und ungefähr 4 Megabyte Speicher und belegt so die Effizienz der Implementation von **DONALD-III**.

1.6.2 Teil II: Vererbungstheorie

Aufgrund der im Laufe des mehrjährigen **DONALD/INVEST** Projektes gemachten Erfahrungen waren aus theoretischer Sicht einige Punkte offengeblieben, die mit der fehlenden Formalisierung der Wissensrepräsentation mit Frames zusammenhängen.

Die Probleme des nichtmonotonen Schließens in **DONALD** und mögliche Lösungsansätze mit Hilfe von Kalkülen der intuitionistischen Modallogik bilden den Hauptteil von [Koch, 1989a]. In der vorliegenden Arbeit geht es

[12] Es handelt sich hierbei um die fast kostenlos verfügbare universitäre Version und nicht um das moderne Franz Common LISP.

[13] Voraussetzung ist hier allerdings mindestens ein Intel 80286 Prozessor und 4 MB Memory.

indessen um die Formalisierung der Wissensrepräsentation mit Strukturierten Objekten. Insbesondere werden Systeme zur hierarchischen Strukturierung von Wissen betrachtet, die als Vererbungssysteme *(Inheritance Systems)* bezeichnet werden.

1.6.2.1 Theoretische Ansätze

Wissensrepräsentationssysteme für den Einsatz in der Künstlichen Intelligenz müssen Informationen effizient darstellen und den schnellen Zugriff auf taxonomisch angeordnetes Wissen erlauben. Zu diesem Zweck bieten sich Systeme an, die auf Netzwerkschemata beruhen. Berechnungen in einem solchen Netzwerk basieren auf Traversierung des Netzwerkes. Die frühesten Systeme gaben entweder überhaupt keine Semantik an, oder spezifierten nur eine prozedurale Semantik. Die Bedeutung einer Darstellung hing also implizit von dem auf ihr arbeitenden Programm ab. Woods erkannte und kritisierte dies in [Woods, 1975]. Seine Bemerkungen führten zu einigem Interesse an einer implementationsunabhängigen Bedeutungstheorie für Netzwerkformalismen.

Die ersten Ansätze lieferte Hayes [Hayes, 1979], der eine *indirekte* Strategie vorschlug. Die Bedeutung eines Netzwerkformalismus wird dabei durch eine Abbildung in die Sprache der klassischen Prädikatenlogik erster Stufe definiert. Liegt so eine Abbildung vor, so kann man das Netzwerk einfach als Variante einer syntaktisch eingeschränkten logischen Theorie betrachten, die nur eine graphische anschauliche Form hat und eventuell noch einige spezielle Inferenzen schneller durchführen kann, als ein allgemeiner Beweis der Prädikatenlogik das vermöchte. Übersetzt man die Kanten eines Netzwerks Γ in eine Menge von äquivalenten Formeln, dann wird eine Aussage genau dann von dem Netz Γ unterstützt, wenn sie zum deduktiven Abschluß dieser Menge gehört. Nachteil dieser Methode ist, daß in der klassischen Logik *jede beliebige* Aussage zum deduktiven Abschluß einer inkonsistenten Menge von Aussagen gehört. Die in Abschnitt 1.6.2.2 vorgeschlagenen Systeme vermeiden ein solches Desaster und können die negativen Effekte von Inkonsistenzen auf eine lokale Umgebung des Konflikts in dem Netzwerk begrenzen. So ein Lokalitätsprinzip findet sich in logischen Formalismen nicht.

Betrachtet man dann Systeme, die auch unsicheres Wissen in hierarchischen Netzwerken darstellen, so stellt sich erneut die Frage nach einer präzisen Definition der Bedeutung. Solche Systeme waren insbesondere FRL [Roberts and Goldstein, 1977a], KRL [Bobrow and Winograd, 1977] und NETL [Fahlman, 1979]. Diese Systeme werden in Kapitel 3 analysiert. Fahlman zeigte selbst, daß die naiven Inferenzalgorithmen seines Systems in einigen Fällen zu bizarren Resultaten führten.

Da bereits die informelle Interpretation der genannten drei Systeme eine nichtmonotone Relation der logischen Konsequenz enthielten, konnte die Übersetzung dieser Formalismen in die klassische Logik nicht gelingen. Es war aber naheliegend, den indirekten Ansatz beizubehalten und weiterhin die Knoten und Kanten der Netzwerke (bzw. die Deskriptoren von KRL) in logische Formeln zu transformieren. Als Ziel der Abbildung mußten nur Sätze einer nichtmonotonen Logik gewählt werden.

Die ersten Ergebnisse lieferte die Arbeit von Etherington und Reiter [Etherington and Reiter, 1983], die Etherington in weiteren Artikeln ausbaute [Etherington, 1986], [Etherington, 1987]. Sie übersetzten vereinfachte Netzwerke des NETL Typs in Formeln einer *Default* Logik. Default Logik wurde von Reiter eingeführt und ist in [Reiter, 1980], [Reiter and Criscuolo, 1981] beschrieben. Mit der Einordnung dieser logischen Formalismen beschäftigt sich unter anderem Kapitel 5.

1.6.2.2 Direkte Semantik der Vererbung

Ganz neue und andersartige Techniken zur Festlegung der Bedeutung eines Netzwerkformalismus für Vererbung mit Ausnahmen entwickelte Touretzky in dem auf seiner Dissertation basierenden [Touretzky, 1986]. Ziel einer semantischen Theorie für einen Formalismus ist es, die Konsequenzen einer Menge von in dem Formalismus dargestellten Fakten zu finden, die Menge der möglichen Konsequenzen zu charakterisieren und schließlich Eigenschaften der ursprünglichen Fakten (wie z.B. Konsistenz) zu überprüfen. Eine Möglichkeit, dies zu erreichen, ist die oben erwähnte Abbildung in einen logischen Formalismus, für den Begriffe wie Konsequenz und Konsistenz bereits definiert sind.

Touretzkys Idee war nun, diesen Zwischenschritt wegzulassen und eine *direkte* semantische Theorie anzugeben. Dazu betrachtet man ein Semantisches Netzwerk als einen endlichen gerichteten Graphen Γ, der direkt gespeicherte Information darstellt. Aus dieser direkten Information ergeben sich aus

Γ durch Vererbungsmechanismen weitere indirekte Schlußfolgerungen. Touretzky übertrug die bestehenden intuitiven Vorstellungen darüber, welche Fakten aus einem Vererbungsnetzwerk herzuleiten sind, in mathematische Definitionen. Betrachtet werden die durch ein Netzwerk möglichen Pfade. Aus diesen Pfaden werden dann Beweise konstruiert, ähnlich dem Vorgehen in der Deduktionstheorie der mathematischen Logik.[14] Die Beschäftigung mit Pfaden als Mittel zur Darstellung von Inferenzen hat den direkten Methoden auch den Namen *pfadbasiert* eingebracht.

Die direkten Formalismen sind ein mehr prozeduraler als deklarativer Ansatz, da sie sich mit der Klassifikation der möglichen Pfade durch ein Netzwerk beschäftigen. Im Gegensatz zur prozeduralen Semantik von NETL ist Touretzkys Theorie aber unabhängig von der aktuellen Implementation und kann zur Überprüfung der Korrektheit einer solchen Implementation dienen.

Die mathematische Logik liefert *zwei* Ansätze zur Untersuchung einer bestimmten Logik. Beim modelltheoretischen oder semantischen Ansatz geht es um die Definition *wahrer* Formeln und die Konstruktion von Modellen, die Formeln erfüllen. Der deduktionstheoretische oder syntaktische Ansatz beschäftigt sich mit dem Begriff der Herleitbarkeit von Formeln ohne Bezug auf wahr oder falsch. Details hierzu finden sich in Kapitel 2. Während Modell- und Beweistheorie durch Vollständigkeitssätze (siehe Theorem 2.13 auf Seite 47) verbunden sind, liefert der pfadbasierte Ansatz nur eine rein syntaktische Konstruktion.

Es liegen allerdings mit [Horty, 1991] auch erste Resultate von John F. Horty (Univerität Maryland) aus dem Team von David S. Touretzky (Carnegie Mellon) und Richmond H. Thomason (Universität Pittsburgh) vor, die für Netzwerke mit *strikten* Verbindungen eine Modelltheorie angeben, die auf einer vierwertigen Logik (siehe [Ginsberg, 1987a], [Ginsberg, 1987b] und [Ginsberg, 1987b]) basiert. Krishnaprasad [Krishnaprasad, 1989] gibt eine auf Circumscription (siehe [McCarthy, 1980], [McCarthy, 1986]) basierende Modelltheorie für den Spezialfall eines baumstrukturierten Systems von Klassen und Eigenschaften an.

[14] Vergleiche auch Abschnitt 2.3.8 auf Seite 46.

1.6.2.3 Erweiterungen des direkten Ansatzes

Das sechste Kapitel behandelt und erweitert die aus Touretzkys Dissertation hervorgegangene Theorie. Ausgehend von der Beobachtung, daß Touretzkys Theorie zu einer Menge von Pfaden, die nicht konfliktfrei ist, nicht nur eine, sondern gleich mehrere Interpretationen liefert, schlugen Thomason, Horty und Touretzky selbst Alternativen vor.

In [Horty *et al.*, 1987a] und der vollständigeren Version [Horty *et al.*, 1987b] wird ein etwas komplizierterer *skeptischer* Ansatz beschrieben, der dieses Problem vermeidet und zu einem gegebenen Netzwerk genau eine einzige Menge von Aussagen liefert, die von diesem Netzwerk unterstützt werden. Touretzkys ursprünglicher Ansatz wird jetzt im Gegensatz zu *skeptisch* als *leichtgläubig (credulous Reasoner)* bezeichnet. Im folgenden soll im allgemeinen die englische Originalbezeichnung verwendet werden, um mögliche Mißverständnisse auszuschließen.

Der Beweis von Aussagen in einem pfadbasierten Formalismus geschieht durch stückweise Verlängerung von möglichen Pfaden durch das Netzwerk. In dem Artikel *A Clash of Intuitions* [Touretzky *et al.*, 1987] werden noch einige weitere technische Details diskutiert, in denen sich Vererbungssysteme unterscheiden können. Im einzelnen wird die Richtung, in der Pfade verlängert werden, die Behandlung negativer Information, und die Klassifikation der Kantentypen in *strikt* und *aufhebbar (defeasible)* betrachtet.

Stellt man ein Netzwerk graphisch dar, so geht man so vor, daß allgemeine Konzepte möglichst weit oben, d.h. hoch in der Hierarchie abgebildet werden. Konkrete Individuen bilden den Gegenpol und sind so möglichst tief angeordnet. Vererbung in einem Netzwerk kann jetzt auf zwei verschiedene Weisen betrachtet werden:

- In der ersten Sicht bewegen sich die Eigenschaften von den allgemeinsten Konzepten durch das Netzwerk nach unten, bis sie schließlich bei den Individuen landen. Man spricht auch von *Downward Reasoning* oder *Coupled Inheritance*, da Teilklassen in Abwesenheit expliziter Ausnahmen bei diesem Vorgehen immer in Übereinstimmung mit ihren Oberklassen stehen müssen und so eine enge Verbindung *(coupling)* zwischen ihnen besteht. Die Darstellung von FRL und NETL suggeriert ebenfalls ein solches *Top Down* Vorgehen. [Touretzky, 1986] ist das einzige System, in dem auch wirklich so vorgegangen wird.

- In der zweiten Sicht erwerben Individuen auf ihrem Weg nach oben zu den allgemeinsten Konzepten oder Klassen Eigenschaften. Dementsprechend spricht man von *Upward Reasoning* oder *Decoupled Inheritance*. Bei genauer Betrachtung von FRL und NETL stellt man auch fest, daß die implementierten Suchalgorithmen immer von einem Individuum ausgehen und dann die Hierachie weiter nach oben durchsuchen *(Bottom Up)*.

Interessanterweise stellt sich bei der Betrachtung der Zeitkomplexität der Vererbung heraus, daß gerade dieser Unterschied zwischen *Upward*- und *Downward*-Schließen darüber entscheidet, ob das Vererbungsproblem NP-schwer oder aber polynomial ist.

Das Mischen von strikten (definitorischen) und aufhebbaren Kanten wird in [Thomason and Horty, 1988] eingehend behandelt und in [Touretzky and Thomason, 1990] noch durch einen effizienten Inferenzalgorithmus erweitert. In [Touretzky and Thomason, 1988] werden auch generische reflexive Aussagen *ohne* die Verwendung von Variablen in Vererbungssysteme integriert. [Horty and Thomason, 1990] gehen das Fehlen von negativen, konjunktiven und disjunktiven Aussagen als einen weiteren Schwachpunkt der obigen theoretischen Arbeiten an.

Ein wichtiger Aspekt von Fahlmans NETL war seine Idee des *Virtual Copying* der zu einem Knoten gehörenden *Rollen*.[15] Touretzky betrachtet in [Touretzky, 1986] zwar die Darstellung von Relationen in einem Netzwerk, läßt aber Rollen und Virtual Copying vollkommen außer Betracht, um seine theoretische Untersuchung nicht weiter zu verkomplizieren. Dieses Versäumnis wird in [Thomason and Touretzky, 1991] nachgeholt. Es stellte sich allerdings heraus, daß das Konzept der Rollen so schwierig zu analysieren ist, daß man sich erst einmal auf den *monotonen* Fall beschränken mußte.

[15] Zu den hier verwendeten Begriffen vergleiche Abschnitt 3.4.2 auf Seite 95 und Abschnitt 3.4.4 auf Seite 99.

Da Semantische Netzwerke nur eingeschränkten Teilmengen logischer Aussagen entsprechen, erscheint der Versuch lohnend, Systeme zu entwickeln, die die Konzepte effizienter pfadbasierter Methoden mit der Mächtigkeit der Formalismen klassischer Theorembeweiser integrieren. Thomason bietet mit [Thomason and Aronis, 1990] einen hybriden Ansatz an. Er geht von der Beobachtung aus, daß Vererbungssysteme zwar intelligent Informationsbeschaffung *(Information Retrieval)* implementieren können, aber im allgemeinen nicht flexibel und mächtig genug sind, um in einem realistischen Wissensbasierten System einzige Inferenzkomponente zu sein. Zu diesem Zweck kombiniert er ein Vererbungssystem mit einem modularen Beweisverfahren für eine vierwertige nichtklassische Logik.

Genau diese Feststellung führte auch zu den in **DONALD-III** getroffenen Designentscheidungen. **DONALD** kombiniert ein effizientes Frame System mit polynomialer Komplexität mit Techniken zur nichtmonotonen Verwaltung von Hypothesen und Lösungsvorschlägen.

Krishnaprasad erweitert in seiner Dissertation [Krishnaprasad, 1989] die bekannten Vererbungsstrategien um den Begriff der *Preferential Inheritance*. Diese erfordert zusätzlich zu der Darstellung des Netzwerks als Graph noch die Spezifikation einer Ordnungsrelation auf den von einem Knoten ausgehenden Kanten. Auch **DONALD** benutzt eine solche Präferenzrelation unter den Oberklassen einer Klasse.

Alle hier angesprochenen Aspekte von Vererbungssystemen werden in Kapitel 6 noch eingehend besprochen.

1.6.2.4 Komplexität der Vererbung

Das Problem der Wissensrepräsentation ist für den allgemeinen Fall sehr schwer maschinell zu behandlen. Ziel der Vererbungssysteme war, *den* Teil der Verarbeitung zu isolieren, der mit effizienten Algorithmen bearbeitet werden kann. Kapitel 7 beschäftigt sich noch ausführlich mit den in Vererbungssystemen auftretenden Komplexitätsproblemen.

Netzwerkbasierte Algorithmen sollten also effizient implementierbar sein. Selman und Levesque zeigen in ihrem Aufsatz [Selman and Levesque, 1991], daß der ursprüngliche, von Touretzky vorgeschlagene, *Top Down* Vererbungsmechanismus NP-schwer und damit in erster Näherung unbrauchbar für die praktische Anwendung ist. Dies gilt sogar für den eingeschränkten Fall total azyklischer und eindeutiger Netzwerke. In [Horty *et al.*, 1987b] ist dagegen ein polynomialer Algorithmus für den Fall des *Upward Reasoning* angegeben.

Der oben erwähnte *skeptische* Ansatz liefert dahingegen nach [Witteveen, 1991] nicht nur für Vererbungssysteme, sondern auch für sogenannte *Reason Maintenance Systeme* (RMS)[16], die zur nichtmonotonen Verwaltung logischer Schlüsse und Annahmen dienen, effiziente Algorithmen. Konkret kann das eindeutige skeptische Modell der dreiwertigen Semantik der dort beschriebenen Reason Maintenance in $O(n^2)$ Zeiteinheiten berechnet werden.

Falls die Menge der Annahmen *(Beliefs)* eines solchen Systems nicht widerspruchsfrei ist, setzt ein Algorithmus zur sogenannten *Belief Revision* ein. Ein RMS verwaltet zusätzlich zu den Annahmen ein Netz von Rechtfertigungen *(Justifications)*, die die Abhängigkeiten zwischen den Annahmen beschreiben. Bisher wurde im Fall von Konflikten ein sogenantes *Dependency-directed Backtracking* mit noch unbekannter Komplexität eingesetzt. [Witteveen, 1991] bietet als Alternative einen $O(n^3)$ Algorithmus zur skeptischen Belief Revision, der auch das Problem mehrerer stabiler Modelle vermeidet. Solche Belief Revision Systeme liegen aber nicht im Hauptinteresse dieser Arbeit und werden deshalb in Kapitel 7 nicht mehr weiter verfolgt. Ihre Darstellung ist aus diesem Grund hier etwas ausführlicher.

1.6.3 Ausblick

Die Schlußbetrachtung enthält neben einer Zusammenfassung der Arbeit noch einen Ausblick auf interessante noch ausstehende Probleme und mögliche Lösungsansätze. Einige neuere Arbeiten auf theoretischem Gebiet, insbesondere die oben erwähnten [Krishnaprasad, 1989] und [Thomason and Aronis, 1990], enthalten noch Ideen, die auch direkt Eingang in die Implementation der Expertensystem-Shell **DONALD-III** finden könnten. Ebenso wäre eine Integration der verwendeten Frames mit dem *Common Lisp Object System*

[16] Jon Doyle diskutiert in seinem Artikel [Doyle, 1983] Weiterentwicklungen seines *Truth Maintenance Systems* (TMS) [Doyle, 1979b].

(CLOS) [Steele Jr., 1990, S. 770–864] sinnvoll und auch möglich. Eine frei verfügbare portable Implementation für CLOS unter dem Namen PCL steht dafür schon zur Verfügung. Die momentan noch bestehenden Probleme im Bereich des Benutzerinterface könnten durch Integration eines auf CLOS basierenden, bereits existierenden Interfaces zu dem Toolkit Motif (CLIM) des Window Systems X11 des MIT gelöst werden.

Das nächste Kapitel bietet einen Einstieg in die verschiedenen Formalismen der Wissensrepräsentation und ihre theoretischen Grundlagen.

Kapitel 2

Formalismen zur Wissensrepräsentation

Dieses Kapitel stellt die wichtigsten Formalismen zur Wissensrepräsentation und deren mathematischen Grundlagen vor und schließt mit einer kurzen Diskussion. Im nächsten Kapitel werden die Implementationen Semantischer Netzwerke ausführlich betrachtet, deshalb ist ihre Darstellung hier recht knapp gehalten.

2.1 Einführung

Es gibt im wesentlichen drei wichtige Formalismen, die bei der Konstruktion von Expertensystemen zur Anwendung kommen: Strukturierte Objekte, regelbasierte Systeme und an klassischer oder auch nicht-klassischer Logik orientierte Ansätze.

Wo es sinnvoll und möglich ist, werden die verschiedenen Formalismen in den folgenden Abschnitten anhand eines Beispiels aus dem Bereich der Wertpapieranlageberatung erläutert.

Ein Investor besitzt u.a. die Eigenschaften Anlagedauer, Anlagebetrag, Anlagezweck, akzeptierbares Risiko, und Steuererparnis. Dargestellt werden soll ein Anleger, der einen geringen Betrag kurzfristig anlegen will und damit auch größere Risiken zur Erlangung einer hohen Rendite in Kauf nimmt. Ausgehend von diesen Informationen soll ihm eine Option empfohlen werden.

Dieses Kapitel beginnt mit einigen Definitionen aus der Theorie der Graphen und Netzwerke. Ebenfalls erforderlich sich einige Grundbegriffe der mathematischen Logik. Gegen den Einsatz des reinen Aussagen- oder Prädikatenkalküls als Mittel zur Wissensrepräsentation spricht aber die Notwendigkeit zur Darstellung *unsicheren* Wissens in Expertensystemen. Unsicher kann auch bedeuten, daß einmal erworbenes oder hergeleitetes Wissen im nachhinein revidiert werden muß. Solches Vorgehen steht im Widerspruch zum Monotonieprinzip der klassischer Prädikatenlogik (siehe Theorem 2.12 auf Seite 46).

Allen Formalismen gemeinsam ist die Verwendung von Mustervergleichen *(Pattern Matching)*, wenn man den Begriff so weit faßt, das er sowohl die syntaktische Unifikation logischer Formeln als auch die Erkennung von Objekten beim Sehen umfaßt.

Thema der folgenden Abschnitte ist nun die Frage, wie Formalismen mit solider mathematischer Grundlage in ein praktisch anwendbares System mit hoher Leistung umgesetzt werden können. Es kann eventuell auch sinnvoll sein, mehrere Formalismen unter Verlust der Uniformität des Wissens zu integrieren.

Nach der Vorstellung der unterschiedlichen Ansätze zur Wissenrepräsentation in den Abschnitten 2.4 bis 2.9 werden die Vor- und Nachteile dieser Formalismen abschließend kurz diskutiert.

2.2 Graphen und Netzwerke: Grundbegriffe

Graphen und ihre Anwendungen in der Künstlichen Intelligenz werden im sechsten Kapitel noch ausführlicher behandelt. Deshalb werden hier nur einige Definitionen gegeben, die für die anschließende Diskussion erforderlich sind.

Zur Widerspiegelung von Beziehungen zwischen Dingen, Sachverhalten u.a. werden in der Mathematik Relationen benutzt:

Definition 2.1 (Relation) *Eine binäre Relation R zwischen Elementen x_1 einer Menge M_1 und Elementen x_2 einer Menge M_2 ist die Teilmenge der Produktmenge $M_1 \times M_2$, die die geordneten Paare (x_1, x_2) enthält, für die die Relation R besteht; in Zeichen $x_1 R x_2$. Allgemeine n-stellige Relationen sind durch Verallgemeinerung von zwei auf n Mengen definiert.*

Zu den wichtigsten Eigenschaften von Relationen zählen *Reflexivität, Symmetrie, Transitivität, Linearität* und *Konnektivität.*

Oft tritt der Fall auf, daß M_1 und M_2 dieselbe Menge sind. In diesem Fall wird die Grundmenge im folgenden mit $X = M_1 = M_2$ bezeichnet. Man spricht dann statt von einer Relation *zwischen* den Elementen von M_1 und M_2 von einer Realtion *auf* X.

Eine binäre Relation R auf einer Menge X ist *reflexiv* genau dann, wenn xRx für alle $x \in X$. R ist *symmetrisch* genau dann, wenn aus xRy yRx folgt. R ist *transitiv* genau dann, wenn aus xRy und yRz xRz folgt. R ist eine *Äquivalenzrelation* genau dann, wenn R reflexiv, symmetrisch und transitiv ist.

Ist R eine Äquivalenzrelation auf X, so bezeichnet $|x|$ die Menge $\{y : y \in X \text{ und } xRy\}$. $|x|$ heißt die von x erzeugte Äquivalenzklasse von R. Folgende Eigenschaften gelten: (i) $x \in |x|$, (ii) $|x| = |y|$ genau dann, wenn xRy und (iii) Aus $|x| \neq |y|$ folgt $|x| \cap |y| = \emptyset$. Für jede Äquivalenzrelation R auf X definiert also die Menge aller Äquivalenzklassen von R eine Zerlegung von X.

Eine binäre Relation R heißt *antisymmetrisch* genau dann, wenn aus xRy und yRx $x = y$ folgt. R heißt *partielle Ordung* genau dann, wenn sie reflexiv, transitiv und antisymmetrisch ist. Beispiel für eine partielle Ordnung ist die Mengeninklusionsrelation $\subseteq$ auf der Menge aller Teilmengen einer festen Menge X.

Ist R eine partielle Ordung auf der Menge X, so heißt ein Element $a \in X$ *größtest* (bzw. *kleinstes*) in Bezug auf R genau dann, wenn xRa (bzw. aRx)

für alle $x \in X$. Ein Element $a \in X$ heißt *maximal* (bzw. *minimal*) in Bezug auf R genau dann, wenn es kein Element $x \in X$ gibt, so daß aRx (bzw. xRa) gilt und $x \neq a$. Für jede partielle Ordnung R gibt es höchstens ein größtes Element in Bezug auf R und höchstens ein kleinstes Element in Bezug auf R. Es kann allerdings viele minimale und maximale Elemente geben. Eine reflexive und transitive binäre Relation heißt *Präordnung*.

Eine binäre Relation R auf X heißt *asymmetrisch* genau dann, wenn es keine Elemente $x, y \in X$ gibt, so daß xRy und yRx. Das muß auch für den Fall $x = y$ gelten. Ist R asymmetrisch und transitiv, dann heißt sie *strikte* partielle Ordnung. Beispiel für eine strikte partielle Ordnung ist die strikte Mengeninklusion $\subset$ auf der Menge aller Teilmengen einer festen Menge X.

Definition 2.2 (Relationsgraph) *Der Relationsgraph einer Relation R zwischen endlichen Mengen M_1, M_2 ist die Darstellung von R durch einen Graphen, indem man den Elementen der Vereinigungsmenge $M_1 \cup M_2$ eindeutig Punkte in der Ebene zuordnet und einen Punkt x genau dann mit einem Punkt y durch einen von x nach y weisenden Pfeil verbindet, wenn xRy.*

Definition 2.3 (Relationsmatrix) *Ebenso kann eine Relation durch eine Matrix $R = (r_{ik})$ dargestellt werden, indem man die Elemente a_i von M_1 und b_k von M_2 entsprechend durchnummeriert und dann R so definiert:*

$$r_{ik} := \begin{cases} 1 & \textit{falls } a_i R b_k \\ 0 & \textit{sonst} \end{cases}$$

Definition 2.4 (Gerichteter Graph) *Sei N eine endliche Menge und $L \subseteq N \times N$. Dann heißt $G := (N, L)$ gerichteter Graph. Die Elemente von N heißen Knoten* (Node)*, ein Paar $(l_i, l_j) \in L$ heißt Kante* (Edge)[1] *von l_i nach l_j. Spielt die Reihenfolge in den Paaren aus L keine Rolle (handelt es sich also um zweielementige Mengen), so heißt G allgemeiner Graph.*

Definition 2.5 (Einfacher Graph) *Ist $G = (N, L)$ ein gerichteter Graph und gilt:*

$$l_i \neq l_j \quad \textit{für alle} \quad (l_i, l_j) \in L,$$

so heißt G einfach.

[1] In der Künstlichen Intelligenz spricht man oft auch von Verbindungen *(Links)*.

Definition 2.6 (Pfad) *Ist $G = (N, L)$ ein gerichteter Graph, dann heißt eine Menge $\{(l_1, l_2), (l_2, l_3), \ldots, (l_{n-2}, l_{n-1}), (l_{n-1}, l_n)\}$ von Paaren aus L Pfad von l_1 nach l_n.*

Definition 2.7 (Wald) *Ist $G = (N, L)$ ein einfacher Graph und enthält L keine Pfade $\{(l_1, l_2), (l_2, l_3), \ldots, (l_{n-2}, l_{n-1}), (l_{n-1}, l_n)\}$ mit $l_1 = l_n$ (Zyklen) so heißt G Wald.*

Definition 2.8 *Ist $G = (N, L)$ ein beliebiger Graph, so daß für alle Paare (l_i, l_j) ein Pfad aus L mit l_i als Anfangspunkt und l_j als Endpunkt existiert, dann heißt G zusammenhängend.*

Definition 2.9 (Baum) *Ein zusammenhängender Wald heißt auch Baum. Oft wird ein Knoten als Wurzelknoten* (Root) *ausgezeichnet. Die übrigen Knoten formen dann eine sich verästelnde Struktur von Nachfolgern der Wurzel. Knoten ohne Nachfolger heißen Blätter* (Leaf).

In der Graphentheorie werden die Kanten eines gerichteten Graphen oft mit Zahlen versehen, die ein Maß für die Kosten der Traversierung dieser Kante angeben sollen. Solche Graphen heißen dann *Netzwerke*. Typische Modellprobleme sind die Bestimmung von Pfaden mit minimalen Kosten zwischen zwei Knoten. Im Rahmen der Künstliche Intelligenz bedeutet das Vorliegen einer Kante, daß die Knoten in einer gewissen Relation stehen. Die Kanten sind also nicht nur mit Zahlen, sondern mit beliebigen Relationen beschriftet. Die folgende Definition von Netzwerk ist wesentlich schwächer, entspricht aber dafür mehr der Praxis.

Definition 2.10 (Netzwerk) *Für ein Paar (N, L) von Knoten und beschrifteten Kanten ist jede Teilmenge von $N \times L \times N$ ein Netzwerk.*

Durch ein solches Netzwerk können z.B. kausale Abhängigkeiten zwischen Ereignissen oder auch Klassifikationshierarchien beschrieben werden. Solche Netzwerke wurden erstmals zur Darstellung von Wortbedeutungen innerhalb eines Computersystems verwendet [Quillian, 1967]. Sie sollten also Wörtern ihre *Semantik* zuordnen, und so entstand der Name *Semantisches Netzwerk* [Quillian, 1968].

Da Netzwerke jetzt auch für ganz andersartige Zwecke eingesetzt werden, sollte man besser von *Assoziativen Netzwerken* reden. Der ursprüngliche Begriff hat sich jedoch bereits soweit etabliert, daß er auch hier weiter verwendet werden soll.

2.3 Aussagen- und Prädikatenlogik: Grundbegriffe

Außer Graphen und Netzwerken spielen auch die Aussagen- und Prädikatenlogik eine wichtige Rolle bei der Formalisierung von Wissensrepräsentationen, die zum Zweck einer mathematischen Analyse unerläßlich ist.

Die moderne klassische Logik wurde als formales System zur Behandlung von logischen Aussagen der Mathematik entwickelt. Die folgenden Abschnitte sollen eine sehr knappe Übersicht über Begriffe geben, die im Verlauf dieser Arbeit benötigt werden. Deshalb wurde zum größten Teil auf Beweise verzichtet, oder nur eine Beweisidee angedeutet. Eine ausführliche Darstellung findet sich z.B. in [Mendelson, 1979]. Im Bereich der *nichtklassischen* Logiken bietet [Lukaszewicz, 1990] eine wesentlich weiterführende Betrachtung, an der sich auch das weitere Vorgehen teilweise orientiert.

2.3.1 Syntax der Aussagenlogik

Ein *Alphabet* der (klassischen) Aussagenlogik besteht aus *primitiven Symbolen* der folgenden vier Klassen:

(i) Eine abzählbare Menge von *Aussagenkonstanten*.

(ii) Eine *Wahrheitskonstante:* **W**.

(iii) Die Satzverbinder: $\supset$ (Implikation) und $\neg$ (Negation).

(iv) Die Klammersymbole: '(' und ')'.

Die Klassen *(ii)–(iv)* sind fest, während die Klasse *(i)* von Alphabet zu Alphabet variieren kann und auch eventuell leer sein kann. Normalerweise werden die Kleinbuchstaben p, q und r mit Indizes als Aussagenkonstanten verwendet.

Definition 2.11 (Formel) *Die Menge der Formeln über einem Alphabet $\mathcal{A}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt:*

(i) *W und jede Aussagenkonstante aus $\mathcal{A}$ ist eine Formel.*

(ii) *Falls A und B Formeln sind, dann sind auch $(\neg A)$ und $(A \supset B)$ Formeln.*

Die Menge aller Formeln über einem Alphabet $\mathcal{A}$ heißt die Aussagensprache über $\mathcal{A}$ und wird mit $L_{\mathcal{A}}$ bezeichnet.

Als syntaktische Variablen zur Darstellung von Formeln dienen A, B, C und D. Als weitere Satzverbindungen kann man '$\vee$' (oder), '$\wedge$' (und) und '$\equiv$' (äquivalent) einführen:

(i) $(A \vee B) := ((\neg A) \supset B)$

(ii) $(A \wedge B) := (\neg((\neg A) \vee (\neg B)))$

(iii) $(A \equiv B) := ((A \supset B) \wedge (B \supset A))$

(iv) $\mathbf{F} := \neg W$

Zur Vermeidung unnötiger Klammergebirge führt man eine Vorrangfolge unter den Satzverbindern ein: $\neg, \wedge, \vee, \supset, \equiv$ (von links nach rechts fallende Priorität). Des weiteren sind die binären Operatoren rechtsassoziativ.

2.3.2 Semantik der Aussagenlogik

Definition 2.12 *Eine* Interpretation *für eine Aussagensprache $L_{\mathcal{A}}$ ist eine Funktion, die jeder Aussagenkonstante aus $\mathcal{A}$ ein Element der Menge $\{0, 1\}$ zuordnet.*

Definition 2.13 *Sei m eine Interpretation für $L_{\mathcal{A}}$ und sei $A \in L_{\mathcal{A}}$. Der* Wahrheitswert *von A in m, $V^m(A)$ ist ein Element von $\{0,1\}$ und durch folgende Rekursion definiert:*

(i) $V^m(\mathbf{W}) = 1$

(ii) $V^m(p) = m(p)$ *für alle* $p \in \mathcal{A}$

(iii) $V^m(\neg B) = 1 - V^m(B)$

(iv) $V^m(B \supset C) = 1$ *genau dann, wenn* $V^m(B) = 0$ *oder* $V^m(C) = 1$

Man spricht von „A ist *wahr* (bzw. *falsch*)“ genau dann, wenn $V^m(A) = 1$ (bzw. $V^m(A) = 0$). A ist erfüllbar *(satisfiable)*, genau dann, wenn $V^m(A) = 1$ für eine geeignete Interpretation m. A ist eine *Tautologie* genau dann, wenn $V^m(A) = 1$ für alle Interpretationen m. Falls $V^m(A) = 1$, dann heißt m ein *Modell* von A.

Das Gültigkeitsproblem der Aussagenlogik ist die Aufgabe, für eine gegebene Formel zu entscheiden, ob sie eine Tautologie ist oder nicht. Mit Hilfe von Wahrheitstabellen der Größe 2^n bei n verschiedenen Aussagenkonstanten läßt sich dieses Problem lösen, und so ergibt sich:

Theorem 2.1 *Das Gültigkeitsproblem für die Aussagenlogik ist entscheidbar.*

Theorem 2.1 wird auch oft so formuliert, daß die Aussagenlogik *entscheidbar* ist.

Zwei Formeln A und B heißen *logisch äquivalent*, in Zeichen $A \Leftrightarrow B$, genau dann, wenn beide denselben Wahrheitswert in jeder Interpretation annehmen. Sie haben dann exakt dieselben Modelle. Während das Symbol $\equiv$ ein Satzverbindungszeichen in Formeln ist, handelt es sich bei $\Leftrightarrow$ um ein metasprachliches Symbol für Aussagen *über* Formeln. Das folgende Theorem gibt einige wichtige Eigenschaften äquivalenter Formeln an:

Theorem 2.2

(i) $A \Leftrightarrow B$, *genau dann, wenn* $A \equiv B$ *eine Tautologie ist.*

(ii) *$A \Leftrightarrow B$ und $B \Leftrightarrow C$ implizieren $A \Leftrightarrow C$ (Transitivität der Äquivalenzrelation).*

(iii) *Ist B eine Teilformel von A und A' durch Ersetzen eines oder mehrerer Vorkommen von B für C in A, und ist $B \Leftrightarrow C$, dann gilt: $A \Leftrightarrow A'$ (Einsetzen).*

Das obige Theorem 2.2 wird oft benutzt, um zu zeigen, daß man Formeln logisch äquivalent so umformen kann, daß sie besondere syntaktische Form haben.

Definition 2.14 (Konjunktive Normalform) *Bei einer* atomaren Formel *handelt es sich entweder um eine Aussagenkonstante oder um eine Wahrheitskonstante. Eine Formel A ist genau dann in* konjunktiver Normalform, *wenn*

$$A = \bigwedge_{i=1}^{n} \left[\bigvee_{j=1}^{m(i)} A_{ij} \right],$$

wobei die A_{ij} entweder atomare Formeln oder Verneinungen atomarer Formeln sind. Eine Formel B wird als konjunktive Normalform einer Formel A bezeichnet genau dann, wenn B in konjunktiver Normalform ist und $A \Leftrightarrow B$.

Theorem 2.3 *Jede Formel A kann* effektiv *in ihre konjunktive Normalform überführt werden.*

Beweis: Der Beweis verläuft in drei Schritten.

Schritt 1. Die Symbole $\supset$ und $\equiv$ werden unter Ausnutzung der folgenden Tautologien eliminiert:

$$\begin{array}{lcl} (B \supset C) & \rightsquigarrow & (\neg B \vee C) \\ (B \equiv C) & \rightsquigarrow & ((\neg B \vee C) \wedge (B \vee (B \vee \neg C)) \end{array}$$

Schritt 2. Das Symbol $\neg$ wird unter Ausnutzung der folgenden Tautologien nach innen bewegt:

$$\begin{array}{rcl} \neg(B \vee C) & \rightsquigarrow & \neg B \wedge \neg C \\ \neg(B \wedge C) & \rightsquigarrow & \neg B \vee \neg C \\ \neg\neg B & \rightsquigarrow & B \end{array}$$

Schritt 3. Verteile $\wedge$ über $\vee$:

$$\begin{array}{lcl} (B \wedge C) \vee D & \rightsquigarrow & (B \vee D) \wedge (B \vee D) \\ B \vee (C \wedge D) & \rightsquigarrow & (B \vee C) \wedge (B \vee D) \end{array}$$

Mit dem obigen Algorithmus erhält man eine Folge $A_0, \ldots, A_n$ von Formeln, so daß $A = A_0$ und $A_n = A'$. A' ist in konjunktiver Normalform und A_{i+1} entsteht aus A_i für alle $0 \leq i < n$ durch Ersetzen einer äquivalenten Teilformel. Also ist wegen Theorem 2.2 $A_i \Leftrightarrow A_{i+1}$ und wegen der Transitivität der Äquivalenzrelation $A_0 \Leftrightarrow A_n$. Damit ist A' eine konjunktive Normalform von A. ∎

2.3.3 Aussagenlogische Theorien

Eins der zentralen Konzepte der Aussagenlogik ist die *Theorie*. Eine aussagenlogische Theorie ist eine Menge von Formeln über einer Aussagensprache, die zur Beschreibung einer Situation oder Welt dienen soll.

Definition 2.15 (Theorie) *Eine* aussagenlogische Theorie *besteht aus einem Paar $T = \langle L, S \rangle$, wobei L eine Aussagensprache und S eine berechenbare Menge von Formeln über L ist. Die Elemente von S heißen* Axiome. *Ist S leer, dann heißt T* Aussagenkalkül.

Zur Abkürzung identifiziert man Theorien mit der Menge ihrer Axiome. In solchen Fällen besteht L aus allen mit den Aussagenkonstanten der Axiome von T konstruierbaren Formeln.

Eine Theorie ist genau dann *endlich*, wenn die Menge ihrer Axiome endlich ist. Eine Theorie T' ist genau dann eine Untertheorie von T, in Zeichen $T' \subseteq T$, wenn die Axiome von T' eine Teilmenge der Axiome von T bilden. Jeder Formel A kann man eindeutig die Theorie $T = \langle L(A), \{A\} \rangle$ zuordnen, wobei $L(A)$ wie im letzten Abschnitt definiert wird. Formeln können also auch als Theorien betrachtet werden.

Definition 2.16 (Modell) *Eine Interpretation m für T heißt genau dann* Modell *einer Theorie T, wenn alle Axiome von T in m wahr sind. Eine Theorie, die ein Modell besitzt, heißt* erfüllbar*. Ansonsten ist sie* unerfüllbar.

Definition 2.17 (Semantische Konsequenz) *Eine Formel A heißt genau dann* semantische Konsequenz *einer Theorie T, in Zeichen $T \models A$, wenn A in jedem Modell von T wahr ist. A ist genau dann semantische Konsequenz eines Aussagenkalküls über L, in Zeichen $\models A$, wenn A in jeder Interpration m für L wahr ist.*

$\models$ ist eine binäre Relation zwischen Theorien und Formeln. Man spricht statt von semantischer auch von *logischer* Konsequenz.

Theorem 2.4 *Die $\models$ Relation hat die folgenden Eigenschaften:*

(i) *$T \models A$ genau dann, wenn $T \cup \{\neg A\}$ unerfüllbar ist.*

(ii) *$T, B \models A$ genau dann, wenn $T \models B \supset A$.*

(iii) *$\{A_1, \ldots, A_n\} \models B$ genau dann, wenn $\models A_1 \wedge \cdots \wedge A_n \supset B$. Um zu zeigen, daß eine Formel B logische Konsequenz ist, genügt es also, zu zeigen, daß die Formel $\models A_1 \wedge \cdots \wedge A_n \supset B$ eine Tautologie ist.*

(iv) *Falls $T' \subseteq T$, dann $\{A : T' \models A\} \subseteq \{A : T \models A\}$ (Monotonie). Wenn zu einer Theorie T', in der A aus T' folgt, also neue Axiome hinzugefügt werden, so folgt aus dieser erweiterten Theorie immer noch die Formel A.*

Aussage *(iv)* ist das *Monotonieprinzip*. In den später betrachteten nichtklassischen Logiken wird dieses Prinzip zum Teil verletzt und man spricht dann von nichtmonotoner Logik. Zwei Theorien T und T' heißen logisch äquivalent, in Zeichen $T \Leftrightarrow T'$, falls sie dieselben Modelle haben. Für eine Theorie $T = \{A_1, \ldots, A_n\}$ gilt dann $T \Leftrightarrow \{A_1 \wedge \cdots \wedge A_n\}$.

2.3.4 Deduktionssysteme

Die Relation zwischen Formeln und ihren Wahrheitswerten ist der semantische Aspekt der Aussagenlogik. Deduktionsysteme sind im Gegensatz dazu rein syntaktischer Natur.

Definition 2.18 (Deduktionssystem) *Ein* Deduktionssystem *für die Aussagenlogik ist ein Tripel* $\langle L, S, R\rangle$*, wobei:*

(i) *L ist eine Aussagensprache.*

(ii) *S ist eine berechenbare Teilmenge von L. Die Elemente von S heißen logische Axiome.*

(iii) $R = \{R_1, \ldots, R_n\}$ *ist eine endliche Menge von* Inferenzregeln. *Jedes R_i ist eine partielle Funktion von L^k $(k \geq 1)$ nach L. Falls die $A_1, \ldots, A_k$ im Definitionsbereich von R_i liegen, dann heißt R_i* anwendbar *auf $A_1, \ldots, A_k$, und $R_i(A_1, \ldots, A_k)$ heißt dann die durch R_i vermittelte* direkte *Konsequenz von $A_1, \ldots, A_k$.*

Deduktionssysteme werden auch als Axiomensysteme bezeichnet. Die bekannteste Inferenzregel ist der *Modus Ponens*, der besagt, daß B eine direkte Konsequenz von A und $A \supset B$ ist. Solche Inferenzregeln werden im allgemeinen so geschrieben:

$$\frac{A, A \supset B}{B}$$

Eine Inferenzregel R heißt genau dann *korrekt*, wenn für alle Interpretationen m und für alle wahren $A_1, \ldots, A_k$ aus dem Definitionsbereich von R auch $R(A_1, \ldots, A_k)$ wahr ist. Eine Formel A ist genau dann *beweisbar* oder *ableitbar* in $D = \langle L, S, R\rangle$, wenn es eine Folge von $B_1, \ldots, B_m$ gibt, so daß $A = B_m$ und für alle $1 \leq i \leq m$ entweder $B_i \in S$ oder B_i eine durch eine Regel aus R vermittelte direkte Konsequenz einer vorhergehenden Formel ist. Eine solche Folge heißt *Beweis* von A. Für ein Deduktionssystem $D = \langle L, S, R\rangle$ und eine Theorie T über L heißt eine Formel A genau dann aus T *beweisbar* in D, in Zeichen $T \vdash_D A$, wenn es eine Folge von $B_1, \ldots, B_m$ gibt, so daß $A = B_m$ und für alle $1 \leq i \leq m$ entweder $B_i \in S \cup T$ oder B_i eine durch eine Regel aus R

vermittelte direkte Konsequenz einer vorhergehenden Formel ist. Eine solche Folge heißt *Beweis* von A aus T in D. A ist dann *syntaktische Konsequenz* von T oder A ist ein Theorem von T. Ist T leer, kann es weggelasssen werden: $\vdash_D A$.

D heißt genau dann *korrekt*, wenn aus $\vdash_D A$ folgt, daß A eine Tautologie ist. D heißt genau dann *vollständig*, wenn alle Tautologien beweisbar sind. Das bekannteste solche System ist die Aussagenlogik PL:

Definition 2.19 *Für eine Sprache L bezeichnet $AL(L) = \langle L, S, R\rangle$ das folgende Deduktionssystem:*

(i) *S enthält alle nach den folgenden Schemata konstruierbaren Axiome (A, B und C sind beliebige Formeln aus L):*

- **W**
- $A \supset (B \supset A)$
- $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
- $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$.

(ii) *R enthält nur den Modus Ponens.*

Für eine feste Sprache L wird $\vdash_{AL(L)}$ mit $\vdash$ abgekürzt. Für $\vdash A$ spricht man von „A ist beweisbar in der Aussagenlogik". A ist dann ein Theorem oder auch Satz der Aussagenlogik. Man definiert die Menge Th aller beweisbaren Formeln einer Theorie T durch $Th(T) = \{A : T \vdash A\}$.

Theorem 2.5 *Für Th gelten die folgenden Eigenschaften:*

(i) $T \subseteq Th(T)$.

(ii) $Th(T) = Th(Th(T))$ *(Idempotenz).*

(iii) *Falls $T \subseteq T'$, dann gilt: $Th(T) \subseteq TH(T')$ (Monotonie).*

Über den Zusammenhang zwischen Syntax und Semantik der Aussagenlogik liegt folgendes fundamentale Resultat vor:

Theorem 2.6 *mbox*

(i) *Aus $T \vdash A$ folgt $T \models A$ (Korrektheit).*

(ii) *Aus $T \models A$ folgt $T \vdash A$ (Vollständigkeit).*

2.3.5 Syntax der Prädikatenlogik

Ein *Alphabet* der (klassischen) Prädikatenlogik erster Stufe besteht aus *primitiven Symbolen* der folgenden paarweise disjunkten Klassen:

(i) Eine abzählbare Menge V von *Variablen:* $\{x_1, x_2, \ldots\}$.
(ii) Eine *Wahrheitskonstante:* $\mathbf{W}$.
(iii) Die Satzverbinder: $\supset$ (Implikation) und $\neg$ (Negation).
(iv) Ein *Quantor*: $\forall$ (universeller Quantor).
(v) Die Klammersymbole '(', ')' und das Komma ','.
(vi) Eine abzählbare Menge $\mathcal{P}$ von *Prädikatenkonstanten*. Jedem $P \in \mathcal{P}$ ist eine eindeutige nichtnegative Zahl zugeordnet, die als die *Stelligkeit* von P bezeichnet wird.
(vii) Eine besondere zweitstellige Prädikatenkonstante: '=' (Gleichheit).
(viii) Eine abzählbare Menge $\mathcal{F}$ von *Funktionskonstanten*. Jedem $f\imath\mathcal{F}$ ist wieder eine eindeutige nichtnegative Zahl zugeordnet, die als die *Stelligkeit* von f bezeichnet wird.

Die Klassen *(i)–(v)* sind fest, während die anderen Klassen von Alphabet zu Alphabet variieren können und auch eventuell leer sein können. Nullstellige Prädikatenkonstanten werden auch als *Aussagenkonstanten* bezeichnet und nullstellige Funktionskonstanten als *Individuenkonstanten*.

Definition 2.20 (Term, Formel) *Die Menge $\mathcal{T}$ der* Terme *über einem Alphabet $\mathcal{A}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt:*

(i) Alle Variablen und alle Individuenkonstanten aus $\mathcal{A}$ sind Terme.
(ii) Falls $\alpha_1, \ldots, \alpha_n \in \mathcal{T}$ und f eine n-stellige Funktionskonstante ist, dann ist auch $f(\alpha_1, \ldots, \alpha_n) \in \mathcal{T}$.

Die Menge der atomaren Formeln *über einem Alphabet $\mathcal{A}$ ist die kleinste Menge, die die folgenden Bedingungen erfüllt:*

(i) **W** *und alle Aussagenkonstanten aus* $\mathcal{A}$ *sind atomare Formeln.*

(ii) *Falls* $\alpha_1, \ldots, \alpha_n \in \mathcal{T}$ *und* P *eine* n*-stellige Prädikatenkonstante aus* $\mathcal{A}$ *ist, dann ist auch* $P(\alpha_1, \ldots, \alpha_n)$ *eine atomare Formel.*

(iii) *Für* $\alpha_1, \alpha_2 \in \mathcal{T}$ *ist auch* $(\alpha_1 = \alpha_2)$ *eine atomare Formel, vorausgesetzt* '=' $\in \mathcal{A}$.

Die Menge der Formeln *über einem Alphabet* $\mathcal{A}$ *ist die kleinste Menge, die die folgenden Bedingungen erfüllt:*

(i) *Jede atomare Formel ist auch eine Formel.*

(ii) *Sind* A, B *Formeln und* X *eine Variable, dann sind auch* $(\neg A)$, $(A \supset B)$ *und* $(\forall x(A))$ *Formeln.*

Analog zu dem der Definition 2.11 auf Seite 34 folgenden Abschnitt werden weitere Satzverbindungszeichen und Operatorprioritäten eingeführt und zusätzlich definiert:

$$(\exists x(A)) := (\neg(\forall x(A))).$$

Eine Variable x heißt genau dann *frei* in einer Formel A, wenn sie weder quantifiziert noch gebunden ist. Eine Formel, die keine freien Variablen enthält, heißt *geschlossen*, ansonsten ist sie *offen*. Geschlossene Formeln heißen auch *Sätze*.

2.3.6 Semantik der Prädikatenlogik

Die Semantik der Prädikatenlogik erster Stufe basiert auf dem Konzept des Rahmens *(Frame)*.

Definition 2.21 (Rahmen) *Ein* Rahmen *für eine Sprache* $L_{\mathcal{A}}$ *ist ein Paar* $M = \langle D, m \rangle$, *wobei:*

(i) *D ist eine nichtleere Menge, die als* Universum *bezeichnet wird.*

(ii) *m ist eine Funktion, die jeder von '=' verschiedenen n-stelligen Prädikatenkonstanten aus $\mathcal{A}$ eine n-stellige Relation über D zuordnet. Der Konstanten '=' wird gegebenfalls die Identitätsrelation auf D zugeordnet und jeder n-stelligen Funktionskonstante aus $\mathcal{A}$ eine n-stellige Funktion $f : D^n \to D$.*

Die Funktion m ordnet also jeder Aussagenkonstanten aus $\mathcal{A}$ ein Element aus $\{0, 1\}$ und jeder Individuenkonstanten aus $\mathcal{A}$ ein Element aus D zu. Für Prädikaten- oder Funktionskonstanten K aus $\mathcal{A}$ schreibt man statt $m(K)$ auch $M|K|$. Für einen Rahmen erster Ordnung bezeichnet $Z(M)$ die Menge aller Zuweisungen über M, definiert als die Menge aller Funktionen von der Menge der Variablen nach D. Für $z \in Z(M)$ und eine Variable x wird die Menge aller Zuweisungen über M, die sich von z höchstens auf x unterscheiden, durch $[z]_x := \{z' \in Z(M) : z'(y) = z(y) \quad \forall y \neq x\}$ definiert.

Definition 2.22 *Sei M ein Rahmen erster Ordung für $L_{\mathcal{A}}$.*
Der Wert *$V_z^M(\alpha)$ eines Terms α in M bezüglich der Zuweisung $z \in Z(M)$ ist ein Element von D und durch folgende Rekursion über α definiert:*

(i) $V_z^M(x) = z(x)$.

(ii) $V_z^M(i) = M|i| \quad$ *für Individuenkonstanten i.*

(iii) $V_z^M(f(\alpha_1, \dots, \alpha_n)) = M|f|(V_z^M(\alpha_1), \dots, V_z^M(\alpha_n))$.

Analog zu Definition 2.13 wird der Wahrheitswert *einer Formel A in M bezüglich der Zuweisung $z \in Z(M)$ ist ein Element von $\{0, 1\}$ und durch folgende Rekursion definiert:*

(i) $V_z^M(\mathbf{W}) = 1$.

(ii) $V_z^M(p) = M|p| \quad$ *für Aussagenkonstanten p.*

(iii) $V_z^M(P(\alpha_1, \dots, \alpha_n)) = M|P|(V_z^M(\alpha_1), \dots, V_z^M(\alpha_n)) \quad$ *für Prädikatenkonstanten P.*

(iv) $V_z^M(\alpha = \beta) = 1$ *genau dann, wenn $V_z^M(\alpha) = V_z^M(\beta) \quad$ für Terme α und β.*

(v) $V_z^M(B \supset C) = 1$ *genau dann, wenn* $V_z^M(B) = 0$ *oder* $V_z^M(C) = 1$ *für Formeln* B *und* C.

(vi) $V_z^M(\neg B) = 1 - V_z^M(B)$.

(vii) $V_z^M(\forall x B) = 1$ *genau dann, wenn* $V_{z'}^M(B) = 1$ *für alle* $z' \in [z]_x$.

Eine Formel A heißt genau dann *erfüllt* in einem Rahmen M durch eine Zuweisung $z \in Z(M)$, wenn $V_z^M(A) = 1$. A ist genau dann *erfüllbar*, wenn es einen Rahmen M und eine Zuweisung z gibt, so daß $V_z^M(A) = 1$. Ansonsten ist A *unerfüllbar*. A ist *wahr* in M, falls $V_z^M(A) = 1$ für alle $z \in Z(M)$. Ist A wahr in M, so heißt M *Modell* von A. A ist *gültig*, falls A in jedem Rahmen wahr ist. Das nächste Theorem gibt einige Resultate über Gültigkeit und Erfüllbarkeit an:

Theorem 2.7

(i) *A ist genau dann wahr in M, wenn $\forall A$ in M wahr ist.*

(ii) *A ist genau dann gültig, wenn $\forall A$ gültig ist.*

(iii) *A ist genau dann erfüllbar, wenn $\exists A$ erfüllbar ist.*

(iv) *A ist genau dann gültig, wenn $\neg A$ unerfüllbar ist.*

Unter dem Gültigkeitsproblem der Prädikatenlogik erster Stufe versteht man die Aufgabe, zu entscheiden, ob eine beliebige vorgegebene Formel gültig ist oder nicht.

Theorem 2.8 (Church 1936) *Das Gültigkeitsproblem der Prädikatenlogik erster Stufe ist unentscheidbar. Die Menge der gültigen Formeln über einer Sprache L ist* keine *berechenbare Teilmenge von L. Sie ist aber rekursiv aufzählbar und damit partiell berechenbar.*

Es gibt also Prozeduren, die für eine beliebige vorgegebene Formel $A \in L$ entscheiden, daß A gültig ist, falls dies wirklich so ist. War A aber nicht gültig, so terminiert die Prozedur mit diesem Ergebnis oder die Berechnung ist nicht endlich. Diese Tatsache wird im Gegensatz zu Theorem 2.1 auch als *Halbentscheidbarkeit* bezeichnet.

Analog zu Definition 2.14 und Theorem 2.3 definiert man eine konjunktive Normalform von Formeln der folgenden Form:

$$A = Q_1 x_1 \dots Q_n x_n \left[\bigwedge_{i=1}^{m} \bigvee_{j=1}^{k(i)} A_{ij} \right] \text{ mit } Q_i \in \{\forall, \exists\}.$$

Jede Formel kann wieder effektiv in eine zu ihr logisch äquivalente Fromel in konjunktiver Normalform transformiert werden. Der Beweis ist allerdings etwas komplexer, da eventuell neue Variablen eingeführt werden müssen, um Konflikte zu vermeiden.

2.3.7 Prädikatenlogische Theorien

Wie in Definition 2.15 werden auch in der Prädikatenlogik Theoreme als Paar $\langle L, S \rangle$ aus einer Sprache L und einer Menge S von geschlossenen Formeln (Sätze) aus S definiert. Die Elemente von S heißen auch Axiome. Ist S leer, so spricht man vom *Prädikatenkalkül*. Theorien werden wie gewohnt mit der Menge ihrer Axiome identifiziert. Jedem Satz A kann die Theorie $\langle L(A), \{A\} \rangle$ zugeordnet werden. Die Definition der *semantischen Konsequenz* $\models$ verläuft analog zu Definition 2.17 der Aussagenlogik und es gelten die folgenden Theoreme:

Theorem 2.9 *Die $\models$ Relation hat die folgenden Eigenschaften:*

(i) $T \models A$ *genau dann, wenn* $T \cup \{\neg\forall A\}$ *unerfüllbar ist.*

(ii) $T, B \models A$ *genau dann, wenn* $T \models B \supset A$.

(iii) $\{A_1, \dots, A_n\} \models B$ *genau dann, wenn* $\models A_1 \wedge \dots \wedge A_n \supset B$.

(iv) Falls $T' \subseteq T$, *dann* $\{A : T' \models A\} \subseteq \{A : T \models A\}$ *(Monotonie).*

Theorem 2.10 $T \Leftrightarrow T'$ *genau dann, wenn* $\{A : T \models A\} = \{A : T' \models A\}$.

Theorem 2.11 *Für* $T = \{A_1, \dots, A_n\}$, *gilt* $T \Leftrightarrow \{A_1 \wedge \dots \wedge A_n\}$.

Definition 2.23 *Ein Satz (oder geschlossene Formel) heißt* universell, *wenn ihre Normalform* keine *Existenzquantoren mehr enthält. Eine Theorie heißt* universell, *wenn alle ihre Axiome universell sind.*

2.3.8 Deduktionssysteme

Auch die Deduktionsysteme für die Prädikatenlogik werden ganz analog zu Definition 2.18 definiert. Ebenso werden die Begriffe der Korrektheit und Vollständigkeit übernommen.

Definition 2.24 *Für eine Sprache L bezeichnet $PL(L) = \langle L, S, R\rangle$ das folgende Deduktionssystem:*

(i) S enthält alle nach den folgenden Schemata konstruierbaren Axiome (A, B und C sind beliebige Formeln aus L):

- **W**
- $A \supset (B \supset A)$
- $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
- $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$
- $\forall x A(x) \supset A(\alpha)$*, wobei* α *ein beliebiger für* x *freier Term in* $A(x)$ *ist.*
- $\forall x(A \supset B) \supset (A \supset \forall x B)$*, wobei* x *in* A *nicht frei vorkommt.*

Enthält die Sprache L das '=', so kommen noch einige Axiome für die Gleichheit hinzu.

(ii) R enthält die folgenden beiden Inferenzregeln:

- B *ist direkte Konsequenz von* A *und* $A \supset B$.
- $\forall x A$ *ist direkte Konsequenz von* A.

Für eine feste Sprache L wird $\vdash_{PL(L)}$ mit $\vdash$ abgekürzt. Für $\vdash A$ spricht man von „A ist beweisbar in der Prädikatenlogik“. A ist dann ein Theorem oder auch Satz der Prädikatenlogik. Man definiert die Menge Th aller beweisbaren Formeln einer Theorie T durch $Th(T) = \{A : T \vdash A\}$.

Theorem 2.12 *Für Th gelten die folgenden Eigenschaften:*

(i) $T \subseteq Th(T)$.

(ii) $Th(T) = Th(Th(T))$ *(Idempotenz).*

(iii) *Falls* $T \subseteq T'$, *dann gilt:* $Th(T) \subseteq TH(T')$ *(Monotonie).*

Über den Zusammenhang zwischen Syntax und Semantik der Prädikatenlogik liegt folgendes fundamentales Resultat vor:

Theorem 2.13

(i) *Aus* $T \vdash A$ *folgt* $T \models A$ *(Korrektheit).*

(ii) *Aus* $T \models A$ *folgt* $T \vdash A$ *(Vollständigkeit).*

Eine Theorie T heißt genau dann *konsistent*, wenn es eine Formel $A \in L$ gibt, so daß $T \not\vdash A$. Konsistenz ist das syntaktische Gegenstück zur Erfüllbarkeit:

Theorem 2.14 *T ist genau dann konsistent, wenn T erfüllbar ist. Ebenso gilt das Deduktionstheorem:*

(i) $T, A \vdash B$ *genau dann, wenn* $T \vdash A \supset B$.

(ii) $\{A_1, \ldots, A_n\} \vdash B$ *genau dann, wenn* $\vdash A_1 \wedge \cdots \wedge A_n \supset B$.

2.3.9 Klauseln

Die Behandlung von Formeln in einem Computerprogramm wird durch die Beschränkung dieser auf die spezielle syntaktische Form der *Klauseln* vereinfacht.

Definition 2.25 (Literal) Literale *sind entweder atomare Formeln (positive Literale) oder negierte atomare Formeln (negative Literale). Für Literale l bezeichnet $|l|$ die zugehörige atomare Formel.*

Definition 2.26 (Klausel) *Eine Klausel K ist eine Formel der folgenden Form:*

$$K = \bigvee_{i=1}^{n} l_i, \quad \textit{mit } n \geq 0, \quad l_i \textit{ Literale.}$$

Für $n = 0$ spricht man von der leeren Klausel, in Zeichen $\Box$. Eine Formel ist genau dann in Klauselform, wenn sie folgende Form hat:

$$\forall C_1 \wedge \forall C_2 \wedge \cdots \wedge \forall C_n \quad \textit{mit } n \geq 1, \quad C_i \textit{ Klauseln.}$$

Universelle Sätze können effizient in äquivalente Sätze in Klauselform transformiert werden. Zu jeder universellen Theorie T existiert eine Theorie T' in Klauselform, so daß $T \Leftrightarrow T'$. Bei der Transformation einer Formel in Klauselform wird das Alphabet $\mathcal{A}$ um eine Menge F von *Skolem* Funktionen erweitert, um die Existenzquantoren zu eliminieren. Die Theorie T' in Klauselform ist deshalb eine Theorie über der erweiterten Sprache $L_{\mathcal{A} \cup \mathcal{F}}$. T ist aber genau dann erfüllbar, wenn T' erfüllbar ist. Eine Menge Kl von Klauseln heißt erfüllbar, wenn die Theorie $\{\forall K : K \in Kl\}$ erfüllbar ist und es gilt der für Abschnitt 2.6.1 entscheidende Satz:

Theorem 2.15 *$T \models A$ genau dann, wenn $K(T) \cup K(\neg\forall A)$ unerfüllbar ist, wobei $K(T)$ die Menge der Klauseln eine Klauselform von T ist.*

Die Bedeutung des Resolutionsprinzips liegt in seiner leichten Mechanisierbarkeit in einem Computer. Diese Tatsache wird in der Programmiersprache Prolog (siehe Abschnitt 2.6 auf Seite 61) ausgenutzt. Die aussagekräftigeren und mächtigeren Formalismen der nächsten Abschnitte lassen eine solche Automatisierung nicht mehr zu.

2.3.10 Prädikatenlogik zweiter Stufe

Die klassische Prädikatenlogik zweiter Stufe baut auf der Prädikatenlogik erster Stufe auf. Zusätzlich werden Variablen für Relationen und Funktionen über Individuen zugelassen. Das weitere Vorgehen orientiert sich naheliegenderweise an Abschnitt 2.3.5 auf Seite 41.

Das Alphabet $\mathcal{A}$ wird um eine abzählbare Menge von n-stelligen Prädikatenvariablen erweitert. Nullstellige Prädikatenvariablen heißen Aussagevariablen. Man definiert wiederum die Begriffe *Term*, *Formel* und *atomare Formel*, wobei aber jetzt auch Prädikatenvariablen zugelassen werden. Für A, $B \in L_{\mathcal{A}}$ und eine Variable X, sind nicht $(\neg A)$ und $(A \supset B)$ Elemente von L, sondern auch $(\forall X(A)) \in L_{\mathcal{A}}$. Sprachen zweiter Ordnung enthalten auch immer das '='.

Die Definition eines Rahmens kann komplett übernommen werden, nur eine Zuweisung $z \in Z(M)$ erhält eine zusätzliche Komponente für die Prädikatenvariablen. Damit ergeben sich wieder Definitionen für *Wahrheitswert*, *Erfüllbarleit*, *Wahrheit*, *Modell* und *Gültigkeit*.

Da bereits die Prädikatenlogik erster Stufe unentscheidbar ist, ist natürlich auch die Prädikatenlogik zweiter Stufe als Erweiterung unentscheidbar. Aber die Verhältnisse sind noch schwieriger, wie das nächste Theorem zeigt:

Theorem 2.16 *Sei L eine Sprache der Prädikatenlogik zweiter Stufe. Die Menge der gültigen Formeln ist im allgemeinen* keine *partiell berechenbare Teilmenge von L.*

Die Begriffe der *Theorie*, der *semantischen Konsequenz* und auch des *Deduktionssystems* übertragen sich wiederum. Damit ist ebenso *beweisbar*, *korrekt* und *vollständig* definiert. Das wichtigste Resultat für die Prädikatenlogik zweiter Stufe lautet nun:

Theorem 2.17 (Gödel) *Es gibt* kein *vollständiges Deduktionssystem für die Prädikatenlogik zweiter Stufe.*

2.3.11 Modallogik

Ebenso wie bei der klassischen Logik gibt es auch für die Modallogik eine Aussagenlogik und eine Prädikatenlogik. Nur die letztere wird hier beschrieben. Das Alphabet einer Modallogik ergänzt das Alphabet der klassischen Logik um zwei weitere *modale* Präfix Operatoren $\mathcal{L}$ und $\mathcal{M}$. Die Menge der Terme und atomaren Formeln werden klassisch definiert, nur die Menge der Formeln wird erweitert: Für eine Variable x und $A, B \in L$ sind auch $(\neg A)$, $(A \supset B)$, $(\forall x(A))$, $(\mathcal{LA})$ und $(\mathcal{MA})$ Elemente von L. Formeln der Form $(\mathcal{LA})$ bzw. $(\mathcal{MA})$ sollen bedeuten „A ist nötig“ bzw. „A ist möglich“. Eigentlich ist es nicht erforderlich, zwei neue Operatoren zu definieren, da folgende Äquivalenz besteht:

$$
\begin{aligned}
(\mathcal{MA}) &:= \neg(\mathcal{L}\neg\mathcal{A}) \\
(\mathcal{LA}) &:= \neg(\mathcal{M}\neg\mathcal{A})
\end{aligned}
$$

Die Spezifikation einer Semantik für die Modallogik benutzt wieder den Begriff des Rahmens. Ein Rahmen enthält hier aber zusätzlich noch eine Menge von *möglichen Welten*, auf der eine Relation der Erreichbarkeit definiert ist. Eine Formel ist dann möglich ($\mathcal{MA}$ ist wahr) in einer Welt w, wenn es eine geeignete Welt w' gibt, so daß w von w' aus erreichbar ist.

Definition 2.27 *Ein* Rahmen *für eine modale Sprache L ist ein Quadrupel $\langle D, W, R, m\rangle$, wobei:*

(i) *D ist eine nichtleere Menge, das Universum.*

(ii) *W ist eine nichtleere Menge von* möglichen Welten*.*

(iii) *R ist eine binäre Relation von Erreichbarkeit auf W.*

(iv) *m ist eine Funktion, die jedem Paar aus einer von '=' verschiedenen n-stelligen Prädikatenkonstanten aus $\mathcal{A}$ und einer Welt $w \in W$ eine n-stellige Relation über D zuordnet. Der Konstanten '=' wird gegebenfalls die Identitätsrelation auf D zugeordnet und jeder n-stelligen Funktionskonstante aus $\mathcal{A}$ eine n-stellige Funktion $f : D^n \to D$.*

Während man den *Wert* $V_z^M(\alpha)$ eines Terms wie in der klassischen Logik definiert, muß die klassische Definition 2.22 auf Seite 43 folgendermaßen modifiziert werden:

Definition 2.28 *Der* Wahrheitswert *einer Formel A in M bezüglich der Zuweisung $z \in Z(M)$ und $w \in W$ ist ein Element von $\{0, 1\}$ und durch folgende Rekursion definiert:*

(i) $V^M_{z,w}(\mathbf{W}) = 1$.

(ii) $V^M_{z,w}(p) = M|p, w|$ *für Aussagenkonstanten* p.

(iii) $V^M_{z,w}(P(\alpha_1, \ldots, \alpha_n)) = M|P, w|(V^M_{z,w}(\alpha_1), \ldots, V^M_{z,w}(\alpha_n))$ *für Prädikatenkonstanten* P.

(iv) $V^M_{z,w}(\alpha = \beta) = 1$ *genau dann, wenn* $V^M_{z,w}(\alpha) = V^M_{z,w}(\beta)$ *für Terme* α *und* β.

(v) $V^M_{z,w}(B \supset C) = 1$ *genau dann, wenn* $V^M_{z,w}(B) = 0$ *oder* $V^M_{z,w}(C) = 1$ *für Formeln* B *und* C.

(vi) $V^M_{z,w}(\neg B) = 1 - V^M_{z,w}(B)$.

(vii) $V^M_{z,w}(\forall x B) = 1$ *genau dann, wenn* $V^M_{z',w}(B) = 1$ *für alle* $z' \in [z]_x$.

(viii) $V^M_{z,w}(\mathcal{LB}) = \infty$ *genau dann, wenn* $V^M_{z,w'}(B) = 1$ *für alle* $w' \in W$, *so daß* wRw'.

(ix) $V^M_{z,w}(\mathcal{MB}) = \infty$ *genau dann, wenn* $V^M_{z,w'}(B) = 1$ *für ein geeignetes* $w' \in W$, *so daß* wRw'.

Der Wahrheitswert einer Formel A hängt also sowohl von der Zuweisung z als auch von der Welt w ab. Eine Formel A ist *wahr*, wenn $V^M_{z,w}(A) = 1$ unter allen Zuweisungen und in allen Welten.

Ein Rahmen $M = \langle D, W, R, m\rangle$ heißt T-Rahmen, wenn R reflexiv ist. Ist R auch zusätzlich transitiv, so heißt M $S4$-Rahmen. Für Äquivalenzrelationen R spricht man von $S5$-Rahmen. Eine Formel heißt X-gültig, wenn A in jedem X-Rahmen wahr ist, wobei $X \in \{T, S4, S5\}$. Wie in der klassischen Prädikatenlogik gilt:

Theorem 2.18 *Für alle* $X \in \{T, S4, S5\}$ *ist das Gültigkeitsproblem der* X-*modalen Logik erster Stufe unentscheidbar. Wie in der klassischen Logik, kann man aber einen Algorithmus angeben, der für eine wirklich gültige Formel* A *deren Gültigkeit nachweist und terminiert. Ist* A *nicht gültig, so kann der Fall eintreten, daß der Algorithmus nicht terminiert.*

Im Unterschied zur Prädikatenlogik zweiter Stufe gibt es jedoch für die Modallogik ein vollständiges und korrektes Deduktionssystem:

Definition 2.29 *Sei L eine modale Sprache und $X \in \{T, S4, S5\}$. Das Deduktionssystem $ML(L) := \langle L, S_X, R\rangle$ ist definiert durch die Axiome S_X und die Inferenzregeln R.*

(i) S_T enthält die folgenden Axiome:

- **W**
- $A \supset (B \supset A)$
- $(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
- $(\neg B \supset \neg A) \supset ((\neg B \supset A) \supset B)$
- $\forall x A(x) \supset A(\alpha)$, *wobei α ein beliebiger für x freier Term in $A(x)$ ist.*
- $\forall x(A \supset B) \supset (A \supset \forall x B)$, *wobei x in A nicht frei vorkommt.*
- $\mathcal{LA} \supset \mathcal{A}$
- $\mathcal{L}(\mathcal{A} \supset \mathcal{B}) \supset (\mathcal{LA} \supset \mathcal{LB})$
- $\forall x(\mathcal{LA}) \supset \mathcal{L}(\forall\S\mathcal{A})$

  S_{S4} enthält zusätzlich zu den Axiomen von S_T das Axiom
- $\mathcal{LA} \supset \mathcal{LLA}$

  S_{S45} enthält zusätzlich zu den Axiomen von S_{S4} das Axiom
- $\mathcal{MA} \supset \mathcal{LMA}$

(ii) R enthält die folgenden beiden Inferenzregeln:

- *B ist direkte Konsequenz von A und $A \supset B$.*
- *$\forall x A$ ist direkte Konsequenz von A.*
- *$\mathcal{LA}$ ist direkte Konsequenz von A.*

Das obige Deduktionssystem ML ist korrekt und vollständig.

Nach diesen theoretischen Vorbereitungen werden im nächsten Abschnitt ganz konkrete Methoden der Wissensrepräsentation betrachtet.

2.4 Wissenrepräsentation in LISP

LISP unterscheidet sich in den folgenden drei Eigenschaften von fast allen anderen geläufigen Programmiersprachen:

- Hauptdatenstrukturen sind *Atome* und *Listen*.
- Programme werden ebenfalls als Listen dargestellt.
- Als primitive Operationen stehen Listenmanipulationen zur Verfügung.

Alle Datenstrukturen sind symbolische Ausdrücke *(S-expressions)* oder kurz *S-Ausdrücke*. Der einfachste S-Ausdruck ist das Atom. Es gibt die Atome `T` und `NIL`, die auch für *wahr* und *falsch* stehen, ansonsten sind Atome einfach mit einem Buchstaben beginnende Zeichenketten. Komplexere S-Ausdrücke enstehen durch Kombination nach der folgenden rekursiven Definition:

Definition 2.30 (S-Ausdruck)

(i) *Jedes Atom ist ein S-Ausdruck.*

(ii) *Wenn x, y S-Ausdrücke sind, dann ist auch $(x\ .\ y)$ ein S-Ausdruck.*

(iii) *Falls für $n > 0$ $l = (x_1\ .\ (x_2\ .\ (\dots\ .\ (x_{n-2}\ .\ x_n)\dots)))$ ein beliebiger S-Ausdruck ist, dann ist l eine Liste, genau dann, wenn $x_n =$ `NIL`.*

(iv) *Falls $n = 0$, so ist l die leere Liste* `NIL`.

Zur Vereinfachung schreibt man Listen der Form $l = (a\ .\ (b\ .\ \texttt{NIL}))$ als $l = (a\ b)$. S-Ausdrücke, die weder Listen noch Atome sind, heißen Punktpaare *(dotted pairs)*. Für $l = (a\ .\ b)$ heißt a Kopf von l und b Schwanz von l.

Auch Programme werden durch Listen dargestellt. Die Definition einer Funktion zum Quadrieren von Zahlen sieht in Common LISP z.B. so aus:

```
(defun quadrat (x) (* x x))
```

Für andere LISP-Dialekte sieht das zum Teil anders aus, aber wesentlich ist nur die Intention, daß mit dem Atom `quadrat` die Definition der Funktion verknüpft wird. Funktionsaufrufe haben die Gestalt

```
(◁Funktion▷  ◁Argument▷  ...  ◁Argument▷)
```

Auf die genauen Details der Funktionsdefinition und Funktionsapplikation und deren Zusammenhang mit Churchs λ-Abstraktion soll hier nicht weiter eingegangen werden. Es ist jedoch interessant festzuhalten, daß ein System, bestehend aus dem Atomen `T` und `NIL` sowie den folgenden fünf Basisfunktionen, bereits alle allgemein rekursiven Funktionen berechnen kann.

EQ Testet zwei Atome auf Gleichheit.

ATOM Testet, ob ein S-Ausdruck atomar ist.

CAR Gibt den Kopf eines S-Ausdruckes zurück.

CDR Gibt den Schwanz eines S-Ausdruckes zurück.

CONS Konstruiert aus gegebenem Kopf und Schwanz ein neues Punktpaar.

LISP bietet nun durch die Äquivalenz zwischen Daten und Programmen die Möglichkeit, Repräsentationsprachen in LISP einfacher implementieren und einbetten zu könnnen, als in einer konventionellen Programmiersprache. Betrachtet man Programme als Daten, so kann man Programme schreiben, die andere Programme während der Laufzeit analysieren, transformieren und modfizieren. Diese *Metaprogrammierung* bei gleichzeitiger Einfachheit und Uniformität der Strukturen machen LISP in einmaliger Weise erweiterbar und anpaßbar.

Genau diese Eigenschaften von LISP waren der Grund, LISP als Implementierungssprache für **DONALD** zu wählen.

2.5 Regelbasierte Systeme

Regeln sind ein wichtiger Formalismus in vielen Bereichen der Informatik wie Automatentheorie und Grammatiken von Programmiersprachen. In Expertensystemen dienen Regeln häufig zur Darstellung von empirischen Verknüpfungen zwischen auftretenden Datenmustern und ihren Konsequenzen.

Das klassische Beispiel für ein regelbasiertes Expertensystem ist das schon erwähnte MYCIN [Shortliffe, 1976] zur Behandlung von Blutinfektionen. Eine typische Regel zeigt Abbildung 2.1.

```
RULE009
PREMISE: ($AND (SAME CNTXT GRAM GRAMNEG)
               (SAME CNTXT MORPH COCCUS))
ACTION:  (CONCLUDE CNTXT IDENTITY NEISSERIA TALLY 80)
```

`IF`: 1) The gram stain of the organism is gramneg, and
2) The morphology of the organism is coccus
`THEN`: There is strongly suggestive evidence (0.8) that the identity of the organism is Neisseria

Abbildung 2.1: Eine MYCIN Regel mit Erklärung

2.5.1 Interpretierer für Produktionsregeln

Ein exemplarisches regelbasiertes System besteht aus drei Teilen, nämlich einer Menge von im allgemeinen statischen Regeln, einer dynamischen Datenbasis, die z.B. Zwischenergebnisse und noch durchzuführende Aktionen enthalten kann und einem Regelinterpretierer, der die Regeln auf Anwendbarkeit prüft, eine geeignete auswählt und dann schließlich aktiviert.

Die kleinste Informationseinheit in einem regelbasierten System nimmt die Form eines Objekt–Attribut–Wert Tripels an. Die Umsetzung der Tatsache, daß der Anleger nur kurzfristig anlegen will, aber eine überdurchschnittliche Rendite erwartet, wird dann in den folgenden Tripeln abgelegt:

Produktionsregeln erlauben es nun, aus vorhandener Information über die Eigenschaften von Objekten neue Eigenschaften herzuleiten.

$$\begin{array}{rl} \texttt{if} & C_1 \& \ldots \& C_n \\ \texttt{then} & A_1 \& \ldots \& A_n, \end{array}$$

wobei die C_i Voraussetzungen und die A_i auszuführende Aktionen sind. Bei den Aktionen kann es sich auch um Tripel handeln, die bei Erfüllung der Voraussetzungen als hergeleitet gelten.

Das Beispiel des Optionsanlegers nimmt so folgende Gestalt an:

$$\begin{array}{rl} \textbf{if} & (\texttt{Anleger}, \texttt{AnlageDauer}, \texttt{kurz})\ \& \\ & (\texttt{Anleger}, \texttt{AnlageZweck}, \texttt{Spekulation})\ \& \\ & (\texttt{Anleger}, \texttt{RenditeErwartung}, \texttt{überdurchschnittlich})\ \& \\ & (\texttt{Anleger}, \texttt{Risikobereitschaft}, \texttt{überdurchschnittlich})\ \& \\ & (\texttt{Anleger}, \texttt{Steuerersparnis}, \texttt{nein}) \\ \textbf{then} & (\texttt{Anleger}, \texttt{AnlageEmpfehlung}, \texttt{Optionen}) \end{array}$$

Diese recht vage Formulierung ist leider unumgänglich, da es mehrere unterschiedliche, auf solchen Regeln basierende Arten von Systemen gibt [2]. Das Spektrum reicht dabei von automatischen Theorembeweisern über rückwärtsverkettende Systeme zur Erstellung von Diagnosen aus mit Wahrscheinlichkeiten behafteten Symptomen bis zu dem anderen Ende der Skala, wo sich Produktionssysteme befinden, die durch den Aufbau aus den folgenden drei Komponenten charakterisiert werden können:

1. Der Arbeitsspeicher W *(Working Memory)*: Eine Menge von den Zustand des Systems determinierenden Datenstrukturen.

[2] Im weiteren wird deshalb die Syntax des recht verbreiteten OPS 5 Systems [Brownston *et al.*, 1985] verwendet.

2. Der Produktionsspeicher R *(Production Memory)*: Eine Menge von Regeln der Form

$$\triangleleft\texttt{condition}\triangleright \mapsto \triangleleft\texttt{action}\triangleright.$$

   Eine Regel r ist also ein Paar $r = (c, a)$ aus einer Voraussetzung c und einer Aktion a.

3. Der Regelinterpretierer: Dieser wendet die Produktionsregeln auf das Working Memory an.

Der Regelinterpretierer sucht nach Produktionsregeln, deren linke Seite mit einem Element des Working Memory zusammenpaßt. Das System arbeitet in Zyklen, wobei ein Zyklus aus drei Phasen besteht.

Wichtigste Phase ist die Berechnung des sogenannten *Conflict-Sets* C, der alle Regelinstanzen enthält, deren Voraussetzungen beim momentanen Zustand des Working Memories W erfüllt sind. Eine Regelinstanz ist dabei im Unterschied zu einer Regel des Produktionsspeichers ein Tripel $r = (c, a, e)$ aus den Voraussetzungen *(Conditions)* c, den Aktionen *(Actions)* a und einer Umgebung E von Variablenbindungen *(Environment)* und entsteht durch Erweiterung einer Regel durch einen Satz von Bindungen für die freien Variablen. Aus *einer* Regel des Produktionsspeichers können so eine Vielzahl von Regelinstanzen mit unterschiedlichem Environment hervorgehen.

Dies kann unter Verwendung der Funktionen m, rc und a wie in Abbildung 2.2 genauer dargestellt werden.

Als erstes wird das Working Memory mit einem Starttripel S initialisiert. Die Funktion m *(Match)* überprüft dann die Voraussetzungen aller Regeln aus R auf Verträglichkeit mit dem momentanen Zustand W des Working Memory und gibt als Resultat eine Umgebung E zurück, die so gestaltet ist, daß die Voraussetzungen in c erfüllt werden. Durch Kombination der Regel r mit der Umgebung E entstehen Regelinstanzen i, die in der Menge C zusammmengefaßt werden. C enthält alle *anwendbaren* Regeln. Gibt es keine anwendbaren Regeln mehr, so terminiert der Algorithmus, da es nicht mehr möglich ist, weitere Fakten abzuleiten. Ansonsten dient die Funktion rc *(Resolve Conflict)* zur Auswahl einer Regelinstanz $s \in C$. Dieser Funktion kommt große Bedeutung für das Verhalten des Systems zu. Beispiele für unterschiedliche

$W := \{S\}$
forever do
  $C := \{i = (c, a, E) : r = (c, a) \in R$ und $E = m(c, W)\}$
  if $(C = \emptyset)$ **exit**
  $s := rc(C)$
  $W := a(s, W)$
done

Abbildung 2.2: Ein typischer Interpretierer für Produktionsregeln

Strategien folgen noch. Die Funktion a *(Act)* wendet im letzten Schritt nun die ausgewählte Regelinstanz s auf W an. Dabei wird W durch die von s festgelegten Aktionen in a modifiziert und man erhält ein neues Working Memory W. Diese zyklische Vorgehensweise wird auch als *Recognize-Act*-Zyklus bezeichnet, wobei die Funktion m den Recognize Schritt implementiert.

Falls $|C| > 1$ ist, so muß die Funktion rc aus der Menge C von „feuerbereiten" Regelinstanzen *eine* auswählen. Die folgenden drei Mechanismen haben sich in der Vergangenheit als sinnvoll erwiesen und sind auch noch miteinander kombinierbar:

- Eine Regelinstanz sollte nur einmal mit ein und denselben Daten in ihrem Environment ausgeführt werden. Damit werden unendliche Schleifen *(Loops)* vermieden.

- Werden die Elemente des Working Memory zusätzlich mit einem sogenannten *Time Tag* versehen[3], so kann man Regelinstanzen bevorzugen, in deren Voraussetzungen möglichst *aktuelle* Elemente des Arbeitsspeichers auftreten.

- Die speziellesten Regelinstanzen werden bei der Auswahl bevorzugt. Speziell bedeutet in diesem Fall, daß die Regel eine große Anzahl von Voraussetzungen hat.

[3] Das könnte z.B. die Nummer des Zyklus sein, in dem sie erzeugt wurden.

2.5.2 Metaregeln

In vielen Systemen ist die Strategie zur Konfliktlösung fest eingebaut. Es kann höchstens eine von mehreren Möglichkeiten ausgewählt werden. Einige erlauben es jedoch, sogenannte *Metaregeln* einzusetzen. Metaregeln arbeiten im Unterschied zu den Objektregeln nicht nur mit Elementen des Working Memory, sondern auch auf einer Darstellung der Konfliktmenge C und eventuell noch weiterer Datenstrukturen, die den Systemzustand bestimmen.

Verglichen mit den Formeln des Prädikatenkalküls haben Produktionsregeln eine sehr eingeschränkte Form. Diese Restriktion bringt aber auch den Vorteil, daß das System mit Hilfe solcher Metaregeln seinen eigenen Zustand analysieren kann. Produktionsregeln lassen sich auch relativ direkt in natürliche Sprache übersetzen und so zur automatischen Generierung von Erklärungstexten nutzen.

Beispiele für solche Metaregeln aus dem Bereich der medizinischen Diagnose könnten sein:

- Bevorzuge Regeln, deren Voraussetzungen einfach und schnell zu überprüfen sind (z.B. durch Fragen an den Benutzer).
- Vermeide Regeln, deren Voraussetzungen weitere Untersuchungen erforderlich machen und eventuell mit Schmerzen oder sogar einem Gesundheitsrisiko für den Patienten verbunden sind.
- Benutze bevorzugt Regeln, die von einem *Experten* eingegeben wurden.

2.5.3 Effizienz

Durch die hohe Zahl von notwendigen Vergleichen zwischen den Elementen des Working Memory und den Pattern der möglichen Regeln entsteht bei der Anwendung des Paradigmas der Produktionsregeln ein unvermeidliches Effizienzproblem. Forgy beschreibt in [Forgy, 1982] eine Methode zur Lösung dieses Problems, die auf den folgenden zwei Beobachtungen beruht:

- In jedem Durchlauf des obigen Verfahrens ändern sich nur wenige Elemente des Working Memory.
- Die Voraussetzungen der Regelinstanzen enthalten oft gleiche Muster.

Diese Eigenschaften legen die Kompilation der Voraussetzungen der Regelinstanzen in ein Netzwerk von *Match* Operationen nahe. Es gibt auch Ansätze, diesen Algorithmus in modifizierter Form auf Prolog-basierte Systeme anzuwenden [Merritt, 1989, S. 137ff].

2.5.4 Regeln in Expertensystemen

Produktionsregeln werden in Wissensbasierten Systemen typischerweise in zwei verschiedenen Modi betrieben:

Forward Chaining. Bei der Vorwärtsverkettung werden durch Vergleich der Elemente des Working Memory gegen die *linken* Seiten der Produktionsregeln aus den bekannten Fakten *alle* Schlußfolgerungen gezogen, die die Regeln implizieren.

Backward Chaining. Bei der Rüchwärtsverkettung wird ausgehend von einer Konklusion, die überprüft werden soll[4], durch Vergleich der *rechten* Seiten der Regeln nach hinreichenden Bedingungen für die Konklusion gesucht. Diese Bedingungen werden dann wieder rekursiv als neue Teilziele aufgefaßt. Dieses Vorgehen liefert im allgemeinen ein zielgerichteteres Verhalten.

Man muß allerdings einen Unterschied machen zwischen der Aktivierungsmethode der Produktionsregeln (Forward oder Backward Chaining) und der globalen Problemlösungsstrategie, die in einem speziellen System eingesetzt wird (Forward oder Backward *Reasoning*).

Auch die Expertensystem-Shell **DONALD-III** unterstützt den Einsatz von Produktionsregeln, allerdings in einer durch den Frame-basierten Ansatz kontrollierten Umgebung. Details des verwendeten Interpretierers beschreibt Abschnitt 4.7.

[4] Man spricht auch von einem Ziel oder *Goal*.

2.6 Wissenrepräsentation in Prolog

Ebenso wie LISP ist Prolog eine Programmiersprache, die zur Verarbeitung symbolischer Informationen besonders geeignet ist. Entscheidendes Merkmal ist die deklarative Darstellung der Prozeduren.

Prolog *(Programming in Logic)* entstand in den frühen siebziger Jahren in Zusammenarbeit zwischen Kowalski von der Universität Edinburgh und Colmerauer und Roussel von der Universität Aix-Marseille.

Logische Programmierung kann definiert werden als die Verwendung von symbolischer Logik zur expliziten Darstellung von Problemen und Wissensbasen. Hinzu kommen kontrollierte logische Inferenzen für die effektive Lösung dieser Probleme.

Als Problemdarstellungssprache wird dabei eine Teilmenge des Prädikatenkalküls erster Ordnung in Horn-Klausel Form benutzt. Klauseln sind nach Definition 2.26 auf Seite 48 logische Formeln, die nur aus einer Disjunktion von Literalen bestehen, wobei jedes dieser Literale ein Prädikat oder ein negiertes Prädikat ist. Ein Prädikat hat die Form $p(t)$ mit einer Relation p und einer Liste t von Termen, die wiederum entweder Konstanten, Variablen oder auf neue Terme angewendete Funktoren sind.

Prolog abeitet mit einer weiteren Einschränkung der Klauseln, den sogenannten *Horn*-Klauseln. Dies sind Klauseln mit nicht mehr als *einem* positiven Literal, das auch als Kopf *(Head)* der Klausel bezeichnet wird. Man unterscheidet so drei Fälle:

(i) Fakten sind Aussagen ohne weitere Voraussetzungen:

$$p(t).$$

(ii) Regeln enthalten eine Implikation:

$$(\forall x) \quad (p(x) \vee \neg q(x)).$$

(iii) Anfragen *(Queries)* enthalten nur negative Literale:

$$(\forall x) \quad (\neg p(x) \vee \neg q(x)).$$

Diese Horn-Klauseln besitzen eine einfache prozedurale Interpretation. Klauseln vom Typ (i) und (ii) werden als Prozeduren interpretiert und Klauseln vom Typ (iii) als Zielaussagen oder Anfragen an das System. In allen Fällen werden Voraussetzungen (negierte Prädikate) als Prozeduraufrufe verstanden. Das System geht jetzt von der Anfrage aus und bezeichnet sie als *Current Goal*. Als nächstes wird ein Prozeduraufruf *(Call)* dieses Goals ausgewählt. Dann wird eine Prozedur gesucht, deren Kopf mit dem Prozeduraufruf nach einer gegebenenfalls durchzuführenden Substitution θ unifiziert. Schließlich wird der Körper *(Body)* nach Applikation der Substitution θ für den Call eingesetzt. So werden nach und nach alle Goals bearbeitet. Die Unifikation wird als Mittel zur Kommunikation der Daten zwischen Aufrufen und Prozeduren benutzt.

In Prolog wird eine vereinfachte Syntax benutzt, in der Konstanten klein und Variablen groß geschrieben werden. Konjunktionen werden durch Kommata angedeutet und Quantoren bleiben implizit. Die Horn-Klauseln der obigen drei Formen werden in Prolog so dargestellt:

```
p(t).
p(X) :- q(X).
?- p(X), q(X).
```

2.6.1 Das Resolutionsprinzip

Die oben informell eingeführte Problemlösungsstrategie ist die Resolution [Robinson, 1985]. Die Resolutionsregel ist eine Inferenzregel, die ausreicht, die Unerfüllbarkeit einer Menge von Klauseln zu zeigen. Nach Theorem 2.15 auf Seite 48 kann damit auch gezeigt werden, daß eine Theorie T eine Formel A als logische Konsequenz hat: $T \models A$. Zur Vereinfachung wird die Resolutionsregel nur für variablenfreie Formeln und Klauseln betrachtet. Für Klauseln K_1 und K_2, sowie ein Literal l ist $K_1 \vee K_2$ direkte Konsequenz von $K_1 \vee l$ und $K_2 \vee \neg l$ (siehe Definition 2.18 auf Seite 39):

$$\frac{K_1 \vee l, K_2 \vee \neg l}{K_1 \vee K_2}$$

Für $K_1 = K_2 = \Box$ reduziert sich das auf:

$$\frac{l, \neg l}{\Box}$$

Die Resolutionsregel ist eine korrekte Inferenzregel und $K_1 \vee K_2$ heißt dann Resolvente. Für eine Menge Kl von Klauseln ist eine *Widerlegung* aus Kl eine Folge $K_1, \ldots, K_n$ von Klauseln, so daß $K_n = \Box$ und entweder $K_i \in Kl$ für $1 \leq i \leq n$ oder K_i eine Resolvente von K_j und K_k mit $j, k < i$ ist. Es gilt nun:

Theorem 2.19 *Eine Menge Kl von variablenfreien Klauseln ist genau dann unerfüllbar, wenn es eine Widerlegung mit Klauseln aus Kl gibt.*

Dieses Theorem wird in der Programmiersprache Prolog ausgenutzt, indem Widerlegungen automatisch konstruiert werden. Das soll an einem Beispiel demonstriert werden.

Sei T die folgende Menge von Formeln:

(i) Vogel(Patty),

(ii) Vogel(Patty) $\supset$ Fliegt(Patty),

(iii) $\neg$Geflügelt(Patty) $\supset$ $\neg$Fliegt(Patty). zusammen mit dem Satz:

(iv) Geflügelt(Patty).

Es sei der Satz (iv) zu zeigen. $T \models (iv)$ sei nun zu zeigen, d.h. nach Theorem 2.9 auf Seite 45 die Gültigkeit $(i) \wedge (ii) \wedge (iii) \supset (iv)$. Alternativ kann gezeigt werden, daß $A := \neg[(i) \wedge (ii) \wedge (iii) \supset (iv)]$ unerfüllbar ist. Dazu wird A in eine entsprechende Menge von Klauseln transformiert:

(v) Vogel(Patty),

(vi) $\neg$Vogel(Patty) $\vee$ Fliegt(Patty),

(vii) Geflügelt(Patty) $\vee$ $\neg$Fliegt(Patty),

(viii) $\neg$Geflügelt(Patty).

Es bleibt zu zeigen, daß eine Widerlegung aus $\{(v) - (viii)\}$ konstruiert werden kann:

(ix)	Fliegt(Patty)	aus (v) und (vi).
(x)	$\neg$ Fliegt(Patty)	aus (vii) und $(viii)$.
(xi)	$\square$	aus (ix) und (x).

Betrachtet man auch Formeln, die Variablen enthalten, benötigt man noch die Begriffe der Substitution von Variablen in einer Formel und den Unifikationsalgorithmus, der zu einer Menge von Ausdrücken (Terme oder Literale) eine Menge von Variablenzuweisungen so bestimmt, daß die Ausdrücke nach Substitution identisch werden. Eine formale Beschreibung des erforderlichen Vorgehens ist aus Platzgründen nicht mehr sinnvoll.

Außer der Festlegung auf das Resolutionsprinzip schreibt Prolog auch die Auswahl der zur Widerlegung benutzten Klauseln vor. Prolog benutzt die Regeln und Fakten des Programms von oben nach unten und bearbeitet die in Konjunktionen auftretenden Zwischenziele *(Subgoals)* immer von links nach rechts. Diese Methode entspricht der Suche nach passenden Klauseln, die zuerst in die Tiefe geht *(Depth-First)*. Daß hier keine flexiblere Strategie zur Auswahl der Klauseln möglich ist, gehört zu den Schwachpunkten von Prolog.

Am Schluß soll noch angedeutet werden, wie Wissen konkret in einem Prolog System abgelegt werden kann. Die Übersetzung der MYCIN Regel aus Abbildung 2.1 auf Seite 55 ist in Abbildung 2.3 dargestellt und für das Beispiel des Optionsanlegers `kunde_1` ergeben sich die in Abbildung 2.4 dargestellten Fakten und Regeln in Prolog.

```
identity( Organism, neisseria ) :-
      gram_stain( Organism, gram_negative ),
      morphology( Organism, coccus ).
```

Abbildung 2.3: Eine MYCIN Regel in Prolog

2.7 Strukturierte Objekte

Unter dem Begriff strukturiertes Objekt werden alle Arten von Repräsentationen zusammengefaßt, in denen Informationen in mehr oder weniger natürli-

```
/* Fakten */
    anlageDauer( kunde_1, kurz ).
    anlageZweck( kunde_1, Spekulation ).
    renditeErwartung( kunde_1, überdurchschnittlich ).
    risikobereitschaft( kunde_1, überdurchschnittlich ).
    steuerersparnis( kunde_1, nein ).
/* Regel */
    anlageEmpfehlung( Kunde, optionen ) :-
        anlageDauer( Kunde, kurz ),
        anlageZweck( Kunde, Spekulation ),
        renditeErwartung( Kunde, überdurchschnittlich ),
        risikobereitschaft( Kunde, überdurchschnittlich ).
/* Anfrage */
    ?- anlageEmpfehlung( kunde_1, A ).
/* Antwort */
        A = optionen
```

Abbildung 2.4: Eine Regel aus dem Optionsgeschäft in Prolog cher Weise gruppiert werden. Das steht im Gegensatz zur Modularität von Produktionsregeln (siehe Abschnitt 2.5 auf Seite 55) oder den Formeln der Prädikatenlogik (siehe Abschnitt 2.3.5 auf Seite 41), die im Idealfall voneinander völlig unabhängig gültige, kleine *Wissenspartikel* bilden.

Es wird im wesentlichen von zwei Gruppierungsmöglichkeiten ausgegangen. Es handelt sich um Schemata, die entweder den Knoten und Kanten der Graphentheorie entlehnt sind oder als Verallgemeinerung der aus den klassischen Programmiersprachen bekannten Strukturen (`struct` in `C` [Kernighan and Ritchie, 1988] oder `record` in `Pascal` [Jensen and Wirth, 1978]) aufgefaßt werden können. Bei dem letzteren Vorgehen spricht man auch von *Slot and Filler* Notation.

2.7.1 Semantische Netze

Da Semantische Netzwerke in verschiedener Form vorkommen und auch für verschiedenste Aufgaben eingesetzt werden, ist es einigermaßen schwierig, ihre essentiellen Eigenschaften festzuhalten.

In ihrer grundlegenden Form bestehen Semantische Netzwerke aus Objekten (Knoten, Nodes) und binären Assoziationen (Kanten, Links), wobei die Knoten Individuen oder Konzepte und die Kanten binäre Relationen über diesen Konzepten repräsentieren.

Eine Wissensbasis besteht also hier aus einer Anzahl von Objekten und Relationen zwischen diesen Objekten. Modifikationen der Wissensbasis erfolgen durch Hinzufügen neuer oder Löschen bestehender Objekte oder Relationen.

Seit solche Netzwerke erstmals in [Quillian, 1968] vorgeschlagen wurden, werden ausschließlich binäre Relationen verwendet, um auch n-stellige Relationen zu simulieren. Daher hat jedes Netzwerk eine natürliche graphische Darstellung, in der jeder Knoten ein Objekt und jede gerichtete, beschriftete Kante (n_1, R, n_2) bedeutet, daß $(n_1, n_2) \in R$, wobei R eine Relation der Wissensbasis ist.

Frühe Versionen Semantischer Netzwerke litten oft unter dem ausufernden Wachstum der Anzahl der Relationen. Relationen wurden einfach nach Bedarf für jedes auftretende Problem definiert. Das erschwerte den Überblick über die intendierte Bedeutung der Beschriftungen. Nach Kritik von Woods [Woods, 1975] und anderen wurde versucht, mit wenigen grundlegenden Relationen auszukommen und kompliziertere Zusammenhänge durch Kombination vorhandener, wohl spezifizierter Relationen auszudrücken. Auch die Arbeiten von Fahlman [Fahlman, 1979] und Touretzky [Touretzky, 1986] sind im Licht dieses neuen Trends zu einer formalen Semantik zu sehen.

Einige dieser Schemata sehen Netzwerkwissensbasen nur als bequeme Notationen bzw. Implementationen logischer Wissensbasen an (so u.a. Brachmans KL-ONE System [Brachman and Schmolze, 1985]).

Der entscheidende Punkt bei der Strukturierung des Wissens sind die organisatorischen Dimensionen, die ein Netzwerkschema zur Verfügung stellt. Folgende Dimensionen sind insbesondere zu betrachten:

Klassifikation Jedem Objekt wird sein generischer Typ (bzw. eventuell seine Typen) zugeordnet. Dadurch werden innerhalb des Netzwerkes Unterschiede zwischen Individuen und Typen gemacht. Man spricht auch von dem Unterschied zwischen *Tokens* und *Types*.

Aggregation Diese Dimension ordnet jedem Objekt seine Teile oder Komponenten zu.

Generalisierung Generalisierung setzt Typen unteinander in Relation. Diese Relation wird oft als „ist eine Art von" bezeichnet, da hier ein Typ mit allgemeineren Typen assoziert wird. In der Literatur hat der Name *ISA* die häufigste Verbreitung gefunden, aber auch *AKO* (für a kind of) und *VC* (für virtual copy) werden z.B. in FRL respektive NETL[5] verwendet. Das Konzept der Generalisierung wird oft zur ökonomischen Speicherplatznutzung eingesetzt, da es erlaubt, Eigenschaften spezieller Objekte nicht explizit auszudrücken, sondern sie von allgemeineren Klassen zu vererben.

Partitionierung Zur Darstellung von Zeitabhängigkeiten, nur *hypothetischen* Welten und Überzeugungen hat es sich als sinnvoll erwiesen, das Netzwerk in eine Hierarchie von Gruppen einzuteilen (siehe die Arbeiten von

[5] Siehe Abschnitt 3.4.1 auf Seite 95.

> Hendrix über Partitioned Networks [Hendrix, 1979]). Die Anordnung einer Partition B unterhalb von A bedeutet, daß alles, was in Partition A vorhanden oder sichtbar ist, auch in Partition B sichtbar ist, falls keine explizite Information etwas anderes festlegt.

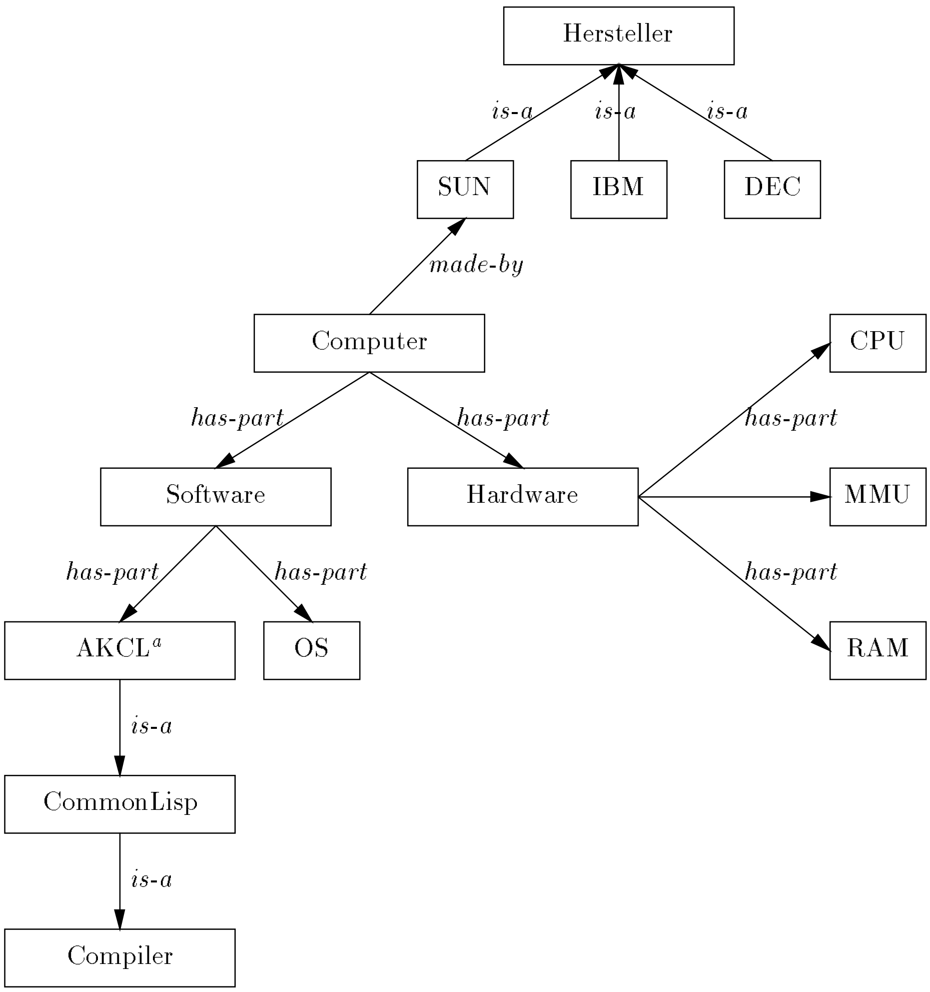
Semantische Netzwerke werden auch durch ihr Modell der Suche nach Informationen charakterisiert. Soll die Beziehung zwischen zwei Knoten (Konzepten) untersucht werden, so läßt man eine Art von Aktivierung von beiden Knoten ausgehen, die sich in allen Richtungen entlang der Kanten zwischen den Knoten durch das Netzwerk fortpflanzt *(Spreading Activation)*. Schneiden sich die Mengen der aktivierten Knoten irgendwann, so stehen die ursprünglichen Knoten in Relation. Dieses Modell liefert recht gute Übereinstimmung mit psychologischen Experimenten.

Das nächste Kapitel wird sich noch ausführlich mit Fahlmans NETL System [Fahlman, 1979] beschäftigen, deshalb soll hier Abbildung 2.5 auf Seite 69 als Demonstration genügen.

2.7.2 Frames

Die Idee der Darstellung von Informationen mit Hilfe von Frames stammt von Marvin Minsky [Minsky, 1975]. Wesentlicher Aspekt dieser Arbeit aus dem Bereich der *Computer Vision* ist die Vorstellung, daß das menschliche Gehirn bei dem Vorgang des Erkennens von Personen und Gegenständen nicht von einem Satz von logischen Formeln ausgeht, der die Identität des zu erkennenden Objekts herleiten kann. Minsky ging vielmehr davon aus, daß der Mensch bei der Bestimmung der Art eines Tieres vom seinem Wissen über eine mehr oder weniger große Anzahl von in einem noch genauer zu charakterisierenden Sinne *typischen* Vertretern dieser Art ausgeht. Erkennen bedeutet dann den erfolgreichen Vergleich der vorliegenden Beobachtungen mit den bekannten *typischen* Vertretern.

Frames ermöglichen diesen Vergleich von mehreren typischen Vertretern, indem sie die zur Verfügung stehenden Informationen über einen Vertreter in kompakter Weise zusammenfassen. Frames stellen insofern eine *objekt orientierte* Wissensrepräsentation dar, für die insbesondere folgende Eigenschaften charakteristisch sind:



[a] Austin Kyoto Common Lisp

Abbildung 2.5: Semantisches Netzwerk zur Darstellung des Begriffs *Computer*

- Ein Frame ist ein Objekt mit benannten Eigenschaften, die als *Slots* bezeichnet werden und mit Werten gefüllt sind. Man spricht deshalb auch von *Slot and Filler* Notation. Außer dem Wert eines Slots kann es noch weitere Aspekte geben, die für die Eigenschaft relevant erscheinen; diese werden in den sogenannten *Facetten* des Slots abgelegt, die wieder benannte Untereinheiten eines Slots sind.

- Frames werden in Form einer Hierarchie von Klassen, Teilklassen und Individuen zusammengefaßt.

Frames wurden unter anderem entwickelt, um prototypische Situationen und Objekte darzustellen. In einem Expertensystem tauchen Frames erstmals in der Reimplementation des auf EMYCIN (siehe Abschnitt 2.5) aufbauenden PUFF Systems auf. Dieses neue, mit Frames arbeitende System namens CENTAUR [Aikins, 1983], [Aikins, 1984] verwendet als allgemeine Problemlösungsmethode das hierarchische Hypothetisieren und Testen, das in Abschnitt 4.2.2 noch genauer beschrieben wird.

Während CENTAUR noch mit einer Mischung aus Frames und Regeln arbeitet, benutzt WHEEZE [Smith and Clayton, 1984] nur noch Frames. CENTAUR benutzt die Frames als Index in die Menge der Regeln und erreicht so eine bessere Fokussierung des Systems auf das aktuelle Problem. WHEEZE stellt durch eine sogenannte Agenda weitergehende Möglichkeiten zur Steuerung des Kontrollflusses zur Verfügung. Die Agenda ist im wesentlichen eine prioritätsgesteuerte Liste von noch durchzuführenden Teilzielen *(Subgoals)*. Jedes Teilziel kann dann z.B. die Untersuchung weiterer Teilziele mit einer vorgegebenen Dringlichkeit *(Priority)* anfordern. So können verschiedenste Strategien von reiner *Depth-First* bis zu reiner *Breadth-First* Suche implementiert werden. Insbesondere kann die Strategie für verschiedene Abstraktionsstufen des Problems unterschiedlich gewählt werden.

Das oben angekündigte Beispiel aus der Wertpapieranlage stellt sich dann wie in Abbildung 2.6 und Abbildung 2.7 dar.

```
(Class Anleger
  (Slots are
    ((OneOf Aktion Optionen RentenPapiere Edelmetalle)
                                    AnlageEmpfehlung)
    ((OneOf kurz mittel lang)       AnlageDauer
     to trigger (kurz .Optionen))
    ((OneOf klein mittel groß regelmäßig)
                                    AnlageBetrag)
    ((OneOf Spekulation Vermögensstreuung Vermögensansammlung)
                                    AnlageZweck)
    ((OneOf normal überdurchschnittlich noch_unklar)
                                    RenditeErwartung)
    ((OneOf normal überdurchschnittlich noch_unklar)
                                    Risikobereitschaft)
    ((OneOf ja nein)                SteuerErsparnis
     to ask "Wollen Sie Ihre Steuer vermindern? "
     to trigger (ja .Aktien .Optionen))
  ...
  ))
```

Abbildung 2.6: Die Klasse Anleger.

```
(Instance Options-Anleger
  (View Anleger with
    (AnlageEmpfehlung ist Optionen)
    (AnlageDauer ist kurz)
    (AnlageZweck ist Spekulation)
    (RenditeErwartung ist überdurchschnittlich)
    (Risikobereitschaft ist überdurchschnittlich)
    (Steuerersparnis ist nein)
  ))
```

Abbildung 2.7: Die Instanz Options-Anleger.

2.8 Objektorientierte Programmierung

Betrachtet man die experimentellen Wissensrepräsentationsprachen, die in den letzten 15 Jahren entstanden sind, so fällt auf, daß die meisten von ihnen *objektorientiert* sind. Fundamentales Organisationsprinzip ist die Integration von Daten und Prozeduren in eine durch einen Vererbungsmechanismus vernetzte Struktur. Während bei den in Abschnitt 2.7 besprochenen Formalismen die Vererbung und Kombination von *Daten* im Vordergrund steht, konzentriert sich die objektorientierte Programmierung auf die *Prozeduren.*

Anstelle des klassischen Konzeptes des Prozeduraufrufs von Funktionen tritt hier die Kommunikation der einzelnen Objekte durch den Austausch von Nachrichten *(Message Passing).*

Aus Sicht der Künstlichen Intelligenz war das in Abschnitt 3.2 noch eingehend besprochene KRL [Bobrow and Winograd, 1977] nach den oben erwähnten Frames von Minsky ein wichtiger Schritt in die objektorientierte Richtung, wie folgendes Zitat [Bobrow and Winograd, 1977, S. 5] belegt:

> ...to explore the consequences of an object-centred factorization of knowledge, rather than the more common factorization in which knowledge is structured as a set of facts, each referring to one or more objects.

Außer den noch zu besprechenden Unterschieden im deklarativen Aspekt leitete KRL auch im prozeduralen Teil eine neue Epoche ein. In klassischen Programmiersprachen sind die Argumente für Prozeduren oft streng typgebunden.[6] Erst neuere Sprachen wie C++ ermöglichen die Implementierung wirklich *polymorpher* Funktionen, bei denen man sich darauf verlassen kann, daß dem Datentyp der Argumente angepaßte Operationen verwendet werden, um eine generische Funktion zu erfüllen.

Objektorientiertes Programmieren bedeutet Implementation in Termen generischer Operationen, wobei Details der Realisierung für unterschiedliche Objekte

[6] Man spricht von *strong typing.* In Fortran oder C gibt es z.B. sogar zum Berechnen des Absolutbetrages von ganzen Zahlen oder reellen Zahlen unterschiedliche Funktionen.

auch unterschiedlich definiert werden können. Abstrakte Datentypen erlauben es dem Programmierer, von den genauen Einzelheiten der Speicherung seiner Daten in der Maschine abzusehen. Ebenso braucht man sich bei der Verwendung generischer Operationen nicht mehr ständig Gedanken über die wirkliche Implementation dieser Funktionen zu machen, sondern kann sich auf das Design des Zusammenspiels der Objekte konzentrieren, das sich in der Definition der von den Objekten verstandenen Kommunikationsprotokolle niederschlägt.

Zusammenfassend kann gesagt werden, daß die wesentliche Eigenschaft objektorientierter Systeme die *kontextabhängige* Interpretation generischer Konzepte ist. Die Implementation eines generischen Konzepts für ein konkretes Objekt wird als *Methode* bezeichnet.

Während der Einfluß von KRL eigentlich mehr theoretischer Natur war, liegen auch außer der Programmiersprache Smalltalk [Goldberg and Robinson, 1983] eine Reihe von Systemen vor, die in LISP geschrieben sind und dieses um objektorientierte Konzepte erweitern. Die bekanntesten sind FLAVORS [Moon and Weinreb, 1980], LOOPS *(Lisp Object Orientied Programming System)* [Bobrow and Stefik, 1981] und das neuere CLOS *(Common LISP Object System)* [Steele Jr., 1990], das sogar in den kommenden ANSI LISP Standard aufgenommen werden wird.

Das Beispiel des Optionsanlegers könnte sich in FLAVORS z.B. wie Abbildung 2.8 darstellen.

Die drei oben erwähnten Systeme unterscheiden sich hauptsächlich in ihrem Ansatz zur *Kombination* der Methoden. Als Besonderheit bei der Vererbung von *Prozeduren* stellte sich nämlich heraus, daß es wesentlich ist, Methoden von seinen Eltern zu übernehmen, aber diese müssen im allgemeinen noch modifiziert oder ergänzt werden. Typisch ist z.B. der Fall, daß erst die Methode der Eltern aufgerufen wird, und dann noch speziellerer Code zur Ergänzung benutzt wird[7]. FLAVORS erreicht dieses Mischen typischerweise durch **before-** und **after-methods**, die vor bzw. nach den Methoden der Eltern angesprochen werden.

[7] Der Name FLAVORS leitet sich auch aus diesem *Mischen* ab. Er soll an das Mischen von Gewürzen erinnern.

```
(defflavor (Anleger)
  AnlageDauer
  AnlageBetrag
  RenditeErwartung
  Risikobereitschaft
  Steuerersparnis
  )

(setq Kunde-1
  (make-instance Anleger
     AnlageDauer 'kurz
     AnlageZweck 'Spekulation
     ...
     ))

(defflavor (Anleger AnlageEmpfehlung)
  (cond ((and (eq AnlageDauer 'kurz)
              (eq AnlageZweck 'Spekulation)
       (eq RenditeErwartung 'überdurchschnittlich)
       (eq Risikobereitschaft 'überdurchschnittlich)
       (eq Steuerersparnis 'nein))
  'Optionen)
 ...
 (t '|Es kann keine Anlageempfehlung gegeben werden|)))
```

Abbildung 2.8: Darstellung des Optionsanlegers mit FLAVORS

Während FLAVORS noch spezielle Methodendefinitionen wie *Wrapper* oder *Whopper* benutzte, um solch ein Mischen zu implementieren, führte LOOPS durch die Funktion ←**Super** *(Super Send)* den Ansatz ein, Nachrichten in veränderter Form an Elternklassen weiter zu schicken, um die dort vorhandene Funktionalität zu nutzen. Daß aber Nachrichten an beliebige Objekte geschickt werden konnten, die in der Klassenhierarchie auch *außerhalb* der Vorfahren eines Objekts lagen, wurde oftmals mißbraucht. CLOS integriert die in den beiden Vorläufersystemen vorhandenen Konzepte.

Instanzen ererben Daten und Methoden von Klassen. Um auch die Methoden zur Erzeugung und Initialisierung der Objekte, die sich ja in den Klassen befinden, vererben zu können, bieten sowohl LOOPS als auch CLOS *Metaklassen* an. Klassen ererben so ihr Verhalten von den Metaklassen.

2.9 Logik

Betrachtet man die Prädikatenlogik erster Stufe ohne Erweiterungen, dann hat man es mit einem extensionalen[8], nicht modalem Kalkül zu tun, der gut verstanden wird. Hier ist auch das allgemeine, vollständige Resolutionsprinzip von Robinson[9] zum automatischen Beweisen von Theoremen anwendbar. Sobald diese Restriktionen fallen gelassen werden, werden die Dinge, wie schon in der Betrachtung der Prädikatenlogik zweiter Stufe (siehe Abschnitt 2.3.10 auf Seite 48) oder der Modallogik (siehe Abschnitt 2.3.11) gesehen wurde, komplizierter und sind schwieriger zu verstehen.

[8] Die *Extension* eines Prädikates P ist die Menge aller Objekte (oder Tupel von Objekten), die P erfüllen. Bei der *Intension* handelt es sich um den schwer zu formalisierenden Begriff der Bedeutung des Prädikats P. Daß hier ein Unterschied vorliegt, wird deutlich, wenn man Prädikate betrachtet, deren Extension die leere Menge ist. Extensional sind diese Prädikate nicht unterscheidbar, intendiert wird aber durchaus etwas anderes. Vergleiche die intensionalen Beschreibungen der Wissensrepräsentationsprache KRL in Abschnitt 3.3 auf Seite 85.

[9] Siehe Abschnitt 2.6.1 auf Seite 62, die Originalarbeiten [Robinson, 1983] und [Robinson, 1985] oder zum Überblick [Nilsson, 1980].

2.9.1 Das Frame Problem

Sei der Zustand einer Welt in symbolischer Form in einem System repräsentiert. Diese Darstellung wird nun im Zeitverlauf durch *Aktionen* modifiziert. Dann handelt es sich bei dem sogenannten *Frame Problem* um die Aufgabe, darzustellen, welche Eigenschaften der Welt von der Aktion unberührt bleiben und welche Eigenschaften sich bei Durchführung der Aktion ändern. Aktionen ändern normalerweise nur einen kleinen Teil der betrachteten Welt. Wenn z.B. ein Klötzchen bemalt wird, so ändert sich nur seine Farbe.[10] Alle anderen Eigenschaften behalten ihren ursprünglichen Zustand. Um diese *invarianten* Eigenschaften explizit darzustellen, sind eine große Anzahl von *Rahmenbedingungen* durch Frame Axiome festzuschreiben. Alle diese Axiome könnten auf natürliche Weise durch eine einzige Regel ersetzt werden, die besagt: „Alle Eigenschaften der Welt bleiben invariant gegenüber Aktionen, außer *den* Eigenschaften, die explizit geändert werden."

Das *Qualifikationsproblem* ist das Problem, daß der Erfolg einer geplanten Aktion von einer großen Zahl von Voraussetzungen oder Qualifikationen abhängt. Selbst in einer relativ einfachen Welt wie der in der Künstlichen Intelligenz oft betrachteten Klötzchenwelt[11] *(Blocks World)* sind viele Qualifikationen zu überprüfen, bevor ein Klotz A auf einen anderen Klotz B gelegt werden kann. Die Oberseite von B muß frei und groß genug sein, um A tragen zu können. A darf nicht zu schwer sein. Es dürfen keine weiteren Klötze auf A liegen. Der Arm des Roboters darf nicht gebrochen sein, usw. Je genauer man die Situation studiert, um so mehr Gründe für ein potentielles Fehlschlagen der Aktion findet man. In realistischeren Welten können viele Qualifikationen gar nicht verifiziert werden. So bleibt keine andere Wahl, als die Aktion zu versuchen, um erst im Fall eines auftretenden Problems korrigierend einzugreifen und die

[10] Bei genauer Betrachtung ändert sich natürlich auch mehr. Die Farbe ändert sein Gewicht, seinen Geruch, seine Oberflächenbeschaffenheit. Es kommt also oft darauf an, Gegenstände *so* detailliert zu betrachten, daß die Darstellung dem Zweck angepaßt ist. Es ist sinnvoll, gewisse Eigenschaften erst einmal als nebensächlich zu betrachten und erst dann einzubeziehen, wenn sich sonst nicht erklärbare Unstimmigkeiten ergeben.

[11] Hier soll ein kleiner Roboter aus verschiedenfarbigen und unterschiedlich gestalteten Klötzchen einen Turm bauen. Viele Programme zum Planen betrachten dieses oder ähnlich gelagerte Probleme.

Planungen zu modifizieren. Auch das Qualifikationsproblem kann auf natürliche Weise gelöst werden: „Solange keine Evidenz für das Gegenteil vorliegt, gehe davon aus, daß eine Aktion erfolgreich sein wird".

Sowohl das Frame Problem als auch das Qualifikationsproblem legen die Beschäftigung mit Formalismen nahe, die die *nichtmonotone* Zurücknahme von zu einem Zeitpunkt t geschlossener oder auch nur angenommer Aussagen im Lichte von neueren, erst zu einem späteren Zeitpunkt t' zur Verfügung stehenden Informationen erlauben.

2.9.2 Nichtmonotone Logik

Ein großer Teil der in der Künstlichen Intelligenz geleisteten Arbeit auf dem Gebiet der Darstellung von *Defaults* und prototypischen Eigenschaften von Objekten hat sich auf Systeme konzentriert, die nichtmonotones und *Default* Schließen verwenden.

McDermott und Doyle erweiterten in [McDermott and Doyle, 1980] und [McDermott, 1982a] die Prädikatenlogik erster Stufe um eine nichtmonotone Komponente. Die Aussage „Vögel fliegen" wird dort z.B. so formalisiert:

$$(\forall x)((\texttt{Vogel}(x) \wedge \mathcal{M}\texttt{fliegt}(\S)) \supset \texttt{fliegt}(\S))$$

Informell kann das interpretiert werden als „Für alle x, falls es stimmt, daß x ein Vogel ist, und es konsistent ist, daß x fliegt, dann schließe, daß x fliegt."

Das oben verwendete Symbol $\mathcal{M}$ darf nicht mit dem Operatorsymbol für die Möglichkeit einer Aussage verwechselt werden, das in der Modallogik (siehe Abschnitt 2.3.11 auf Seite 49) verwendet wird. Hier geht es um die Herleitbarkeit in der Logik selbst.

Auf der anderen Seite wird „Vögel fliegen" in Reiters System des *Default Reasoning* [Reiter, 1980] durch folgende Regel dargestellt:

$$\frac{\texttt{Vogel}(x) : \texttt{fliegt}(x)}{\texttt{fliegt}(x)}.$$

Als Interpretation dient hier: „Falls geschlossen werden kann, daß x ein Vogel ist, und es konsistent ist anzunehmen, daß x fliegt, dann schließe, daß x fliegt."

Eine dritte Variante ist McCarthys *Circumscription* Methode [McCarthy, 1980] [McCarthy, 1986]. Er *umschreibt* (oder *definiert*) ein Abnormalitätsprädikat Ab, um die Anzahl der Vögel zu minimieren, die hinsichtlich des Fliegens nicht normal sind:

$$\forall x \texttt{Vogel}(x) \wedge \neg Ab(x) \supset \texttt{fliegt}(x)$$

Da logische Formalismen noch in Kapitel 5 als Methode zur Definition der Bedeutung eines Semantischen Netzwerks eine Rolle spielen, soll die obige kurze Betrachtung hier genügen.

2.10 Vergleich der Formalismen

Als Implementierungssprache für Wissensrepräsentationen ist LISP prinzipiell weder besser noch schlechter als PROLOG geeignet. Sobald es um wirkliches Programmieren geht, stellen sich doch viele formale Eigenschaften der zugrundeliegenden Theorie als wenig hilfreich heraus. Weder der λ-Kalkül noch Horn-Klauseln können direkt die Erklärung für intelligentes Verhalten liefern.

Die hohe Modularität von Produktionsregeln bringt den Vorteil, daß sich das Verhalten des Systems bei Änderung einer Regel im allgemeinen nicht drastisch ändert. Es fehlen jedoch Ordnungsprinzipien, die das Wissen überschaubar machen.

Diesen Mangel behebt die Wissensrepräsentation mit Strukturierten Objekten. Oft werden Produktionsregeln auch in solche Systeme integriert und führen zu einem hybriden Ansatz, der versucht, die besten Eigenschaften *aller* Formalismen zu bieten.

Dem direkten Einsatz klassischer Logik zur Wissensrepräsentation widersprechen folgende Beobachtungen:

- Die Aussagenlogik ist zwar entscheidbar, aber es gibt bis heute keinen effizienten polynomialen Algorithmus zur Überprüfung der Erfüllbarkeit von Formeln.[12]

- Die Aussagenlogik ist nicht ausdrucksstark genug, um die in der Künstlichen Intelligenz geforderten Beschreibungen in ihren Formalismus zu übernehmen. Wegen des in Abschnitt 2.9.1 angesprochenen Frame Problems reicht dazu noch nicht einmal die Prädikatenlogik erster Stufe.

Aufgrund der Unentscheidbarkeit der Prädikatenlogik zweiter Stufe bleibt nur der Ausweg nichtmonotoner Erweiterungen der klassischen Logik, die oben schon kurz betrachtet wurden. Nichtmonotone Logiken können aber ebenfalls nicht direkt als Wissensrepräsentation verwendet werden. Sie dienen jedoch neben den direkten pfadbasierten Methoden der Kapitel 5 und 6 als Basis zur indirekten Spezifikation einer Semantik von assoziativen Netzwerken mit Hilfe von Übersetzungen der Knoten und Kanten in logische Formeln und werden deshalb noch in Kapitel 5 ausführlicher behandelt.

[12] Vergleiche auch Kapitel 7.

Kapitel 3

Implementation semantischer Netzwerke

Wie schon im letzten Kapitel erwähnt, werden unter dem Begriff „Semantisches Netzwerk“ ganz unterschiedliche Ideen und Zielsetzungen zusammengefaßt.

Semantische Netzwerke werden hier ausführlich behandelt, da sowohl praktische Implementationstechniken als auch ihre theoretischen Eigenschaften in dieser Arbeit genauer untersucht werden.

LISP und Prolog sind die wichtigsten KI-Sprachen, deshalb werden einige Beispiele in diesen Sprachen erläutert. Siehe z.B. [Steele Jr., 1990], [Tatar, 1987], [Wilensky, 1986] oder [Winston and Horn, 1983] als Einführung in LISP und [Clocksin and Mellish, 1990], [Coelho and Cotta, 1988] oder [Sterling and Shapiro, 1986] für Prolog.

3.1 Einführung

In den folgenden Abschnitten werden drei klassische Implementationen eingehend betrachtet. Die Auswahl der untersuchten Systeme orientierte sich dabei an ihrem Einfluß auf die Implementation von **DONALD**. So werden die frühen Ansätze von Quillian aus den sechziger Jahren ([Quillian, 1967] und [Quillian, 1968]) wegen ihrer Beschränktheit auf das Problem der Semantik von Wörten nicht weiter verfolgt.

Hauptgewicht wird vielmehr auf der *Frame Representation Language* (FRL) von Goldstein und Roberts, der *Knowledge Representation Language* (KRL) von Bobrow und Winograd und der *Network Language* (NETL) von Fahlman liegen.

3.2 FRL

FRL (Frame Representation Language) wurde von Ira Goldstein und Bruce Roberts entworfen und in [Roberts and Goldstein, 1977b] und [Roberts and Goldstein, 1977a] beschrieben. Einige grundlegende Ideen sollen im folgenden dargestellt werden.

FRL ist eine, in der Programmiersprache LISP implementierte, Sprache zur Darstellung von Frames. Sie werden als geschachtelte Assoziationslisten gespeichert. Eine Assoziationsliste ist eine benannte Liste von Paaren von Schlüsseln und Werten. Als Beispiel soll die Liste A mit Namen N und den Werten $v_1, v_2, \ldots, v_n$ für die Schlüssel $k_1, k_2, \ldots, k_n$ dienen:

$$A := (N \quad (k_1 \quad v_1) \quad (k_2 \quad v_2) \quad \ldots \quad (k_n \quad v_n))$$

Jeder Frame ist nun eine dreistufig geschachtelte Assoziationsliste, wobei N auf der obersten Ebene den Framenamen enthält, auf der nächsten Stufe den Namen eines Slots und schließlich den einer Facette.

In FRL vorgesehene Facetten sind unter anderem VALUE für den Wert des Slots, DEFAULT für einen Ersatzwert, der bei Fehlen eines richtigen Wertes angenommen werden soll, sowie drei weitere Facetten, deren Werte Funktionen sind, die bei Zugriff auf den Wert des Slots vom System automatisch aktiviert werden. Solche Funktionen werden als *Demons* bezeichnet.

Das Beispiel des Optionsanlegers aus **INVEST** nimmt in FRL folgende Gestalt an:

```
(Anleger
   (AKO Kunde)
   (AnlageEmpfehlung)
   (AnlageDauer) ...
   (Risikobereitschaft (DEFAULT klein))
   (SteuerErsparnis (DEFAULT ja) (IF-NEEDED ask-steuer))
   ...)
```

```
(Options-Anleger
   (AKO (VALUE Anleger))
   (AnlageEmpfehlung (VALUE Optionen))
   (AnlageDauer (VALUE kurz))
   (AnlageZweck (VALUE Spekulation))
   (RenditeErwartung (VALUE überdurchschnittlich))
   (Risikobereitschaft (VALUE überdurchschnittlich))
   (Steuerersparnis (VALUE nein)))
```

Die für ein Framesystem charakteristische Möglichkeit zur Vererbung von Information von Frame zu Frame wird durch den in FRL speziell behandelten `AKO` *(A Kind Of)* Slot gegeben. Die folgenden Zugriffsfunktionen für Slotwerte sind definiert:

`(fget ◁frame▷ ◁slot▷ ◁facet▷)`
: gibt die Liste der Werte der durch den Zugriffspfad `◁frame▷`, `◁slot▷` und `◁facet▷` bezeichneten Facette zurück. Existiert kein Wert, so ist das Ergebnis die leere Liste (in LISP traditionell mit `NIL` bezeichnet).

`(fput ◁frame▷ ◁slot▷ ◁facet▷ ◁value▷)`
: fügt `◁value▷` der über den Zugriffspfad bezeichneten Liste hinzu.

`(fremove ◁frame▷ ◁slot▷ ◁facet▷)`
: entfernt die Facette `◁facet▷` aus dem über den Zugriffspfad `◁frame▷` und `◁slot▷` bezeichneten Slot.

`(fget-v-d ◁frame▷ ◁slot▷)`
: gibt die Liste aller Werte, die in dem angegebenen Frame und Slot unter der VALUE Facette abgelegt sind, zurück. Falls kein Wert eingetragen ist, so wird der unter der DEFAULT Facette eingetragene Wert zurückgegeben.

`(fget-i ◁frame▷ ◁slot▷)`
: durchsucht den Slot `◁slot▷` nicht nur in `◁frame▷`, sondern auch rekursiv in allen mit diesem durch den AKO Slot verbundenen Frames nach einer VALUE Facette.

`(fget-n ◁frame▷ ◁slot▷)`
: arbeitet ähnlich wie `fget-i`, falls jedoch kein Wert gefunden wurde, so wird die AKO Hierarchie nochmals nach DEFAULT Facetten und eventuell auch noch nach einer IF-NEEDED Funktion (siehe unten) durchsucht.

`(fget-z ◁frame▷ ◁slot▷)`
: unterscheidet sich von `fget-n` dadurch, daß die AKO Hierarchie erst in dem Fall weiter durchsucht wird, wenn VALUE, DEFAULT und IF-NEEDED Facetten des Slots keine Resultate liefern.

Die oben erwähnte IF-NEEDED Facette enthält ebenso wie die IF-ADDED und IF-REMOVED Facetten in der FRL Implementation die Namen von parameterlosen LISP Funktionen. Diese Funktionen werden dann von FRL bei den ensprechenden Zugriffen aktiviert. Durch lokale Variablen wie `*frame*`[1] und `*slot*` in den Zugriffsfunktionen `fget` und `fput` können diese Demons

[1] Durch die Sterne soll angedeutet werden, daß es sich um *dynamische* Variablen handelt. Diese Bezeichnung entspricht allgemeiner Konvention.

jedoch in einem LISP System mit *dynamic scoping*[2] sehr einfach etwas über ihre Aufrufumgebung erfahren. Das soll aber nicht heißen, daß so etwas mit *static scoping* nicht möglich wäre, sondern es ist eben für Demonstatrationszwecke einfacher. Ganz im Gegenteil verwenden alle modernen LISP Systeme wie z.B. Common LISP [Steele Jr., 1990] und auch LISP-ähnliche Systeme wie Scheme, das auch in dem Lehrbuch [Abelson and Sussman, 1985] verwendet wird, heute alle *static scoping*.

Als Beispiel soll der oben in der IF-NEEDED Facette benutzte `ask-steuer` Demon kodiert werden:

```
(defun ask-steuer ()
  (format t "Wollen Sie Ihre Steuer vermindern? ")
  (read))
```

Ein einfacher Demon zum Erfragen der Eigenschaft `color` eines beliebigen Frames F unter Ausnutzung der Variablen `*frame*` und `*slot*` könnte so aussehen:

```
(defun ask ()
  (format t "Please enter a value ~%
          for the ~S of ~S: " *slot* *frame*)
  (read))

(fput 'F 'color 'IF-NEEDED 'ask)
```

Dieser rudimentären Form der Darstellung von Wissen folgte mit der Knowledge Representation Language von Bobrow und Winograd, der der nächste Abschnitt gewidmet ist, ein wesentlich komplexeres System mit sehr differenzierten Ausdrucksmöglichkeiten.

[2] Zur Erläuterung des Unterschiedes zwischen *dynamic* und *static* Scoping siehe z.B. [Stoyan and Görz, 1984].

3.3 KRL

3.3.1 Überblick

KRL (Knowledge Representation Language) wurde von Daniel G. Bobrow und Terry Winograd [Bobrow and Winograd, 1977] zur Verwendung in Systemen zum Verstehen natürlicher Sprache entworfen und stellt einen Versuch zur Integration von prozeduralem und deklarativem Wisssen dar. Die logische Struktur des Wissens kann auf verschiedenste Art ausgedrückt werden, um das prozedurale Wissen flexibel speziellen Strukturen zuorden zu können, und um die Zugriffspfade auf eine Gedächtniszelle so zu organisieren, daß der Zugriff auf oft gebrauchte Strukturen beschleunigt wird. KRL wurde von Lehnert [Lehnert, 1977] kritisiert, die Antwort auf diese Kritik findet sich in [Bobrow and Winograd, 1979].

Die Wissensrepäsentationssprache KRL stellt deklaratives Wissen durch strukturierte Objekte mit assoziierten Deskriptoren dar. Diese Objekte bilden ein Netzwerke von Gedächtniseinheiten *(Units)*[3] mit verschiedenartigen Verbindungen, die jeweils unterschiedliche Bedeutung für die in diesem Netzwerk ablaufenden Suchprozesse haben. Jedem Objekt können aber auch direkt Prozeduren zugeordnet werden. Dadurch wird die Durchführung einer speziellen Operation auf einem Objekt von der Charakteristik dieses Objekts beeinflußt oder auch gänzlich bestimmt. Genau diese Idee liegt allen heutigen objektbasierten Programmiersprachen zugrunde.

Die Autoren von KRL waren davon überzeugt, daß jedes *intelligente* Programm zur Verarbeitung von Wissen folgende Ideen berücksichtigen sollte:

- Wissen sollte in konzeptionellen Einheiten mit assoziierten Beschreibungen und Prozeduren organisiert werden.
- Eine solche Beschreibung sollte partielles und unvollständiges Wissen über ein Objekt darstellen, und es unter verschiedenen Blickwinkeln mit Hilfe mehrerer Deskriptoren beschreiben können.

[3] Diese Units entsprechen in etwa den in anderen Systemen gebräuchlichen Frames.

- Eine wichtige Methode zur Beschreibung ist der Vergleich mit einem bekannten Objekt mit genauerer Spezifikation der Unterschiede zu diesem Prototyp.
- Logisches Schließen wird durch einen Erkennungsprozeß dominiert, in dessen Verlauf neue Objekte und Ereignisse mit gespeicherten Mengen von erwarteten Prototypen, die eventuell noch mit speziellen Strategien ausgerüstet sein können, verglichen werden.
- Intelligente Programme erfordern mehrere gleichzeitig aktive Prozesse mit explizit vom Benutzer angegebenen Algorithmen für die Vorrangsteuerung und die Zuordnung von Betriebsmitteln.
- Informationen sollten so gruppiert sein, daß die Unterschiede in der Zugriffsmethode reflektiert werden.
- Eine Sprache zur Wissensrepräsentation sollte eine flexible Menge von Werkzeugen zur Verfügung stellen, anstatt spezielle Annahmen über Strategien und die darzustellenden Objekte zu machen.

KRL enstand im wesentlichen aus dem Bemühen, deklaratives Wissen objektbezogen zu faktorisieren, anstatt es durch eine Menge von Fakten darzustellen, die sich auf ein oder mehrere Objekte beziehen. Beispiele für die konzeptionellen Einheiten, die in KRL beschrieben werden, sind Objekte, Relationen, Szenen und Ereignisse. Diese Beschreibungen sind intensional[4] und erlauben damit die Identifikation und den Vergleich mit anderen Einheiten.

Drei wichtige Operationen mit solchen Beschreibungen liegen dem System zugrunde. Dies sind die Erweiterung einer Beschreibung um neues Wissen, der Vergleich *(Matching)* zweier Beschreibungen, ob sie in gewissem Sinne kompatibel sind und die Suche nach Einheiten, die zu einer vorgegebenen Einheit passen. Folgende Spezifikationen sind möglich:

- Da eine Beschreibung aus mehreren Deskriptoren besteht, kann ein Objekt sowohl aus verschiedenen Gesichtpunkten beschrieben werden, als auch durch zusätzliche Behauptungen näher spezifiziert werden.

[4] Zum Unterschied zwischen extensionalen und intensionalen Beschreibungen vergleiche Abschnitt 2.9 auf Seite 75.

- Ein Objekt kann durch seine Einordnung als Mitglied einer Kategorie oder durch Darstellung seiner Rolle innerhalb eines komplexen Objekts oder Ereignisses, sowie als Teil einer Beziehung zwischen Objekten beschrieben werden.

- Ebenso kann ein Objekt durch einen eindeutigen Identifikator, eine komplexe logische Formel, die für das Objekt wahr sein soll, spezifiziert sein.

- Außerdem kann es ein Element einer Menge sein, oder eine Menge wird durch die in ihr enthaltenen Objekte beschrieben.

In KRL wird großes Gewicht auf die Möglichkeit der Beschreibung eines Objektes durch Vergleich gelegt. Dadurch ist es möglich, daß ein System eine funktional vollständige Repräsentation eines Objektes besitzt, das heißt Antworten auf alle relevanten Fragen geben kann, ohne über eine wirkliche Definition zu verfügen.

Beschreibt man Objekte durch Vergleich, so liegt es oft nahe, ein Objekt zum Vergleich heranzuziehen, daß in gewissem Sinne typisch für ein Mitglied der Klasse solcher Objekte ist. Das impliziert oft, daß so ein Prototyp überhaupt nicht real existiert, sondern nur zur Sammlung von Default-Wissen dient.

Ein Objekt kann so durch Bezugnahme auf mehrere verschiedene Prototypen jeweils eventuell mit Angabe weiterer spezifizischer Information beschrieben werden. So erhält man eine Vererbung von Eigenschaften, ähnlich wie in FRL.

Soll ein mit nur unvollständiger Information ausgestattetes System sinnvoll arbeiten, so erscheint die Verwendung von typischen und erwarteten Eigenschaften von Objekten, die solange als richtig erachtet werden, wie sie mit anderem Wissen verträglich sind, unabdingbar. Die Autoren von KRL halten die Aufteilung des Wissens über ein Objekt unter mehreren Gesichtpunkten (views) in eine prototypische Komponente und eine spezifische Komponente nicht nur für einen technischen Trick, sondern sind der Überzeugung, daß auch die menschliche Informationsverarbeitung ihr Wissen in vergleichbarer Form strukturiert.

3.3.2 Syntax

KRL Datenstrukturen bestehen aus Beschreibungen, die in sogenannten *units* zusammengefaßt werden. Jede Unit hat einen Namen, einen Kategorietyp und einen oder mehrere benannte Slots, die Beschreibungen von Objekten enthalten, die mit der Unit assoziiert sind. Jeder Slot hat einen Slotnamen, der innerhalb der Unit eindeutig ist. Jede Unit enthält einen ausgezeichneten Slot (`SELF` genannt), der die Unit selbst beschreibt. Weiterhin kann eine Menge von Prozeduren, die unter bestimmten Bedingungen aktiviert werden, mit einem Slot assoziiert werden.

Jede Beschreibung besteht aus einer Liste von Deskriptoren, die jeder mit einer Menge von Features versehen sein können. In KRL-0 gibt es eine feste Anzahl von Deskriptoren, die sich in ihrer syntaktischen Form unterscheiden. Ihnen allen ist gemeinsam, daß sie prinzipiell LISP Syntax haben, das heißt, ihre Baumstruktur wird durch Klammerpaare definiert. Im Unterschied zu LISP gibt es jedoch vier Klammertypen, wobei die eckigen Klammern [...] Units umschließen, die spitzen $< \ldots >$ die einzelnen Slots. Jeder komplexe Deskriptor wird duch die einfachen Klammern (...) begrenzt, während die geschweiften Klammern {...} mehrere Deskriptoren zu einer komplexen Beschreibung zusammenfassen. In jedem modernen LISP System läßt sich jedoch der Reader so parametrisieren, daß eine solche Struktur einfach mit der Funktion `read` eingelesen werden kann.

3.3.3 Deskriptoren in KRL

Die in KRL definierten Deskriptoren haben alle die Gestalt von LISP Listen oder Atomen. Die KRL Schlüsselwörter sind dabei zur Verdeutlichung **fett** hervorgehoben. Die ersten Elemente der Listen (in LISP ist das ihr `car`) ordnen sie einer der folgenden Deskriptorfamilien zu[5]:

Direkte Zeiger

[5] Man spricht auch von einer *tagged* List.

Format Ein Unitname, eine Zahl, eine Zeichenkette oder ein durch ein vorangestelltes **QUOTE** vor weiterer Evaluation geschütztes LISP-Objekt.

Gebrauch Dient als eindeutiger Identifikator.

Beispiele Kunde, Wertpapier, 2.718281828, "ja",
(**QUOTE** (Integral (-1.0 1.0) (exp (∗ x x))(d x)))

Perspektive

Format
(**a** prototype **with** $\text{identifier}_1 = \text{filler}_1 \ldots$
$\text{identifier}_n = \text{filler}_n$)

Gebrauch Beschreibt ein Objekt als Mitglied einer Kategorie durch Vergleich mit `prototype`, wobei einige Slots durch die Filler genauer spezifiziert sind.

Beispiele
(**a** Car **with** color = red
owner = Pi)

Spezifikation

Format
(**the** slotSpecifier **from** view targetDescription)

Gebrauch Beschreibt den Filler des Slots, den targetDescription unter der Perspektive view enthält.

Beispiele
(**the** owner **from** Car (**a** Car **with** color = red))

Prädikation

Format
(**which** predicateName . predicateArgs)

Gebrauch Beschreibt ein Objekt durch seine Relation zu den Argumenten des Prädikats.

Beispiele
(**which** Owns (**a** Car))

Logisch

Format
(**OR** . booleanArgs) (**XOR** . booleanArgs)
(**NOT** . booleanArgs)

Gebrauch Klar, **AND** ist implizit durch {...}.

Beispiele
(**OR** (**a** Car) (**a** Train) (**a** Plane))

Restriktion

Format
(**theOne** restrictionDesc)

Gebrauch Markiert restrictionDesc als ausreichend, um ein Objekt im aktuellen Kontext eindeutig zu identifizieren.

Beispiele
(**theOne** {(**a** Person) (**which** Owns (**a** Car))})

Selektion

Format
(**using** selectionDesc
selectFrom pattern$_1$ $\sim$ selection$_1$ ...
selectFrom pattern$_n$ $\sim$ selection$_n$
otherwise defaultSelection)

Gebrauch Deklarative Form eines `switch` Statements der Programmiersprache C (siehe z.B. [Harbison and Jr., 1991]).

Beispiele
(**using** (**the** age **of** Person Pi)
selectFrom (**which** isLessThan 2) $\sim$ Infant
selectFrom (**which** isAtLeast 18) $\sim$ Adult
otherwise Child)

Mengenspezifikation

Format
(**SetOf** setElementDescription)

(**In** setDescription)
(**Items** . elements)
(**NotItems** . setDescription)
(**AllItems** . elements)
(**ListOf** . setDescription)
(**Sequence** . elements)

Gebrauch Beschreibt eine Menge durch ihre Elemente und Objekte durch ihre Mitgliedschaft in einer Menge.

Beispiele
(**SetOf** {(**an** Integer) (**which** hasFactor 2)})
(**AllItems** 1 2 64 {(**an** Integer) (**which** hasFactor 3)})
(**In** (SetOf {(**an** Integer) (**which** isPrime)}))

Möglichkeit

Format
(**during** timeSpecification **then** contingentDescription)

Gebrauch Beschreibt ein zeitabhängiges (oder nur in einer hypothetischen Welt existierendes) Objekt.

Beispiele
(**during** (**an** accident **with** date = yesterday) **then**
(**a** policeman))

3.3.4 Unit Kategorien

Die oben definierten Deskriptoren werden in Units mit folgender syntaktischer Form zusammengefaßt:

```
◁unit▷       := [◁unitname▷ UNIT ◁category▷
                            ◁selfslot▷ ◁slot▷*]
◁selfslot▷   := SELF ◁description▷
◁slot▷       := ◁slotname▷ ◁description▷
◁category▷   := Abstract | Basic | Specialization |
                Individual | Manifestation | Relation |
```

Proposition

◁`description`▷	:=	◁`descriptor`▷ | ◁`feature`▷ | {◁`description`▷+}
◁`feature`▷	:=	◁`descriptor`▷ ; ◁`facet`▷
◁`facet`▷	:=	DEFAULT | ...

Hierbei stehen die nichtterminalen Symbole ◁`unitname`▷ und ◁`slotname`▷ für direkte Zeiger im obigen Sinne. Der jeder Unit zugeordnete Kategorietyp hat folgende Bedeutung:

Basic partitioniert die Welt in disjunkte Teilmengen. Kein Individuum kann gleichzeitig in verschiedenen Basic-Kategorien sein.

Specializations sorgen für nicht notwendig disjunkte weitere Unterteilung in den Basic-Kategorien.

Abstract dient im allgemeinen dazu, Beschreibungen und Prozeduren so zusammenzufassen, daß sie an alle Units weitervererbt werden können, die durch eine Perspektive beschrieben werden, für die die abstrakte Kategorie ein Prototyp ist.

Individuals werden vom Matcher als grundsätzlich verschieden angenommen.

Manifestations dienen zur Gruppierung von Informationen, um z.B. eine ganze Menge von zeitabhängigen Beschreibungen unter einem Namen zu fassen, der dann in der oben erwähnten (**during** ... **then** ...) Form verwendet werden kann.

Relations können als abstrakte Abbildung mit Hilfe von Slots und Deskriptoren dargestellt werden.

Propositions sind Instanzen einer Relation, deren Wahrheitswert nicht durch ihre reine Existenz in einer Datenbasis, sondern durch explizite Angabe definiert ist.

3.3.5 Matching

KRL-0 enthält ein Grundgerüst zum Vergleich *(Matching)* von Deskriptoren. Es stehen einige grundlegende Strategien zur Verfügung, auf denen ein Matcher für spezielle Anwendungsgebiete erstellt werden kann. Für Deskriptoren, die Zeiger auf Individuen sind, kann der Matcher einfach entscheiden, ob sie identisch sind oder nicht. Komplexere Deskriptoren werden rekursiv in einfachere Strukturen zerlegt, die dann wieder verglichen werden. Es findet ein sogenanntes *Alignment* statt, um alle Perspektiven eines Deskriptors zu behandeln.

Die Standardalgorithmen zum Matching versuchen nur eine Reihe von einfachen Tests durchzuführen. Der Benutzer kann aber die Reihenfolge, in der diese Tests angewendet werden, frei wählen und ebenfalls eigene, spezialisierte Routinen ansprechen. So gewinnt man große Flexiblität auf Kosten von Programmieraufwand für den Benutzer.

Der Matcher von KRL-0 bietet nun eine Reihe von attraktiven Konzepten zur Verwendung innerhalb des Vergleichsprozesses an:

- Der Vergleich muß nicht nur direkt spezifizierte Eigenschaften berücksichtigen, sondern eventuell auch nur *herleitbare* Informationen.
- Der Benutzer kann für Teilaufgaben selbstdefinierte, spezialisierte Algorithmen festlegen.
- Der Vorgang des Matchens kann im Fall noch fehlender Informationen so lange aufgehalten werden, bis diese endlich zur Verfügung stehen. Das System arbeitet währenddessen an Fällen, für die ausreichende Daten vorliegen.
- Der Matcher liefert als Resultat nicht nur ein definitives *JA* oder *NEIN*. Als weitere Möglichkeiten können partielle Resultate der Form *bis jetzt noch nicht entscheidbar* auftreten. Grund dafür kann einmal eine vorher angegebene Resourcenbeschränkung für diese Teilaufgabe sein oder aber die Unentscheidbarkeit aufgrund der Unvollständigkeit der Algorithmen.
- Im obigen Fall der Resourcenbeschränkung soll der Matcher die genaue Ursache für das Scheitern geben können.

- Für bestimmte Anwendungen ist es wichtig, nicht nur ein *JA* oder *NEIN* als Antwort zu erhalten, sondern auch ein Maß für die Qualität der Übereinstimmung.
- Der Matcher kann nicht nur Fragen der Art „Paßt dies zusammen?“ beantworten, sondern auch „Welche Annahmen müßte man machen, damit dies zusammenpaßt?“

3.4 NETL

3.4.1 Übersicht

Scott Fahlmans NETL [Fahlman, 1979] ist einer der ersten und ambitioniertesten Versuche, ein mächtiges Wissensrepräsentationsystem zu entwickeln, das auf dem Prinzip des semantischen Netzwerkes basiert:

Wissen ist in seinem System konzeptionell in Form eines Verbindungsmusters zwischen sehr einfach aufgebauten Prozessorelementen gespeichert.

NETL verfolgt dabei die folgenden Ziele:

- Kompatibilität mit einer Implementation als paralleles Netzwerk. Ein wesentliches Ziel der Arbeit von Fahlman, die Möglichkeit zu einer effizienten Implementation der Beschreibungssprache NETL auf einem Netz von parallel arbeitenden Prozessoren, die sich durch Austausch von kleinen Markierungen *(Marker Bits)* verständigen, zu schaffen, spielt im Rahmen dieser Arbeit keine wesentliche Rolle und soll deshalb nur am Rande Erwähnung finden.
- Vollständigkeit: Das System sollte all das darstellen können, was auch Menschen darstellen können. Diese Forderung ist wohl so zu interpretieren, daß Fahlman versucht, *jedes* auftretende Darstellungsproblem mit NETL zu lösen und sich nicht auf eine sehr spezielle Domäne zu beschränken. Eine solche Einschränkung kann helfen, viele Probleme gar nicht erst entstehen zu lassen. Das wird bei der Entwicklung von Expertensystemen und im speziellen auch bei **DONALD** ausgenützt. Dieses Ziel wird wohl in jedem Fall unerreicht bleiben.

- Semantische Präzision. Diese erfordert zumindest eine algorithmische Spezifikation der Arbeitweise von NETL.
- Einfachheit und intuitive Klarheit. NETL benutzt nur eine kleine Anzahl verschiedener Elementtypen und Verbindungen, die dann zur Konstruktion komplexerer Dinge verwendet werden.
- Ökonomische Darstellung. Als Mittel zur redundanzfreien Speicherung von Informationen dient in NETL insbesondere die hierarchische Anordnung aller Konzepte in einer in erster Näherung baumförmigen Struktur. Explizite Darstellung von Eigenschaften der Konzepte kann so vermieden und durch die relative Position in der Hierarchie kodiert werden.

Zur konzeptionellen Klarheit des Systems trägt der Begriff der virtuellen Kopie *(Virtual Copy)* einer gespeicherten Beschreibung wesentlich bei. Eine virtuelle Kopie bedeutet dabei, daß sich die Wissensbasis genauso verhält als sei ein Teil von ihr (eventuell mit gewissen, relativ kleinen Änderungen) kopiert worden, der Kopiervorgang aber nicht die Erzeugung einer redundanten Gedächtnisstruktur beinhaltet.

So wird es möglich, Strukturen beliebiger Größe mit diese virtuellen Methode zu duplizieren, ohne einen nicht vertretbaren Aufwand an neuen Verbindungen und Prozessorelementen zu erforden. Eine komplette Struktur von internen Beziehungen kann in ihrer Gesamtheit auf einfache Weise vererbt werden.

3.4.2 Individuen

In der Literatur findet sich als eines der Standardprobleme die Darstellung von Clyde, dem Elephanten, das auch an dieser Stelle als einführendes Beispiel dienen soll.

Dazu werden vier verschiedene Basistypen von Netzwerkknoten benötigt:

- *INDV Knoten zur Darstellung eines spezifischen Individuums in einem real existierenden oder nur imaginären Universum. Dabei muß es sich aber nicht um physikalische Objekte handeln, sondern auch unter anderem um Ereignisse oder Aussagen. Individuen können auch eine bestimmte Rolle *(Role)* in einer Typ-Beschreibung (s.u.) spielen.

- *TYPE Knoten beschreiben eine ganze Menge von Individuen, die die Struktur und Eigenschaften dieses Knotens erben.
- *VC Knoten verbinden *INDV und *TYPE Knoten zum Zweck der Vererbung.
- *MAP Knoten werden für die Rollen in einer neuen virtuellen Kopie eines *TYPE Knotens erzeugt, falls explizite Informationen über das die Rolle ausfüllende Individuum vorliegen. Andernfalls bleiben sie implizit.

Nachdem im ersten Schritt ein *INDV Knoten (im folgenden mit CLYDE bezeichnet) zur Darstellung des Elephanten angelegt wurde, bleibt noch die Art dieses Individuums und sein Existenz- oder Geltungsbereich festzulegen.

Ersteres geschieht mit Hilfe eines VC Links zu einem *TYPE node. In diesem Fall handelt es sich um den Typ ELEPHANT. Als allgemeinster Typ steht THING zur Verfügung, so daß bei Fehlen genauerer Informationen ein Link zu THING genau dieses Fehlen anzeigt. Normalerweise besteht wegen der Existenz wenig spezifischer Typen wie PHYSICAL-OBJECT, SET, AREA wenigstens die Möglichkeit einer groberen Klassifikation. Die folgende Abbildung 3.1 stellt diese Situation graphisch dar.

Durch diesen VC Link wird CLYDE zu einer *Virtual Copy* von ELEPHANT und ebenfalls wird CLYDE zu einem Element der Menge aller Elephanten, dargestellt durch das Individuum ELEPHANT-SET. Jedem TYPE node x ist immer die Menge x-SET zugeordnet, der jede *Virtual Copy* des Typs angehört. Wie jedes Individuum haben auch diese Mengen ein VC Link zu einem TYPE Node, in diesem Fall zu einer Spezialisierung von SET, z.B. EMPTY-SET oder NONEMPTY-SET. Die Mitgliedschaft in diesen Mengen bleibt aber implizit durch die Existenz des VC Links; NETL stellt sie nicht explizit dar.

Da die Darstellung solcher VC Links eine separate Link Unit erfordert und damit recht aufwendig ist, jedes Individuum aber mindestens ein VC Link besitzt, steht in jedem Individuum ein spezieller *Parent-Wire* zur Verfügung. So kann die unbestreitbare Mitgliedschaft in einer Klasse effizient realisiert werden. Genereller Nachteil dieser speziellen Wires ist, daß sie nicht wie eine vollständige Link Unit selbst mit einem Geltungsbereich versehen oder sogar

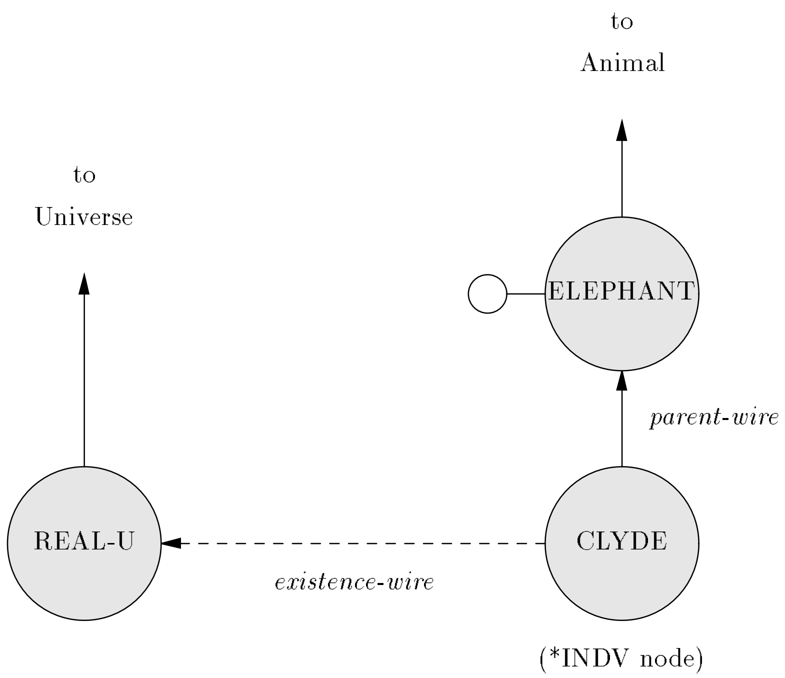


Abbildung 3.1: NETL: Der *INDV Knoten Clyde

verneint werden können. In solchen Fällen wäre die Umwandlung des Parent-Wire in eine echtes VC Link unvermeidlich.

Bleibt noch Ort und Zeit der Existenz eines Individuums zu spezifizieren, was durch ein EXIN Link geschieht. Wie oben steht zusätzlich ein spezieller *Existence-Wire* zur Verfügung. In unserem Fall könnte der *Existence-Wire* auf das real existierende Universum führen. In weniger eindeutigen Fällen oder auch z.B. zur Darstellung einer nur hypothetischen Situation muß auf ein echtes EXIN Link zurückgegriffen werden.

Alle *Existence-Wires* zeigen auf Individuen, die einem Untertyp des AREA Typs angehören. Eine AREA kann dabei eine örtliche, zeitliche oder thematische Einschränkung sein. Als Beispiel soll die Gegend Transsylvanien, das 18. Jahrhundert oder die Relativitätstheorie genügen. AREA können auch zur weiteren Einschränkung kombiniert werden (z.B. Münster um 1960). Natürlich sind fast alle AREAS Teile von anderen, allgemeineren AREAS (Münster in Westfalen, in Deutschland, in Europa, auf der Erde, im Sonnensystem). Die durch PART-OF Relationen repräsentierte Hierarchie ähnelt in gewisser

Weise der durch VC Links gebildeten Typ-Hierarchie. Hier gibt es aber keinen allumfassenden Wurzelknoten wie THING, sondern nur gegebenenfalls mehrere AREAS, die nicht mehr Teile anderer AREAS sind. Diese werden als UNIVERSES bezeichnet.

Das UNIVERSE REAL-U ist nur ein Universum, andere können zum Durchspielen hypothetischer Situationen dienen (was wäre, wenn?). Solche Universen werden als *Possible Worlds* bezeichnet. Da auch Universen Individuen sind, müssen sie ebenfalls einen *Existence-Wire* besitzen, der auf ein Universum führt. Zu diesem Zweck existiert das Metauniversum META-U, das in sich selbst existiert und in dem alle anderen Universen existieren.

Ein Individuum I in einer Area A existiert so in allen Areas, die A enthalten. Diese Kette endet erst bei Erreichen eines Universums.

Die explizite Information über die Existenz eines Individuums dient drei Zwekken. Als erstes erlaubt sie natürlich die Beantwortung direkter Fragen über Ort und Zeit. Zweitens können so alle Individuen gefunden werden, die ebenfalls in diesem Universum existieren. Solche Information kann z.B. bei der Mustererkennung, die auch in **DONALD** eine zentrale Rolle spielt, sehr nützlich sein, um von Anfang an eine sinnvolle Menge möglicherweise passender Kandidaten zu haben. Als letzer Punkt bleibt die Möglichkeit, ganze Universen virtuell zu kopieren, da die Existenz eines Individuums in der neuen virtuellen Kopie implizit durch die Existenz in dem kopierten Universum dargestellt ist. Erst bei Modifikationen in der virtuellen Kopie wird diese in eine richtige Kopie umgewandelt.

Als Option bei der Erzeugung eines *INDV Knotens ist noch die Angabe **SPLIT möglich. Zwei *INDV Knoten können mit einem *EQ Link verbunden werden. So ein Link sagt aus, daß die beiden Knoten dasselbe externe Objekt bezeichnen. Ist aber **SPLIT gesetzt, so kann der Knoten nicht mehr mit einem anderen Knoten derselben Area gleichgesetzt werden, dessen **SPLIT Option ebenfalls gesetzt ist.

3.4.3 Typen

Neue Typen werden immer als Spezialisierung *(Subtype)* eines bereits bestehenden Typen erzeugt. Ausnahme ist natürlich der Typ THING, der, wie oben erwähnt, kein Untertyp mehr ist, sondern an der Spitze der Hierarchie steht. THING ist aber bereits in NETL fest eingebaut.

Nach der wesentlichen Entscheidung für den bereits bestehenden Elterntyp S schreitet die Erzeugung des neuen Typs T als Untertyp von S folgendermaßen fort, wobei wieder der Elephant Clyde als Bespiel dienen soll:

- Erzeugung des *INDV Knotens T-SET.
- Verbinden des *Parent-Wire* von T-SET mit dem Typ SET.
- Verbinden des *Existence-Wire* mit REAL-U.

Falls die AREA genauer spezifiert werden soll, ist sie auf jeden Fall so zu wählen, daß sie alle AREAs der Individuen des neuen Typs umfaßt.

Da nun T-SET existiert, kann ein neuer TYPE Knoten für T selbst kreiert werden. Dessen *Parent-Wire* läuft von T nach S (in unserem Bespiel von ELEPHANT zu MAMMAL), sein *Set-Wire* zu T-SET. T hat **keinen** *Existence-Wire*, da die Typbeschreibung eigentlich nicht als wirkliches Individuum existiert, sondern sich mehr wie ein Universum verhält.

Die Namen ELEPHANT oder ELEPHANT-SET werden also nicht dargestellt. Um einem Objekt einen Namen zu geben, ist es erforderlich, einen separaten Knoten des Typs WORD zu erzeugen und diesen dann mit einem *Proper-Name* Link mit ELEPHANT zu verbinden.

Das Ergebnis dieser Operationen für das Beispiel T = ELEPHANT und S = MAMMAL (d.h. ELEPHANT wird als neuer Untertyp von MAMMAL definiert) ist in Abbildung 3.2 graphisch dargestellt.

3.4.4 Rollen

Ein *TYPE node dient als Kern einer Beschreibung, die sich aber über eine beliebige große Portion des semantischen Netzwerkes erstrecken kann. Eine Rolle *(Role)* ist nun einfach ein *INDV Knoten, der entweder ein einzelnes Objekt oder eine Menge darstellt, dessen *Existence Wire* mit einem *TYPE

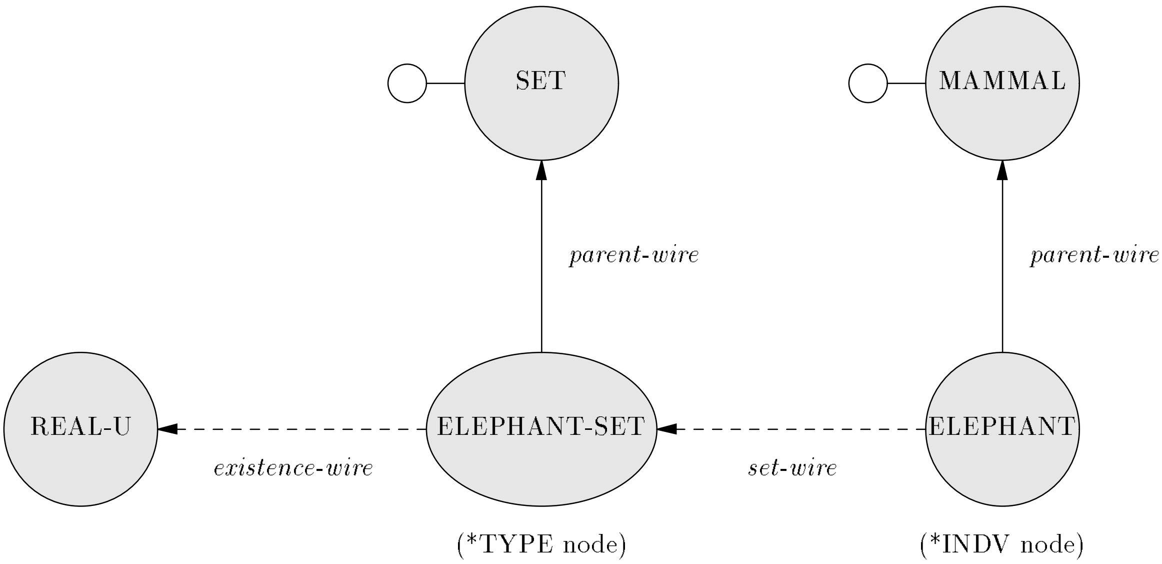


Abbildung 3.2: NETL: der *TYPE Knoten Elephant

Knoten an Stelle einer bestimmtem Area oder eines Universums verbunden ist. Dieser *TYPE node wird als Eigentümer *(Owner)* einer Rolle bezeichnet. Ein Rollenknoten ist eine vollständig eigenständige Einheit mit individuellen Eigenschaften und Verbindungen zu anderen nodes. Ist eine Rolle Abkömmling des SET types, ist die Situation noch etwas komplexer, da ihr dann noch ein eigener *TYPE Knoten zugeordnet ist, der einen typischen Vertreter dieser Menge beschreibt; dieser *TYPE kann wieder eine weitere Menge von Rollen enthalten, usw.

Individuelle Rollen sind entweder vom Typ IN oder OF. IN impliziert dabei, daß der Owner Knoten als AREA im weiteren Sinne gesehen werden kann, innerhalb dessen das Individuum existiert. Sowohl IN als auch OF Rollen stellen eine eindeutige Korrespondenz zwischen Rolle und Owner her. Eine OF Rolle stellt also einfach eine schwächere Version einer IN Rolle dar. Für die Funktion des Systems spielt diese Unterscheidung keine große Rolle, sie soll nur den sprachlichen Unterschied zwischen IN und OF wiederspiegeln.

Zur Darstellung einer IN Rolle wird also einfach ein *INDV Knoten erzeugt und mit einem *EXIN Link eine Verbindung von dem Rollenknoten zu dem *TYPE Knoten des Owners hergestellt. Das ist eigentlich genau das, was zur Darstellung der Area eines Individuums entwickelt wurde, mit dem einzigen Unterschied, daß die Area hier kein *INDV sondern ein *TYPE Knoten ist.

Eine OF Rolle wird statt durch ein *EXIN durch ein *EXFOR Link dargestellt. Dabei bedeutet ein *EXIN Link von *INDV A nach *TYPE B, daß für alle Knoten vom Typ B ein Abbild *(Map)* des Individuums A existiert.

Diese Rollen können natürlich auch entlang eines *VC Links vererbt werden. Solange Informationen über diese Rollen nur abgerufen werden, ist es nicht erforderlich, neue Knoten für diese Rollen zu erzeugen. Die Lage ändert sich, wenn man neue Information über die Rollen ablegen will. Dann wird für die entsprechenden Rollen ein *MAP Knoten erzeugt, dessen *Map-Wire* zu der abgebildeten Rolle zeigt und dessen *Owner-Wire* zu dem Owner der Rolle zeigt.

Auch wenn eine Rolle von einer *TYPE Beschreibung zur anderen „gemapped" wird, erfüllt sie weiterhin ihre Aufgabe als Rolle. Diese möglicherweise recht lange Kette endet erst bei einem frei für sich stehenden Individuum.

3.4.5 Aussagen

Außer den bereits vorgestellten Knotentypen existiert auch ein Konzept zur Darstellung deklarativer Information. Diese Aussagen *(Statements)* werden entweder mit dem Handle-Knoten der Linkelemente oder mit Hilfe von speziellen *IST *(Individual Statement)* Knoten dargestellt.

Ähnlich der Situation bei VC Links sind *IST Knoten nur für die komplizierteren Fälle erforderlich. Hier kann dann ein Statement wieder als eigenständiges Objekt mit Eigenschaften und Klassenzugehörigkeiten gesehen werden. Es ist z.B. sinnvoll, festzuhalten, ob eine Aussage nur ein *Default* ist, oder ob es sich um eine unbestreitbare Tatsache handelt, die niemals in Frage gestellt werden darf. Solche Information kann bei der Auflösung von Widersprüchen im gespeicherten Wissen nützlich sein. Eine andere Anwendung stellt neben der Verläßlichkeit die Herkunft einer Information dar.[6]

Eine anderer Grund für ein eigenständiges Objekt könnte die Implementation eines sogenannten *Truth Maintenance Systems (TMS)* sein. Doyle beschreibt

[6] Auch **DONALD** hält zu jeder Information diese Eigenschaft im `origin` Strukturfeld jedes Slots fest und verwendet sie vielfach (siehe Abschnitt 4.5.6 auf Seite 132).

in [Doyle, 1979a], [Doyle, 1979b] und seinem späteren *Reason Maintenance System (RMS)* [Doyle, 1983] Systeme, die die explizite Darstellung der Herkunft *(source)* eines Statements und der Menge der unterstützenden Aussagen *(Set of Support)* erfordern.

Jedes Statement ist Instanz eines Statement-Typs, ebenso wie jedes Individuum die Instanz eines Objekt-Types ist. Dazu wird wiederum ein VC Link benutzt. Statement-Typen sind Abkömmlinge des *TYPE Knotens STATEMENT.

Individuelle Statements sind innerhalb einer Area gültig, die als *Scope* des Statements bezeichnet wird. Falls der Scope eines Statements aktiv ist, so nimmt das Statement an allen Operationen des Systems teil, ansonsten spielt es keinerlei Rolle und wird für den Moment als nicht existent betrachtet. Die Verbindung zwischen Statements und Area geschieht durch ein oder mehrere *SCOPE Links. Ein Statement ist dabei aktiv, falls mindestens eine seiner Areas betrachtet wird. Zur Vereinfachung existiert auch ein spezieller *Scope-Wire* für die einfachen Fälle.

So ergibt sich in Abbildung 3.3 eine detailliertere Sicht der Abbildung 3.1 von Seite 97.

NETL behandelt noch weitere, komplizierte Probleme der Wissensrepräsentation mit semantischen Netzwerken:

Definierte Klassen. Hier geht es um den Unterschied zwischen der *Definition* einer Klasse und der schwächeren *Beschreibung* der typischen Vertreter einer Menge. NETL stellt dafür den *EVERY Knoten zur Verfügung, der einem *TYPE gleicht, bei dem einige Eigenschaften und Aussagen speziell als *definierend* gekennzeichnet sind.

Negation. Aussagen können einfach negiert werden, in dem die **NOT Option gesetzt wird. Für NETL bedeutet das, daß die Aussage normalerweise ignoriert wird. Werden neue Aussagen in ein semantisches Netz eingefügt, so müssen alle negierten Aussagen auf eventuelle Widersprüche untersucht werden (NETL erzeugt dann einen *Clash*).

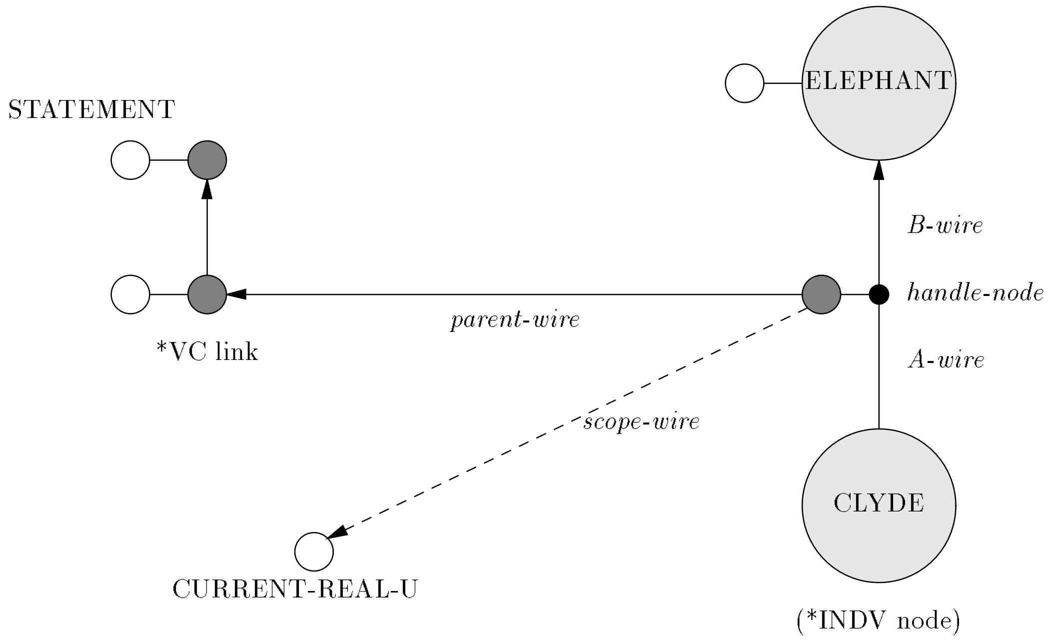


Abbildung 3.3: NETL: Der *INDV Knoten Clyde im Detail

Ereignisse, Aktionen und Kontexte. Ereignisse werden als Ursache für Übergänge zwischen einem Zustand der Welt und einem neuen Kontext gesehen. Fahlman [Fahlman, 1979] entwickelt zu diesem Zweck Mechanismen, die aber nur auf einer relativ kleinen Menge von einfachen Testfällen sinnvolle Resultate liefern.

3.5 Bewertung

3.5.1 FRL

Nachteile dieser recht einfachen Konzeption von FRL ergaben sich bei einem ersten Versuch, mit diesem Ansatz Wissen darzustellen:

- Oft war unklar, welche der Funktionen der `fget-?` Familie richtig und angebracht ist. Sie unterscheiden sich ja sowohl im Berechnungsaufwand, als auch in der berechneten Information. Da Zugriffe sich aber auch über die Demons z.B. in direkte Fragen an den Benutzer auswirken können, wird sogar das sichtbare Verhalten des Systems von der Wahl der Funktion beeinflußt.

- FRL definiert nur den Zugriff auf einzelne Facetten von Slots. Es werden keine Möglichkeiten zur Manipulation kompletter Frames geboten. Es kann z.B. kein Maß für die Ähnlichkeit zweier Frames angegeben werden.

- Um unvorhersehbare Effekte zu vermeiden, müssen die Slotnamen in der gesamten AKO-Hierarchie eindeutig sein.

- Es stellten sich Effizienzprobleme ein, genaueres dazu folgt noch in Abschnitt 4.8.1.

3.5.2 KRL

Die in [Bobrow and Winograd, 1977] vorgestellten Ideen sind beeindruckend. Das Projekt muß aber im ganzen als gescheitert angesehen werden. Es gab nach KRL-0 keine weiteren neuen Implementationsaktivitäten. Nach ca. zwei Monaten intensiver Beschäftigung mit der Implementation eines kleinen Subset von KRL in LISP im Rahmen von [Heuer, 1987] wurde ein Grund dafür offensichtlich. Es war zwar möglich, die Funktionen zum Matchen zweier Frames soweit zu entwickeln, daß sie mit einfachen Beispielen durchaus zufriedenstellend und korrekt arbeiteten, in komplizierten Fällen war aber nicht mehr klar, was überhaupt in diesem Fall die richtige Antwort sein müßte und natürlich noch weniger, wie das zu implemtieren wäre. Mit KRL läßt sich zwar sehr elegant Wissen deklarieren, aber es fehlt eine genaue Spezifikation der Semantik.

Deshalb mußten für den Prototyp einige Einschränkungen gefordert werden, die es ermöglichen, überhaupt einen solchen zu erstellen. Andererseits enthält das in Kapitel 4 beschriebene System **DONALD** eine ganze Reihe von Features, die direkt auf KRL zurückgehen. Dazu gehören in erster Linie die Views, die Möglichkeit Frames innerhalb von Frames anordnen zu können, sowie als wichtigste Operation auf Frames das Matchen zweier Deskriptoren.

3.5.3 NETL

NETL stellt eigentlich alles an Ausdrucksmöglichkeiten zur Verfügung, was unabdingbar erforderlich ist. Auch aufwendigere Fragestellungen wie Negation, Ereignisse und zeitliche Abläufe werden berücksichtigt. Wie nicht anders zu erwarten, können solche Probleme allerdings nicht mehr vollständig mit der von Fahlman [Fahlman, 1979] angestrebten hochgradigen Parallelität gelöst werden. Er greift dann zu schnellen *marker-passing* Algorithmen, die in einfachen Fällen schnelle Resultate liefern. Ansonsten müssen serielle Methoden eingesetzt werden, die dann eventuell sehr viel langsamer sind, und nur dann angesprochen werden, wenn genügend Zeit vorhanden ist und das System im Moment keine wichtigen anderen Aufgaben hat.

NETL verwendet auch eine Methode zur *heuristischen Klassifikation*, die in ähnlicher Weise in **DONALD** Verwendung findet. Das Erkennungssystem darf *TYPE nodes untersuchen und *vorschlagen*, in welche Typmenge ein neues Individuum am besten paßt. Dies funktioniert durch eine Art gewichtetes Mittel über alle verfügbaren Eigenschaften, es gibt dafür keine formale Definition.

Das schwerwiegendste Problem ist neben der Implementation von NETL auf einem Simulator, der die parallelen Prozessorelemente nachbildet, die Tatsache, daß die Bedeutung eines NETL-Netzes von den spezifizierten Algorithmen abhängt. Es gibt keine *formale* Definition von richtig und falsch. Deshalb entstand in neuerer Zeit, beginnend mit der Arbeit *The Mathematics of Inheritance Systems* von Touretzky [Touretzky, 1986] die Tendenz, die Theorie der semantischen Netzwerke zu formalisieren, jedoch nur unter Verzicht auf einen großen Teil der in NETL vorhandenen Komplexität.

Die Arbeit *Inheritance Theory and Networks with Roles* [Thomason and Touretzky, 1991] ist ein erster Ansatz zur Formalisierung monotoner Vererbungssysteme *ohne* die Existenz der oben beschriebenen *Rollen* zu vernachlässigen. Ebenso ist hier die Spezifikation von Gleichheit zwischen Knoten möglich.

Einerseits ist die resultierende Theorie immer noch weniger aussagekräftig als der Prädikatenkalkül erster Ordnung; wie alle Vererbungssysteme stellt sie eine Art schwache Logik dar, die auf der Konstruktion von Pfaden durch Graphen basiert. Die Inferenzregeln und Beweise sind aber andererseits wesentlich komplexer als in einfachen IS-A Hierarchien.

Es resultiert ein Algorithmus zur Herleitung von Antworten für Netzwerke mit Rollen, der dem Parsen eines Fragments des Netzwerkes mit einer kontextfreien Grammatik entspricht. Die nichtterminalen Symbole dieser Grammatik sind dabei die Inferenzregeln und die terminalen die Links des Netzwerks. Beweise in diesem Formalismus sind immer noch pfadbasiert, haben aber die Form eines Baumes und lassen sich nicht mehr als Ketten darstellen. Genaueres zu den hier verwendeten Begriffen findet sich im fünften Kapitel über Vererbungshierarchien.

Kapitel 4

Die Expertensystem-Shell DONALD-III

4.1 Einführung

In diesem Kapitel wird die Expertensystem-Shell **DONALD** dargestellt. Nach der Einführung in die Klassifikation von Problemen und der Diskussion möglicher Lösungsparadigmen für diese Klassen werden auch technische Details angesprochen und Begründungen für die getroffenen Designentscheidungen gegeben.

Betrachtet man die zum Thema Wissensrepräsentation verfügbare Literatur, so fällt auf, daß es außer der reichhaltigen Auswahl an Arbeiten und Aufsätzen mit theoretischen Beiträgen eigentlich nur Darstellungen des Einsatzes von existierenden Systemen gibt. Die von Universitäten entwickelten Systeme sind zum Teil recht ausführlich beschrieben, dafür aber leider oft recht veraltet. Als Beispiel wäre hier MYCIN zu nennen, daß in [Jackson, 1990] recht ausführlich beschrieben wird. Die Originalarbeit findet sich in [Shortliffe, 1976].

Unter dieser Voraussetzung schien es sinnvoll, die bei der Konstruktion eines Expertensystems auftretenden Probleme und ihre Lösung nicht nur in der Literatur aufzuspüren, sondern sie vielmehr auch auf experimentelle Weise zu erforschen. Das erforderte einerseits die Wahl eines Studiengebietes und andererseits die Erstellung einer Expertensystem-Shell, in der das Wissen aus diesem Gebiet repräsentierbar ist.

Der Aufbau einer Wissensbasis erfordert intensive Interviews und Diskussionen mit den Experten, deren Wissen im Computer abgebildet werden soll. Nach anfänglichen Versuchen im Verlauf einer Diplomarbeit, diesen Problembereich dadurch zu entschärfen, daß als Anwendungsgebiet ein Thema gewählt wird, für das man sich selbst als Experte fühlte, wurde doch offensichtlich, daß so eine ganz wichtiger Aspekt der Wissensrepräsentation ausgespart wurde.

Im Rahmen des **DONALD**-Projektes konnte jedoch ein Anwendungsgebiet gefunden werden, daß die obigen Anforderungen in fast idealer Weise erfüllt. In Zusammenarbeit mit zwei Experten wurde auf der Basis der Expertensystem-Shell **DONALD** ein System names **INVEST** zur Wertpapieranlage erstellt. Diese Beratungsaufgabe hat einerseits diagnostische Aspekte, in denen die Situation des Anlegers eingeschätzt werden muß, andererseits synthetisiert das System auch ganz konkrete Anlagevorschläge, die unter globalen Zielvorstellungen *(Constraints)*, wie niedriges Risiko oder hohe Rendite, zu optimieren sind.

Ein System, das in der Lage ist, eine solch komplexe Aufgabe zu lösen, umfaßt nicht nur eine Wissensrepräsentationskomponente, sondern auch eine Problemlösungskomponente. Deshalb werden in den folgenden Absätzen Wissensbasierte Systeme klassifiziert und mögliche Ansätze zur Problemlösung beschrieben und bewertet.

4.2 Strategien zur Problemlösung

[Buchanan *et al.*, 1983] legt das Vorgehen in folgenden Schritten nahe (siehe Abbildung 4.1):

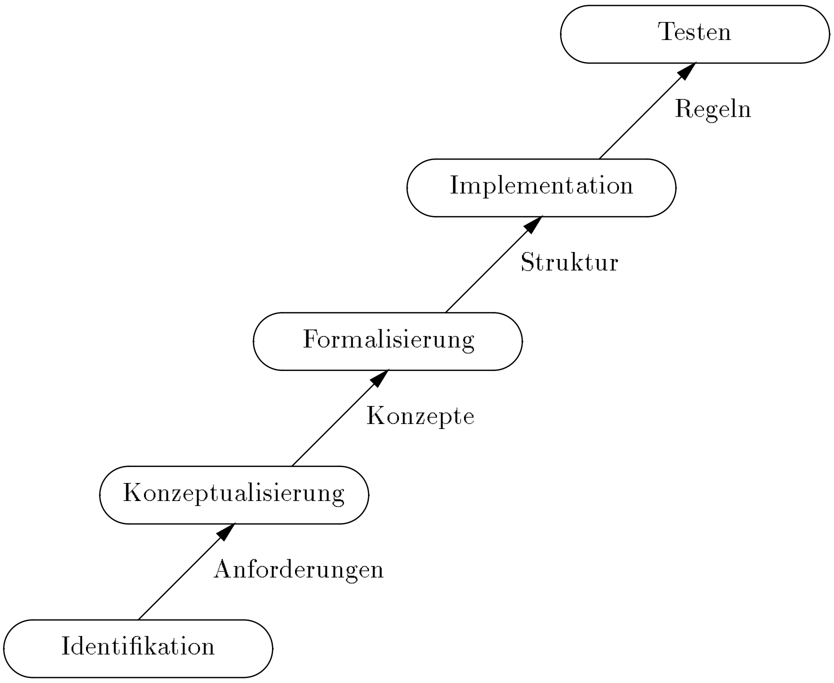


Abbildung 4.1: Stufen bei dem Problemlösungsprozeß

Identifikation: Identifizierung der Klasse von Problemen, die das System lösen muß. Dazu kommen die Daten, mit denen das System arbeiten wird, sowie geforderte Eigenschaften der Lösungen. Dem stehen die zur Verfügung stehende Expertenwissen und die Ausstattung mit Computern und Geld gegenüber.

Konzeptualisierung: Die Suche nach Schlüsselkonzepten und Relationen zwischen diesen. Das umfaßt die Charakterisierung der verschiedenen Daten, den Informationsfluß und die zugrundeliegende Struktur des Arbeitsgebiets im Hinblick auf kausale, zeitliche, räumliche und Teil-Ganzes Abhängigkeiten und Relationen.

Formalisierung: Das Verstehen der relevanten Eigenschaften des zugrundeliegenden Suchraumes und welche Anforderungen an den Suchalgorithmus dadurch impliziert werden. Von besonderer Bedeutung sind hier die Zuverlässigkeit, Vollständigkeit und Konsistenz der zur Verfügung stehenden Informationen aus eventuell unterschiedlichen gegebenen Quellen.

Implementation: Aus der Formalisierung des Wissens wird nun ein ablauffähiges Programm durch genaue Spezifikation von Kontrollstrukturen und Informationsfluß. Entscheidungen über Datenstrukturen und den Grad der Unabhängigkeit zwischen den einzelnen Modulen des Systems müssen getroffen werden.

Testen: Die Evaluation eines Wissensbasierten Systems ist weit davon entfernt, eine exakte Wissenschaft zu sein. Klar scheint aber der Wert einer großen und repräsentativen Auswahl an Testfällen zu sein. So können insbesondere Fehler durch die ungewollte Interaktion einzelner Wissenspartikel[1] schnell erkannt werden.

4.2.1 Heuristische Klassifikation

Nach [Hayes-Roth *et al.*, 1983] können Expertensysteme folgendermaßen klassifiziert werden:

- *Interpretationssysteme* leiten Situationsbeschreibungen aus Beobachtungen oder Sensordaten ab.
- *Vorhersagesysteme* leiten wahrscheinliche Konsequenzen aus Situationen oder Ereignissen ab.
- *Diagnosesysteme* leiten Fehlerdiagnosen aus beobachteten Symptomen ab.
- *Designsysteme* entwickeln Konfigurationen von Objekten, die bestimmte Bedingungen erfüllen.
- *Planungssysteme* generieren Folgen von Aktionen, die vorgegebene Ziele erreichen.
- *Monitorsysteme* betrachten das Verhalten eines anderen Systems im Zeitverlauf zur Verhinderung von Abweichungen gegenüber Zielvorgaben.

[1] Im allgemeinen handelt es sich hierbei um Produktionsregeln, wohingegen das Wissen in **DONALD** überwiegend in anderer Form gespeichert wird.

- *Debuggingsysteme* finden Abhilfe bei Systemfehlern.
- *Reparatursysteme* finden und administrieren Abhilfe bei Systemfehlern.
- *Instruktionssysteme* erkennen und behandeln Mißverständnisse von Studenten.
- *Kontrollsysteme* leiten das Verhalten eines Systems durch Erkennen von Problemen, Planen von Lösungen und Überwachen der nötigen Aktionen.

Diese Klassifikation weist einige Überschneidungen zwischen den einzelnen Kategorien auf. Planen kann z.B. als Spezialfall von Design aufgefaßt werden (Design von Aktionen).

Clancey [Clancey, 1985] bietet als alternative Analyse die Betrachtung der generischen Operationen auf einem System. *Synthetische* Operationen zur Konstruktion und *analytische* Operationen zur Interpretation stehen sich dabei gegenüber.

Mit dieser Methode erhält man die in den Abbildungen 4.2 und 4.3 dargestellte Einteilung in Klassen.

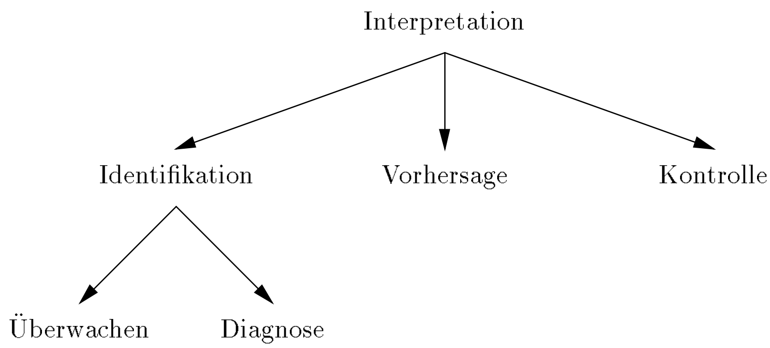


Abbildung 4.2: Generische Operationen zur Analyse eines Systems

Essentielle Charakteristik von Klassifikationsaufgaben ist die Tatsache, daß der Experte eine Kategorie aus einer festen und bereits bekannten Menge von möglichen Lösungen *selektiert*. In einfachen Fällen reichen schon offensichtliche Eigenschaften eines Objekts zu seiner Klassifikation aus; in komplexeren Fällen ist es erforderlich auf *heuristische Klassifikation* auszuweichen. Hier wird in drei Schritten vorgegangen (siehe Abbildung 4.4):

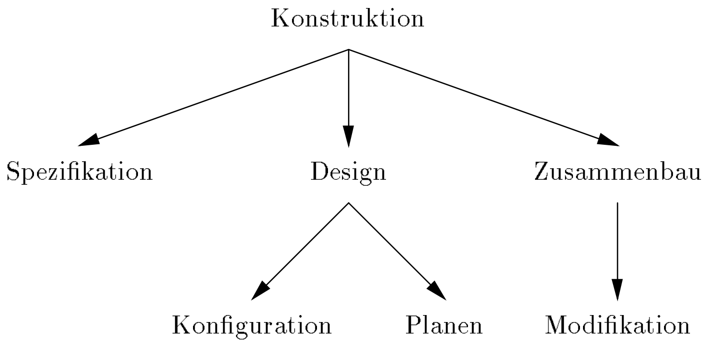


Abbildung 4.3: Generische Operationen zur Synthese eines Systems

Abbildung 4.4: Heuristische Klassifikation

- *Datenabstraktion.* Es wird von den Daten eines speziellen Falls abstrahiert. Sensordaten werden z.B. von Zahlen in Kategorien wie *höher als normal* transformiert.

- *Heuristischer Vergleich.* Es ist oft wesentlich einfacher, eine Verbindung zwischen den im ersten Schritt vorgenommenen Datenabstraktionen und den möglichen Klassen zu finden, als direkt die Rohdaten zu klassifizieren. Daten *triggern* so Hypothesen auf einem relativ abstrakten Niveau. Der Vergleich wird als *heuristisch bezeichnet*, da die Daten nicht eins zu eins auf die Hypothesen abgebildet werden können.

- *Verfeinerung der Lösung.* Nachdem eine Lösungsabstraktion gefunden wurde, und damit der Suchraum für Lösungen schon stark eingeschränkt ist, ist es noch erforderlich, mögliche Kandidaten im Lösungsraum zu identifizieren und zu bewerten. So können weitere Daten benötigt werden, die dann eventuell wieder kategorisiert werden müssen.

Das oben erwähnte MYCIN paßt sich in dieses Modell folgendermaßen ein: Der Vergleich zwischen Daten- und Lösungsabstraktionen wird durch Produktionsregeln vorgenommen, und die heuristische Natur dieses Vergleiches spiegelt sich in den Konfidenzfaktoren *(Certainty Factors)* der Regeln wieder.

Zusammmenfassend kann gesagt werden, daß das entscheidende Merkmal der heuristischen Klassifikation der von vorneherein aufzählbare Lösungsraum ist. In vielen Fällen müssen dagegen Lösungen *konstruiert* und nicht nur gesucht werden. In der Praxis besteht eine Lösung oft auch aus vielen relativ unabhängigen Teilaspekten, so daß beim Aufzählen aller möglichen Lösungen alle Kombinationen dieser Teillösungen auftreten. Es macht dann nicht viel Sinn, diesen enormen Suchraum von Kombinationen zu betrachten.

4.2.2 Hierarchisches Hypothetisieren und Testen

Hierarchisches Hypothetisieren und Testen kann als Kombination von Aspekten der heuristischen Klassifikation und des konstruktiven Problemlösens verstanden werden. Klassisches Anwendungsgebiet ist die medizinische Differentialdiagnose, die zu einem beobachteten Satz von Symptomen eines Patienten nicht nur eine einzige Ursache findet, sondern eventuell auch eine Kombination von Krankheiten. In diesem Fall trifft zwar das allgemeine Schema von Abbildung 4.4 wieder zu, es sind aber noch mehrere Teillösungen zu einer Gesamthypothese zur Erklärung aller vorhandenen Daten zusammenzufassen.

Diese zusammengesetzten Hypothesen komplizieren die Sache erheblich und erfordern die Strukturierung des Suchraums. Das hierarchische Hypothetisieren und Testen löst dieses Problem durch eine explizite taxonomische Darstellung des Raumes der Hypothesen. Im allgemeinen ensteht dadurch ein Baum von Hypothesenknoten, dessen Blätter die Lösungselemente sind. Auch die generelle Kontrollstruktur des Systems benutzt dann diese Taxonomie als Hilfsmittel zur Organisaton des Problemlösungsprozeßes.

Der verwendete Algorithmus kann folgendermaßen charakterisiert werden:

1. Einlesen der Anfangsdaten.

2. Für jede in Betracht[2] kommende Hypothese wird bestimmt, zu welcher Menge an Daten sie paßt.

3. Bestimme den Knoten n, der die beste Anpassung liefert.

4. Wenn n bereits ein Blatt ist, dann ist es die Lösung und der Algorithmus ist beendet. Ansonsten partitioniere den Hypothesenraum in zwei Mengen K und L, so daß K die Nachfolger von n enthält und L die übrigen Konkurrenten von n.

5. Sammle weitere Daten zur Unterscheidung der Hypothesen aus K, und bringe sie in eine Rangfolge.

6. Sei k der beste Knoten in K und l der beste Knoten in L.

7. Falls k besser als l ist, dann setze $n = k$ ansonsten setze $n = l$.

8. Gehe zu Punkt 4.

4.2.3 Konstruktives Problemlösen

Entscheidendes Merkmal des konstruktiven Problemlösens ist die Schwierigkeit, daß die Lösungen aus primitiveren Komponenten zusammensetzt und nicht nur aus einer im voraus bekannten Menge von Lösungen ausgesucht werden müssen.

Typischerweise geht es bei konstruktiven Problemen um Planen, Design und spezielle Diagnoseanwendungen. Speziell beim Planen sind die Lösungskomponenten Aktionen und die Lösungen selbst sind Folgen von Aktionen, die dann ein gefordertes Ziel erreichen. Da es zu viele mögliche Anordnungen von Aktionen gibt, ist es unmöglich, den Lösungsraum zu fixieren. Naive Ansätze zur Lösung von Planungsaufgaben führen sofort auf Algorithmen mit exponentieller Komplexität.

Da das Problem der Wertpapieranlage zu den konstruktiven Problemen gehört, stellt sich als nächstes die Frage, ob das vorhandene Spezialwissen ausreicht,

[2] Im allgemeinen schlagen schon die eingebenen Daten eine gewisse Menge von Starthypothesen vor. Zu diesem Zweck enthalten die Datenspezifikationen sogenannte *Trigger*, die auch in **DONALD** verwendet werden (siehe Abschnitt 4.6.1 auf Seite 136).

um zu jedem Zeitpunkt zu entscheiden, was das *richtige* weitere Vorgehen ist. Die Antwort darauf hängt nun von der Darstellung und dem Abstraktionsgrad der verwendeten Hypothesen ab. In **INVEST** ist es gelungen, eine Repräsentation zu finden, die ein Backtracking im Verlauf des Problemlösens auf der höchsten Abstraktionsstufe nicht erforderlich macht. Anders sieht es bei der Behandlung von Meinungsänderungen des Kunden aus. Hier ist durch explizite Darstellung aller Abhängigkeitsverhältnisse und eine Art Checkpoint-Methode durchaus eine nichtmonotone Modifikation des Planungszustandes möglich. Der Verzicht auf Backtracking erfordert aber die gleichzeitige Verfolgung unterschiedlicher Lösungsansätze.

Für Systeme, bei denen ein solches Backtracking nicht erforderlich ist, bietet sich das sogenannte „Vorschlagen und Anwenden“ *(Propose and Apply)* an, bei dem das Hauptaugenmerk auf der Auswahl des richtigen Operators[3] zur Erweiterung partieller Lösungen liegt.

Als Beispiel für ein solches System kann R1 dienen [McDermott, 1982b], das in Zusammenarbeit zwischen der Carnegie-Mellon Universität und der Digital Equipment Corporation zur Konfiguration von Computern der VAX-Reihe entwickelt wurde. R1 basiert auf der Verwendung von Produktionsregeln und wurde in OPS5 [Brownston *et al.*, 1985] implementiert. R1 wird noch durch XSEL [McDermott, 1982c] um eine Komponente zur Beratung bei dem Problem der Auswahl des richtigen Computersystems für einen Kunden erweitert.

Die Methode *Propose-and-apply* besteht aus folgenden Schritten:

1. *Initialize-Goal.* Ein Kontrollelement für die momentane Aufgabe *(task)* wird erzeugt und alle überflüssigen Kontrollelemente für bereits erledigte Aufgaben werden entfernt.

2. *Propose-Operator.* Operatoren schlagen plausible Vorgehensweisen vor.

3. *Prune-Operator.* Operatoren werden grob nach globalen Kriterien, wie z.B. vordefinierter Preferenzreihenfolge, bewertet und dann eventuell nicht mehr weiter betrachtet.

[3] Zum Begriff des Operators vergleiche die kurzen Anmerkungen zum GPS von Newell und Simon im ersten Kapitel auf Seite 5ff.

4. *Eliminate-Operator.* Durch paarweisen Vergleich der verbliebenen, konkurrierenden Operatoren wird ein Operator favorisiert, alle anderen werden eliminiert.

5. *Select-One-Operator.* Durch Untersuchung der Resultate der Schritte 2 bis 4 wird genau ein Operator ausgewählt.

6. *Apply-Operator.* Durch Anwendung des vorher bestimmten Operators auf den aktuellen Problemstand wird eine partielle Konfiguration erweitert.

7. *Evaluate Goal.* Falls das globale Ziel mit dieser neuen Konfiguration erreicht wurde, wird der Zyklus abgebrochen, ansonsten wird weiter iteriert.

Wie im folgenden erläutert, wird in **DONALD** das obige Problemlösungsparadigma verwendet.

4.3 Motivation des Propose-and-Apply Ansatzes

Schon nach relativ kurzer Zeit zeigte sich in den Gesprächen mit den Experten der Volksbank Münster und der Westdeutschen Genossenschaftszentralbank (WGZ), daß die Extraktion und Formalisierung des umfangreichen Wissens- und Erfahrungsschatzes dieser Experten eine sehr interessante Aufgabe werden würde.

Es trat weiterhin ein auch in der Expertensystemliteratur beschriebenes Problem auf, nämlich, daß das Expertenwissen nicht systematisch strukturiert vorliegt, sondern wie kompiliert erscheint, d.h. kollabiert und reduziert auf eine minimale und effiziente Form, die es schwieriger macht, es zu extrahieren.[4]

Waterman schreibt an anderer Stelle [Waterman, 1986, S. 153]:

[4] An expert's knowledge seems compiled, that is, collapsed and reduced to a minimal and efficient form, which makes it more difficult to extract [Waterman, 1986, S. 155].

> He [the expert] makes complex judgements rapidly, without laboriously reexamining and restating each step in his reasoning process. The pieces of basic knowledge are assumed and are combined so quickly that it is difficult for him to describe the process.[5]

Experten kommen zu richtigen Lösungen und haben um so mehr Mühe, den Lösungsweg zu beschreiben, je kompetenter sie sind. Das ist, was Johnson die Paradoxie des Expertenwissens nennt.[6]

In Anlehnung an klassische Ansätze wie in den oben erwähnten Systemen MYCIN und R1 war zuerst die Verwendung von Produktionsregeln geplant. Der Versuch, aus der von den Experten zu der Lösung gelieferten Argumentation Regeln zu bilden, mißlang aber fast durchweg. Regelformulierungen wurden von den Experten verworfen, weil sie zu allgemein waren und deshalb zu viele Ausnahmen hatten.[7] Andererseits machen zu spezielle Regeln die Wissensbasis unnatürlich und schwer verständlich.

Vor allem aber ist es so, daß Experten beim Problemlösen solche "wenn-dann"-Regeln anscheinend nicht bewußt anwenden. Vielmehr sieht es so aus, als assoziierten sie bei der Schilderung eines Problems bestimmte in Frage kommende Musterfälle. Ein solcher Verdacht erzeugt dann konkrete Fragen, deren Beantwortung weitere speziellere Assoziationen auslöst.

Zwei Ideen bestimmten nun die weitere Vorgehensweise:

1. Der Aufbau der Wissensbasis wird um so leichter, je ähnlicher die Verarbeitung des Wissens im Computer den wirklichen Denkprozessen ist.
2. Effizienz und Kompetenz des Expertensystems sind um so größer, je ähnlicher das Computermodell die wirkliche Problemlösungsstrategie simuliert.

[5] Er (der Experte) kommt rasch zu komplexen Einschätzungen, ohne mühsam jeden Schritt seines Denkprozesses wieder und wieder zu überprüfen. Die Bestandteile des Grundwissens werden so schnell vermutet und kombiniert, daß es für ihn schwierig ist, diesen Vorgang zu beschreiben.

[6] the paradox of expertise [Johnson, 1983], [Waterman, 1986, S. 154].

[7] Diese Problem erwähnt auch Marvin Minsky: „What I'm getting at is that there is a problem with exceptions. It is very hard to find things that are always true.“

Verständlicherweise wurde nicht länger versucht, das Expertenwissen in Regelform zu bringen. Stattdessen wurden die vermuteten Musterfälle (fortan *Prototypen* genannt) identifiziert. Die Bearbeitung solcher Prototypen bildet nun ein Kernstück der Wissensverarbeitung in **DONALD**.

Die in **DONALD** eingesetzte Lösungsstrategie ist das Resultat des Bemühens, eine möglichst natürliche Abbildung des Fachwissens, der Erfahrungen und der Strategien der Experten zu schaffen. Sie sieht in Anlehnung an die oben bereits beschriebenen Algorithmen wie folgt aus:

1. **DONALD** fordert (durch Fragen) eine Problembeschreibung vom Benutzer.

2. Jede erhaltene Antwort kann einen oder mehrere Verdachtsmomente (fortan *Hypothesen* genannt) „wecken".

3. Jede Hypothese ist mit einem oder mehreren Prototypen verknüpft. Die Hypothesen werden geprüft, indem die Problembeschreibung mit den Prototypen verglichen wird. Eine Hypothese wird als bestätigt angesehen, wenn der oder die assoziierten Prototypen genügend Ähnlichkeit mit der Problembeschreibung aufweisen.

4. Die bestätigten Hypothesen bestimmen, welche weiteren Fragen gestellt werden müssen. Darüberhinaus liefern sie die Mosaiksteine, aus denen die Lösung aufgebaut wird.

5. Die Mosaiksteine des vorherigen Schrittes werden in die Liste der aktuellen Lösungsansätze eingepaßt.

6. Der Kreislauf beginnt wieder bei Punkt 1 und endet, wenn keine weiteren Fragen mehr erforderlich sind.

Dieser Ablauf wird in Anhang A später anhand eines konkreten Beispiels aus dem Expertensystem **INVEST** detaillierter dargestellt werden. Abbildung 4.5 stellt den Ablauf einer Beratung mit **INVEST** graphisch dar.

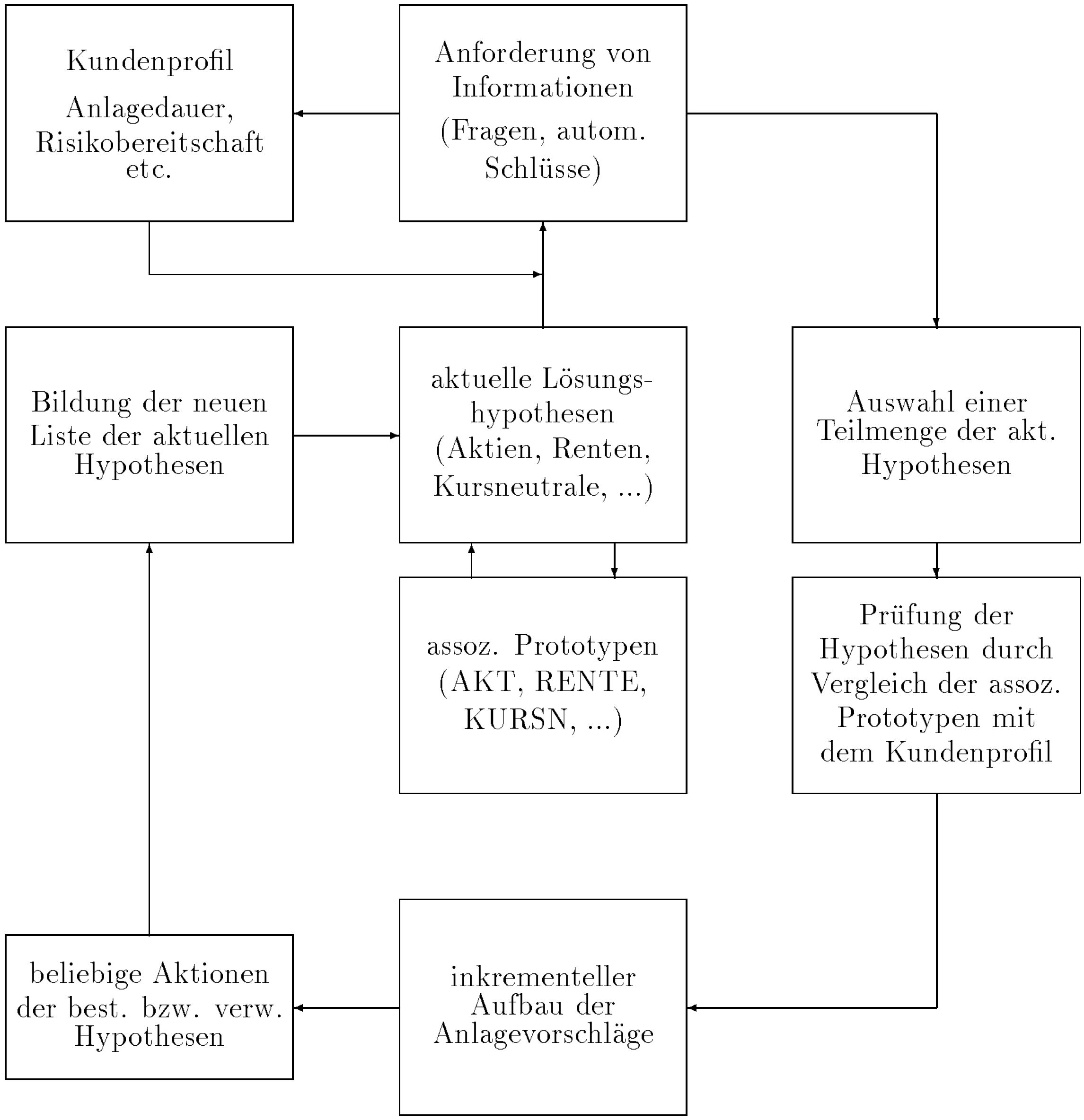


Abbildung 4.5: Flußdiagramm einer Beratung mit **INVEST**

4.4 Wahl der Implementierungssprache

Nach der Entscheidung für ein angemessenes Problemslösungsparadigma stellte sich die Frage nach der Wahl einer geeigneten Implementierungssprache. Die Erfüllung der folgenden Anforderungen ist dabei zu gewährleisten:

- *Rapid Prototyping.* Die Sprache sollte Rapid Prototyping [Budde *et al.*, 1984] unterstützen. So wird es möglich, die Systemfunktionen zwar in ihrer beabsichtigten endgültigen Form zu präsentieren, aber nur einen Teil davon wirklich zu implemetieren *(vertical Prototyping)* bzw. die Funktionen nur so zu implementieren, daß eine Demonstration möglich wird *(horizontal Prototyping)*.

- *Programmierumgebung.* Die Sprache sollte in eine Umgebung *(Programming Environment)* eingebettet sein, die die interaktive Programmentwicklung wirksam unterstützt. Das bedeutet insbesondere, daß fehlerhafte Routinen nicht zum Zusammenbruch des gesamten Systems führen und daß Werkzeuge zur Verfügung stehen, die es erlauben, den Zustand des Systems nach Auftreten eines Fehlers zu analysieren, die fehlerhafte Routine zu korrigieren und dann mit der Ausführung fortzufahren.

  Da ein Wissensbasiertes System immer eine große Menge Daten enthält, kann es nämlich sehr viel aufwendiger sein, genau diese Situation wieder herzustellen, als nur ein Neuladen des Programms.

- *Portabilität.* Die Sprache sollte auf einer ausreichend großen Zahl von Computern zur Verfügung stehen und so standardisiert sein, daß eine Portierung des Systems nur geringen Aufwand erfordert.

- *Effizienz.* Die Sprache sollte so effizient auf den verfügbaren Rechnern ablaufen können, daß die Implementation des Interpretieres für die Module der Wissensbasis immer noch vertretbare Ausführungszeiten für eine interaktive Benutzung bietet.

Die obigen Bedingungen legen nahe, keine der klassischen Programmiersprachen zu verwenden. Es gibt durchaus (insbesondere kommerzielle) Wissensbasierte Systeme, die z.B. in Fortan oder C geschrieben sind, aber in fast allen

Fällen handelt es sich dabei um Reimplementationen von Programmen, die vorher in einer der KI-Sprachen vorlagen. Zur Entwicklung von **DONALD** kamen drei Sprachen in Frage, die in den folgenden Abschnitten betrachtet werden.

4.4.1 Smalltalk

Smalltalk [Goldberg and Robinson, 1983] [Krasner, 1984] stellt hohe Anforderungen an die Hardware, die zu Beginn des Projektes von keinem der vorhandenen Rechner erfüllt werden konnten. Ein hochauflösendes Bitmapterminal ist ebenso erforderlich wie eine schnelle CPU, da Smalltalk zwar einen Compiler enthält, dieser aber keinen Code für die CPU selbst *(Native Code)* erzeugt, sondern nur für eine virtuelle Smalltalk Maschine. Der Compiler[8] ist so sehr portabel, aber die virtuelle Maschine muß dann noch durch einen kleinen Interpretierer simuliert werden.

Smalltalk ist eine objektorientierte Sprache, die dieses Konzept vollkommen einheitlich für *alle* Datentypen durchsetzt. Sogar solche Basisoperationen wie das Addieren zweier kleiner ganzer Zahlen wird objektorientiert abgehandelt. Man kann Objekten des Typs *Integer* z.B. eine Nachricht *(Message)* mit dem Inhalt „addiere dich zu 42“ schicken und erhält ein neues *Integer*-Objekt als Resultat. Aus theoretischer Sicht ist die mit einem solchen Vorgehen verbundene Uniformität zu begrüßen, der damit verbundene Effizienzverlust ist jedoch durch die nochmalige Interpretation nicht zu vernachlässigen. Modernere objektorientierte Programmiersprachen wie C++ [Stroustrup, 1986] erweitern einen herkömmlichen Kern von Programmiermethoden um objektorientierte Ansätze, so daß die Effizienzvorteile klassischer Sprachen auf niedriger Abstraktionsstufe ausgenutzt werden können. Die weitere Beschäftigung mit Smalltalk wäre sicherlich interessant gewesen, mußte aber leider unterbleiben, da keine geeignete Hardware zur Verfügung stand.

[8] Solche Compiler, die nur in einen Zwischencode kompilieren, werden auch als *Bytecompiler* bezeichnet, da die Befehle für die virtuelle Maschine typischerweise in ein Byte passen.

4.4.2 Prolog

Prolog [Sterling and Shapiro, 1986] war schon zu Beginn des Projektes auf einem IBM Personal Computer verfügbar. Das Prolog-1 System von Expert Systems Limited hatte durch einschneidende Speicherplatzbeschränkungen des segmentierten Memorymodells der 8086 CPU zwar nur den Charakter eines Experimentalsystems, aber durch die schnelle Entwicklung des europäischen Prolog-Markts war damit zu rechnen, daß in Zukunft leistungsfähigere Implementationen erhältlich sein würden. Mit diesem Prolog wurde z.B. zu Testzwecken ein kleiner Interpretierer für Produktionsregeln implementiert.

Prolog erlegt dem Programmierer jedoch die Beschränkung auf einen einzigen Programmierstil auf. Bekannte Algorithmen können oft nicht natürlich dargestellt werden. Objektorientierte Methoden lassen sich ebenso wie iterative Formulierungen nur sehr umständlich umsetzen.

4.4.3 LISP

LISP ist die klassische Programmiersprache der Künstlichen Intelligenz. LISP ist nach Fortan die älteste Sprache überhaupt und hat sich durch fehlende Standardisierung sehr ausufernd entwickelt. Durch die Fähigkeit, Programme als Daten zu manipulieren, ist LISP im Gegensatz zu Prolog in der Lage, mehrere verschiedene Programmierstile parallel zu unterstützen.

Kaum eines der heute in LISP erstellten Systeme ist eigentlich in LISP programmiert, vielmehr dient LISP nur als Vehikel zur Implementation einer eigenen problemangepaßten Sprache. Die Konstrukte dieser neuen Sprache werden dann automatisch in LISP Funktionen übersetzt. In LISP geschrieben sind eigentlich nur diese Umwandlungsfunktionen.

Diese Flexibilität, verbunden mit der sehr guten Programmierumgebung, ist verantwortlich für die immer noch große Attraktivität der LISP Systeme.

Größtest Argument *gegen* LISP war in der Vergangenheit eigentlich die große Vielzahl von *Dialekten*[9], die recht bedeutende Unterschiede aufweisen und damit die Portierung von Systemen erschweren.

[9] Zu den wichtigsten Vertretern gehören Interlisp, Maclisp, Zetalisp, Franzlisp und NIL *(New Implementation of Lisp)*.

In den letzten Jahren war jedoch ein eindeutiger Wille zur Standardisierung zu verzeichnen. Guy L. Steele definierte bereits in der ersten Auflage von [Steele Jr., 1990] aus dem Jahre 1984 Common LISP als Integration der bedeutendsten LISP Dialekte. Inzwischen liegt auch ein ANSI Draft Standard vor. Common LISP ist dabei nicht einfach als kleinste gemeinsame Teilmenge aller Dialekte entstanden, sondern stellt im Gegenteil verhältnismäßig hohe Anforderungen. Die Beschränkung auf einen zu kleinen Sprachumfang war ja bereits der Programmiersprache PASCAL [Cooper, 1983] zum kommerziellen Verhängnis geworden. So ist es nicht erforderlich, sich bei der Programmierung in Common LISP auf einen kleinen Satz von Funktionen und Datentypen zu beschränken, um das resultierende System portabel zu halten. Man kann sich z.B. auf die Existenz von Vektoren, Bitarrays und Hashtabellen verlassen, mit denen eine kompaktere Speicherung oder ein schnellerer Zugriff zu realisieren ist, als mit dem für LISP charakteristischen Datentyp der einfach verketteten Liste.

Common LISP steht als AKCL *(Austin Kyoto Common LISP)* der Universitäten von Kyoto (Japan) und Austin (Texas) für eine Reihe von UNIX Implementationen in Form einer kostenlosen Lizenz zur Verfügung. Es handelt sich bei AKCL um ein vollständiges Common Lisp System mit einem optimierenden Compiler. AKCL wurde komplett in C geschrieben und verwendet auch bei der Kompilation von LISP Funktionen als Zwischensprache C. Der entstehende C Sourcecode wird mit dem C Compiler des UNIX Systems in Objectcode übersetzt und anschließend dynamisch in das laufende LISP gelinkt. Die Optimierung des LISP Compilers geht dabei soweit, daß entsprechend deklarierte LISP Variablen wirklich in CPU Registern untergebracht werden. Einziger Nachteil dieser Methode sind die relativ langen Kompilationszeiten.

4.4.4 Bewertung

Nur LISP unterstützt mit der nötigen Effizienz eine ganze Vielzahl von Programmierstilen. Es lassen sich iterative und rekursive Algorithmen natürlich formulieren. Mit der Aufnahme von CLOS *(Common Lisp Object System)* ist auch die objektorientierte Programmierung durch die Aufnahme in Steeles Standardwerk portabel geworden. Es stehen leistungsfähige Implementationen für Rechner aller Größenklassen vom Personal Computer bis zur Cray

(S1-LISP) zur Verfügung, davon eine kostenlos für Workstations unter UNIX. Für Prolog spricht die eingebaute Deduktionskomponente zur Interpretation von Hornklauseln mit Backtracking. Ähnliches läßt sich aber im Notfall auch in LISP recht einfach nachbilden. So fiel die Entscheidung für LISP.

Die nächsten Abschnitte beschäftigen sich mit den Details der in **DONALD** verwendeten Wissensrepräsentation.

4.5 Wissensrepräsentation mit Frames

DONALD-III besteht aus einem Kern mit Funktionen zur Erzeugung und Manipulation von Frames. Diesem Kern kommt große Bedeutung zu, da alle wichtigen Daten innerhalb eines auf **DONALD-III** basierenden Expertensystems in Frames abgelegt sind. Die Effizienz der Grundoperationen bestimmt deshalb auch in hohem Maße die Effizienz des gesamten Systems. Daher ist beim Design der Grundoperationen auch die effiziente Implementierbarkeit ein wichtiger Gesichtpunkt.

In diesem Abschnitt wird die Syntax für Frames in **DONALD** definiert, und es werden Details der Implementation der heterogenen Frame-Hierarchie erläutert.

Bei den in **DONALD** auftretenden, als Frames dargestellten Objekten handelt es sich zum einen um Begriffe der realen Welt, die für die konkrete Anwendung eines mit **DONALD** erstellten Expertensystems von Bedeutung sind (im Fall der Anlageberatung sind das zum Beispiel Personen, Wertpapiere, wirtschaftliche Rahmenbedingungen), zum anderen um abstrakte Begriffe, die zu **DONALD**s Inferenz- bzw. Wissensrepräsentationsmechanismus gehören (zum Beispiel Hypothesen, Lösungsvorschläge). Es folgen Beispiele verschiedener Typen von Objekten und ihre Darstellung als Frames.

4.5.1 Klassen, Teilklassen und Instanzen

Frames treten in **DONALD** in drei Grundformen auf. Jeder Frame ist entweder eine *Klasse*, eine *Teilklasse* oder eine *Instanz*. Durch Klassen werden

Objekte oder Begriffe beschrieben, die nicht als Spezialfälle eines allgemeineren, schon bekannten Begriffes definiert werden können. Klassen enthalten Definitionen für *Slots*, das heißt Definitionen der Merkmale, durch die die Mitglieder einer Klasse charakterisiert werden.

Im Gegensatz zu den Klassen stellen Instanzen konkrete Objekte dar, die keine weitere Spezialisierung mehr zulassen. Instanzen enthalten deshalb keine Slotdefinitionen, sondern *Views*. Ein View beschreibt einen Blickwinkel, aus dem eine Instanz als Mitglied einer Klasse betrachtet wird. Ein View ist die Aufzählung der in der zugehörigen Klasse definierten Slots zusammen mit den für die Instanz charakteristischen Werten. Instanzen, die einen View auf eine bestimmte Klasse besitzen, sind Elemente dieser Klasse.

Die Abbildungen 4.6–4.7 zeigen als Beispiel eine mögliche Darstellung der Klasse aller Aktien und als ein Mitglied dieser Klasse die Instanz "Beispiel-AG".

```
(Class Aktie
   (Slots are
         (Symbol Bezeichnung)
         (Real   Nominalwert)
         (Real   Kurs)
         (Real   KGV10)))
```

Abbildung 4.6: Die Klasse Aktie

Da Klassen nicht mehr durch den Vergleich mit einem umfassenderen Konzept beschrieben werden können, besitzen sie keine Views. Umgekehrt haben Instanzen, weil sie keine Elemente mehr haben können, keine Definitionen für Slots.

Die dritte Gruppe von Frames sind die *Teilklassen*. Sie enthalten sowohl Slotdefinitionen als auch Views. Teilklassen sind Teilmengen der Klassen (oder

[10] KGV steht für Kurs-Gewinn-Verhältnis. Diese Kennziffer besagt, das Wievielfache des aus den Jahresabschlußdaten geschätzten Reingewinns je Aktie der Kurs einer Aktie ausmacht. Das KGV ist in der Aktienanalyse ein Maß für die Preiswürdigkeit einer Aktie.

```
(Instance BAG
  (View Aktie with
        (Bezeichnung = Beispiel-AG)
        (Nominalwert = 50.0)
        (Kurs = 271.3)
        (KGV = 5.3)))
```

Abbildung 4.7: Die Instanz BAG

Teilklassen), für die sie Views besitzen. Sie ermöglichen eine strukturierte Darstellung einer Menge von Objekten und Begriffen mit vielen Abstufungen unterschiedlichen Abstraktionsgrades.

```
(Class Wertpapier
  (Slots are
        ((OneOf klein mittel hoch) Risiko)
        (Symbol                    Emittent)
        (Symbol                    Land)
        (Integer                   Nominalwert)
        ((OneOf ja nein)           börsengängig)))
```

Abbildung 4.8: Die Klasse Wertpapier

4.5.2 Die Definition von Slots in Klassen und Teilklassen

Die Slots in Klassen und Teilklassen dienen nicht dem Speichern von Werten (obwohl auch das möglich ist), sondern der Definition der Merkmale, durch die sich die Mitglieder dieser Klasse unterscheiden lassen.

Jeder Slot hat einen Typ. Das ist eine Angabe darüber, welche Werte für diesen Slot erwartet werden. Mögliche Typen sind zum Beispiel *Integer* (ganze

```
(SubClass KursneutralesAnlagepapier
  (Slots are
        ((OneOf verzinslich abgezinst) Zinsmodus)
        (Real                          Nominalzins)
        (Real                          Rendite)
        (Integer                       Laufzeit))
  (View Wertpapier with
        (Risiko ist klein)
        (börsengängig = nein)))
```

Abbildung 4.9: Die Teilklasse Kursneutrales Anlagepapier

```
(SubClass Bundesschatzbrief
  (Slots are
        ((VectorOf Real) Nominalzinsen)
        ((OneOf A B)     Typ)))
  (View Wertpapier with
        (Emittent ist Bund)
        (Land ist BundesrepublikDeutschland)
        (Nominalwert ist 100)
  (View KursneutralesAnlagepapier))
```

Abbildung 4.10: Die Teilklasse Bundesschatzbrief

```
(SubClass Bundesschatzbrief-A
  (View KursneutralesAnlagepapier with
        (Zinsmodus ist verzinslich)
        (Laufzeit ist 6))
  (View Bundesschatzbrief with
        (Typ ist A)))
```

Abbildung 4.11: Die Teilklasse Bundesschatzbrief-A

```
(Instance BSB-A-1987/4
  (View KursneutralesAnlagepapier with
        (Rendite ist 5.84)
        (Rückzahlung ist 1.9.1993)
        (Zinstermine ist (List 1.9.)))
  (View Bundesschatzbrief with
        (Nominalzinsen sind (Vector 4.0 5.0 6.0 6.25 6.5 8.0)))
  (View Bundesschatzbrief-A))
```

Abbildung 4.12: Ein spezielles Wertpapier

Zahlen), *Real* (Gleitkommazahlen), *Symbol* (Symbole) und andere mehr. Auch zusammengesetze Typen wie *List* (Listen) oder *Vector* (Vektoren von Basistypen) sind erlaubt. Der Typangabe folgt der Name des Slots. Danach kann ein sogenannter *Default* angegeben werden. Das ist ein Wert, der für alle Instanzen und Teilklassen, die einen View mit diesem Slot haben, als Wert dieses Slots angenommen wird, sofern für sie nicht ausdrücklich ein anderer Wert angegeben wurde. Ein Beispiel dazu folgt im Abschnitt 4.8 über Vererbung. Die Abbildungen 4.8–4.12 illustrieren die Ausführungen in diesem Abschnitt.

4.5.3 Die Definition von Views in Teilklassen und Instanzen

Wie schon beschrieben, dienen Views dazu, neue Objekte oder Begriffe als spezielle Vertreter einer allgemeineren Klasse zu beschreiben.

Teilklassen und Instanzen können beliebig viele Views haben. Dabei müssen die (Teil-) Klassen, auf die die Views deuten, schon zuvor definiert sein. Das garantiert, daß der durch die Teilmengen- bzw. Elementrelation gebildete Graph aller Frames azyklisch ist.

Views sind Listen, die mit der Sequenz "View *Name* with" beginnen.

Um wichtige Gesichtspunkte von weniger wichtigen unterscheiden zu können, kann eine ganze Zahl $n \geq 1$ als Gewicht eines Views angegeben werden. Ohne diese Angabe wird als Gewicht 1 angenommen.

Darauf folgt optional eine Folge von Tripeln. Das sind Listen aus dem Namen eines Slots aus der zuvor durch *Name* bezeichneten (Teil-) Klasse, einem Zuweisungsoperator (z.B. "=") und einem Ausdruck, der der Wert dieses Slots in dieser Instanz bzw. Teilklasse ist.

4.5.4 Die Syntax für Frames in DONALD

Die in den vorangegangenen Beispielen illustrierte Syntax für Frames wird in diesem Abschnitt formal definiert.

Frames werden als Sätze der Grammatik mitgeteilt, die in der nachfolgenden Tabelle 4.1 auf Seite 130 dargestellt ist.

Dabei sind nicht in Anführungszeichen (◁,▷) eingeschlossene Wörter terminale Symbole. Ein nachgestelltes Sternchen ($*$) heißt, daß der vorangegangene Ausdruck beliebig oft (auch keinmal) auftreten darf. Ein nachgestelltes Pluszeichen (+) heißt, daß der vorangegangene Ausdruck beliebig oft wiederholt werden darf, aber mindestens einmal auftreten muß. Eckige Klammern ([,]) umfassen Ausdrücke, die optional sind. Der senkrechte Strich (|) trennt Alternativen voneinander.

Kursiv gesetzte Begriffe bezeichnen Lisp-Atome. Diese Begriffe stellen selbst durch ihre Bedeutung die an das Lisp-Atom gestellte Bedingung dar. Zum Beispiel bezeichnet *Slotname* den Platz, an dem ein Lisp-Atom stehen darf, das der Name eines Slots ist.

◁`atom`▷ und ◁`s-expr`▷ sind durch den zugrundeliegenden Lisp-Interpreter festgelegt und bezeichnen gültige Lisp-Atome bzw. Lisp-S-Ausdrücke [Steele Jr., 1990].

◁`w-integer`▷ bezeichnet eine natürliche Zahl ≥ 1. ◁`cf-integer`▷ ist eine ganze Zahl im Bereich von -1000 bis +1000.

Diese Darstellung von Frames heißt die *externe Form*. Sie ist, wie man der Grammatik ansehen kann, ein gültiger Lisp-S-Ausdruck und läßt sich als solcher mit den vom Lisp zur Verfügung gestellten Funktionen anzeigen, ausdrukken, auf Hintergrundspeicher (Platten, Disketten, Streamer) schreiben und auch wieder einlesen.

<table>
<tr><td>◁frame▷</td><td>::=</td><td>◁class▷ | ◁subclass▷ | ◁instance▷</td></tr>
<tr><td>◁class▷</td><td>::=</td><td>(Class Classname ◁slotdef▷)</td></tr>
<tr><td>◁slotdef▷</td><td>::=</td><td>(Slots are ◁class-slot▷+)</td></tr>
<tr><td>◁class-slot▷</td><td>::=</td><td>(◁type▷ Slotname [◁assignop▷ ◁value▷] ◁facet▷*)</td></tr>
<tr><td>◁subclass▷</td><td>::=</td><td>(SubClass SubClassname [◁slotdef▷] ◁view▷+)</td></tr>
<tr><td>◁instance▷</td><td>::=</td><td>(Instance Name ◁view▷+)</td></tr>
<tr><td>◁view▷</td><td>::=</td><td>(View { Classname | SubClassname } [weight ◁w-integer▷]
[with ◁slot▷+])</td></tr>
<tr><td>◁slot▷</td><td>::=</td><td>(Slotname ◁assignop▷ ◁value▷ ◁facet▷*)</td></tr>
<tr><td>◁assignop▷</td><td>::=</td><td>= | is | are | ist | sind</td></tr>
<tr><td>◁type▷</td><td>::=</td><td>◁atomic-type▷ | (OneOf ◁atom▷+) |
(◁ctype-tag▷ ◁atomic-type▷)</td></tr>
<tr><td>◁atomic-type▷</td><td>::=</td><td>Integer | Real | Symbol | String |
Classname | SubClassname | Other</td></tr>
<tr><td>◁ctype-tag▷</td><td>::=</td><td>SetOf | ListOf | VectorOf | HListOn | AssocListOn</td></tr>
<tr><td>◁value▷</td><td>::=</td><td>◁atom▷ | (Set ◁value▷*) | (List ◁value▷*) |
(Vector ◁value▷*) | (HList ◁h-pair▷*) |
(AssocList ◁pair▷*)</td></tr>
<tr><td>◁pair▷</td><td>::=</td><td>(◁atom▷ . ◁value▷)</td></tr>
<tr><td>◁h-pair▷</td><td>::=</td><td>(◁atom▷ . ◁cf-integer▷)</td></tr>
<tr><td>◁facet▷</td><td>::=</td><td>[to | if | when | wenn | falls] Facetname ◁s-expr▷</td></tr>
</table>

Tabelle 4.1: Framesyntax in **DONALD**

4.5.5 Implementation der Frames

Eine FRL-ähnliche Implementation der früherern Version **DONALD-I** hatte gezeigt, daß der Zugriff auf Frameinhalte durch die Darstellung der Frames als geschachtelte Listen stark verlangsamt wird.

Eine parallel entwickelte, auf einer Vektordarstellung beruhende Repräsentation der Frames in **DONALD-II** [Heuer, 1987] ergab kürzere Zugriffszeiten und weniger Speicherplatzbedarf. Die Zugriffszeit auf eine Vektorkomponente ist konstant, während die Zugriffszeit auf ein Listenelement proportional zur Länge der Liste anwächst.

Das Bemühen, bei der Implementation Kompatibität zu Common LISP zu erzielen, legte es nahe, die *Structure*-Möglichkeiten von CommonLisp zu benutzen. Common LISP Strukturen sind Pascal-ähnliche Datenstrukturen. Bei der Definition einer solchen Struktur erzeugt der Interpreter bzw. der Compiler automatisch Zugriffsfunktionen auf die angegebenen Datenfelder und Funktionen zum Erstellen und Kopieren solcher Strukturen.

Die Definition eines Frames sieht dann so aus:

```
(defstruct (frame :named :conc-name (:copier copy-frame-i)
                  (:print-function frame-print))
  (name nil) (bindings nil) (longname nil) (source-code nil)
  (created nil) (modified nil) (version nil) (database nil)
  (position nil) (backup nil) (flags nil) (author nil)
  )
```

Die aktuelle Implementation in **DONALD-III** rückt wieder ein wenig vom völlig an Vektoren orientierten Vorgehen ab. Zunächst war für jeden View einer Instanz bzw. einer Teilklasse ein Vektorfeld vorgesehen [Heuer, 1987]. Das macht das Verändern einer Instanz durch Hinzufügen oder Löschen eines Views kompliziert, da die Länge eines Instanzvektors ab dem Zeitpunkt der Definition der Instanz fest ist. Das gleiche gilt für Viewvektoren und die darin enthaltenen Slotvektoren.

Um hier wieder mehr Bewegungsraum zu schaffen, sind die Viewvektoren wieder zu einer Liste zusammengefaßt worden, die in einer Vektorzelle der Instanz gespeichert ist. Analog war das Vorgehen bei den Slotvektoren in den Views.

4.5.6 Die interne Darstellung der Frames

Die Transformation von der *externen* zur *internen* Darstellung geschieht durch das Einlesen der externen Form durch den Lisp-Interpreter. `Class`, `SubClass` und `Instance` sind Lisp-Makros, die beim Einlesen zu einem Aufruf der Funktion `create-frame` expandiert werden.

Die externe Form ist also gerade der Lisp-S-Ausdruck, dessen Auswertung als Seiteneffekt die Herstellung der internen Darstellung hat. Diese interne Form ist in Tabelle 4.2 dargestellt.

Name	Name des Frames
Bindings	derzeit unbenutzt
Longname	lange Namensversion zu Ausgabezwecken
Source-Code	externe Form des Frames
Created	Datum der Erzeugung des Frames
Modified	Datum der letzten Änderung
Version	Versionsnummer
Database	die zugehörige Datenbasis
Position	Position innerhalb der Datenbasis
Backup	Position des letzten Backups
Flags	zur Zustandskontrolle des Frames
Author	der Autor dieses Frames
Views	alphabetische Liste der Viewvektoren
Default[a]	Default-Setzungen der Slots
Instances[a]	Instanzen-Liste der (Teil-) Klassen
Subclasses[a]	Teilklassen-Liste der (Teil-) Klassen

[a] Diese Strukturfelder sind nur für Klassen und Teilklassen definiert.

Tabelle 4.2: Die interne Form der Frames (als Struktur)

Zu Tabelle 4.2 sind folgende Bemerkungen zu machen:

- Im Feld *Database* steht der Name der Wissensbasis, zu der dieser Frame gehört.

Slots	Liste der Slotvektoren
Upward	Zeiger auf den Framevektor
Class	Name der definierenden Klasse des Views
Definition	Zeiger auf den Klassenvektor
Weight	Gewicht des Views
Bindings	derzeit unbenutzt

Tabelle 4.3: Die interne Form der Views (als Struktur)

Name	Name des Slots
Upward	Zeiger auf den Viewvektor
Type	Typangabe des Slots
Value	Wert des Slots
Facets	Facetten des Slots
Demons	Demons des Slots
Origin	Herkunft des Slots

Tabelle 4.4: Die interne Form der Slots (als Struktur)

- Flags markieren verschiedene Zustände des Frames. Sie zeigen zum Beispiel an, daß die interne und die externe Form nach einem Schreibzugriff auf den Frame nicht mehr übereinstimmen. Wird die externe Form benötigt und das entsprechende Flag ist gesetzt, so wird zunächst die neue externe Form aus der internen Darstellung berechnet, das Flag gelöscht und die neue Form weitergegeben.

- Auch Klassen haben aus Gründen der Uniformität ein *Views*-Feld, das aber stets nur die leere Liste enthält.

- Das Feld *Default* enthält einen Viewvektor mit den Default-Setzungen der (Teil-) Klasse.

- Die Felder *Instances* und *Subclasses* enthalten Listen mit den Namen der Instanzen bzw. der Teilklassen.

Ebenso wie Frames haben auch die Views und Slots eine interne Darstellung als Strukuren. Sie sind in den Tabellen 4.3 und 4.4 abgebildet.

Die *Upward* Felder enthalten einen Zeiger auf den Framevektor (Viewvektor), in dem der Viewvektor (Slotvektor) enthalten ist. Mit *Herkunft des Slots* ist gemeint, aus welcher Quelle der Inhalt dieses Slots stammt. Mögliche Werte sind `:user`, `:demon`, `:default` und `NIL`. Diese Reihenfolge bildet auch die Priorität beim Schreibzugriff auf einen Slot: zum Beispiel kann eine vom Benutzer eingegebene Antwort einen Slotinhalt stets überschreiben, weil sie `origin` = `:user` hat. Umgekehrt kann ein Default-Wert weder einen von einem Demon errechneten Wert noch eine Benutzereingabe überschreiben.

4.6 Facetten und Demons in DONALD

Demons und *Facetten* sind Begriffe, die in den vorangegangenen Abschnitten, vor allem im Zusammenhang mit FRL, schon erläutert worden sind. In diesem Abschnitt wird ihre Implementation und ihre Wirkungsweise beschrieben werden.

4.6.1 Facetten

Facetten sind in Slots auftretende Speicherplätze, auf deren Inhalt mit einem Schlüsselwort, dem Facettennamen (siehe Tabelle 4.4), zugegriffen werden kann. Sie sind als Lisp-*Association-Lists* [Wilensky, 1986] im Strukturfeld *Facets* gespeichert.

Im Unterschied zu FRL werden die Speicherplätze für den Wert des Slots (vgl. die VALUE Facette) und für angebundene Prozeduren, die beim Zugriff auf den Slot ausgeführt werden, (vgl. die IF-NEEDED Facette) nicht als Facetten bezeichnet.

In der derzeitigen Version von **DONALD** sind die Facetten *Ask, Explain, Trigger, Weight* und *Constraint* implementiert. Abbildung 4.13 zeigt eine Slotdefinition aus der Klasse *Anleger* und illustriert die Syntax für Facetten und Demons.

Die folgende Aufzählung erläutert die Bedeutung der Facetten im einzelnen.

```
(Class Anleger
   (Slots are
      ...
      ((HListOn |ich weiss noch nicht| nein ja) AnlageVerfügbarkeit
       to ask “Muss das Geld während der Anlagedauer verfügbar sein? ”

       to trigger     (ja (.RENTE .INVESTMENT)
                      nein (.OPT .AUSL .AKT))
       to explain     “Verfügbar sind Geldanlagen im weitesten Sinne im- ~@
                      mer, da man schlimmstenfalls die Anlage beleihen ~@
                      kann. Verfügbar soll hier heißen, daß die Anlage ~@
                      liquidierbar ist unter (teilweisem) Verzicht auf ~@
                      die angestrebten Anlageziele, aber ohne zusätz- ~@
                      liche Kosten wie Kreditzinsen. ~%”
       put> (InCase(eq (GetSlot AnlageVerfügbarkeit) ’nein)
                      (Forward (KURSN-Anleger Liquidität/Rendite Rendite)
                               (BSB-Anleger Sperrfrist ja))
                      (Forward (KURSN-Anleger Liquidität/Rendite -)
                               (BSB-Anleger Sperrfrist -))))
      ...
       ))
```

Abbildung 4.13: Facetten und Demons eines Slots

- Die Facetten *ask* und *explain* enthalten einen String, der den Formatanforderungen der Lisp-Ausgabefunktion *format* genügt. Der ask-String wird benutzt, wenn eine Frage nach dem Wert des Slots für den Benutzer formuliert wird. Ist diese Facette leer, wird ein Standardfragetext benutzt. Der explain-String wird ausgegeben, wenn der Benutzer auf eine Frage hin weitere Erläuterungen anfordert.

- Die *trigger*-Facette enthält eine Lisp-association-list. Die möglichen Slotwerte sind darin die Schlüssel. Die Werte dazu sind ein Hypothesenname oder eine Liste von Hypothesennamen. Auf eine Benutzerantwort hin wird an die entsprechende(n) Hypothese(n) je ein Triggerpunkt vergeben.

- Die Facette *weight* (siehe Abbildung 4.23 auf Seite 157) enthält eine natürliche Zahl $n \geq 1$, die für das Gewicht dieses Slots innerhalb des Views steht (siehe Abschnitt 4.9 über das Matchen).

- Die *constraint*-Facette ermöglicht die Angabe von Bedingungen, die ein für diesen Slot zulässiger Wert erfüllen muß. Die Angabe der *constraints* folgt der Syntax in Tabelle 4.5.

`◁constraint▷`	::=	(`◁atomic-constr▷`) |
		([Or | And] (`◁atomic-constr▷`)+)
`◁atomic-constr▷`	::=	([not] `◁atomic-constr▷`) |
		(Range `◁number▷` `◁number▷`) |
		(`◁predicate▷` `◁s-expr▷`∗)

Tabelle 4.5: Syntax für *Constraints* in **DONALD**

Ein Beispiel für eine Constraint Facette wäre

`constraint(And` $(\geq 2)(\leq 4))$

was zu der Angabe

`constraint(Range` 24)

äquivalent ist und nur numerische Werte zwischen 2 und 4 als Slotinhalt zuläßt.

Facetten liefern Zusatzinformationen zu einem Slot, die in vorgegebenen Situationen vom Programm abgefragt werden können. Demons dagegen sind unabhängiger von der Kontrolle, die das Programm ausübt.

4.6.2 Demons

Demons sind Prozeduren, die zu jedem Zeitpunkt während des Programmablaufs durch einen Schreib- oder Lesezugriff auf einen Slot angestoßen werden können. Meist erfüllen sie Aufgaben wie, die Slotwerte zu aktualisieren oder leeren Slots einen hergeleiteten Wert zu geben. Da Demons ihrerseits wieder Zugriffe auf andere Slots enthalten können, kann es sein, daß größere Programmsequenzen, die in der Wissensbasis auftreten, unabhängig von der kontrollierenden Shell ablaufen.

Die Wissensrepräsentation durch Frames ist ein deklarativer Formalismus, der aber in speziellen Fällen durch Demons als wesentlichem Teil der prozeduralen Anbindung *(procedural attachment)* von **DONALD** ergänzt wird. Solche prozeduralen Komponenten dienen z.B. folgenden Zwecken:

Effizienz. Falls für ein Problem ein wohldefinierter Algorithmus existiert, dann ist kompilierter Kode in einer klassischen Programmiersprache oder auch LISP im allgemeinen um Größenordnungen schneller als ein KI-System. Als extremes Beispiel ist die Benutzung der eingebauten arithmetischen Operationen dem Einsatz eines Theorembeweisers auf der Basis der Peanoschen Axiome vorzuziehen.

Externe Schnittstellen. Ein KI-Programm steht typischerweise nicht allein für sich. Oft ist die Interaktion mit bereits existierenden Programmpaketen, die in anderen Programmiersprachen implementiert sind, erforderlich oder Datenbankzugriffe und direkte Betriebssystemaufrufe sind nötig. In **INVEST** stellt sich das Problem, auf die Daten über das bestehende Wertpapierdepot des Kunden in den vorhandenen DB2-Datenbanken des Großrechners zugreifen zu müssen. Auch aktuelle Bewertungen einzelner Aktien und Anleihen sind über Dienste wie Reuter kurzfristig elektronisch verfügbar. Zusatzprodukte zur Kursanalyse einzelner Wertpapiere stehen zur Verfügung, basieren aber auf grundsätzlich verschiedenen Ansätzen, so daß eine direkte Integration solcher Komponenten nicht sinnvoll ist. Die Integration von KI-Systemen mit Programmen der klassischen EDV stellt im allgemeinen noch ein großes Problem dar.

Ungelöste Probleme. Rein deklarative Systeme erfordern eine vollständige Axiomatisierung für jeden Aspekt der Fragestellung. Wird ein Problem nicht oder noch nicht vollständig verstanden, so gibt es aber doch oft Lösungsansätze, mit denen sich in wichtigen Spezialfällen durchaus sinnvoll arbeiten läßt.

Die Syntax für Demons entspricht der für Facetten (siehe Tabelle 4.1). Anstelle des Facettennamens steht eines der vier Lisp-Atome *>get, get>, >put* oder *put>*. Sie bezeichnen den Zeitpunkt der Ausführung des Demons. Dabei stehen *get* und *put* für einen Lese- bzw. Schreibzugriff. Das vorangestellte und nachgestellte >-Zeichen bedeutet *vor* bzw. *nach* dem angegebenen Zugriff. Der darauf folgende Lisp-S-Ausdruck kann ein beliebiges Stück Lisp-Quellcode sein. In **DONALD** sind vier Demons bereits vorgesehen, nämlich *Forward, Fetch, GetSlot* und *InCase*. Die folgende Aufstellung erklärt die Bedeutung dieser Demons.

- *Forward* wird mit der Syntax

  (Forward (*Viewname Slotname* [◁`value`▷])+)

  aufgerufen. Forward gibt für jedes Paar bzw. Tripel den mit *value* bezeichneten Wert an den genannten Slot im View *Viewname* desselben Frames weiter. In die angesprochenen Slots geschrieben wird der Wert nur, wenn der bisherige `origin` der Slots es erlaubt (vgl. Tabelle 4.4). Ist kein Wert angegeben, benutzt Forward den Wert des Slots, in dem es selber steht. Als Beispiel kann Abbildung 4.13 dienen.

- *Fetch* folgt der Syntax

  (Fetch (*Viewname Slotname*)+)

  Fetch durchsucht die durch *Viewname* und *Slotname* bezeichneten Slots desselben Frames, bis ein Wert gefunden wird. Dieser Wert wird in dem Slot, in dem der Demon steht, als Wert eingetragen.

- *GetSlot-Demons* werden durch einen Ausdruck der Form

  (GetSlot [*Slotname* | (*Slotname*) | (*Viewname Slotname*)])

  angegeben. GetSlot liefert als Ergebnis den Wert des Slots *Slotname* im selben View bzw. im View *Viewname*, falls ein Viewname angegeben ist. Ist weder ein Slot noch ein View angeführt, ist das Ergebnis der Wert des Slots, in dem der Demon steht.

- Der *InCase-Demon* ist eine Liste der Gestalt

  (InCase "*condition*" "*then-part*" ["*else-part*"])

  condition, then-part und *else-part* sind Lisp-S-Ausdrücke. Ist der Wert von *condition* nicht `nil`, wird *then-part* evaluiert; sonst, falls vorhanden, der *else-part*.

Die Speicherung der Facetten und Demons als Lisp-association-lists macht den Zugriff auf sie etwas weniger effektiv, als wenn jede Facette und jeder Demon ein eigenes Strukturfeld in der internen Darstellung des Slots erhalten hätte. Da aber das Auftreten von Demons und Facetten eher die Ausnahme ist, hätte man dadurch in jedem Slot eine Reihe von Strukturfeldern reserviert, die meist leer geblieben wären. Die Speicherplatzersparnis hatte hier Vorrang vor der Uniformität der Darstellung.

4.7 Produktionsregeln

Als alternative prozedurale Komponente bieten Produktionsregeln die Möglichkeit, auch ohne LISP-Kenntnisse Slots zu bearbeiten. Der in **DONALD-III** integrierte Regelinterpretierer benutzt die hierarchische Darstellung der Frames zur Klassifikation der Regeln. Alle Regeln sind Mitglieder der in Abbildung 4.14 dargestellten Klasse `Rule`. Die Teilklassen von `Rule` partitionieren dann die Menge der vorhandenen Regeln. Produktionsregelklassen oder auch einzelne Produktionsregeln werden genau wie Hypothesen ge*triggered* und dann je nach Wert des Slots `TimeOfEvaluation` vor oder nach Benutzereingaben bearbeitet.

Eine typische Regel zum Berechnen alle Zahlenpaare (i, j) mit $1 \leq i < j \leq 4$ zeigt Abbildung 4.15.

Das Beispiel in Abbildung 4.16 ist direkt aus **INVEST** übernommen. Dort wird auch der Slot `AttenuationFactor` benutzt, der eine Zielhypothese (Slot `TargetHypothesis`) durch Modifikation des Konfidenzfaktors unwahrscheinlicher macht. Folgende Formel wird verwendet:

```
(Class Rule
   (Slots are
      (Action Antecedent)
      (Action Consequent)
      (Integer AttenuationFactor)
      (Integer TruthThreshold is 600)
      (Integer FalsityThreshold is 200)
      (Symbol TargetHypothesis)
      ((ListOf Rule) TriggerIfTrue)
      ((ListOf Rule) TriggerIfFalse)
      ((OneOf early late) TimeOfEvaluation)
      (Integer TriggerPoints)))
```

Abbildung 4.14: Die Definiton der Klasse `Rule`

```
(Instance Test-Regel-3
   (View Rule with
            (Antecedent = (if (member ?x (1 2 3 4))
                              (member ?y (1 2 3 4))
                              (< ?x ?y)))
            (Consequent = (then (show ?x) (show ?y)))))
```

Abbildung 4.15: Regeln mit Variablen in **DONALD**

```
(Instance Investment-Regel-1
    (View Rule with
        (Antecedent is
            (if (the AnlageBetrag of Anleger ?Kunde is klein)
                (the AnlageZweck of Anleger ?Kunde is Vermögensstreuung)))
        (Consequent is (say "Investment-Regel-1 gefeuert."))
        (AttenuationFactor is 500)
        (TargetHypothesis is .INVESTMENT)))
```

Abbildung 4.16: Eine Regel für Investment Anleger

```
(defun adjust-cf (hypothesis output-certainty)
  (let ((x (fget (symbol-value hypothesis) 'Hypothesis 'ConfidenceFactor))
        (y output-certainty)
        (z 0))
       (if (and (>= x 0) (>= y 0))
           (setq z (+ x (/ (* y (- 1000 x)) 1000))))
       (if (or (and (< x 0) (>= y 0))
               (and (< y 0) (>= x 0)))
           (if (= (min (abs x) (abs y)) 1000)
               (setq z 0)
               (setq z (/ (* 1000 (+ x y))
                          (- 1000 (min (abs x) (abs y)))))))
       (if (and (< x 0) (< y 0))
           (setq z (+ x (/ (* y (+ 1000 x)) 1000))))
       (setf (fget (symbol-value hypothesis) 'Hypothesis 'ConfidenceFactor)
             z)
       ))
```

Der Regelinterpretierer arbeitet im *Forward Chaining Modus* und betrachtet alle anwendbaren Regeln so lange, bis keine neuen Fakten mehr herleitbar sind. Die Berechnung von neuen Slotwerten, die Ausgabe von Texten und auch die Ausführung beliebiger LISP Ausdrücke stehen als mögliche Aktionen zur Verfügung. Der grundsätzliche Aufbau ist recht einfach. Zuerst werden für alle ge*triggerten* Regeln sogenannte *Rule Activation Records* (RAR) angelegt, die Voraussetzungen, Aktionen und eine leere Variablenbindung enthalten.[11] Alle entstandenen RARs werden dann in eine Warteschlange `es` (für *Execution State*) eingereiht, die in einer Schleife bearbeitet wird:

[11] Hier ist noch eine Erweiterung geplant, die Variablenbindungen dauerhaft in dem zugehörigen Regel Frame speichert. Das erforderliche Strukturfeld ist schon vorhanden.

```
(defun run-rules (es)
  (until (empty-queue-p es)
    (enqueue-list (process-rule (dequeue es)) es)))
```

Die Funktion `process-rule` entfernt das erste Element aus der Warteschlange und bearbeitet es. Ergebnis ist eine Liste von neuen RARs, die aber auch leer sein kann. Dieser Fall tritt auf, wenn die bearbeitete Regel angewendet werden konnte. Ansonsten wird eventuell die Menge der Variablenbindungen erweitert und die Menge der Voraussetzungen reduziert. Aus einem RAR entstehen aber auch durchaus eine Vielzahl von neuen RARs, wenn Prädikate wie `member` angetroffen werden.

4.8 Informationsvererbung in DONALD

Im ersten Teil dieses Abschnitts wird ausgeführt werden, wie **DONALD** Nutzen aus der hierarchischen Anordnung von Frames zieht. Das Thema *Vererbung von Informationen* wurde bereits im vorangegangenen Kapitel im Zusammenhang mit semantischen Netzen und FRL angerissen worden.

4.8.1 Vererbungsstrategie in FRL

Zur Motivation der aktuellen Implementation soll hier ein kurzer Rückblick auf die Erfahrungen mit der früheren FRL-basierten Version **DONALD-I** gegeben werden.

Im Abschnitt 3.2 wurde ein kleines Beispiel mit der Funktion `fget-as-needed` gezeigt, in dem Informationen, die in einem in der AKO-Hierarchie höher postierten Frame gespeichert waren, für einen tiefer stehenden ausgenutzt wurden.

Bei der Diskussion in Abschnitt 3.5.1 auf Seite 103 wurden die mit FRL gemachten Erfahrungen bereits angesprochen. Da der Vergleich von kompletten Frames als Schlußfolgerungsmechanismus in **DONALD** überwiegt, werden jeweils die Werte *aller* Slots dieser Frames benötigt. Die Berechnung dieser hohen Zahl von Werten stellte sich als sehr rechenintensiv heraus.

Desweiteren fiel auf, daß die Unterscheidung der verschiedenen Zugriffsfunktionen eigentlich nur Effizienzgründe hatte. Man versucht einfach, für simple Fälle Rechenzeit einzusparen, anstatt *immer* die allgemeinsten Funktionen mit Durchsuchen der Vererbungshierarchie zu wählen. **DONALD-III** stellt dagegen nur *eine* generische Zugriffsfunktion zur Verfügung, die die volle Allgemeinheit der Vererbung abdeckt und *gleichzeitig* auf hohe Laufzeiteffizienz hin optimiert ist.

Typischerweise werden in **INVEST** zwischen zwei Benutzerinteraktionen circa 100 Frames verglichen. Während **DONALD-I**, das im großen und ganzen eine Implementation im Stil von FRL war, bis zu fünf Minuten für so viele Vergleiche benötigte, liegt die Zeit in der aktuellen Version unter einer Sekunde (ca. 10 ms pro Vergleich).

4.8.2 Vererbungsstrategie in DONALD

Das aufwendige Durchsuchen der Frame-Hierarchie nach brauchbaren Slotwerten während einer laufenden Konsultation von **DONALD-I** machte die Antwortzeiten des Expertensystems zu lang. Ein weiteres Problem von FRL war, daß nicht klar war, in welcher Situation welche der `fget`-x Funktionen anzuwenden war. Schließlich benötigen sie stark unterschiedliche Rechenzeiten und liefern unterschiedliche Ergebnisse. Die Semantik der Frame-Hierarchie war unklar: je nach Suchstrategie wurde ein entfernter VALUE einem nahen DEFAULT vorgezogen oder umgekehrt.

Das Berechnen der ererbten Slotwerte *während* einer Beratung ist nur dann nötig, wenn das Expertenwissen sich im Laufe einer Konsultation ändert. Für die Expertensystem-Shell **DONALD** wurde von der Prämisse ausgegangen, daß das in der Wissensbasis niedergeschriebene Wissen langlebiger ist und höchstens zwischen zwei Beratungen geändert werden muß. So können die Slotwerte, die sich für Instanzen und Teilklassen ergeben, im voraus errechnet werden. Während einer Beratung liegt die Datenbasis dann in *kompilierter* Form vor.

Genau diese Kompilation geschieht bei der Transformation eines Frames von seiner externen in seine interne Form. Das macht es möglich, den Quellcode eines Frames kurz und übersichtlich zu gestalten. Bei der Übersetzung wird dann automatisch die gesamte für einen Slot verfügbare Information gesammelt und in *einer* Struktur gespeichert. Nur noch *eine* Zugriffsfunktion ist nötig, um den Wert eines Slots zu erfahren:

(`fget`◁`frame`▷◁`view`▷◁`slot`▷)

Das Lesen eines Slots ist auf diese Weise wesentlich effizienter geworden. Das Vergleichen zweier Frames (das *Matchen*), das ja den Hauptinferenzmechanismus von **DONALD** darstellt und das eine Fülle solcher Leseoperationen umfaßt, ist deutlich schneller geworden.

Die folgende Abbildung 4.17 einer Frame-Hierarchie soll den noch zu beschreibenen Vererbungsalgorithmus illustrieren.

Die Instanz I1 hat via SC1 auch einen View auf C1 geerbt: (`fget` I1 'C1 'CS2) liefert den Default-Wert 2.0 als Ergebnis. (`fget` I1 'C1 'CS1) ergibt den aus SC1 ererbten Wert 1.5, der den Default 1.0 aus C1 überschreibt. (`fget` I2 'C1 'CS1) hat als Resultat 2.0, da sowohl der Default aus C1 als auch der ererbte Wert 1.5 aus SC1 durch die explizite Setzung des Slotwertes überschrieben wurden.

Die in **DONALD-III** implementierte Vererbungsstrategie geht nun in drei Schritten vor:

- Bei der Generierung von Teilklassen und Instanzen liest **DONALD** den Quellcode des Frames und transformiert zunächst den Quellcode der angegebenen (der *eigenen*) Views in Strukturen.
- Alle Views aus den Teilklassen, auf die eigene Views verweisen, werden in einer Liste gesammelt. Für jedes Element dieser Liste wird geprüft, ob der zu generierende Frame schon einen View gleichen Namens besitzt. Ist das nicht der Fall, wird der View für den neuen Frame übernommen. Andernfalls werden der eigene und der ererbte View zu einem zusammengefaßt. Das geschieht Slot-weise so, daß die Informationen im eigenen View erhalten bleiben und lediglich die Lücken aus dem ererbten View gefüllt werden.

```
(Class C1
   (Slots are
      (Real CS1 = 1.0)
      (Real CS2 = 2.0)))

(SubClass SC1
   (Slots are
      (Integer SCS1 = 1))
   (View C1 with
      (CS1 = 1.5)))

(Instance I1
   (View SC1))

(Instance I2
   (View SC1 with
      (SCS1 = 2))
   (View C1 with
      (CS1 = 2.0)
      (CS2 = 3.0)))
```

Abbildung 4.17: Eine Frame-Hierarchie in **DONALD-III**

- Im letzten Schritt werden noch verbliebenene Lücken in den eigenen Views aus den Default-Views der Klassen (vergl. Abschnitt 4.5.6), bzw. Teilklassen, auf die sie verweisen, gefüllt.

Der Algorithmus ist so konzipiert, daß die auf direktem Weg ererbten Daten sich gegen die, die auf indirektem Weg erhalten wurden, durchsetzen. Potentielle Probleme ergeben sich in den Fällen, in denen *keine* direkten Daten vorliegen, aber *mehrere* indirekte Daten, die sich gegeneinander ausschließen. Eine solche Situation wird als Konflikt bezeichnet.

Treten bei der Vererbung Konflikte auf, so werden diese dadurch gelöst, daß Views in der Bearbeitungsreihenfolge von links nach rechts im Source-Code bewertet werden. **DONALD-III** zieht also die zuerst genannten Views allen anderen vor und implementiert damit einen Spezialfall der *Preferential Inheritance* [Krishnaprasad, 1989].

Theoretische Eigenschaften, Korrektheit und Effizienz dieses Algorithmus werden im nächsten Abschnitt kurz angesprochen und dann in den Kapiteln 6 und 7 vertieft.

Die Anfrage (`fget` Tweety 'Tier 'kann-fliegen) ergibt *nein*, weil der View auf *Tier* mit der Slotsetzung *nein* vor dem View auf *Vogel* spezifiziert wurde. Durch den View auf *Vogel* wäre ansonsten die Slotsetzung *ja* ererbt worden.

4.8.3 Korrektheit der Vererbung

Übersetzt man den in **DONALD** eingesetzten Frame-Formalismus zur Wissensrepräsentation in ein Semantisches Netzwerk und damit in eine mathematisch wohldefinierte Darstellung als gerichteten Graphen, ergibt sich die Möglichkeit einer exakten Analyse der Eigenschaften der verwendeten Formalismen und Algorithmen.

Ergebnis der Übersetzung ist ein Netzwerk $\Gamma = \langle I, K, P, E^+, E^- \rangle$ mit Kanten und Knoten unterschiedlichen Typs, wobei:

(i) I ist eine nichtleere Menge von Individuen.

```
(Class Tier
  (Slots are
    ((Oneof ja nein) Kann-fliegen)))

(SubClass Vogel
  (Slots are
    ((Oneof ja nein) Hat-Flügel = ja))
  (View Tier with
    (Kann-fliegen = ja)))

(SubClass Nicht-fliegender-Vogel
  (View Tier with
    (Kann-fliegen = nein))
  (View Vogel))

(Instance Tweety
  (View Nicht-fliegender-Vogel))
```

Abbildung 4.18: Ein Vererbungsbeispiel

(ii) K ist eine nichtleere Menge von Klassen.

(iii) P ist eine Menge von Eigenschaften *(Properties)*.

(iv) $E^+ \subseteq (I \cup K) \times (K \cup P)$ ist eine Menge von positiven Kanten.

(v) $E^- \subseteq (I \cup K) \times (K \cup P)$ ist eine Menge von negativen Kanten.

Die Transformation einer Menge von Individuen mit Slots, Klassen und Teilklassen in der Terminologie des Systems **DONALD** geht dazu folgendermaßen vor sich:

1. Individuen werden zu Elementen von I.

2. Klassen werden zu Elementen von K.

3. Die von den Klassen und Individuen spezifizierten Oberklassen definieren eine Relation *Teilklasse* auf $I \cup K$, die mit dem Symbol $\nearrow$ bezeichnet wird. Gilt $i \nearrow k$ für $i \in I \cup K$ und $k \in K$, dann wird das Paar $\langle i, k \rangle$ in E^+ aufgenommen.

4. Slots werden zu Elementen von P (Eigenschaften). Slots haben aber noch einen Typ mit einem möglicherweise komplexen Wertebereich. Im Rahmen der Analyse der Vererbung spielen Typ und Wert aber keine Rolle, da es nur noch um den abstrakten Begriff des *Konflikts* geht. Ein Konflikt liegt immer dann vor, wenn nicht übereinstimmende Information aus verschiedenen Klassen geerbt werden kann. Es genügt also, nur die Fälle Übereinstimmung und Konflikt zu unterscheiden und mit positiv bzw. negativ zu bezeichnen.

5. Für jeden Slot $p \in P$ innerhalb der Klasse oder des Individuums $e \in I \cup K$ werden positive oder negative Verbindungen $e \to p$ in E^+ bzw. E^- aufgenommen. Dabei soll $e \to p$ nur eine suggestive Schreibweise für $\langle e, p \rangle \in E^+$ und $e \not\to p$ für $\langle e, p \rangle \in E^-$ sein. Die Verbindungen werden positiv übersetzt, solange bis Konflikte auftreten. Im Fall eines Konflikts mit einem bereits bestehenden Wert wird der widersprechende Wert durch eine negative Verbindung dargestellt.

Aus dieser Übersetzung folgt schon eine wichtige Beobachtung. Für die Kanten $\sigma = i \to k$, die sich im dritten Punkt der obigen Vorschrift ergeben, gilt:

$\sigma \in (I \cup K) \times K$. In Punkt fünf ergeben sich zwar neue Kanten für E^+ und E^-, aber die Zielelemente der neuen Verbindungen τ sind niemals Klassen, sondern immer Eigenschaften. Deshalb gilt: $\tau \in (I \cup K) \times P$. Aufgrund dieser Eigenschaft der transformierten Framehierarchie läßt sich Unterpunkt (v) der obigen Definition abschwächen:

(v) $E^- \subseteq (I \cup K) \times P$ ist eine Menge von negativen Kanten.

Damit erfüllen die aus Frames hervorgehenden Übersetzungen die Voraussetzungen einer sogenannten *Tangled Hierarchie* (siehe Definition 6.30 auf Seite 211). Zusammen mit Baumstrukturierten Netzwerken und Class-Property Systemen bilden sie wichtige Einschränkungen Semantischer Netzwerke. Die weitere Ausarbeitung dieser Ideen und der allgemeinen mathematischen Grundlagen der Vererbung von Informationen folgt in den nächsten beiden Kapiteln.

4.9 Das Matchen von Frames

In Abschnitt 3.3 wurde bereits auf die Bedeutung des Vergleichens von Frames als Schlußfolgerungsmechanismus hingewiesen:

> Reasoning is dominated by a process of recognition in which new objects and events are compared to stored sets of expected prototypes ... [Bobrow and Winograd, 1977, S. 5].[12]

In diesem Abschnitt wird beschrieben werden, wie dieses Vergleichen in **DONALD** geschieht.

Die Funktion *Match* kann auf je zwei beliebige Lisp-Ausdrücke (z.B. Frames, Views, Slots, Listen oder Atome) angewandt werden. Um zwei Frames miteinander zu vergleichen, berechnet *Match* rekursiv die Ähnlichkeit der gemeinsamen Views und darin die Ähnlichkeit der Slotsetzungen. Als Ergebnis liefert *Match* eine ganze Zahl zwischen -1000 und $+1000$. Dabei bedeutet -1000,

[12] Denken ist ein Prozeß des Wiedererkennens, der bestimmt wird durch das Vergleichen neuer Objekte und Ereignisse mit gespeicherten Mengen erwarteter Prototypen.

daß die Frames völlig verschieden sind, und +1000, daß sie, soweit es ihre gemeinsamen Views betrifft, gleich sind.

Der Algorithmus zum Vergleichen zweier Frames F_1 und F_2 sieht wie folgt aus:

1. Bilde die Listen VL_1 und VL_2 der Views von F_1 und F_2.

2. Jeder View ist mit einem Gewicht, d.h. einer Zahl $w \geq 1$ (Standardsetzung ist $w = 1$) versehen, die es ermöglicht, innerhalb eines Frames nach wichtigen und weniger wichtigen Gesichtspunkten zu differenzieren (siehe Tabelle 4.3 auf Seite 133).

   Wende *match* auf alle Paare $[V_1, V_2]$ (mit $V_1 \in VL_1$ und $V_2 \in VL_2$) von Views auf dieselbe (Teil-) Klasse an. Das Resultat des Matchens zweier Views ist das gewichtete Mittel der Match-Resultate der Slots (s.u.). Das Gewicht des Ergebnisses ist das Maximum der Gewichte von V_1 und V_2.

   Gibt es kein Paar $[V_1, V_2]$ von Views auf dieselbe (Teil-) Klasse, gib als Ergebnis 0 zurück.

   Sonst bilde das gewichtete arithmetische Mittel aller Ergebnisse und gib es als Maß für die Ähnlichkeit von F_1 und F_2 zurück.

3. Jeder Slot ist mit einem Gewicht, d.h. einer Zahl $w \geq 1$ (Standardsetzung ist $w = 1$) versehen, die es ermöglicht, innerhalb eines Views nach wichtigen und weniger wichtigen Merkmalen zu differenzieren (siehe Tabelle 4.4 und Abschnitt 4.6.1).

   Wende *Match* auf alle Paare $[S_1, S_2]$ (mit S_1 in V_1 und S_2 in V_2) von gleichnamigen, gesetzten Slots an, d.h. Slots, deren Werte ungleich `NIL` sind. Das Gewicht des Ergebnisses ist das Maximum der Gewichte von S_1 und S_2.

   Gibt es kein Paar $[S_1, S_2]$ von gleichnamigen gesetzten Slots, gib als Ergebnis 0 zurück.

   Sonst bilde das gewichtete arithmetische Mittel aller Ergebnisse und gib es als Maß für die Ähnlichkeit von V_1 und V_2 zurück.

4. Der Vergleich zweier Slots geschieht durch das Matchen der Werte der Slots. Je nach Typ des Slots stehen dafür schon spezielle Funktionen

(z.B. *Match-String, Match-Integer, Match-Oneof, Match-HList*), die eigens zum Vergleich von Werten dieses Typs geschaffen worden sind, zur Verfügung. Definiert der Anwender neue Typen, so kann er für sie ebenfalls spezielle Matchfunktionen implementieren und in **DONALD** einbinden.

Die Darstellung der Frames als Strukturen, die alphabetische Anordnung der Views nach den Namen ihrer definierenden Klasse und die Vorausberechnung aller ererbten Informationen dienen dazu, das Matchen möglichst schnell zu machen [Koch und Heuer, 1986], [Koch und Heuer, 1987].

4.10 Prototypen und Hypothesen

Im Abschnitt 4.3 ist der Einsatz von Prototypen und Hypothesen motiviert worden. Im folgenden wird die Art und Weise ihrer Implementation im Detail beschrieben werden.

4.10.1 Hypothesen

Unter Hypothesen in **DONALD** sind Instanzen der Klasse *Hypothesis* zu verstehen. Das *Auslösen* und *Überprüfen* von Hypothesen ist der wichtigste Inferenzmechanismus der Expertensytem-Shell **DONALD**.

Das Auslösen von Hypothesen (meist *Triggern* genannt) war ja schon im Zusammenhang mit den eingebauten *Facetten* angesprochen worden. Es geschieht während des Sammelns von Informationen. Zum Beispiel lösen Antworten des Benutzers auf Fragen des Expertensystems Hypothesen aus, d.h. diese Hypothesen werden vorgemerkt, um später überprüft zu werden. So soll dafür gesorgt werden, daß nur Hypothesen, für die zumindest ein Anhaltspunkt spricht, ausgewertet werden. Andernfalls verschwendete man Rechenzeit auf die Überprüfung völlig aussichtsloser Kandidaten.

Die Klasse *Hypothesis* ist in Abbildung 4.19 dargestellt.

```
(Class Hypothesis
   (Slots are
          ((ListOf Hypothesis) MoreGeneral)
          ((ListOf Hypothesis) MoreSpecific)
          ((ListOf Hypothesis) IncompatibleWith)
          (Symbol              Prototype)
          ((ListOf Symbol)     Proposition)
          (Symbol              CompareWith)
          (Other               IfConfirmed)
          (Other               IfDisproved)
          (Integer             TriggerPoints)
          (Integer             ConfidenceFactor)))
```

Abbildung 4.19: Die Klasse Hypothesis

Der Slot *TriggerPoints* nimmt als Wert eine ganze Zahl ≥ 0 auf, die angibt, wie oft die entsprechende Hypothese ausgelöst worden ist.

Die Slots *MoreGeneral* und *MoreSpecific* dienen dazu, für jede Hypothese die Listen mit den Namen der Vorgänger und der Nachfolger, d.h. der allgemeineren bzw. der spezielleren Hypothesen, aufzunehmen.

Aus den einfachen Hypothesen, die vom Expertensystem für richtig erachtet werden, versucht es durch Kombination komplexere Gruppen aufzubauen, die zu Lösungsvorschlägen führen. Um diese Gruppierungen zu ermöglichen, muß bekannt sein, welche Hypothesen mit welchen anderen "verträglich" sind. Dazu dient der Slot *IncompatibleWith*, der zu jeder Hypothese eine unter Umständen leere Liste der Hypothesen aufnehmen kann, die nicht mit ihr kombinierbar sind.

Der Slot *Prototype* enthält einen oder mehrere Namen von Frames, die zur Überprüfung der Hypothese benutzt werden sollen.

Ist dort nur ein Name angegeben, wird der Frame mit diesem Namen mit dem Frame, der im Slot *CompareWith* genannt ist, durch die Funktion *Match* verglichen (siehe Abschnitt 4.9). Das Ergebnis dieses Vergleiches wird im Slot

ConfidenceFactor gespeichert und repräsentiert die Größe der Gewißheit, mit der das Expertensystem diese Hypothese “glaubt”.

Ist der Inhalt des Prototype-Slots von der Form (And *Liste von Framenamen*) bzw. (Or *Liste von Framenamen*), werden alle angegebenen Frames wie zuvor durch *Match* überprüft und das Minimum bzw. das Maximum der Ergebnisse als Konfidenzfaktor gespeichert.

Ist der Konfidenzfaktor größer bzw. kleiner als ein voreingestellter oberer bzw. unterer Schwellenwert, gilt die Hypothese als bestätigt bzw. als verworfen.

In den Slots *IfConfirmed* und *IfDisproved* kann beliebiger Lisp-Quellcode gespeichert werden, der für den Fall, daß die Hypothese bestätigt bzw. verworfen wurde, ausgeführt wird.

Der Slot *Proposition* enthält eine Liste von Views, der Syntax für Views in Instanzen und Teilklassen entsprechend, die zum stückweisen Aufbau von Lösungvorschlägen benutzt werden, wenn die Hypothese bestätigt worden ist. Mehr über die Konstruktion von Lösungen folgt in einem späteren Abschnitt.

Die Abbildung 4.20 zeigt eine Hypothese aus dem Expertensystem **INVEST**. Dabei handelt es sich um die Hypothese `RENTENF`, d.h. den Vorschlag, einen Anteil an einem Investmentfonds, der auf festverzinsliche Wertpapiere spezialisiert ist, zu kaufen.

```
(Instance .RENTENF
  (LongName is “Rentenfonds”)
  (View Hypothesis with
        (MoreGeneral are (List .INVESTMENT .AUSLF .THESF))
        (MoreSpecific are (List .RENTENFKL .RENTENFLL))
        (IncompatibleWith are (List .AKTIENF .MISCHF))
        (Prototype is RENTENF)
        (Proposition is (List (View Rentenfonds)))
        (ConfidenceFactor is 0)))
```

Abbildung 4.20: Die Instanz .RENTENF

Die Einordnung dieser Hypothese in einen größeren Kontext zeigt die Abbildung 4.21, die einen Ausschnitt aus einem semantischen Netz zeigt, daß durch die Verweise in den Slots *MoreGeneral, MoreSpecific, IncompatibleWith* und *Prototype* geknüpft wird. In der Abbildung sind der Übersichtlichkeit halber nicht alle Kanten eingetragen.

Das Beispiel zeigt, daß Frame-basierte Systeme eine echte Verallgemeinerung des Ansatzes mit semantischen Netzen sind. Die *is-a-* bzw. *ako*-Kanten (in **DONALD** sind das die Kanten zwischen Instanzen, Teilklassen und Klassen) sind zwar wegen ihrer besonderen Bedeutung für die Vererbung hervorgehoben, aber auch andere Arten von Kanten lassen sich durch entsprechende Einträge in Slots leicht realisieren.

4.10.2 Prototypen

Die Prototypen in **DONALD** sind Instanzen von Klassen, die Situationen abbilden.

Ein Beispiel aus **INVEST** soll das verdeutlichen: Der Benutzer von **INVEST** ist ein potentieller Wertpapierkäufer, ein Anleger. Die Klasse *Anleger* enthält die Slotdefinitionen, die nötig sind, um die näheren Umstände darzustellen, in denen sich der Benutzer befindet, unter dem Gesichtspunkt, daß er ein Anleger ist.

Zu Beginn einer Konsultation wird ein neuer Frame mit einem vom Benutzer gewählten Namen geschaffen. Dieser Name wird in den getriggerten Hypothesen in den CompareWith-Slot eingetragen. Zunächst enthält dieser Frame nur einen View auf die Klasse Anleger. Dieser View wird durch Antworten des Benutzers gefüllt und gibt einen ersten Eindruck der Benutzersituation wieder.

Prototypen sind Instanzen, die spezielle Situationen verkörpern. Zum Beispiel gibt es in **INVEST** Prototypen, die ebenfalls einen View auf die Klasse Anleger haben und für bestimmte Anlagearten typische Situationen darstellen. So bilden zum Beispiel die Instanzen *AKT, RENTE* und *KURSN* prototypische Anleger für Aktien, Rentenpapiere und kursneutrale Anlagepapiere ab.

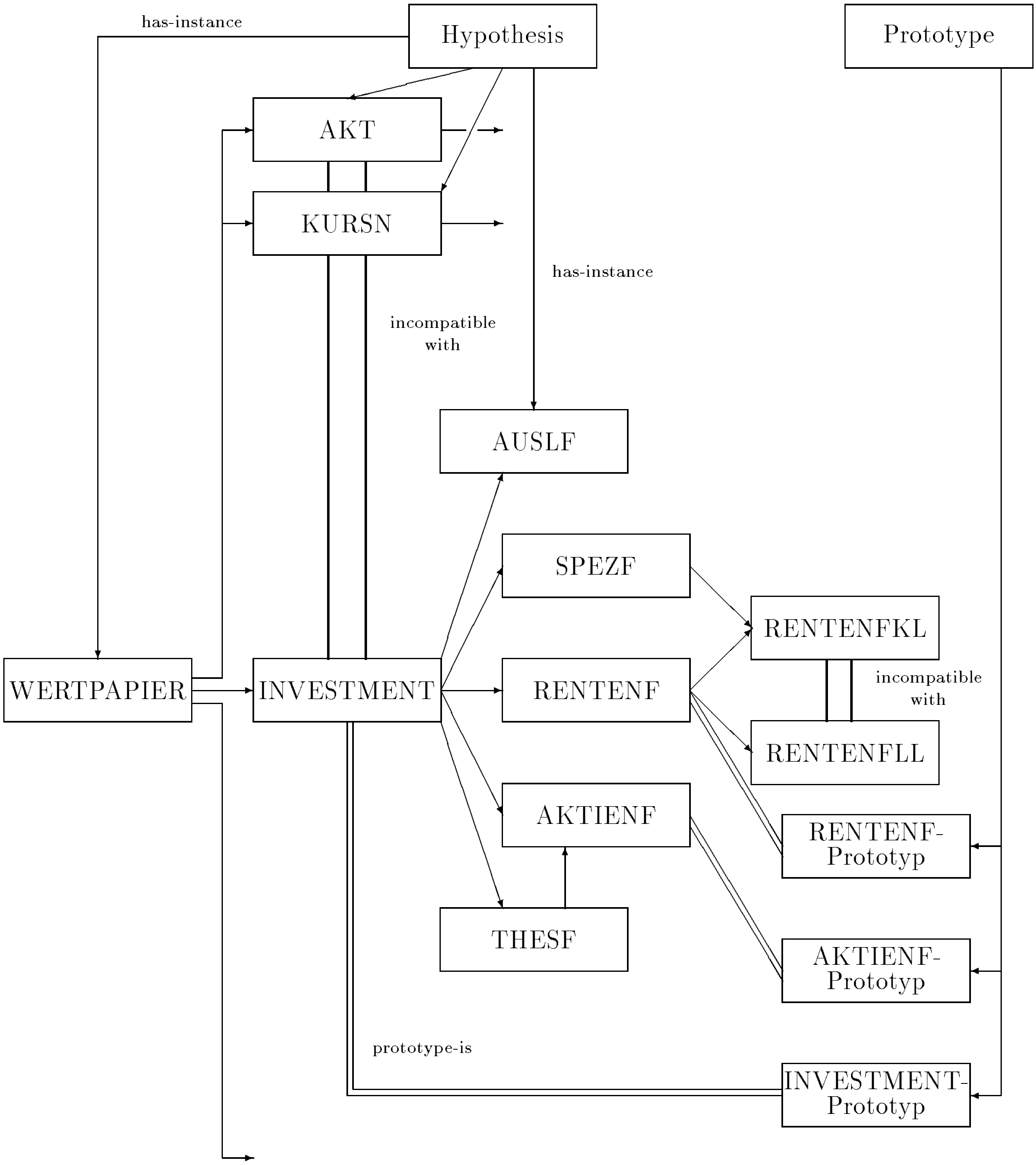


Abbildung 4.21: Ein semantisches Netz von Hypothesen und Prototypen

```
(Instance Kunde-1
  (View Anleger with
        (AnlageDauer = mittel)
        (AnlageBetrag = mittel)
        (AnlageZweck = Vermögensstreuung)
        (KundenVorstellung = Zinspapiere)
        (RenditeErwartung = normal)
        (RisikoBereitschaft = klein)
        (AnlageVerfügbarkeit = nein)
        (KundenSteuerproblem = ja)))
```

Abbildung 4.22: Die Instanz Kunde-1

Die getriggerten Hypothesen bestimmen durch ihre Prototype-Slots, welche Prototypen mit dem Frame, der die Benutzersituation zeigt, ge*match*t werden. Die Größe der Ähnlichkeit zwischen Benutzer-Frame und den Prototyp-Frames ist entscheidend dafür, welche Hypothesen bestätigt bzw. verworfen werden. Abbildung 4.22 auf Seite 156 zeigt einen typischen Benutzer-Frame nach der ersten Runde einer Konsultation. Abbildung 4.23 auf Seite 157 zeigt den Prototyp *KURSN*.

4.10.3 Der Datentyp HList

Mit den eingebauten Datentypen *Interger, Float, Symbol, String, Fuzzyset, OneOf,* usw. liefert die Funktion *Match* auf Slots entweder den Wert -1000 oder +1000 und zeigt damit vollständige Übereinstimmung oder ihr Gegenteil an.

Bei der Entwicklung von **INVEST** zeigte sich, daß das in vielen Situationen ein zu grobes Maß ist. Auch bei Slots sollte auf Ähnlichkeit und nicht auf völlige Übereinstimmung geprüft werden.

Ein Beispiel aus **INVEST** soll das veranschaulichen: Eine der Einstiegsfragen bei jeder Konsultation von **INVEST** ist die Frage nach dem Zweck der ge-

```
(Instance KURSN
  (LongName is "kursneutral")
  (View Anleger with
        (AnlageDauer ist
            (HList (ich weiss noch nicht . 350)
            (lang . 250)
            (mittel . 1000)
            (kurz . 1000))
            weight 2)
        (AnlageZweck ist
            (HList (Vermögensansammlung . 1000)
            (Vermögensstreuung . 500)
            (Spekulation . -500))
            weight 1)
        (KundenVorstellung ist
            (HList (nein . 250)
            (Optionen . -1000)
            (Investment . -500)
            (Zinspapiere . 1000)
            (Aktien . -1000))
            weight 3)
        (RenditeErwartung ist
            (HList (ich weiss noch nicht . 250)
            (überdurchschnittlich . -750)
            (normal . 1000))
            weight 2)
        (RisikoBereitschaft ist
            (HList (ich weiss noch nicht . 0)
            (überdurchschnittlich . -500)
            (normal . 500)
            (klein . 1000))
            weight 4)))
```

Abbildung 4.23: Der Prototyp KURSN

planten Anlage in Wertpapieren. Die Antwortmöglichkeiten sind *Vermögensansammlung, Vermögensstreuung* und *Spekulation.*

Ist nun die Wertpapiergattung A besonders gut zum Ansparen von Vermögen geeignet, würde der Slot *AnlageZweck* des A-Prototypen den Wert *Vermögensansammlung* enthalten. Es ist aber nicht unwahrscheinlich, daß die Wertpapiere der Gruppe A auch zum Risikostreuen, zwar nicht optimal, aber gut geeignet sind. Die Benutzerantwort *Vermögensstreuung* würde aber ohne Rücksicht darauf beim Matchen der Slots -1000 ergeben.

Die Lösung, im A-Prototyp in diesem Slot mit *(Or Vermögensansammlung Vermögensstreuung)* beide Alternativen vorzusehen, würde dazu führen, daß bei beiden Antworten das Ergebnis des Matchens +1000 wäre. So kommt aber das Wissen, daß Wertpapiere der Gattung A zum Ansammeln besser geeignet sind als zum Streuen, nicht zum Ausdruck.

Für diese Fälle ist der Datentyp *HList* (Heuristic List) vorgesehen. Eine HList ist eine Liste von Paaren, deren erstes Element ein möglicher Slotinhalt ist und deren zweites Element der korrespondierende Wert der Match-Funktion ist (vergleiche Abbildung 4.23 in Abschnitt 4.10.2).

Beim Matchen dieses Prototyps gegen den Frame Kunde-1 (siehe die Abbildung 4.22) steuert der Slot *AnlageZweck* einfach gewichtete 500 Punkte bei.

Die numerischen Werte in den HLists sind mit einer groben Semantik versehen. Die neun Vielfachen von 250 (-1000, -750, ..., 750, 1000) repräsentieren die Einschätzungen „paßt überhaupt nicht“ bis „paßt sehr gut“.

Durch HLists wird das Matchen aussagefähiger. Sie definieren *Umgebungen* von Antworttupeln um den idealen Prototypen. Alle Tupel, die in einer solchen Umgebung liegen, gelten als zum Prototyp ähnlich. Mit HLists wird zum Ausdruck gebracht, daß eine Reihe von schwächeren Ähnlichkeiten ebenso gut auf einen Protoypen hinweisen können wie einige wenige starke Übereinstimmungen.

Schon die Antworten auf die Einstiegsfragen in **INVEST** lassen circa 17000 verschiedene Antwortkombinationen zu. Die HLists machen es möglich, diese Kombinationen mit einer relativ geringen Anzahl von Prototypen zu klassifizieren. Ein aus Gründen der graphischen Darstellbarkeit stark vereinfachtes Modell, in dem Anlageempfehlungen nur von der Anlagedauer, dem Anlagezweck und der Risikobereitschaft abhängig gemacht werden, zeigt Abbildung 4.24. Die darin angegebenen Zahlenwerte sind grob gerundete *ConfidenceFactors*.

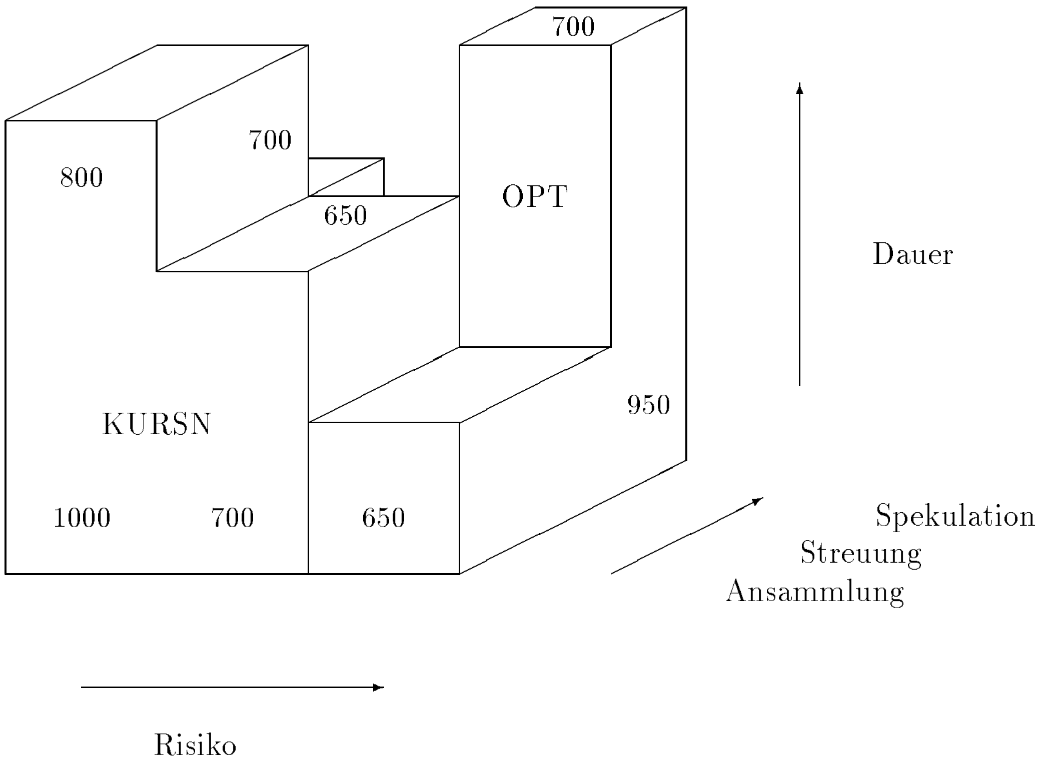


Abbildung 4.24: Umgebungen der Prototypen OPT und KURSN

HLists sind vor allem für Antwortauswahlen vorgesehen, die sich nicht auf einer Skala anordnen lassen. Bei Fragen nach Quantitäten (Dauer, Betrag, Rendite, Gewicht, Länge etc.) wurden schon spezielle Match-Funktionen implementiert. Zum Beispiel können Slots vom Typ *Integer* und *Real* mit der speziellen Match-Funktion `nummatch` verglichen werden.

Für zwei Zahlen x und y ist

$$\texttt{nummatch}(x, y) \quad := \quad \begin{cases} 1000 & \text{falls } x = 0, y = 0 \\ 1000 * [1 - 2\frac{|x-y|}{|x|+|y|}] & \text{sonst} \end{cases}$$

Diese Funktion berücksichtigt neben der Differenz der beiden Zahlen auch ihre Größenordnung. So ist `nummatch`(1,100) = -960 und `nummatch`(1001,1100) = +906.

Um Slots auf Ähnlichkeit zu prüfen, gibt es also zwei Möglichkeiten. Entweder man definiert eine Match-Funktion speziell für Daten eines besonderen Typs, oder man benutzt HLists. HLists lassen sich als Spezialfall der ersten Möglichkeit auffassen, weil sie einer Wertetabelle für eine spezielle Match-Funktion entsprechen.

HLists sind vielleicht die weniger elegante Lösung, aber sie sind ein allgemein anwendbarer Formalismus, der die für das Programmieren einer speziellen Match-Funktion nötigen Lisp-Kenntnisse entbehrlich macht.

4.11 Die Synthese der Lösung

Im Abschnitt 4.10.1 wurde bereits erwähnt, daß **DONALD** versucht, aus den einfachen bestätigten Hypothesen größere Gruppen aufzubauen. Dies ist einer der Punkte, in denen **DONALD** sich von herkömmlichen Expertensystem-Shells zur Diagnose und Klassifikation unterscheidet, deren Ziel es ist, ein Objekt aus einer im voraus bekannten Menge zu finden (Fehlerdiagnose) oder ein Objekt einer im voraus bekannten Klasse zuzuordnen.

DONALD konstruiert aus den Hypothesen-Bausteinen Lösungsvorschläge, wobei die Freiheit dieser Konstruktion nur durch das Wissen darüber, welche Hypothesen sich gegenseitig ausschließen, beschränkt wird. Die Überlegung, die zu dieser Konzeption geführt hat, war, daß es ökonomischer ist, nicht alle möglichen Kombinationen von Hypothesen als Objekte zu repräsentieren, sondern lieber das Wissen darüber, wie zulässige Kombinationen gebildet werden können, abzubilden.

So können zum Beispiel in **INVEST** aus den circa 30 terminalen Hypothesen (i.e. den Hypothesen, die keine spezielleren Nachfolger mehr haben) die rund 100 wichtigsten Gruppen von Wertpapieren geformt werden.

In diesem Abschnitt wird beschrieben werden, wie der Aufbau von Lösungen vollzogen wird.

Eine Hypothese V heißt *Vorgänger* der Hypothese H, wenn sie im Slot *More-General* von H aufgeführt ist.

Eine Hypothese A heißt *Ahne* der Hypothese H, falls

1. A ein Vorgänger von H ist, oder falls
2. es einen Vorgänger von H gibt, dessen Ahne A ist.

Zwei Hypothesen H_1 und H_2 heißen *inkompatibel*, wenn

1. im Slot *IncompatibleWith* der Hypothese H_i die Hypothese H_j aufgeführt ist, wobei $i, j \in \{1, 2\}, i \neq j$; oder
2. wenn im *IncompatibleWith*-Slot eines Ahnen H_A von H_i eine Hypothese H_B genannt ist, die ein Ahne von H_j ist, $i, j \in \{1, 2\}, i \neq j$.

Ein (partieller) Lösungsvorschlag LV ist eine Liste, die

1. die Liste mit dem Namen der allgemeinsten Hypothese (der *Wurzelhypothese*) als einzigem Element enthält und
2. die n ($n \geq 0$) weitere Elemente enthält, die geordnete Paare (Listen aus zwei Elementen) von Namen bestätigter[13] Hypothesen sind. Dabei ist der erste Name der Name des Paares und gibt an, für welche Hypothese das Paar steht. Der zweite Name ist der Name des Vorgängers, d.h. der Hypothese, als deren Spezialisierung die erste bestätigt wurde.
3. Jede Hypothese, die in einem Paar als zweites Element auftritt, muß einmal auch als erstes Element eines Paares auftreten (*Fundiertheit*).
4. Unter allen in LV vorkommenden Hypothesen dürfen nicht zwei auftreten, die inkompatibel sind (*Konsistenz*).

[13] **DONALD** betrachtet eine Hypothese als bestätigt, wenn sich bei dem Vergleich der durch die Hypothese dargestellten prototypischen Situation mit der aktuellen Situation ein ausreichend hoher Wert ergibt, der standardmäßig bei 60% liegt, aber jederzeit dynamisch verändert werden kann.

Eine Liste, die diesen Bedingungen genügt, heißt *fundierter, konsistenter, (partieller) Lösungsvorschlag.* Ein Lösungsvorschlag LV heißt *maximal-konsistent*, wenn es kein noch nicht in LV enthaltenes Paar $P = (H, V)$ aus einer bestätigten Hypothese H und ihrem Vorgänger V gibt, so daß P zu allen Paaren in LV kompatibel wäre. Zwei Paare sind dabei kompatibel, wenn die Hypothesen H kompatibel sind. Die Vorgänger V spielen keine Rolle mehr.

Zu Beginn einer Konsultation von **DONALD** besteht die Liste der partiellen Lösungsvorschläge nur aus einem Element. Das ist der Lösungsvorschlag, der als einziges Element die Liste enthält, die nur aus der Wurzelhypothese WH besteht. Diese gibt durch ihren *MoreSpecific*-Slot die Hypothesen $NH_1, ..., NH_n$ an, die in der ersten Runde der Konsultation überprüft werden sollen. Aus den bestätigten Hypothesen $NH_{i_1}, ..., NH_{i_m}, m \leq n$, und ihren Vorgängern (in diesem Fall stets WH) werden die Paare

$$(NH_{i_1}, WH), \ldots, (NH_{i_m}, WH)$$

gebildet, die durch die partiellen Lösungsvorschläge erweitert werden sollen. Das geschieht dadurch, daß die Paare der Reihe nach mit dem *Propose*-Algorithmus bearbeitet werden.

Die Routine *Propose* nimmt als Argumente ein Paar $P = (H, V)$ aus einer bestätigten Hypothese H und ihrem Vorgänger V, sowie die Liste der partiellen Lösungsvorschäge LLV, soweit sie bisher aufgebaut worden ist.

Für jeden partiellen Lösungsvorschlag LV in LLV versucht Propose, LV um P zu erweitern. Taucht V in LV nicht auf, oder ist P schon in LV vertreten, geht Propose weiter zum nächsten noch nicht bearbeiteten Lösungsvorschlag.

Andernfalls bildet Propose aus allen in LV auftretenden Hypothesen, die zu H inkompatibel sind, die Konflikt-Menge CS und aus den übrigen die Differenz-Menge DS. Ist CS leer, wird P zum Lösungsvorschlag LV hinzugefügt. Im anderen Fall wird DS um P erweitert. Ist diese neue Liste fundiert und nicht bereits in LLV enthalten, so wird sie als weiterer Lösungsvorschlag zu LLV hinzugefügt.

Sind alle Paare bearbeitet worden, folgt ein weiterer Test: Enthält ein Lösungsvorschlag LV ein Paar (H, V), für das kein Nachfolgepaar (N, H) in LV existiert, obwohl alle direkten Nachfolger N (d.h. die im *MoreSpecific*-Slot von

H aufgeführten Hypothesen) schon geprüft worden sind, so wird LV aus LLV entfernt. So werden bestätigte Hypothesen, deren sämtliche Nachfolger bei der Überprüfung unbestätigt geblieben sind, von der Konstruktion der Lösungsvorschläge ausgeschlossen. Ebenso werden Lösungsvorschläge, die aufgrund von Inkompabilitäten nicht mehr erweitert werden können, aus der Liste der Lösungsvorschläge gelöscht.

So kann es sein, daß zu Beginn und in der Mitte einer Konsultation die Anzahl der Lösungsvorschläge anwächst und gegen Ende wieder abnimmt oder sogar wieder zu Null wird. Diese Fähigkeit, Arbeitshypothesen zu bilden, und sie auch wieder verwerfen zu können, diese sogenannte *Nichtmonotonie*, ist eines der wichtigsten Konstruktionsmerkmale der Expertensystem-Shell **DONALD** und wird noch Gegenstand eines folgenden Abschnitts sein.

Der Propose-Algorithmus liefert nach jeder Konsultationsrunde eine Liste von maximal-konsistenten, fundierten (partiellen) Lösungsvorschlägen. Die Views, die in den *Proposition*-Slots der Hypothesen, die in einem Lösungsvorschlag auftreten, angeführt sind, werden mit Hilfe der `add-view` Funktion der Liste der Proposals hinzugefügt. Jede dieser Instanzen *Proposal*-n, $n \geq 1$ enthält eine vom Expertensystem vorgeschlagene Lösung, die z.B. wie folgt aussehen kann:

```
(Instance Proposal-1                  ; Die Hypothesen "aus-
   (View Proposal)                    ; ländisch" und "Aktien"
   (View Wertpapier with              ; wurden revidiert. Die
        (Herkunft = Inland)           ; bestätigten Hypothesen
        (Risiko = enorm)              ; "Option" und "Option/Op-
        (börsengängig = ja))          ; tionsschein" haben Views
   (View Börsengängigeswertpapier)    ; zu diesem Proposal bei-
   (View Terminpapier)                ; getragen.
   (View Optionen))
```

4.12 Nichtmonotones Schließen

In diesem Abschnitt wird das nichtmonotone Vorgehen der Expertensystem-Shell **DONALD** noch einmal näher untersucht.

Das nichtmonotone Verhalten beruht auf im Programm berücksichtigten Heuristiken. Somit existiert keine formale Syntax, nach der dieses Schließen stattfände, abgesehen von der formalen Darstellung in der Programmiersprache. Zum Verständnis dieses Prozesses wäre eine formale Semantik wünschenswert, und zur Rechtfertigung der Schlüsse eine dazu passende Syntax.

Dieses Problem wird in [Koch, 1989a], basierend auf einer von Dov Gabbay [Gabbay, 1982] vorgeschlagenen Semantik, angegangen und eine nichtmonotone Logik J4 definiert, die als Grundlage des Schließens dienen soll.

Ein neuerer interessanter Ansatz in dieser Richtung ist in [Lamarre, 1991] beschrieben. Basis des Ansatzes von Lamarre ist eine monotone Logik mit einer Präferenzrelation, die auf der Menge ihrer Interpretationen definiert ist. In monotoner Logik korrespondiert eine Formel A semantisch mit der Menge aller Interpretationen, in denen A wahr ist. Um eine nichtmonotone Logik zu erhalten, kam er auf die Idee, nicht *alle* Interpretationen zu betrachten, sondern nur die in einem gewisse Sinne bevorzugten *(preferred)*. Lamarres Ansatz hat direkten Bezug zu der in Abschnitt 2.3.11 auf Seite 49 betrachteten Modallogik $S4$, da auch die Modallogik binäre Relationen auf möglichen Welten in der Definition ihrer Semantik verwendet.

Unter Verwendung der in [Koch, 1989a] formal definierten Begriffe soll an dieser Stelle die Nichtmonotonie von **DONALD** (eher informell) beschrieben werden. Die Problematik inkompatibler Lösungen soll dabei der Einfachheit halber ausgelassen werden. Wie im **INVEST**-Beispiel soll eine Menge von Hypothesen als Baum angeordnet sein (siehe Abbildung 4.25).

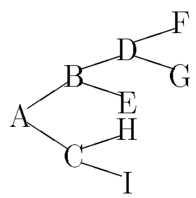


Abbildung 4.25: Hierarchie von Hypothesen

Diese Baumstruktur wird durch die folgenden Axiome repräsentiert:

1. $A \rightarrow B \vee C$

2. $B \to D \vee E$
3. $C \to H \vee I$
4. $D \to F \vee G$

Das bedeutet z.B.: wenn die Hypothese A bewiesen ist, ist B oder C bewiesen.

Als neue Bezeichnung dafür, daß eine Hypothese A "geglaubt" wird, wird $\diamond A$ verwendet. Syntaktisch steht $\diamond A$ als Abkürzung dafür, daß das Matchen des (oder der) Prototypen für A mit dem Vergleichs-Frame einen hinreichend großen Wert ergeben hat. $\diamond A$ soll deshalb gelesen werden als: "es ist widerspruchsfrei, A anzunehmen".

Zu den oben angeführten Axiomen kommt noch für jeden Knoten das Axiom $\diamond K \to K$ hinzu. Diese Axiome entsprechen in etwa den *Normal Defaults* in Reiters *Default Reasoning*[14] [Reiter, 1980].

Eine Formel A soll *nichtmonoton herleitbar* heißen (in Zeichen $\mid\!\sim A$), wenn sie aus den Axiomen unter Rückgriff auf Annahmen der Form $\diamond B$ bewiesen werden kann. Dabei müssen die Annahmen mit dem bisher Bewiesenen konsistent sein.

Zu Beginn einer **DONALD**-Sitzung (zum Zeitpunkt t_0) wird nur die Start-Hypothese geglaubt:

$$t_0 : \diamond A$$

Wegen $\diamond A \to A, \diamond A \vdash A$ gilt also $\mid\!\sim A$.

Zum nächsten Zeitpunkt (t_1) seien auch B und C durch `match` bestätigt worden:

$$t_1 : \diamond B, \diamond C \Longrightarrow \; \mid\!\sim B \text{ und } \mid\!\sim C$$

Zum Zeitpunkt t_2 bestätigt `match` die Hypothese D, lehnt aber E, H und I ab:

[14] Vergleiche auch Abschnitt 5.10.

$$t_2 : \Diamond D, \neg\Diamond E, \neg\Diamond H, \neg\Diamond I$$

Aus $\neg\Diamond K$ folgt $\neg K$. Also gilt:

$$\neg H \wedge \neg I \rightarrow \neg C \text{ (Kontraposition von Axiom 3)}$$

und damit $\vdash \neg C$. Dadurch wird zum Zeitpunkt t_2 die Anwendung der Annahme $\Diamond C$ blockiert. Sie ist nicht mehr widerspruchsfrei. Jetzt ist C nicht mehr nichtmonoton herleitbar:

$$\not\mathrel{|\!\sim} C$$

Der nächste Schritt ergebe

$$t_3 : \Diamond F, \neg\Diamond G, \text{ also } \mathrel{|\!\sim} F, \not\mathrel{|\!\sim} G$$

Zum Zeitpunkt t_3 gilt deshalb:

$$\mathrel{|\!\sim} A \wedge B \wedge D \wedge F$$

Dieses Beispiel demonstriert, wie durch im Laufe der Zeit anwachsendes Wissen die Rücknahme ehemals gültiger Schlüsse erzwungen wird.

Kapitel 5

Vererbungshierarchien

Ziel dieses Kapitels ist die Diskussion der Vor- und Nachteile hierarchischer Strukturen zur Vererbung von Informationen in den Wissensrepräsentationssystemen der Künstlichen Intelligenz. Die aus theoretischer Sicht einfachste Form der Hierarchie ist eine baumartige Struktur, in der jedes Objekt nur einen Vorgänger besitzt. Um wirklich sinnvoll einsetzbar zu sein, ist es jedoch erforderlich, mehr als einen Vorgänger zuzulassen. Dadurch treten Probleme mit Mehrdeutigkeit und potentieller Inkonsistenz auf. Die verschiedenen Ansätze auch aus dem Bereich der nichtmonotonen Logiken zur Lösung dieser Problematiken werden vorgestellt. Im nächsten Kapitel wird dann der auf der Arbeit von Touretzky [Touretzky, 1986] aufbauende pfadbasierende Ansatz zur Spezifikation einer *direkten* Semantik für solche Strukturen weiter betrachtet.

5.1 Einführung

Ein Vererbungssystem ist ein Repräsentationssystem, das auf der hierarchischen Strukturierung von Wissen basiert. Fast alle Wissensrepräsentationssprachen und objektorientierten Programmiersprachen (bis auf die regelbasierten Formalismen) sind so organisiert. Diese Tatsache wurde in Kapitel 2 bereits ausführlich diskutiert.

Obwohl die essentielle Idee der hierarchischen Ordnung von Objekten oft auf kompliziertere Anordnungen als die rein hierarchisch aufgebaute Baumstruktur verallgemeinert wird, bildet sie doch die Basis für viele wohlbekannte Systeme.

Als Beispiel für KI-Systeme mit Vererbung können FRL [Roberts and Goldstein, 1977a], KRL [Bobrow and Winograd, 1977], KL-ONE [Brachman and Schmolze, 1985] und NETL [Fahlman, 1979] dienen. Aber auch in Programmiersprachen wie z.B. Smalltalk [Goldberg and Robinson, 1983], [Krasner, 1984], Ada [Ledgard, 1980] und C++ [Stroustrup, 1986], [Dewhurst and Stark, 1990], sowie in verschiedenen objektorientierten Systemen für LISP (FLAVORS [Moon and Weinreb, 1980], LOOPS (Lisp Object Oriented Programming System) [Bobrow and Stefik, 1981] und CLOS (Common Lisp Object System) [Steele Jr., 1990]) findet das hierarchische Konzept weitgefächerte Anwendung, und es gibt durchaus Überschneidungen zwischen den Problemen der KI, Datenbanken und Programmiersprachen. Der Band *Conceptual Modelling* [Brodie *et al.*, 1984] und insbesondere der Artikel [Mylopoulos and Levesque, 1983] kann hier als Integrationsansatz verstanden werden.

Trotz dieses hohen Verbreitungsgrades blieben Vererbungssysteme der komplexeren Art, d.h. Systeme, bei denen Ausnahmen beim Vererbungsprozeß zugelassen sind, lange Zeit ohne theoretisch fundierte Basis. So blieben einige Ungereimtheiten existierender Systeme lange Zeit im Verborgenen.

David Touretzky [Touretzky, 1986] von der Carnegie Mellon University war wohl der erste, der in seiner Dissertation eine formale, mathematische Theorie der Vererbung mit Ausnahmen in semantischen Netzwerken entwickelte.

5.2 Taxonomische Hierarchien

Bei der Organisation von Wissen geht es auch in der Künstlichen Intelligenz darum, Regelmäßigkeiten durch Abstraktion auszunutzen. Eine Abstraktion im speziellen Sinn ist eine Zusammenfassung von Eigenschaften, die den Mitgliedern einer Menge von Objekten gemeinsam sind. Nicht nur Individuen, sondern auch Abstraktionen können sich ihrerseits wiederum Eigenschaften teilen.

Da Abstraktionen also Mengen beschreiben, existiert zwischen Abstraktionen eine natürliche Relation, die durch die Inklusionsrelation zwischen Mengen definiert ist. Ist N die Menge aller Abstraktionen und L definiert durch Mengeninklusion, dann definiert $T := (N, L)$ nach Definition 2.4 auf Seite 31 einen gerichteten Graphen, da die Mengeninklusion nicht symmetrisch ist.

Handelt es sich bei diesem Graphen T um einen Baum (siehe Definition 2.9), so spricht man von einer taxonomischen Hierarchie oder speziell in der Künstlichen Intelligenz von einer Vererbungshierarchie. Definierende Eigenschaft eines Baumes ist, daß jedes Objekt nur höchstens einen Vorgänger haben kann. Das stellt eine starke Einschränkung dar, wie die nächsten beiden Abschnitte über mehrfache Vererbung und Ausnahmen belegen.

5.3 Vorteile hierarchischer Strukturen

Der wesentliche Vorteil einer hierarchischen Anordnung ist, daß sie eine sehr effiziente Methode zur kompakten Speicherung von Wissen ist.

Der zweite Vorteil ist die Möglichkeit, in solch baumartigen Strukturen mit Hilfe einer Art von *binärer* Suche sehr viel effizienter und schneller Informationen zu suchen, als in einer ungeordneten Liste von Aussagen über die einzelnen Objekte. Typischerweise erhält man statt eines $O(n)$ komplexen Zeitverhaltens eines der Form $O(\log(n))$.

Da oft nicht nur eine einzige Suchaufgabe für die dargestellten Objekte zu lösen ist, ist es erforderlich, mehrere verschiedene Organisationen der Objekte gleichzeitig zur Verfügung zu haben. Deshalb wird eine mehrfache, sich eventuell überlappende, orthogonale[1] Gruppierung nach Eigenschaften erforderlich. Man spricht dann von mehrfacher Vererbung *(Multiple Inheritance)*.

[1] Orthogonal bedeutet hier, daß die Objekte unter verschiedenen Gesichtspunkten völlig unterschiedlich hierarchisch angeordnet sein können.

Wenn der Graph T die Form eines Baumes hat, so hat jedes Objekt nur *einen* Vorgänger, von dem geerbt werden kann. Die durch Multiple Inheritance ermöglichte Flexibilität wird allerdings mit dem Auftreten von semantischen Problemen erkauft, die bei der Beschränkung auf taxonomische Hierarchien vermieden werden können.

5.4 Die Notwendigkeit von Ausnahmen

Zur Darstellung von Wissen über Objekte und Beziehungen der *realen* Welt erweist es sich als zu unflexibel, zu fordern, daß bei der Vererbung von Eigenschaften einer Abstraktion auf ein Individuum auch wirklich alle Eigenschaften zutreffen müssen. Ganz im Gegenteil gibt es hier fast zu jeder Verallgemeinerung eine Ausnahme. Damit würde bei Festhalten an der Forderung nach Freiheit von Ausnahmen die Zahl der vererbbaren Eigenschaften nur sehr klein sein und so die Vorteile der hierarchischen Darstellung zunichte gemacht. Klassisches Beispiel ist hier das in Abbildung 5.1 dargestellte *Tweety Triangle*, das einen Vogel beschreibt, der nicht fliegen kann.

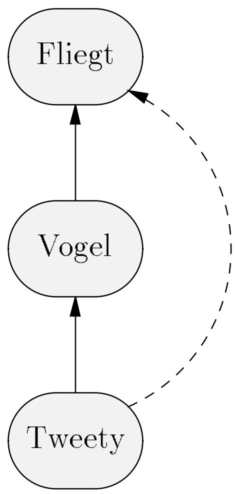


Abbildung 5.1: Das Tweety Triangle

Aus den angeführten Gründen erlauben die meisten Systeme die Spezifikation von Ausnahmen, indem Individuen den Widerspruch zwischen ererbter Information und direkt für das Individuum spezifizierter Information zu Gunsten letzterer auflösen.

Auch die durch Abstraktion gebildeten Klassen können wieder Ausnahmen enthalten, ebenso sind dann Ausnahmen von Ausnahmen möglich. Auf dem Cover von [Gelfond, 1987] befindet sich z.B. die Darstellung eines Pinguins, der sich dank selbstgebastelter Behelfsflügel in die Luft erhebt. Pinguine sind Vögel. Vögel können im allgemeinen fliegen. Pinguine können bis auf diese eine Ausnahme von der Ausnahme nicht fliegen.

5.5 Normative Inferenzen

Vererbung, wie sie in Programmiersprachen benutzt wird, beschreibt jeweils eine Menge von durch den Programmierer definierten Abstraktionen. Diese haben also keinen direkten Einfluß auf die Spezifikation des Programmes selbst. Das in der Künstlichen Intelligenz geforderte plausible Schließen mit Objekten der realen Welt verwendet im Gegensatz dazu direkt den Vererbungsmechanismus. Damit beeinflußt dieser auch das beobachtbare Verhalten des Programms.

Normative Aussagen sind Aussagen, die für wahr gehalten werden können, solange keine Evidenz für gegenteilige Information besteht. Sie treffen auf die meisten Individuen einer Klasse oder den *typischen* Vertreter einer Klasse zu und sind also nur *im allgemeinen* richtig.

Im Rahmen der Künstlichen Intelligenz wird normative Inferenz benötigt, um ein Programm auch im Falle unvollständiger Information über seine Umgebung effizient schlußfolgern lassen zu können.

Die Herleitung von Eigenschaften durch Vererbung muß aber nicht notwendig normativen Charakter haben. Die Darstellung einer Ansammlung von Individuen kann durch die richtige Überschreibung der falsch vererbten Information in jedem Fall korrigiert werden. Es sind also durchaus Fälle von stark spezialisierten Wissensbasen denkbar, in denen nicht-normative Aussagen zu einer kompakteren Repräsentation führen. Das liegt daran, daß die *typischen* Vertreter einer Klasse nicht unbedingt prozentual gesehen in der Mehrheit sein müssen. Normative Aussagen haben ja insbesondere nur bei *fehlender* Information ihre Berechtigung.

Man betrachte die folgenden Aussagen:

Säugetiere haben vier Beine.

Das typische Säugetier hat vier Beine.

Normale Säugetiere haben vier Beine.

Fast alle Säugetiere haben vier Beine.

Es kann angenommen werden, daß Säugetiere vier Beine haben.

Säugetiere haben standardmäßig vier Beine.

Alle diese Aussagen drücken etwas geringfügig anderes aus, würden aber in einem Vererbungssystem gleich dargestellt. Somit wird deutlich, daß die Darstellung normativer Aussagen nur approximativ möglich ist.

5.6 Mehrfache Vererbung

Falls es sich bei dem Vererbungsgraphen um einen Baum handelt, so kann Information nur von einer Oberklasse vererbt werden. Zur Darstellung des Individuums I mit den Eigenschaften der Klassen C_1 und C_2 ist es dann erforderlich, entweder eine neue Klasse S_1 als Unterklasse von C_1 mit den zusätzlichen Eigenschaften von C_2 oder umgekehrt S_2 als Unterklasse von C_2 künstlich zu definieren. Beide Lösungen sind unbefriedigend, da Information doppelt spezifiziert werden muß. Außerdem gibt es keine Begründung, sich für die eine oder andere Möglichkeit zu entscheiden.

Läßt man dagegen zu, daß Information nicht nur von einer Klasse, sondern auch von mehreren Klassen vererbt werden kann, so muß weder S_1 noch S_2 definiert werden. In solchen Systemen hat man dann aber keinen Vererbungsbaum, sondern einen gerichteten Vererbungsgraphen.

Leider zeigen die Laufzeitabschätzungen vieler Algorithmen ein nicht zu unterschätzendes Problem auf. Das liegt an der im schlechtesten Fall exponentiell von der Anzahl der Knoten abhängigen Zahl der Pfade durch einen allgemeinen Graphen. Die höhere Allgemeinheit muß also auch hier mit größerer Komplexität erkauft werden. Die Komplexität von solchen auf Pfaden basierenden Vererbungsalgorithmen ist noch Thema von Kapitel 7. Ein weiterer Ansatz zur Lösung dieses Problems besteht in dem Einsatz massiv paralleler Computerarchitekturen wie im Fall des spezialisierten NETL [Fahlman, 1979] oder der allgemeineren Connection Machine [Hillis, 1985][2].

5.7 Probleme mit Mehrfacher Vererbung

Läßt man in einem System mit multipler Vererbung Ausnahmen zu, so ergeben sich sofort zwei semantische Probleme. Betrachtet man ein einfaches Beispiel, das in der Literatur vielfach verwendet wird und in Abbildung 5.2 graphisch dargestellt ist, so wird klar, daß hier ein unbefriedigendes Verhalten vorliegt:

Elephanten sind grau.
Königselephanten sind Elephanten.
Königselephanten sind nicht grau.
Clyde ist ein Königselephant.

Im Hinblick auf die Frage, ob Clyde grau ist, kann man auf doppelte Art und Weise argumentieren. Da Clyde ein Königselephant ist, und Königselephanten nicht grau sind, ist auch Clyde nicht grau. Andererseits ist Clyde ein Königselephant, Königselephanten sind Elephanten, Elephanten sind grau, also ist auch Clyde grau. Offensichtlich ist das ein Widerspruch. Intuitiv ist klar, das Clyde nicht grau ist, obwohl er ein Elephant ist, da er ein besonderer Elephant ist, eben ein Königselephant. Sowohl die Systeme FRL, NETL als auch **DONALD** stimmen hier natürlich noch überein, da ihr Inferenzalgorithmus so entworfen ist, daß Unterklassen gegenüber Klassen bevorzugt werden. Was geschieht aber, wenn man eine weitere Aussage den vorhandenen Axiomen hinzufügt?

Clyde ist ein Elephant.

[2] Die Connection Machine kann in einem geringfügig erweiterten LISP programmiert werden, während NETL nur primitive Operationen zur Manipulation von Markierungen zur Verfügung stellt. Da für NETL auch immer noch keine parallele Implementation existiert, bleiben viele Ansätze von rein theoretischem Interesse.

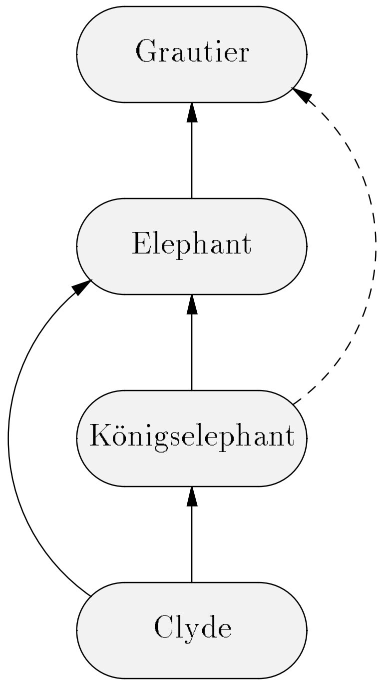


Abbildung 5.2: Ein Netzwerk mit redundanter Information

Auch aus dem ursprünglichen Satz von Axiomen folgte diese Aussage bereits, da Königselephanten Elephanten sind. Das Hinzufügen dieser Aussage sollte also alle Schlußfolgerungen über Clyde unbeeinflußt lassen. Betrachtet man jedoch die Implementation dieser Aussagen in NETL oder FRL, so ergibt sich in diesem System ein direkter Link von Clyde sowohl zu Königselephant als auch zu Elephant. Da in FRL der Vererbungsmechanismus alle vorhandenden Werte sammelt, ist hier Clyde grau und gleichzeitig nicht grau.

Andere Implementationen beenden die Suche nach vererbter Information, sobald ein Wert gefunden wurde. Solange man sich nicht mit *mehrfacher* Vererbung beschäftigt, ist diese Suche des nächsten Wertes äquivalent zur Wahl des kürzesten Weges *(Shortest Path)* zu einem Wert. Alle diese Pfade stimmen aufgrund des eindeutig bestimmten Elternknotens jeder Teilklasse in ihren Anfangssegmenten überein. Diese Äquivalenz wird durch das Vorhandensein mehrfacher Vererbungswege zerstört und damit ist die Pfadlänge kein gültiges Entscheidungskriterium mehr, das die Auswahl des korrekten Wertes erlaubt. Im obigen Beispiel wird durch die Einführung der redundanten Aussage „Clyde ist ein Elephant“ die Antwort auf die Frage, ob Clyde grau ist, von der Suchstrategie des Systems abhängig. Damit hängt seine Farbe von etwas ab, das nicht in der Menge der vorgegebenen Aussagen enthalten war.

Der Suchalgorithmus von NETL zeigt ebenso dieses Problem, da NETL bei der Suche nach Informationen eine Reihe von Markierungen durch das Semantische Netzwerk propagiert, die bei der Suche konkurrieren. Welche Markierung als erste auf eine Antwort stößt, hängt wieder einzig und allein von der Länge der zurückzulegenden Pfade ab, die aber, wie oben gezeigt, durchaus im Widerspruch zur intuitiv vorhandenen Klassenrelation stehen kann.

Die erste Idee zur Beseitigung dieses Problems ist einfach das Verbot redundanter Aussagen. Dieses Vorgehen ist aber insofern nicht haltbar, als es wenig sinnvoll erscheint, unbestreitbar wahre Aussagen, die zudem auch noch zu der Menge der ableitbaren Aussagen gehören, nicht zuzulassen. Die Nichtmonotonie eines Systems könnte auch die explizite Aufnahme einer redundanten Aussage erforderlich machen, falls sich im Laufe der Zeit herausstellt, daß die Voraussetzungen für diese Aussage nicht mehr gültig sind. Betrachtet man z.B. Doyles *Truth Maintenance System* (TMS) [Doyle, 1979b] [Doyle, 1979a], so hat dort jede Aussage u.a. einen sogenannten *Set of Support*, der auch für im Moment redundante Aussagen erforderlich ist. Doyles System wurde von de Kleer in [de Kleer, 1986a], [de Kleer, 1986b] und [de Kleer, 1986c] durch die Einführung von sogenannten minimalen Kontexten verbessert. Auch sein *Assumption Based Truth Maintenance System* (ATMS) ist algorithmischer und nicht deklarativer Natur, scheint aber im Hinblick auf die Formalisierbarkeit Vorteile zu bieten.

In einem System, das nicht frei von Ausnahmen ist, gibt es auch gar keine wirklich redundanten Aussagen. Das wird an folgendem Beispiel deutlich:

> Elephanten sind grau.
> Graue Dinge sind sandfarben.
> Elephanten sind sandfarben.
> Königselephanten sind Elephanten.
> Königselephanten sind nicht grau.

„Elephanten sind sandfarben“ erscheint redundant, da Elephanten grau sind und graue Dinge sandfarben. Sobald es aber einen anderen, von dieser Argumentation unabhängigen Grund für die sandige Farbe von Elephanten gibt, kann diese scheinbar redundante Aussage wichtig werden: Obwohl Königselephanten nicht grau sind, sind sie nämlich deshalb sandfarben.

Das zweite semantische Problem ist die bei der Kombination von mehrfacher Vererbung und Vererbung mit Ausnahmen mögliche Mehrdeutigkeit. Klassisches Beispiel ist hier das in Abbildung 5.3 dargestellte *Nixon diamond*:

Quäker sind Pazifisten.
Republikaner sind keine Pazifisten.
Nixon ist ein Quäker.
Nixon ist ein Republikaner.

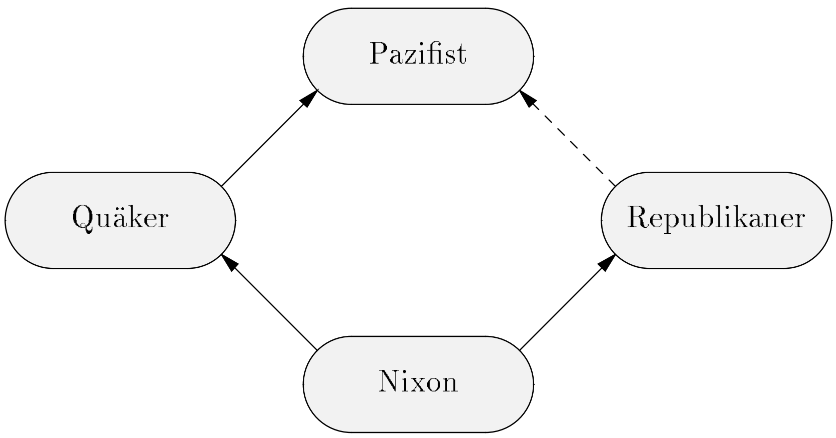


Abbildung 5.3: Das Nixon Diamond

Ist Nixon nun ein Pazifist oder nicht? FRL beantwortet diese Frage mit *beides*. Im Gegensatz zu dieser inkonsistenten Antwort ist NETLs Verhalten nicht vorhersehbar und hängt von irrelevanten Implementationsdetails ab. Offensichtlich gibt es in diesem Fall keine Basis für eine Beantwortung, da das Netzwerk nicht eindeutig ist. In der Sprache klassischer Prädikatenlogik bedeutet das die Existenz zweier in sich konsistenter, aber untereinander inkompatibler Theorien. Wie man dieses Problem auch löst, entscheidend ist zunächst, daß das System in der Lage ist, solche Situationen überhaupt zu erkennen.

5.8 Vererbung und Prädikatenlogik

Die Prädikatenlogik erster Stufe hat eine wohldefinierte Semantik (siehe Abschnitt 2.3.6), d.h. jeder Satz hat eine Bedeutung, die aus den einzelnen Komponenten (Termen) und Konnektiven durch die Anwendung einfacher Regeln hervorgeht (siehe Definition 2.22 auf Seite 43). So kam Patrick Hayes in [Hayes, 1979] auf die Idee, die Semantik eines Vererbungssystems einfach dadurch zu definieren, daß er eine Abbildung angibt, die ein Semantisches Netzwerk in Sätze der Prädikatenlogik transformiert. Leider ist diese Abbildung bei der Betrachtung nichtmonotoner Vererbung mit Ausnahmen nicht ohne weiteres zu bewerkstelligen. Es sieht so aus, als sei es geeigneter, als Ziel dieser Abbildung Sätze einer nichtmonotonen Logik zu wählen. Dies wird im folgenden noch eingehender betrachtet.

Die gebräuchliche Übersetzung von Frames in die Prädikatenlogik geht so vor: Frames, die Individuen darstellen, werden zu Konstanten, die Slots eines Frames werden zu partiellen Funktionen. Das bekannte Beispiel des Elephanten Clyde liefert hier die Funktion `farbe()`. Seine Farbe wäre dann `farbe(clyde)`. Das Füllen des Slots mit dem Wert grau entspricht dann der logischen Aussage `farbe(clyde) = grau`.

Frames zur Darstellung von Klassen werden zu einstelligen Prädikaten. Der Frame Elephant wird so zum Prädikaten Elephant und die AKO Verbindung von Clyde zu Elephant wird zu `Elephant(clyde)`. Beschränkt man sich auf ein System ohne Ausnahmen wie FRL, so ist AKO eine strikte Implikation und man hat damit folgende Übersetzung:

$$(\forall x) \quad \texttt{Elephant}(x) \rightarrow \texttt{Grautier}(x).$$

Die Übersetzung eines Vererbungssystems in die Prädikatenlogik ist einfach und naheliegend, solange man keine Ausnahmen zuläßt. Mit den durch Ausnahmen und anfechtbare Annahmen entstehenden Problemen beschäftigt sich die nichtmonotone Logik und die Default Logik. Ursprüngliches Ziel dieser Formalismen war die adäquate Lösung des Frame Problems und des Qualifikationsproblems.[3]

[3] Diese beiden Probleme wurden schon in Abschnitt 2.9.1 auf Seite 76 dargestellt.

5.9 Nichtmonotone Logik

Wenn die Menge der aus einer Reihe von Axiomen herleitbaren Formeln *nicht* monoton wächst, sobald die Menge der Axiome erweitert wird, so spricht man von einer nichtmonotonen Logik. Die klassische Prädikatenlogik ist nach Theorem 2.12 auf Seite 46 ein monotones System.

Der Begriff der nichtmonotonen Logik bezeichnet nun eine ganze Familie von nichtmonotonen Formalismen, die modale Operatoren zur Darstellung von Begriffen wie „Konsistenz“ und „Überzeugung“ *(Belief)* verwenden. Die Sprache dieser Formalismen entsteht aus der klassischen Logik durch Hinzunahme des zusätzlichen Operators $\mathcal{M}$ mit der Interpretation „es ist konsistent, anzunehmen“. Üblicherweise wird ein weiterer zu $\mathcal{M}$ dualer modaler Operator $\mathcal{L}$ als Abkürzung für $\neg\mathcal{M}\neg$ eingeführt: „Es wird geglaubt“ oder „Es ist nichtmonoton beweisbar“.

In der modalen nichtmonotonen Logik werden nichtmonotone Regeln immer in der Objektsprache selbst ausgedrückt. Klassisches Beispiel ist die Aussage „Vögel fliegen“:

$$\forall x \texttt{Vogel}(x) \wedge \mathcal{M}\texttt{fliegt}(\S) \supset \texttt{fliegt}(\S)$$

Das Problem ist die Definition und intendierte Bedeutung des Operators $\mathcal{M}$, das oft durch eine Fixpunkt Konstruktion gelöst wird.

Der erste Ansatz zur Definition einer nichtmonotonen Logik kam von McDermott und Doyle [McDermott and Doyle, 1980]. Ihre wichtigsten Definitionen und Theoreme werden im folgenden kurz dargestellt. Die Darstellung orientiert sich dabei unter anderem an [Lukaszewicz, 1990]. Zentraler Begriff ist die Erweiterung *(Extension)* einer *Theorie*. Theorien sind berechenbare Mengen von Sätzen über einer Sprache und sollen bekannte Voraussetzungen darstellen. Eine Erweiterung stellt dann eine mögliche Menge von aus diesen Voraussetzungen ableitbaren Überzeugungen dar. Für Erweiterungen S ist unter Verwendung des Begriffs der Beweisbarkeit $\vdash$ aus der klassischen Prädikatenlogik (siehe Definition 2.24) die Erfüllung der folgenden Bedingungen wünschenswert:

(i) S soll deduktiv abgeschlossen sein: $S = Th(S)$.

(ii) S sollte die Voraussetzungen enthalten: $S \supseteq T$.

(iii) S sollte alle Sätze der Form $\mathcal{MA}$ enthalten, vorausgesetzt, daß $S \not\vdash \neg A$.

Definition 5.1 (Erweiterung) *Sei T eine Theorie. Eine Menge $S \subseteq L_T$ von Sätzen ist genau dann eine* Erweiterung *von T, wenn*

$$S = Th(T \cup \{\mathcal{MA} : \mathcal{A} \textit{ ist ein Satz von } \mathcal{L_T} \textit{ und } \neg\mathcal{A} \notin \mathcal{S}\}).$$

Sätze der Form $\mathcal{MA}$ heißen Annahmen (Assumptions).

Sei T eine Theorie und $\lambda = \{A_i : i = 0, 1, \ldots\}$ eine Aufzählung aller Sätze aus L_T. Dann bezeichnet $S(T, \lambda)$ die folgende Menge

$$S(\lambda, T) = Th\left(\bigcup_{i=0}^{\infty} S_i\right), \text{ mit}$$

$$\begin{array}{rcl} S_0 & = & T, \text{ und für } i \geq 0 \\ S_{i+1} & = & \left\{ \begin{array}{ll} L_T & \text{falls } \mathcal{MB} \in \mathcal{S}_\rangle \text{ und } \mathcal{S}_\rangle \vdash \neg\mathcal{B} \\ & \quad \text{für ein geeignetes } B \in L_T \\ S_i \cup \{\mathcal{MA}_\rangle\} & \text{sonst, wenn } S_i \not\vdash \neg A_i \\ S_i & \text{sonst} \end{array} \right. \end{array}$$

Es gelten folgende Theoreme:

Theorem 5.1 *Für alle T und λ gilt: Ist $S(\lambda, T)$ konsistent, so ist es eine Erweiterung von T.*

Theorem 5.2 (McDermott und Doyle, 1980) *Sei T eine Theorie. Sind S_1 und S_2 Erweiterungen von T und gilt $S_1 \subseteq S_2$, dann ist $S_1 = S_2$.*

Diese Theoreme belegen zwar die Minimalität von Erweiterungen in Bezug auf die Mengeninklusion, beantworten aber immer noch nicht die Frage nach der Menge der nichtmonoton herleitbaren Theoreme. Für den Fall einer einzigen Erweiterung einer Theorie liegt es nahe, diese zu wählen. Wie Beispiele zeigen, gibt es aber Theorien *ohne* Erweiterungen, Theorien mit mehreren Erweiterungen und sogar Theorien mit unendlich vielen Erweiterungen. Die Menge $TH(T)$ der herleitbaren Theoreme muß also anders definiert werden:

Definition 5.2 (McDermott und Doyle, 1980) *Sei T eine Theorie und bezeichne $E(T)$ die Menge aller Erweiterungen von T. Dann ist die Menge aller nichtmonoton aus T herleitbaren Theoreme $TH(T)$ definiert durch:*

$$TH(T) = \begin{cases} \bigcap\{S : S \in E(T)\} & \textit{für } E(T) \neq \emptyset \\ L_T & \textit{sonst} \end{cases}$$

Diese so entstehende Logik hat einige überraschende Eigenschaften:

Theorem 5.3 (McDermott und Doyle, 1980) *Sogar eine Teiltheorie einer nichtmonoton konsistenten Theorie kann nichtmonoton inkonsistent sein.*

Im Nachhinein wurden verschiedene Beispiele gefunden, die zeigen, daß die obige Logik nicht nur theoretisch wenig wünschenswerte Eigenschaften hat, sondern auch die geplante Bedeutung ihrer modalen Operatoren nicht befriedigend darstellt. Dies hatte zur Folge, daß McDermott in [McDermott, 1982a] eine Gruppe stärkerer Formalismen entwickelte, die auf einer *modalen* deduktiven Struktur aufbauen. Deren Darstellung soll aber hier nicht mehr folgen. Stattdessen wird im nächsten Abschnitt auf einen anderen Formalismus zur Lösung des Frame Problems eingegangen.

5.10 Default Logik

Default Logik *(Default Logic)* ist ein weiterer nichtmonotoner Formalismus und gehört zu den am besten bekannten und verstandenen Ansätzen. Reiter legte mit seinem Artikel [Reiter, 1980] die Grundlagen zu seinem konzeptionell recht einfachen Ansatz. Weiterführende Ergänzungen finden sich in [Reiter and Criscuolo, 1981], [Etherington, 1987], [Lukaszewicz, 1990] und [Delgrande and Jackson, 1991].

Default Logik und modale nichtmonotone Logik unterscheiden sich im wesentlichen in der Darstellung der nichtmonotonen Schlußregeln. Während die modale nichtmonotone Logik diese Regeln durch Formeln der Objektsprache selbst ausdrückt, werden diese in der Default Logik durch spezielle linguistische Ausdrücke (eben die Defaults) dargestellt.

Daß Vögel im allgemeinen fliegen können, wird durch folgende Default Regel festgehalten:

$$\frac{\texttt{Vogel}(x) : \texttt{fliegt}(x)}{\texttt{fliegt}(x)}.$$

Als Interpretation dient hier: „Für jedes Individuum x gilt: Falls x ein Vogel ist und es konsistent ist, anzunehmen, daß x fliegen kann, dann soll geschlossen werden, daß x fliegen kann.“ Wir sind also berechtigt anzunehmen, daß Patty fliegen kann, falls wir wissen, daß sie ein Vogel ist. In dem Moment, in dem wir erfahren, daß Patty ein Pinguin ist und wir bereits wissen, daß Pinguine nicht fliegen können, wird es inkonsistent anzunehmen, daß Patty fliegen kann, und die Anwendung der Default Regel wird blockiert.

Informell wird Allgemeinwissen über eine zu modellierende Welt in einer Default Theorie dargestellt. Eine solche Theorie ist ein Paar, bestehend aus einer Menge von Sätzen der Prädikatenlogik erster Stufe, genannt die *Axiome* der Theorie, und einer Menge von Defaults. Die Axiome stellen logisch gültige, aber im allgemeinen leider unvollständige Informationen über die Welt dar. Die Defaults erweitern diese Information, indem bestimmte plausible, aber nicht notwendigerweise wahre Schlüsse erlaubt werden.

Jede Menge von Annahmen (oder *Beliefs*), die man über eine durch eine Default Theorie beschriebene Welt haben kann, wird als eine Erweiterung *(Extension)* dieser Theorie bezeichnet. Die formale Definition einer Extension ist technisch komplex. Intuitiv enthält eine Extension alle Sätze, die aus den Axiomen entweder mit klassischer Prädikatenlogik oder durch die Anwendung von Default Regeln abgeleitet werden können. Diese Regeln werden folgendermaßen definiert:

Definition 5.3 (Reiter, 1980) *Ein Default ist ein Ausdruck der Form*

$$\frac{A(\vec{x}) : B_1(\vec{x}), \ldots, B_k(\vec{x})}{C(\vec{x})}$$

wobei $A(\vec{x}), B_1(\vec{x}), \ldots, B_k(\vec{x})$ *und* $C(\vec{x})$ *Formeln erster Ordung sind, deren freie Variablen unter denen von* $\vec{x} = (x_1, \ldots, x_n)$ *sind.* $A(\vec{x})$ *heißt Voraussetzung* (Prerequisite)*,* $B_1(\vec{x}), \ldots, B_k(\vec{x})$ *sind die Begründungen* (Justifications) *und* $C(\vec{x})$ *Konsequenz* (Consequent) *der Default Regel.*

Definition 5.4 (Reiter, 1980) *Eine Default Theorie ist eine Tripel* $T = \langle L, W, D\rangle$*, wobei* L *eine Sprache erster Ordung,* W *eine Menge von Sätzen aus* L *(die* Axiome *von* T*) und* D *eine Menge von Defaults ist, deren Voraussetzungen, Begründungen und Konsequenzen Formeln aus* L *sind.*

Ein Default heißt *offen*, wenn wenigstens einer der gegebenen Terme $A(\vec{x})$, $B_1(\vec{x}), \ldots, B_k(\vec{x})$ oder $C(\vec{x})$ eine freie Variable enthält; ansonsten heißt er *geschlossen*. Eine Default Theorie heißt *offen* genau dann, wenn sie mindestens einen offenen Default enthält; ansonsten heißt sie *geschlossen*.

Freie Variablen werden als implizit universell quantifiziert angesehen. Ihr Gültigkeitsbereich *(Scope)* umfaßt den gesamten Default. Ein offener Default stellt also ein allgemeines Schema dar, das auf unterschiedliche Tupel von Instanzen angewandt werden kann.

Eine Instanz eines offenen Defaults entsteht durch Ersetzen aller freien Vorkommen von Variablen durch Grundterme. Eine Instanz eines offenen Defaults $A(\vec{x}) : B_1(\vec{x}), \ldots, B_k(\vec{x})/C(\vec{x})$ mit $\vec{x} = (x_1, \ldots, x_n)$ ist also jeder abgeschlossene Default $A(\vec{\alpha}) : B_1(\vec{\alpha}), \ldots, B_k(\vec{\alpha})/C(\vec{\alpha})$, wobei $\vec{\alpha}$ ein beliebiges n-Tupel von Grundtermen ist.

Offene Defaults bereiten technische Schwierigkeiten, deshalb definiert man zu einer Default Theorie ihren *Abschluß* durch einen Transformationsprozeß nach Art der Skolemisierung in der Prädikatenlogik. So kann die folgende Theorie sich auf die Beschäftigung mit geschlossenen Theorien beschränken und den Begriff der Erweiterung definieren:

Definition 5.5 (Reiter 1980) *Sei* $T = \langle W, D\rangle$ *eine abgeschlossene Default Theorie über einer Sprache* L *erster Stufe. Für eine Menge* $S \subseteq L$ *von Sätzen, sei* $\Gamma(S)$ *definiert als die kleinste Menge von Sätzen aus* S*, die den folgenden Bedingungen genügt:*

(i) $\Gamma(S) = Th(\Gamma(S))$.

(ii) $W \subseteq \Gamma(S)$.

(iii) *Ist* $(A : B_1, \ldots, B_k/C) \in D, A \in \Gamma(S)$ *und* $\neg B_1 \notin S, \ldots, \neg B_k \notin S$*, dann ist* $C \in \Gamma(S)$.

Eine Menge $E \subseteq L$ *von Sätzen heißt genau dann* Erweiterung *von* T*, wenn* $E = \Gamma(E)$*, d.h.* E *ist ein Fixpunkt der Operators* Γ*.*

Diese Fixpunktdefinition ist natürlich nicht sehr intuitiv und konstruktiv, was aber der folgende Satz ausgleicht:

Theorem 5.4 (Reiter 1980) *Sei* $T = \langle W, D\rangle$ *eine abgeschlossene Default Theorie, dann ist eine Menge* E *von Sätzen genau dann eine Erweiterung von* T*, wenn*

$$E = \bigcup_{i=0}^{\infty} E_i, \ \ mit$$

$$\begin{aligned} E_0 &= W, \ \ und\ für\ i \geq 0 \\ E_{i+1} &= Th(E_i) \cup \{C : (A : B_1, \ldots, B_k/C) \in D, \\ &\quad wobei\ E_i \vdash A\ und\ \neg B_1 \notin S, \ldots, \neg B_k \notin E\}. \end{aligned}$$

Leider tritt auch hier wieder wie in [McDermott and Doyle, 1980] das Problem nicht existierender oder nicht eindeutiger Erweiterungen auf, obwohl wieder ein Minimalitätssatz gilt. Reiter konnte aber eine eingeschränkte Form von Defaults finden, für die die Existenz von Erweiterungen garantiert ist.

Definition 5.6 *Ein Default der Form*

$$\frac{A(\vec{x}) : B_1(\vec{x}), \ldots, B_k(\vec{x})}{B(\vec{x})}$$

heißt normal*. Eine Theorie* $\langle W, D\rangle$ *heißt* normal*, wenn alle ihre Defaults aus* D *normal sind.*

Für abgeschlossene normale Theorien konnte Reiter die Existenz von Erweiterungen zeigen. Existieren mehrere Erweiterungen, so ist ihre Vereinigung inkonsistent. Normale Defaults ermöglichen es auch, die Erweiterungen auf nichtrekursive Weise zu charakterisieren. Die Erweiterungen normaler Theorien genügen sogar einer Monotonierelation:

Theorem 5.5 (Semimonotonie) *Seien D und D' zwei abgeschlossene Mengen normaler Defaults, so daß $D' \subseteq D$. Sei E' eine Erweiterung von $T' = \langle W, D' \rangle$ und sei $T = \langle W, D \rangle$. Dann hat T einer Erweiterung E, so daß $E' \subseteq E$.*

Aufgrund dieser positiven Eigenschaften sind normale Theorien ein Kandidat zur Repräsentation von Wissen. Viele der in der Praxis auftretenden nichtmonotonen Regeln lassen sich in normale Form bringen. Betrachtet man jedoch die Regeln in einem breiteren Kontext, so ergeben sich durch Interaktion der Defaults Probleme [Reiter and Criscuolo, 1981]. Die für normale Defaults geltenden Einschränkungen waren zu stark und das führte zur folgenden abschwächten Version von Definition 5.6.

Definition 5.7 *Ein Default der Form*

$$\frac{A(\vec{x}) : B_1(\vec{x}), \ldots, B_k(\vec{x}) \wedge C(\vec{x}}{B(\vec{x})}$$

heißt seminormal*. Eine Theorie $\langle W, D \rangle$ heißt* seminormal*, wenn alle ihre Defaults aus D seminormal sind.*

Es stellte sich aber heraus, daß seminormale Theorien nicht mehr die positiven Eigenschaften von normalen Theorien haben. Etherington konnte jedoch in Zusammenarbeit mit Reiter in [Etherington and Reiter, 1983] zeigen, daß *geordnete* seminormale Theorien eine Erweiterung besitzen. Jede geordnete seminormale Theorie ist wieder auch eine normale Theorie.

Die von Etherington benötigte Ordungsrelation ist eine Abhängigkeitsordnung zwischen Formeln. Die Idee ist, daß $A \ll B$, wenn B von A abhängen könnte. So eine Abhängigkeitsrelation unterscheidet sich aber stark von der in Vererbungssystemen benutzen Ordnung zwischen Klassen und Teilklassen.

Relativ schnell wurde klar, daß die Erweiterungen der Default Logik häufig Aussagen enthalten bzw. nicht enthalten, die der intuitiven vorhandenen Vorstellung über die Menge der herleitbaren Fakten widersprechen. Delgrande und Jackson versuchen in [Delgrande, 1988] und [Delgrande and Jackson, 1991] diesem Problem durch etwas geänderte Definitionen der Konsistenz und Einführung spezieller Formen von Defaults Rechnung zu tragen.

5.11 Bewertung

Wie schon die nichtmonotone Logik ist auch die Default Logik im allgemeinen unentscheidbar. Selbst wenn für einen Formalimus bewiesen wurde, daß eine Erweiterung existiert, muß es noch keinen Algorithmus geben, der diese auch konstruieren kann.

Zwar konnte in [Etherington and Reiter, 1983] gezeigt werden, wie sich die Vererbung mit Ausnahmen in Default Logik übersetzen läßt, aber dazu war es erforderlich, alle Ausnahmen zu einer seminormalen Default Regel explizit aufzulisten. Das steht in krassem Gegensatz zu einer hierarchischen Ordnung, in der *implizit* Information über die Anwendung der Inferenzregeln abgelegt ist.

Bei der Formalisierung nichtmonotoner Vererbungssysteme kann man sich auf den Standpunkt stellen, daß die oben betrachteten Schlußsysteme nichtmonotoner Logiken die Problematiken der Vererbung als Spezialfall enthalten und lösen. Diese Systeme haben eine mehr traditionelle Grundlage auf der Basis mathematischer Logik und verfügen über eine semantische Motivation in Form einer Modelltheorie. Als Nachteil muß gewertet werden, daß sie auf viele spezielle Feinheiten der Vererbung in hierarchischen Strukturen noch nicht erfolgreich angewendet werden konnte. Diese Tatsache motivierte die Entwicklung direkter Methoden zur Analyse von Hierarchien, die auf der Betrachtung der möglichen Pfade durch ein Netzwerk basieren. Diese Methoden stellen zwar einen vollkommen konträren Ansatz dar, aber insgesamt läßt sich ein positiver Einfluß der direkten auf die indirekten Formalismen und umgekehrt beobachten.

Touretzky hält in [Touretzky, 1986] die Benutzung der hierachischen Ordung von Klassen zur impliziten Kontrolle der Inferenzen für die definierende Eigenschaft von Vererbungssystemen. Aufbauend auf seiner Dissertation entwickelten sich eine ganze Reihe von pfadbasierten Formalismen zur Vererbung, die sich in wichtigen Eigenschaften unterscheiden. Erst in den letzten beiden Jahren ergaben auch Untersuchungen der Komplexität solcher Systeme, daß es durchaus wesentliche Feinheiten des Designs zu betrachten gilt, die den Unterschied zwischen NP-schweren und polynomialen Algorithmen ausmachen. Das nächste Kapitel beschäftigt sich ausführlich mit diesen direkten Formalismen, die ohne mathematische Logik auskommen.

Kapitel 6

Direkte pfadbasierte Vererbungssysteme

Dieses Kapitel stellt die schon angekündigten *direkten pfadbasierten* Methoden zur Formalisierung der Wissensrepräsentation in einem Semantischen Netzwerk vor. Ausgangspunkt ist die Dissertation von David Touretzky *The Mathematics of Inheritance Systems* [Touretzky, 1986]. Als er sich näher mit der von Fahlman entworfenen Netzwerksprache NETL (siehe Abschnitt 3.4 auf Seite 94) beschäftige, wurde ihm deutlich, daß die in NETL verwendeten Algorithmen auf einer sehr schwachen theoretischen Grundlage standen.

Die hier gegebene Darstellung integriert und erweitert Aspekte der Arbeit Touretzkys mit einer ganzen Reihe von neueren Arbeiten [Thomason *et al.*, 1986] [Horty *et al.*, 1987a] [Horty *et al.*, 1987b] [Touretzky *et al.*, 1987] [Touretzky and Thomason, 1988] [Thomason and Horty, 1988] [Krishnaprasad, 1989] [Thomason, 1990] [Horty and Thomason, 1990] [Horty, 1991] [Thomason and Touretzky, 1991] [Stein, 1991] [Touretzky *et al.*, 1991] auf diesem Gebiet. Besonderer Wert wird auf die in Abschnitt 4.8.3 auf Seite 146 begonnene Integration der Implementation der Expertensystem-Shell **DONALD-III** als praktischem Teil der vorliegenden Arbeit mit aktuellen Forschungsergebnissen gelegt.

6.1 Einführung

NETL [Fahlman, 1979] implementiert die Suchfunktionen in einem Semantischen Netz aus theoretischer Sicht durch einen Graphfärbungsalgorithmus. Eine Anfrage der Form „Hat Objekt a die Eigenschaft p“ löst eine Welle von Farbreaktionen aus, wobei der dem Objekt a entsprechende Knoten die Farbe p hat, falls die Antwort *ja* sein soll. Die Algorithmen zur Traversierung des Graphen beruhen auf einer Maschine, die die Knoten des Netzwerks durch Prozessorelemente darstellt, die eine endliche Anzahl von Markierungen (Farben) aufnehem können. Instruktionen für diese Maschine haben dann etwa die Form: „Wenn die Markierungen M_1 gesetzt und M_3 nicht gesetzt sind, dann setze auch M_4.“ oder „Falls Markierung M_2 gesetzt, dann propagiere die Markierung M_6 über alle mit *VC *(Virtual Copy)*[1] beschrifteten Kanten“. Da alle Prozessorelemente gleichzeitg aktiv sind, spricht Fahlman von einer Parallel Marker Propagation Machine (PMPM). Da jedes Prozessorelement nur eine kleine endliche Menge an Markierungen verwalten kann[2], ist die Klasse der mit ihr realisierbaren Algorithmen eingeschränkt.

Benutzt man eine solche Maschine zur Darstellung von Graphfärbungsalgorithmen, so ist naheliegend, daß die Entfernung[3] zwischen zwei Knoten eine Rolle für das Ergebnis der Operation spielt. Das ist insbesondere dann der Fall, wenn man kritische Netzwerke betrachtet, die der Situation der im letzten Kapitel betrachteten mehrfachen Vererbung mit Ausnahmen entsprechen. Dieser Tatsache wird mit dem Begriff *Shortest Path Reasoner* Rechnung getragen. Entscheidend für die endgültige Farbe eines Zielknotens ist nämlich die

[1] Eine *VC Kante ist Fahlmans Notation für die Vererbungsrelation, die in FRL (siehe Abschnitt 3.2) mit ISA und in **DONALD** mit *View* bezeichnet wird. Vergleiche auch die Beschreibung auf S. 95ff.

[2] Fahlman wollte damit sicherstellen, daß seine Maschine auch einmal direkt in Hardware realisiert werden kann. Das ist bis heute aber nicht geschehen.

[3] Das ist im allgemeinen die *Anzahl* der Kanten, da die Kanten kein unterschiedliches Gewicht haben.

Entfernung aller anderen Knoten, von denen aus unterschiedliche Farbströme über die *VC Kanten durch das Netzwerk fließen.

Touretzky beobachtete nun, daß die in NETL verwendeten Algorithmen in gewisser Weise *falsch* wahren. Das Ergebnis der Operationen hing in komplizierten Fällen von in den Algorithmen nicht spezifizierten Implementationsdetails ab. Außerdem konnte er Beispiele konstruieren, die wie in Abschnitt 5.7 betrachtet, von NETL nicht intuitiv befriedigend gelöst werden konnten. In [Touretzky, 1986] lieferte er zur Lösung dieser Probleme eine mathematische Definition der aus der Spezifikation eines Netzwerks herleitbaren Eigenschaften der Objekte. Seine Konstruktionen im ersten Teil seiner Arbeit erinneren an die Deduktionstheoreme der Aussagen- und Prädikatenlogik.[4]

Um die in Abschnitt 5.7 genannten zwei Probleme der redundanten Aussagen und der Mehrdeutigkeit zu lösen, ersetzte Touretzky in [Touretzky, 1986] die bis dahin in NETL als Entscheidungskriterium benutzte Pfadlänge durch eine neue Ordnungsrelation zwischen Knoten, die sicherstellt, daß die in dem Netzwerk vorhandene Klassenhierarchie *richtig* wiedergegeben wird.

Kernpunkt seiner vom ihm *Inferential Distance Ordering* getauften Ordnung ist die folgende Vorschrift: Ein Individuum oder eine Klasse A ist näher zu der Klasse B als zur der Klasse C genau dann, wenn es einen Inferenzpfad von *A durch B hindurch* nach C gibt. Durch diese Bedingung ergibt sich eine nur partielle Ordnung, so daß sich Klassen möglicherweise nicht vergleichen lassen. In diesen Fällen spricht Touretzky von einer Mehrdeutigkeit *(Ambiguity)*.

Das Vorgehen in den nächsten Abschnitten wird zunächst die informelle Darstellung der Idee der direkten pfadbasierten Methoden sein. Dann folgt die formale Definition der benötigten Konzepte, daran schließt sich die Diskussion der Eigenschaften des allgemeinen Basisformalismus an. Beschränkt man die volle Allgemeinheit, so ergeben sich interessante Spezialfälle, die definiert und

[4] Im zweiten Teil beschäftigt er sich mit der Umsetzung seines neu definierten Korrektheitsbegriffs in die Implementation von Algorithmen für Fahlmans PMP Maschine. Dazu konditioniert er das Netzwerk durch Hinzufügen neuer Kanten, so daß auch die ursprünglichen Programme der PMP Maschine jetzt *die* Resultate liefern, die nun mathematisch als korrekt definiert sind. So kann er dann die Effizienz der PMP Maschine nutzen. Es stellte sich aber bei der Komplexitätbetrachtung (siehe [Selman and Levesque, 1991] und Kapitel 7) heraus, daß sein Konditionierungsprozeß NP-schwer ist.

dann zur Einordnung des Vererbungsmechanismus von **DONALD-III** in die Theorie verwendet werden.

6.2 Die Idee der direkten Methode

Die direkte Methode betrachtet ein zur Wissensrepräsentation verwendetes Semantisches Netzwerk als einen endlichen gerichteten Graphen Γ (siehe Definition 2.4 auf Seite 31), der direkt gespeicherte Informationen enthält. Zu zeigen ist jetzt, wie aus dieser direkten Information in Γ durch Vererbungsmechanismen weitere Schlußfolgerungen gezogen werden können. Man stellt sich also vor, daß wie bei einer Datenbankabfrage die Gültigkeit von Aussagen oder Formeln in der Vererbungssprache abgefragt wird.

Zu diesem Zweck wird die Verbindung zwischen den Knoten x und y in einem Graphen Γ mit einer Aussage der Form $x \to y$ identifiziert, wobei die verschiedenen Kantentypen durch unterschiedliche Pfeiltypen ersetzt werden. Man erhält so zu jedem Graphen eine Menge von Aussagen. Aus diesen atomaren Aussagen lassen sich jetzt durch Aneinanderhängen *Pfade* durch das Netzwerk konstruieren.

Naheliegend wäre jetzt die Definition der in einem Netzwerk gültigen Aussagen als die durch die Menge *aller* konstruierbaren Pfade unterstützten Aussagen. Das erweist sich jedoch nicht als sinnvoll, da Pfade durch widersprechende Kantentypen und gleiche Anfangs- und Endpunkte in Konflikt zueinander stehen können. Diese Menge muß also durch die Definition *zulässiger* Pfade eingeschränkt werden. Ein Pfad ist zulässig *(permitted)*, wenn er nicht in direktem Konflikt zu einem anderen Pfad steht, und er nicht ausgeschlossen *(precluded)* ist. Ein Pfad ist precluded, wenn es in dem Netzwerk eine Verbindung von einem Zwischenknoten des Pfades *(Intermediary)* zu dem Endknoten mit widersprechender Beschriftung gibt. Alternativ formuliert sind Pfade permitted, wenn alle widersprechenden Pfade verhindert *(preempted)* werden.

Hat man die Menge aller zulässigen Pfade definiert, so gibt man die durch einen Pfad ermöglichten *(enabled)* Aussagen an, und spricht dann von der Menge der von den zulässigen Pfaden unterstützten *(supported)* Aussagen. Die neuen Aussagen werden als Erweiterung des ursprüngliche Netzwerks Γ

betrachtet. Diese Erweiterung oder *Extension* wird dann zur Beantwortung von Abfragen benutzt.

Es sind also die folgenden Fragen zu beantworten:

- Wie werden aus atomaren Aussagen und Pfaden neue Pfade konstruiert?
- Welche Pfade widersprechen sich?
- Welche konstruierbaren Pfade werden verhindert oder ausgeschlossen?
- Welche Aussagen werden von einem Pfad unterstützt?

6.3 Terminologie

Ein Vererbungsnetzwerk *(Inheritance Network)* ist ein beschrifteter gerichteter Graph, dessen Knoten Individuen und Klassen darstellen und dessen Kanten Relationen zwischen diesen Knoten bezeichnen. Statt von Individuen spricht man auch oft von Objekten und statt von Klassen von Eigenschaften *(Properties)* oder Sorten *(Kinds)*.

Im folgenden bezeichnen Buchstaben vom Anfang des Alphabets (a bis d) Individuen (auch Objekte genannt); Buchstaben aus der Mitte des Alphabets (m bis t) stehen für Klassen oder Eigenschaften. Buchstaben vom Ende des Alphabets (u bis z) stehen für Variablen, die sowohl Objekte als auch Eigenschaften bezeichnen können.

Ist y eine Eigenschaft, so stellen die Kanten $x \Rightarrow y$ und $x \not\Rightarrow y$ positive bzw. negative **strikte** Relationen dar. Ist x selbst wieder eine Eigenschaft, so stellt $p \Rightarrow q$ eine Aussage der Form „Alle p sind auch q" der Prädikatenlogik dar. Ebenso steht $p \not\Rightarrow q$ für „Es gibt kein p, das auch q ist". Für Objekte a gibt es folgende Äquivalenz zwischen Kanten und klassischen atomaren und negierten atomaren Aussagen: $a \Rightarrow p$ bzw. $a \not\Rightarrow p$ stehen für $p(a)$ bzw. $\neg p(a)$.

Im Gegensatz dazu gibt es für den Kantentyp $\rightarrow$, der die primäre Relation ISA darstellt, keine solche direkte Entsprechung in der klassischen Logik. Sind sowohl p als auch q Eigenschaften, so ist $p \rightarrow q$ nur eine *generische* Aussage.

$p \to q$ und $r \not\to q$ könnten z.B. so etwas wie „Vögel fliegen“ oder „Säugetiere können nicht fliegen“ bedeuten. Insbesondere ist es *nicht* zulässig, diese Kanten in die logischen Aussagen $\forall x(p(x) \supset q(x))$ bzw. $\forall x(r(x) \supset \neg q(x))$ zu übersetzen.

Griechische Großbuchstaben stellen Netzwerke dar, wobei ein Netzwerk aus einer Menge I von Individuen, einer Menge K von Klassen und eventuell einer Menge P von Eigenschaften *(Properties)* sowie einer Menge L von Kantentypen zusammen mit einer endlichen Teilmenge von $(I \cup K \cup P) \times L \times (I \cup K \cup P)$ (den Kanten) besteht. Statt Kanten spricht man in der Vererbungstheorie auch von *Links*.

Kleine griechische Buchstaben stehen für allgemeine Folgen von Kanten (Linksequenzen) und die speziellen Bedingungen genügenden *Pfade* (siehe Definition 2.6 auf Seite 32). Pfade werden noch als atomar oder zusammengesetzt, strikt oder anfechtbar und positiv oder negativ klassifiziert.

Ein Vererbungsnetzwerk heißt **baumförmig**, falls man seine Knoten und Kanten in Form eines Baumes anordnen kann. Im Gegensatz dazu erlaubt ein System mit mehrfacher Vererbung seinen Knoten, mehr als einen direkten Nachfolger zu haben. Eine Einschränkung wird hier aber typischerweise noch gemacht: Der Vererbungsgraph muß azyklisch sein (siehe Definition 6.23 auf Seite 200).

Ein **bipolares** Vererbungssystem enthält sowohl positive ($\to$) als auch negative ($\not\to$) Verbindungen. Ein **unipolares** System enthält nur positive Verbindungen. Damit können negative Aussagen wie „Clyde ist kein Vogel“ nicht direkt dargestellt werden. Man muß dann einen speziellen Knoten „Nicht-Vogel“ einführen, so daß die negative Aussage dann durch eine positive Verbindung von Clyde zu Nicht-Vogel dargestellt werden kann. NETL und das in [Touretzky, 1986] beschriebene System (im folgenden mit TMOIS bezeichnet) sind bipolar, während FRL und die Typsysteme von objektorientierten Programmiersprachen wie C++ [Dewhurst and Stark, 1990] unipolar sind.

Ein **nichtmonotones** Vererbungssystem erlaubt es, Ausnahmen zu den ererbten Eigenschaften zu spezifizieren. Sowohl NETL als auch TMOIS benutzen bipolare Verbindungen um diese Nichtmonotonie zu erreichen: Negative Verbindungen überschreiben positive ererbte Information und positive Verbindungen können wiederum negative ererbte Information überschreiben. Alternativ dazu gibt es auch Systeme, die einen besonderen Typ von expliziten Ausnahmeverbindungen benutzen (siehe z.B. [Etherington, 1987]).

Bipolarität und Monotonie sind voneinander unabhängige Eigenschaften. Die Typsysteme von objektorientierten Programmiersprachen sind unipolar monoton. Framesysteme wie FRL sind unipolar und nichtmonoton. **DONALD** ist dagegen bipolar und nichtmonoton. Ein neueres bipolares monotones System wird in [Thomason *et al.*, 1986] beschrieben.

Ein **homogenes** Vererbungssystem ist entweder monoton (alle Verbindungen sind strikt) oder durchgängig nichtmonoton (alle Verbindungen sind anfechtbar oder überschreibbar). Auch heterogene Systeme wurden schon vorgeschlagen (z.B. in [Etherington, 1987] und [Thomason and Horty, 1988]); diese enthalten dann eine Mischung aus strikten und überschreibbaren Verbindungen und können darstellen, daß gewisse Eigenschaften für Klassen *definierend* sind.

Die *Theorie* eines Netzwerks ist als Fixpunkt einer Menge von Pfaden definiert. Als Operator dient dabei die Anwendung von Inferenzregeln zur Konstruktion zulässiger Pfade. Inferenzregeln leiten neue Pfade aus bestehenden ab, wobei die Kanten des ursprünglichen Netzwerkes mit Pfaden der Länge 1 identifiziert werden und diese dann als Startmenge dienen.

Fixpunkte von nichtmonotonen Netzwerken werden statt Theorie **Erweiterung** genannt, da Theorien deduktiv sind und damit eindeutigen Charakter haben. Nichtmonotone Inferenz muß nicht rein deduktiv sein, da es je nach Definition des Begriffs des konstruierbaren und zulässigen Pfads außer Systemen mit eindeutig konstruierbaren Fixpunkten auch solche mit mehr als einem Fixpunkt gibt. Somit können nichtmonotone Netzwerke mehr als eine Erweiterung haben.

Jeder Erweiterung einer Menge von Aussagen wird die Menge der *Konklusionen* dieser Erweiterung zugordnet. Im speziellen Fall des bipolaren homogen nichtmonotonen Netzwerks gilt: $x_0 \rightarrow x_n$ ist eine Konklusion, falls die Erweiterung den Pfad $x_0 \rightarrow \cdots \rightarrow x_n$ enthält und entsprechend ist $x_0 \not\rightarrow x_n$ eine Konklusion, falls sie den Pfad $x_0 \rightarrow \cdots \rightarrow x_{n-1} \not\rightarrow x_n$ enthält. Es ist durchaus möglich, daß zwei unterschiedliche Erweiterungen dieselbe Menge von Konklusionen besitzen, falls diese durch unterschiedliche Inferenzpfade generiert wurden.

6.4 Formale Definitionen

Die in den oben betrachteten Ideen benutzten Begriffe sollen jetzt formal definiert werden.

Definition 6.1 (Netzwerk) *Ein Netzwerk Γ in der Vererbungstheorie ist ein Tupel $\langle I, K, P, L, E\rangle$, wobei*

(i) I ist eine Menge von Individuen.

(ii) K ist eine Menge von Klassen.

(iii) P ist eine Menge von Eigenschaften.

(iv) L ist eine Menge von Kantentypen.

(v) E ist eine beliebige Teilmenge von $(I \cup K \cup P) \times L \times (I \cup K \cup P)$.

Des öfteren wird auch eine alternative Formulierung für spezielle Kantentypen L gebraucht. Ist $L = \{+, -\}$ verzichtet man auf die Menge E und betrachtet stattdessen zwei neue Mengen: $E^+ \subseteq (I \cup K \cup P) \times (I \cup K \cup P)$ und $E^- \subseteq (I \cup K \cup P) \times (I \cup K \cup P)$ stellen die positiven und negativen Kanten dar.

Für einen beliebigen Kantentyp $l \in L$ wird die Kante $e = \langle x, l, y\rangle$ auch alternativ in der suggestiveren Form $e = x \rightsquigarrow y$ geschrieben. Einige der folgenden Definitionen machen nur für den Fall $L = \{+, -\}$ Sinn. Das wird aber im einzelnen vermerkt werden.

Definition 6.2 (Konzepte) *Die Menge aller Konzepte Π ist die Vereinigung von I, K und P:*

$$\Pi := I \cup K \cup P$$

In der Vererbungstheorie wird nur eine eingeschränkte Klasse von Kantentypen betrachtet. Die Kanten sich also nicht beliebig beschriftet. Der allgemeinste Fall ist hier $L \subseteq (M \times B) = (\{-, =\} \times \{+, -\})$.

Definition 6.3 (Homogenität) *Sei* $\Gamma = \langle I, K, P, L, E\rangle$ *mit* $L = M \times B$. *Dann heißt* Γ homogen, *wenn* M *nur genau ein Element enthält. Für* $M = \{-\}$ *heißt* Γ nichtmonoton, *für* $M = \{=\}$ monoton.

Definition 6.4 (Polarität) *Sei* $\Gamma = \langle I, K, P, L, E\rangle$ *mit* $L = M \times B$. *Dann heißt* Γ unipolar, *wenn* B *nur genau ein Element enthält, das dann immer mit '+' bezeichnet wird. Ansonsten heißt* Γ bipolar.

Für einen beliebigen Kantentyp $l \in L$ können also vier Fälle unterschieden werden, die durch unterschiedliche Symbole bei der Schreibweise einer mit l beschrifteten Kante e von x nach y anstelle des allgemeinen $x \rightsquigarrow y$ angedeutet werden.

$l = \langle =, + \rangle$	$e = x \Rightarrow y$	strikt positiv
$l = \langle =, - \rangle$	$e = x \not\Rightarrow y$	strikt negativ
$l = \langle -, + \rangle$	$e = x \rightarrow y$	anfechtbar positiv
$l = \langle -, - \rangle$	$e = x \not\rightarrow y$	anfechtbar negativ

Definition 6.5 (Aussagen) *Aussagen in einem Netzwerk* Γ *sind geordnete Paare von Konzepten der Form* $a = x \rightsquigarrow y$. *Für den interessantesten Fall* $L \in \{+, -\}$ *werden Aussagen mit* $x \rightarrow y$ *und* $x \not\rightarrow y$ *bezeichnet werden. Die Menge aller Aussagen in* Γ *wird mit* $A(\Gamma)$ *bezeichnet.*

In leichter Abwandlung des Begriffs eines Pfades aus Kapitel 2 (Definition 2.6 auf Seite 32) wird eine Linksequenz eingeführt:

Definition 6.6 (Linksequenz) *Eine Linksequenz* σ *der Länge* $n = \text{length}(\sigma)$ *in einem Netzwerk* $\Gamma = \langle I, K, P, L, E\rangle$ *ist definiert als Folge* $\{x_0 \rightsquigarrow x_1, x_1 \rightsquigarrow x_2, \cdots, x_{n-2} \rightsquigarrow x_{n-1}, x_{n-1} \rightsquigarrow x_n\}$ *von Kanten aus* E *mit Konzepten* $x_i \in \Pi$. *Eine solche Folge heißt Linksequenz von* x_0 *nach* x_n *und wird folgendermaßen geschrieben:*

$$\sigma = x_0 \rightsquigarrow x_1 \rightsquigarrow \cdots \rightsquigarrow x_n$$

Die Menge aller Linksequenzen in Γ *wird mit* $\Theta(\Gamma)$ *bezeichnet. Die in einer Linksequenz enthaltenen Knoten* $x_i, 0 \leq i \leq n$ *heißen auch* Token.

Für Linksequenzen $\sigma = x_0 \rightsquigarrow \cdots \rightsquigarrow x_n$ und $\tau = y_0 \rightsquigarrow \cdots \rightsquigarrow y_m \in \Theta(\Gamma)$ ist $\sigma \rightsquigarrow \tau$ einfach als Aneinanderkettung von σ und τ definiert: $\sigma \rightsquigarrow \tau := x_0 \rightsquigarrow \cdots \rightsquigarrow x_n \rightsquigarrow y_0 \rightsquigarrow \cdots \rightsquigarrow y_m$. Die einzelnen Teile einer Linksequenz werden auch als *Segmente* bezeichnet und können als Spezialfall leer sein.

Typischer Fall ist ein Pfad $\tau = x \to \sigma \to y$ mit beliebigen σ. τ steht für einen beliebigen Pfad mit Anfangspunkt x und Endpunkt y. Für den Fall $\sigma = \emptyset$ ist $\tau = x \to y$.

Zur Vereinfachung wird statt $\Gamma = \langle I, K, P, L, E\rangle$ unter Verwendung des Begriffs des Konzeptes auch $\Gamma = \langle \Pi, L, E\rangle$ betrachtet, falls die Unterscheidung zwischen I, K und P nicht erforderlich ist.

Definition 6.7 (Polarität von Netzwerken) *Ein Netzwerk $\Gamma = \langle \Pi, L, E\rangle$ mit $|L| = 1$, in der es also nur einen Typ von Kanten gibt, heißt* unipolar. *Enthält L nur zwei Elemente, d.h. L kann mit der Menge $\{+, -\}$ oder auch $\{\to, \not\to\}$ identifiziert werden, so heißt L* bipolar.

Definition 6.8 (Atomare Pfade) *Unter einem* atomaren *(oder* einfachen*) Pfad $\sigma = x \rightsquigarrow y$ versteht man eine Kante von Γ. Nicht atomare Pfade heißen* zusammengesetzt.

Die folgenden Definitionen beschäftigen sich zunächst nur mit homogen nichtmonotonen Netzwerken, die nur die beiden Kantentypen $\to$ und $\not\to$ enthalten.

Definition 6.9 (Homogen nichtmonotone Pfade) *Die Menge Σ der möglichen Pfade zu einem Netzwerk $\Gamma = \langle I, K, P, L, E\rangle$ wird rekursiv definiert:*

(i) Alle atomaren Pfade sind auch Pfade.

(ii) Für $p, q \in K \cup P$ und $\sigma \to p \in \Sigma$ ist sowohl $\sigma \to p \to q$ als auch $\sigma \to p \not\to q$ ein Pfad.

Bemerkung: Man beachte, daß Pfade nur durch Verbindungen zu Klassen oder Eigenschaften verlängert werden dürfen. Damit ist sichergestellt, daß Individuen in Pfaden nur als führende Elemente auftauchen. Ebenso sind alle Verbindungen positiv homogen, bis auf möglicherweise die letzte, da Pfade mit einem negativen Endstück nicht mehr weiter verlängerbar sind.

Definition 6.10 (Verallgemeinerte homogene Pfade) *Die Menge Σ' aller verallgemeinerten Pfade durch ein Netzwerk Γ wird rekursiv definiert:*

(i) Alle atomaren Pfade sind auch verallgemeinerte Pfade: $\Sigma' \supseteq A(\Gamma)$.

(ii) Für $\sigma \in \Sigma'$ und $p \in K \cup P$ ist sowohl $\sigma \to p$ als auch $\sigma \not\to p$ ein verallgemeinerter Pfad.

Pfade können ebenso wie die Kanten in Typen eingeteilt werden.

Definition 6.11 (Polarität von nichtmonotonen Pfaden) *Ein Pfad σ in einem Netzwerk Γ heißt* negativ*, wenn er ein negatives Segment enthält. Ansonsten heißt er* positiv*. Diese Eigenschaft wird auch als Polarität des Pfades bezeichnet.*

Bemerkung: Nach der vorhergehenden Bemerkung kann nur das allerletzte Segment eines Pfades negativ sein, deswegen ist die Polarität eines Pfades identisch mit der Polarität der letzten Kante. Für inhomogene Pfade ist die Situation etwas komplizierter, da man von strikten Endsegmenten abstrahieren muß (siehe Abschnitt 6.6.4 auf Seite 210).

Definition 6.12 (Unterstützte nichtmonotone Pfade) *Ein Pfad der Form $\tau = x \to \sigma \to y \in \Sigma$ mit beliebigem $\sigma \in \Theta$ unterstützt die Aussage $x \to y$ und $\tau = x \to \sigma \not\to y$ unterstützt ebenso $x \not\to y$. Dabei kann auch der Fall $\sigma = \emptyset$ auftreten.*

Definition 6.13 (Konflikt von Pfaden) *Ein Pfad $\sigma = x_0 \rightsquigarrow \cdots \rightsquigarrow x_n$ steht in einer Menge Φ von Pfaden in* Konflikt *genau dann, wenn es in Φ einen Pfad τ mit identischen Anfangs- und Endpunkten x_0 und x_n, aber entgegengesetzter Polarität gibt.*

Beispiel: Die Pfade $\sigma = x \to y \to z$ und $\tau = x \to p \to q \to r \not\to z$ stehen zueinander in Konflikt, da sie dieselben Anfangs- und Endpunkte haben, aber die letzte Kante von entgegengesetzter Polarität ist.

Konstruierbare Pfade sind dann *vererbbar*, wenn sie nicht mit anderen Pfaden in Konflikt stehen und nicht durch weitere Pfade verhindert werden. Die Konfliktrelation ist natürlich abhängig von der Menge der zu betrachtenden Pfade, die im folgenden mit Φ bezeichnet wird. In der englischen Literatur spricht man statt vererbbar von *permitted* oder *inheritable* sowie statt Konflikt und Verhinderung von *contradicted* und *preempted*.

Ein Pfad σ kann nun verhindert werden durch einen Pfad, der von einem intermediären Knoten von σ zum Endpunkt von σ führt und die σ entgegengesetzte Polarität hat. Dazu muß erst einmal geklärt werden, was ein Zwischenknoten eines Pfades ist. Dafür werden zwei alternative Definitionen gegeben, die mit *On-path* und *Off-path* bezeichnet werden. Welche unterschiedlichen Motivationen für die eine oder andere Definition sprechen, wird in Abschnitt 6.6.3 auf Seite 208 noch diskutiert werden. Es soll noch einmal betont werden, daß es sich *nicht* um *äquivalente* Definitionen handelt.

Definition 6.14 (On-path intermediäre Token) *Ein Token y eines Pfades $\sigma = x_0 \to \cdots \to x_n$ in Φ heißt genau dann* on-path intermediäres *Token von σ, wenn es ein $i \leq n$ gibt, so daß mindestens eine der beiden folgenden Bedingungen erfüllt ist:*

(i) $y = x_i$.

(ii) *Es exisitiert ein positiver Pfad $\tau = x_0 \to \cdots \to x_{i-1} \to y_0 \to \cdots \to y_m \to x_i$ in Φ dergestalt, daß $x = y_j$ für geeignetes j. Die Pfade σ und τ müssen also in ihren Anfangsstücken bis zu dem Token x_{i-1} einschließlich übereinstimmen.*

Definition 6.15 (Off-path intermediäre Token) *Ein Token y eines Pfades $\sigma = x_0 \to \cdots \to x_n$ in Φ heißt genau dann* off-path intermediäres *Token von σ, wenn es in Φ einen positiven Pfad τ von x_0 nach x_{n-1} gibt, so daß y ein Token von τ ist. Die Pfade σ und τ müssen also außer in ihrem Anfangspunkt x_0 und dem gemeinsamen Token x_{n-1} in* keinem *Token übereinstimmen.*

Aufbauend auf der Definition der intermediären Token eines Pfades kann jetzt der Begriff der *Preemption* unabhängig von der Entscheidung für on-path oder off-path erfolgen.

Definition 6.16 (Preemption) *Ein Pfad* $\sigma = x_0 \rightsquigarrow \cdots \rightsquigarrow x_n$ *wird genau dann in einer Menge* Φ *von Pfaden* verhindert (preempted), *wenn es ein intermediäres Token* y *von* σ *in* Φ *gibt und ein atomarer Pfad von* y *nach* x_n *mit entgegengesetzter Polarität in* Φ *enthalten ist.*

Definition 6.17 (Aufwärtskonstruktion von Pfaden) *Ein Pfad der Form* $\sigma = x_0 \rightsquigarrow \cdots \rightsquigarrow x_{n-1} \rightsquigarrow x_n$ *ist genau dann* aufwärts konstruierbar *aus einer gegebenen Menge* Φ *von Pfaden, wenn:*

(i) $x_{n-1} \rightsquigarrow x_n \in \Phi$ *und*

(ii) $x_0 \rightsquigarrow \cdots \rightsquigarrow x_{n-1} \in \Phi$.

Man spricht auch von Upward Concatenation.

Definition 6.18 (Abwärtskonstruktion von Pfaden) *Ein Pfad der Form* $\sigma = x_0 \rightsquigarrow x_1 \rightsquigarrow \cdots \rightsquigarrow x_{n-1} \rightsquigarrow x_n$ *ist genau dann* abwärts konstruierbar *aus einer gegebenen Menge* Φ *von Pfaden, wenn:*

(i) $x_1 \rightsquigarrow \cdots \rightsquigarrow x_n \in \Phi$ *und*

(ii) $x_0 \rightsquigarrow \cdots \rightsquigarrow x_{n-1} \in \Phi$.

Man spricht auch von Downward Concatenation.

Definition 6.19 (Vererbbarer Pfad) *Ein Pfad* σ *ist genau dann* vererbbar (inheritable) *in* Φ, *wenn er die folgenden drei Bedingungen erfüllt:*

(i) σ *ist konstruierbar mit Hilfe von Pfaden aus* Φ.

(ii) σ *steht nicht in Konflikt mit Pfaden aus* Φ.

(iii) σ *wird nicht verhindert (preempted) durch Pfade aus* Φ.

Für σ *ist vererbbar in* Φ *schreibt man auch* $\Phi \models \sigma$.

Ein Pfad ist also vererbbar in einer Menge Φ von Pfaden, wenn er herleitbar ist und nicht durch andere Pfade ungültig gemacht werden kann. Die Suche nach einer passenden Menge Φ zu einem gegebenen Netzwerk Γ und Methoden zu ihrer Erweiterung steht noch aus. Auch hier gibt es wieder zwei Möglichkeiten. Die Definition aus Touretzkys Originalarbeit wird heute als *credulous* bezeichnet, da sie so angelegt ist, daß eine möglichst große Menge von Pfaden vererbbar ist. Seinen Ansatz kann man auch als *leichtgläubig* oder *wissenshungrig* einordnen. Zuerst wird nun definiert, wann eine Menge unter Vererbbarkeit abgeschlossen ist.

Definition 6.20 (Abschluß) *Eine Menge Φ von Pfaden heißt abgeschlossen unter Vererbbarkeit oder* closed under inheritance *genau dann, wenn Φ bereits alle in Φ vererbbaren Sequenzen enthält.*

Definition 6.21 (Grounded) *Eine Menge Φ von Pfaden heißt* grounded *in einer Menge $S \subseteq \Phi$ genau dann, wenn jede Sequenz in $\Phi \setminus S$ vererbbar in Φ ist.*

Bemerkung: Groundedness und closed under inheritance sind voneinander unabhängig. Sei $\Gamma = \{a \to b, b \to c\}$.

1. $\Phi = \Gamma$ ist grounded aber nicht closed.
2. $\Phi = \Gamma \cup \{a \to b \to c\}$ ist grounded und closed.
3. $\Phi = \Gamma \cup \{d \to e\}$ ist weder grounded noch closed.
4. $\Phi = \Gamma \cup \{a \to b \to c, d \to e\}$ ist nicht grounded aber closed.

Definition 6.22 (Credulous Grounded Extension) *Die Menge Φ ist genau dann eine* credulous grounded Extension *einer Menge von Kanten Γ, wenn die beiden folgenden Bedingungen erfüllt sind:*

(i) $\Phi \supseteq \Gamma$ *und*

(ii) Für alle Pfade σ gilt:

$$\sigma \in \Phi \setminus \Gamma \quad \Leftrightarrow \quad \Phi \models \sigma.$$

Leider liefert diese Definition zu einem gegebenen Γ nicht in allen Fällen eine eindeutig bestimmte credulous grounded Extension. Ein Netzwerk kann daher mehrdeutig *(ambigous)* sein. Credulous impliziert also, daß ein so arbeitendes System eventuell eine ganze Anzahl von Alternativen in unterschiedlichen Erweiterungen betrachten muß.

Diese Tatsache motivierte Horty, Thomason und Touretzky in [Horty *et al.*, 1987b] zu der Entwicklung einer Definition von Erweiterung, die die Existenz einer eindeutig bestimmten Extension garantiert. Dieser neue Ansatz wird im Gegensatz zu der obigen *wissenshungrigen* Methode als *skeptisch* bezeichnet. Die skeptische Methode definiert eine Erweiterung induktiv durch Hinzunahme von weiteren vererbbaren Pfaden nur unter der Bedingung, daß der neue Pfad nicht durch die Vererbbarkeit eines Pfades mit gleichem Anfangs- und Endpunkt, aber entgegengesetzter Polarität verhindert wird. Bevor eine solche induktive Definition angegeben werden kann, muß noch ein Begriff für die Komplexität eines Pfades gefunden werden.

Entscheidend für die Möglichkeit induktiver Definitionen und Beweise ist der Begriff des *Grades* eines Pfades. Die Pfade mit dem minimalem Grad 1 sind gerade die Kanten des Netzwerks selbst und bilden die Grundlage der Induktionen. Damit der Grad endlich bleibt, werden nur azyklische Pfade bzw. Netzwerke betrachtet.

Definition 6.23 (Azyklische Netzwerke) *Eine Menge von Kanten des gegebenen Netzwerks Γ heißt genau dann* azyklisch*, wenn der von den Konzepten von Γ gebildete Graph azyklisch ist. Ein Graph ist azyklisch, wenn es keine Pfade gibt, in denen ein und derselbe Knoten mehr als einmal vorkommt.*

Definition 6.24 (Grad) *Der Grad eines Pfades $\tau = x \rightsquigarrow \sigma \rightsquigarrow y$ in einem Netzwerk Γ mit der Menge G_Γ von azyklischen verallgemeinerten Pfaden ist definiert als die Länge des längsten Pfades von x nach y in G_Γ:*

$$\deg_\Gamma(\tau) := \max_{\tau' = x \rightsquigarrow \sigma' \rightsquigarrow y \in G_\Gamma} length(\tau')$$

Bemerkung: Für $A \subseteq B$ gilt $\deg_A(\sigma) \leq \deg_B(\sigma)$, da bei der Maximumsbildung über B mehr Pfade in Betracht gezogen werden.

Bemerkung: Für Pfade σ mit $\deg_\Gamma(\sigma) = 1$ gilt $\sigma = x \rightarrow y$ und damit ist σ atomar. Die Umkehrung gilt aber nicht, da noch weitere eventuell längere Pfade von x nach y existieren können.

Definition 6.25 (Skeptical Grounded Extension) *Die Menge Φ ist genau dann eine skeptical grounded Extension einer Menge von Kanten Γ, wenn*

$$\Phi = \bigcup_{i=1}^{\infty} \Phi_i, \ \textit{wobei}$$

(i) $\Phi_1 := \Gamma$ *und*

(ii) $\Phi_{i+1} \supseteq \Phi_i$ *und enthält alle Pfade σ mit $\deg_\Gamma(\sigma) = i + 1$, die die folgenden Bedingungen erfüllen:*

- *σ ist konstruierbar aus zwei Pfaden in Φ_i.*
- *Es gibt keinen atomaren oder vererbbaren Pfad in Φ_i, der denselben Anfangs- und Endpunkt aber entgegengesetzte Polarität hat.*

Anstelle von Definition 6.25 findet sich in [Horty *et al.*, 1987b] die folgende Originaldefinition:

Definition 6.26 ([Horty *et al.*, 1987b]) *Ein Pfad σ ist in einem Netzwerk Γ vererbbar, in Zeichen $\Gamma \models \sigma$, falls die folgende Fallunterscheidung gilt:*

(i) *Für einfache Pfade der Form $\sigma = x \rightarrow y$ oder $\sigma = x \not\rightarrow y$ gilt genau dann $\Gamma \models \sigma$, wenn σ bereits ein Element von Γ war.*

(ii) *Für einen zusammengesetzten Pfad σ mit $\deg_\Gamma(\sigma) = n, n > 2$ sind zwei symmetrische Fälle zu unterscheiden:*

σ positiv: *Für einen positiven Pfad σ von der Form $x \rightarrow \sigma_1 \rightarrow u \rightarrow y$ gilt $\Gamma \models \sigma$ genau dann, wenn:*

1. $\Gamma \models x \rightarrow \sigma_1 \rightarrow u$,

2. *$u \to y \in \Gamma$,*
3. *$x \not\to y \notin \Gamma$,*
4. *Für alle v, für die $\Gamma \models x \to \tau \to v$ mit $v \not\to y \in \Gamma$ gilt, gibt es ein z ($z \neq v$), so daß $\Gamma \models x \to \tau_1 \to z \to \tau_2 \to v$ und $z \to y \in \Gamma$.*

<u>*σ negativ:*</u> *Für einen negativen Pfad σ von der Form $x \to \sigma_1 \to u \not\to y$ gilt $\Gamma \models \sigma$ genau dann, wenn:*

1. *$\Gamma \models x \to \sigma_1 \to u$,*
2. *$u \not\to y \in \Gamma$,*
3. *$x \to y \notin \Gamma$,*
4. *Für alle v, für die $\Gamma \models x \to \tau \to v$ mit $v \to y \in \Gamma$ gilt, gibt es ein z ($z \neq v$), so daß $\Gamma \models x \to \tau_1 \to z \to \tau_2 \to v$ und $z \not\to y \in \Gamma$.*

Hierbei implizieren die Punkte 1 und 2 einen Forward Chaining Algorithmus (vergleiche auch Abschnitt 2.5.4 auf Seite 60). Wie man sieht, kommt diese Definition ohne Rückgriff auf die vorher gegebenen Definitionen aus. Sie ist dementsprechend kompliziert und unverständlich.

Es bleibt nur noch das in der Vererbungstheorie zu lösende Problem zu definieren:

Definition 6.27 (Konklusionsmenge) *Eine Menge Ψ von Kanten ist genau dann eine* Konklusionsmenge *für ein Netzwerk Γ, wenn es eine grounded Extension Φ von Γ gibt, so daß Ψ der Menge aller von Pfaden aus Φ unterstützten Aussagen (Konklusionen) entspricht.*

Definition 6.28 (Vererbungsproblem) *Zu einem gegebenem azyklischen Netzwerk Γ bestimme eine Konklusionsmenge von Γ.*

In [Horty *et al.*, 1987b] wird das Vererbungsproblem etwas enger gefaßt: Bestimme, ob eine gegebene Aussage von der eindeutigen Erweiterung des azyklischen Netzwerks Γ unterstützt wird. Wie in Abschnitt 7.6 auf Seite 226 noch genauer erklärt wird, ist dieses Problem aber nicht grundsätzlich einfacher, da sich die gesamte Erweiterung durch $O(n^2)$ Anfragen dieser Art berechnen läßt.

6.5 Eigenschaften

Theorem 6.1 *Ist Φ eine grounded Extension von Γ, dann gilt:*

$$x \to y \in \Phi \quad \Leftrightarrow \quad x \to y \in \Gamma.$$

Beweis: Für $x \to y \in \Gamma$ gilt $x \to y \in \Phi$, da $\Phi \supseteq \Gamma$.
Umgekehrt sei $x \to y \in \Phi$, dann ist $x \to y \in \Gamma$, da Φ grounded ist und Sequenzen der Länge eins nicht inheritable in Φ sind. ■

Theorem 6.2 *Ist Φ eine grounded Extension von Γ mit $\sigma = x_0 \to x_1 \to \cdots \to x_n \in \Phi$, dann gilt:*

$$x_0 \to x_1 \to \cdots \to x_j \in \Phi \quad \textit{für alle} \quad 0 < j \leq n$$

Beweis: Per Induktion über $d = \deg(\sigma)$. Für $d = 1$ ist die Aussage trivialerweise richtig. Sei die Aussage jetzt für all $d < n$ bereits gezeigt. Aus $\sigma \in \Phi$ ergibt sich, daß $\Gamma \models \sigma$, da $d > 1$ und Φ grounded ist (Definition 6.21). Daraus folgt mit Definition 6.17 bzw. Definition 6.18 auf Seite 198: $\Gamma \models x_0 \to x_1 \to \ldots \to x_{n-1}$ und damit $x_0 \to x_1 \to \ldots \to x_{n-1} \in \Phi$, da Φ eine grounded Extension von Γ ist. ■

Bemerkung: Touretzkys stärkere Version von Theorem 6.2, die besagt, daß

$$\Gamma \models x_i \to \ldots \to x_j \quad \text{für alle} \quad 1 \leq i < j \leq n$$

bleibt unter Verwendung von Definition 6.17 nicht mehr richtig, wie das Gegenbeispiel in Abbildung 6.1 zeigt:

Abbildung 6.1 ist die graphische Darstellung der folgenden Situation:

$$\Gamma := \{a \to p, p \to q, p \to r, a \not\to r, q \to s, r \not\to s\}$$

(i) $\Gamma \models a \to p \to q \to s$, da der potentiell widersprechende Pfad $a \to p \to r \not\to s$ wegen des Widerspruchs zwischen $a \to p \to r$ und $a \not\to r$ nicht konstruierbar ist.

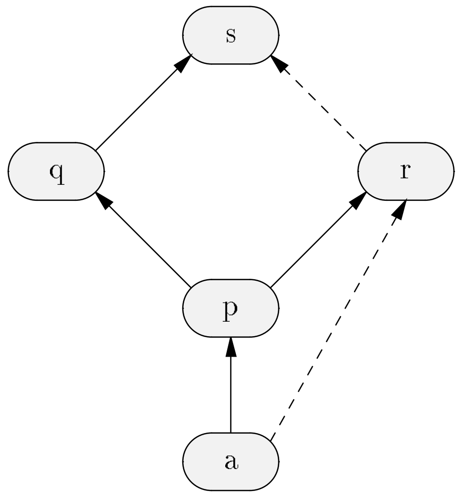


Abbildung 6.1: Gegenbeispiel

(ii) $\Gamma \not\models p \to q \to s$, da diesem Pfad $p \to r \not\to s$ widersprechen würde.

(iii) Aus (i) und (ii) ergibt sich, daß die Bemerkung im Fall skeptischer Vererbung nicht mehr gültig ist, dann sonst wären die *beiden* Segmente $a \to p \to q$ und $p \to q \to s$ vererbbar in Γ.

Korollar 6.1 *Falls* $\Gamma \models x_0 \to \ldots \to x_n$, *dann gilt:*

$$\Gamma \textit{ supports } x_i \to x_{i+1} \quad \textit{für alle} \quad i < n.$$

Theorem 6.3 *Für grounded Extensions* Φ_a *und* Φ_b *von* Γ *gilt:*

$$\Phi_a \subseteq \Phi_b \quad \Leftrightarrow \quad \Phi_a = \Phi_b$$

Beweis: Die Rückrichtung ist klar. Angenommen $\Phi_a \subseteq \Phi_b$ aber $\Phi_a \neq \Phi_b$. Dann gibt es ein $\sigma = x_0 \to \delta \to x_n \in \Phi_b \setminus \Phi_a$ mit folgenden Eigenschaften: $\Gamma \models \sigma$ und σ ist von minimalem Grad mit dieser Eigenschaft. Da σ inheritable ist, gilt: $\Gamma \models x_0 \to \delta$ und $x_{n-1} \to x_n \in \Gamma$. Da σ minimal war, muß $x_0 \to \delta \in \Phi_a \cap \Phi_b$ sein.
Wenn aber nun $x_0 \to \delta \in \Phi_a$ und $x_{n-1} \to x_n \in \Gamma$, dann gilt: $\Phi_a \models x_0 \to \delta \to x_n$ und damit $x_0 \to \delta \to x_n \in \Phi_a$, da Φ_a grounded in Γ ist, im Widerspruch zur Annahme. ∎

Korollar 6.2 *Grounded Extensions sind minimal.*

Konzentriert man die Betrachtung auf die skeptische on-path upward Vererbung, so gelten die folgenden zwei Theoreme und das nächste Lemma. Für Beweise wird auf [Horty *et al.*, 1987b, S. 17ff.] verwiesen.

Theorem 6.4 (Korrektheit) *Falls ein Netzwerk* Γ *sowohl* $x \rightarrow y$ *als auch* $x \not\rightarrow y$ *supported, dann ist sowohl* $x \rightarrow y \in \Gamma$ *als auch* $x \not\rightarrow y \in \Gamma$.

Ist ein Algorithmus in gewisser Weise unabhängig von redundanter Information, so spricht man von einem *stabilen* Algorithmus. Es gibt im allgemeinen eine Reihe von weiteren verschiedenen Stabilitätseigenschaften.

Theorem 6.5 (Stabilität) *Für eine atomare Aussage* A *gilt folgendes: Falls* Γ *allein schon* A *unterstützt, dann gilt für alle Aussagen* B:

$$\Gamma \cup \{A\} \text{ unterstützt } B \quad \Leftrightarrow \quad \Gamma \text{ unterstützt } B$$

Grundlage für den Beweis des vorhergehenden Theorems bildet das folgende Lemma:

Lemma 6.1

(i) *Angenommen* $\Gamma \models a \rightarrow \delta \rightarrow p$. *Dann gilt:*

$$\Gamma \models \sigma \quad \Rightarrow \quad \Gamma \cup \{a \rightarrow p\} \models \sigma$$

$$\Gamma \cup \{a \rightarrow p\} \models \sigma \quad \Rightarrow \quad \Gamma \models \sigma^*$$

wobei σ^* *durch Ersetzen aller Kanten* $a \rightarrow p$ *in* σ *durch den Pfad* $a \rightarrow \delta \rightarrow p$ *ensteht.*

(ii) *Angenommen* $\Gamma \models a \rightarrow \delta \not\rightarrow p$. *Dann gilt:*

$$\Gamma \models \sigma \quad \Rightarrow \quad \Gamma \cup \{a \not\rightarrow p\} \models \sigma$$

$$\Gamma \cup \{a \not\rightarrow p\} \models \sigma \quad \Rightarrow \quad \Gamma \models \sigma^*$$

wobei σ^* *jetzt durch Ersetzen aller Kanten* $a \not\rightarrow p$ *in* σ *durch den Pfad* $a \rightarrow \delta \not\rightarrow p$ *ensteht.*

6.6 Theoretische Alternativen

Die folgende Abschnitte diskutieren die Gründe für die in Definition 6.14 bis Definition 6.22 noch verbleibenden Freiheitsgrade zur Definition eines konkreten Vererbungssystems und den Einfluß der Entscheidung für eine spezielle Definition auf die Konklusionsmenge eines Netzwerks Γ.

6.6.1 Wissenshungrige und skeptische Methoden

Für den Begriff einer Erweiterung wurden zwei unterschiedliche Definitionen angeben, nämlich in Definition 6.22 eine wissenshungrige oder leichtgläubige und in Definition 6.25 eine mehr skeptische Definition.

Ein skeptisches System weigert sich, in mehrdeutigen Situationen noch weitere Schlüsse zu ziehen. Das klassische Beispiel für ein mehrdeutiges Netzwerk ist das in Abbildung 5.3 auf Seite 176 dargestellte *Nixon diamond*. Ein leichtgläubiges System würde in dieser Situation versuchen, möglichst viel herzuleiten. Um Inkonsistenzen innerhalb einer Erweiterung zu vermeiden, ist ein solches System gezwungen, mehr als eine Erweiterung zu konstruieren. So werden die sich widersprechenden Aussagen in verschiedenen Erweiterungen voneinander isoliert. Im Fall des Nixon Diamond würde das System [Touretzky, 1986] zwei Erweiterungen zulassen. In der einen ist der Pfad Nixon $\rightarrow$ Quäker $\rightarrow$ Pazifist enthalten und in der anderen Nixon $\rightarrow$ Republikaner $\not\rightarrow$ Pazifist.

Skeptische Systeme erzeugen dagegen immer eindeutig bestimmte Erweiterungen, so daß es nicht erforderlich ist, mehrere Erweiterungen weiterzuverfolgen. Die Vermutung liegt jetzt nahe, daß es sich bei einer skeptischen Erweiterung um den Durchschnitt aller leichtgläubigen Erweiterungen handelt. Das in Abbildung 6.2 dargestellte kaskadierte Nixon Diamond belegt aber, daß diese Annahme falsch ist.

Nixons Pazifismus ist nicht eindeutig feststellbar, und deswegen auch nicht seine Gefühle dem Militär gegenüber. Also enthält der Durchschnitt der leichtgläubigen Erweiterungen keine dieser Aussagen. Ein skeptisches System würde die zweite Mehrdeutigkeit nicht erkennen, da nicht entscheidbar ist, ob Nixon

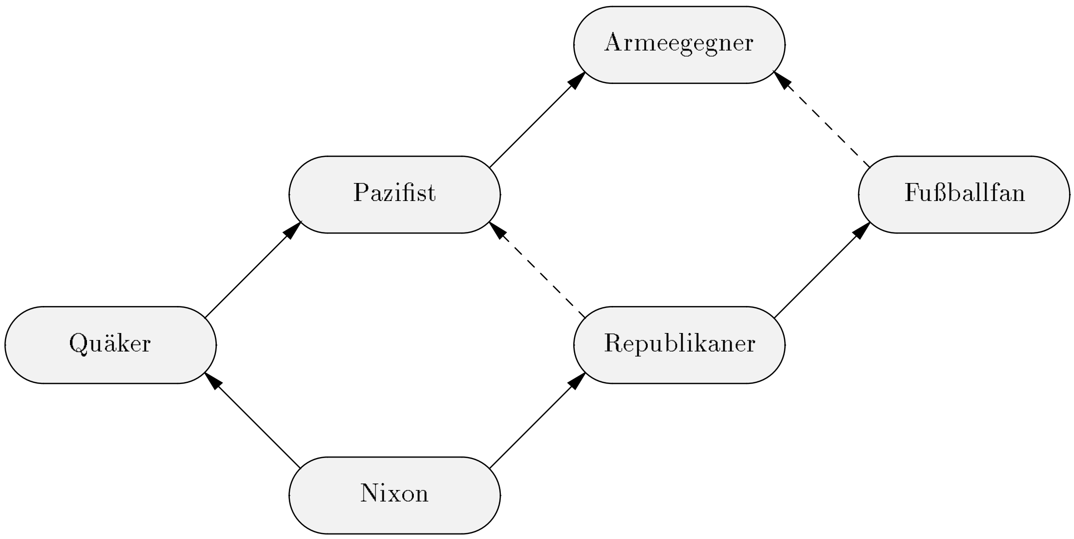


Abbildung 6.2: Das kaskadierte Nixon-Diamond.

ein Pazifist ist oder nicht. Demzufolge gibt es keine gültigen Pfade von Nixon über Pazifist zu Armeegegner in der skeptischen Erweiterung. Also steht dem Pfad von Nixon über Fußballfan zu dem Gegenteil von Armeegegner nichts mehr im Wege, und es kann geschlossen werden, daß Nixon *kein* Armeegegner ist.

6.6.2 Aufwärts- und Abwärtskonstruktion

Auch die durch Auswahl von Definition 6.17 oder Definition 6.18 getroffene Entscheidung hat wesentlichen Einfluß auf das Verhalten des resultierenden Systems.

Konstruiert man Pfade abwärts, so spricht man auch von *Backward Chaining*. Diese Methode impliziert, das die Konklusionen über einen Pfad in einem Netzwerk von seinen direkten Vorgängern abhängen. Der Pfad $x \to u \to \sigma \to y$ gehört z.B. nur dann zu einer Erweiterung, wenn der Pfad $u \to \sigma \to y$ ebenfalls in ihr enthalten ist (vergleiche auch die Bemerkung nach Theorem 6.2). Ein Pfad $x \to u \to \sigma \not\to y$ gehört nur dann zu einer Erweiterung, wenn der Pfad $u \to \sigma \not\to y$ in ihr enthalten ist.

Wie Abbildung 6.1 auf Seite 204 zeigt, kommt es dagegen beim *Forward Chaining* (Aufwärtskonstruktion) zu einer Entkopplung *(decoupling)* von den direkten Vorgängern in dem Sinne, daß Objekte Eigenschaften besitzen können, die keiner ihrer direkten Vorgänger besessen hat. Das Netzwerk Γ aus Abbildung 6.1 unterstützt $a \to p \to q \to s$, da das Anfangssegment $a \to p \to r$ des Pfades $a \to p \to r \not\to s$ in Konflikt mit $a \not\to r$ steht. Obwohl nun Γ den Pfad $a \to p \to q \to s$ erlaubt und damit die Konklusion $a \to s$ unterstützt, ist $p \to q \to s$ nicht erlaubt und damit wird $p \to s$ *nicht* unterstützt.

Absolut entscheidende Bedeutung kommt der Unterscheidung zwischen *upward* und *downward* jedoch noch aus einem anderen Grunde zu. Diese Eigenschaft eines Vererbungssystems bestimmt nämlich die Komplexität des Vererbungsalgorithmus. Während Forward Chaining in polynomialer Zeit durchgeführt werden kann, ist das *Backward Chaining* **NP**-schwer. Mit dieser Fragestellung wird sich aber Kapitel 7 noch eingehend beschäftigen.

6.6.3 On-path und Off-path intermediäre Token

Die letzte verbleibende Designentscheidung ist nun bezüglich Definition 6.14 und Definition 6.15 auf Seite 197 zu treffen. Dort werden die intermediären Token eines Pfades definiert, die wieder direkt die Interpretation des Begriffs der Verhinderung *(Preemption)* von Pfaden (Definition 6.16 auf Seite 198) bestimmen.

On-path Verhinderung ist der strengere Begriff: Ein Pfad kann einen anderen nur dann verhindern, wenn der zu verhindernde Pfad eine redundante Kante enthält, die ansonsten ein Segment des Verhinderers kurzschließen würde. Zur Verdeutlichung betrachte man Abbildung 6.3. Wenn ein Knoten eine redundante Kante unterbricht, so wie der afrikanische Elephant die Kante von Clyde nach Elephant unterbricht, dann gilt die Verhinderungsrelation nicht länger. Damit generiert das abgebildete Netzwerk die beiden widersprüchlichen Pfade Clyde $\to$ Königselephant $\not\to$ Grautier und Clyde $\to$ Königselephant $\to$ Elephant $\to$ Grautier.

Man kann sich aber auf den Standpunkt stellen, daß eine weniger strenge Regel, nämlich die off-path Preemption, sinnvoller ist, indem man folgendermaßen argumentiert: Sowohl der Pfad von Königselephant zu Grautier als

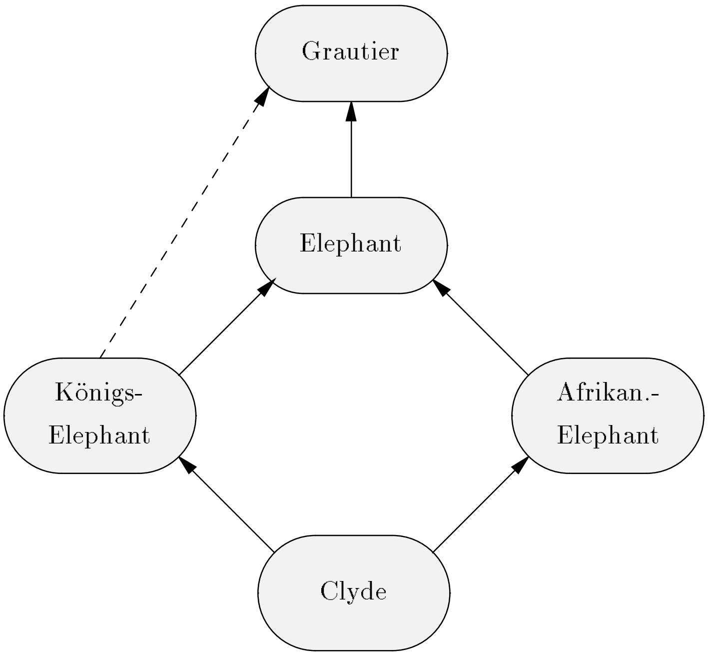


Abbildung 6.3: Das Problem des Königselephanten

auch von afrikanischer Elephant zu Grautier gehen durch den Knoten Elephant. Das afrikanische Elephanten typischerweise grau sind, sollte Clyde nicht beeinflussen, da afrikanische Elephanten nicht *explizit* grau sind, sie erben diese Information nur von Elephant. Königselephanten überschreiben die Farbe eines Elephanten mit spezifischerer Information. Diese Argumentation unterstützt die Aussage „Clyde ist *nicht* grau“. Die off-path Verhinderung ist im allgemeinen eher gewillt, einen Pfad einen anderen überschreiben zu lassen.

Zusammenfassend muß gesagt werden, daß man in der Tat nicht von *der* direkten Vererbungsmethode sprechen kann. Es handelt sich um eine ganze Familie von Ansätzen. Während der Einsatz des Backward Chaining sich in einem objektiven feststellbaren Effizienzproblem gegenüber dem Forward Chaining manifestiert, lassen sich für die anderen Alternativen Gründe für und gegen den Einsatz anführen.

6.6.4 Erweiterungen

Es wurden auch schon Versuche unternommen, die Vererbung weiter zu verallgemeinern, um den Nachteil der vergleichsweise geringen Ausdrucksstärke gegenüber der Prädikatenlogik zu verkleinern:

Thomason und Horty gelingt in [Thomason and Horty, 1988] die Integration strikter Kantentypen in einem inhomogenen Vererbungssystem. Die Idee ihrer Konstruktion ist die Aufteilung der Pfade in strikte und anfechtbare Segmente, die nur in bestimmten Positionen innerhalb eines Pfades auftreten dürfen. Zentrale Rolle nimmt die Definition eines inhomogenen Pfades ein, die aufgrund von zehn Fallunterscheidungen für die verschiedenen Kombinationen der möglichen Segmenttypen wesentlich komplexer ist als Definition 6.9 auf Seite 195.

[Touretzky and Thomason, 1988] gibt für generische reflexive Aussagen der Form „Elephanten lieben sich selbst" eine Darstellung in Form eines Netzwerks mit speziellen zusätzlich eingeführten Knoten. Das System kann sowohl neue generische Aussagen als auch Aussagen über Individuen herleiten. Im Gegensatz zu allen logischen Formalismen werden *keine* Variablen verwendet.

Die Repräsentation von Booleschen Ausdrücken wie Negation, Konjunktion und Disjunktion ist Thema des Artikels [Horty and Thomason, 1990]. Zur Vereinfachung der theoretischen Analyse von Semantischen Netzwerken war diese Problematik immer ausgespart worden. Der vorgestellte Lösungsansatz erscheint aber recht kompliziert und eine Komplexitätsbetrachtung steht noch aus.

Thomason kann in [Thomason, 1989] auch Komplexitätsabschätzungen für monotone Netzwerke mit Relationen[5] und Identität herleiten. Er erhält eine Schranke von $O(l^5 \cdot n^{10})$, wobei l die Anzahl der verschiedenen relationalen Kantentypen ist und n die Zahl der Knoten.

[Thomason and Touretzky, 1991] beschäftigt sich mit der Integration der von Fahlman in NETL [Fahlman, 1979] definierten *Rollen* in monotone Vererbungsnetzwerke. Rollen sind dabei partielle Funktionen von Individuen auf

[5] In einem solchen System gibt es nicht nur die vier Basiskantentypen.

Individuen und dienen zur Darstellung von Aussagen der Form „Jede Person hat einen Vater“. Die definierten Inferenzregeln und die daraus resultierenden Beweise sind recht komplex, da man nicht mehr mit einfachen Ketten von Beweisen auskommt, sondern sich Beweisbäume ergeben.

In [Touretzky *et al.*, 1991] werden die in dieser Arbeit verwendeten Definitionen, die bereits eine Verallgemeinerung der Definitionen von [Touretzky, 1986] sind, im Hinblick auf den Begriff Preemption noch weiter gefaßt. Zusätzlich zu den bekannten *Conflictors* und *Preemptors* werden hier noch *Situator*, *Defeater*, *Reinstater* und *Zombies* eingeführt. Ziel der Diskussion ist die Klärung der Frage, auf welche verschiedene Art und Weise anfechtbare Information ihrerseits selbst wieder angefochten werden kann.

6.7 Klassifikation von Netzwerken

Definition 6.29 (Taxonomische Hierarchie) *Ein Netzwerk Γ heißt* Taxonomische Hierarchie, *wenn alle Kanten von Γ positiv sind. In der Terminologie von Definition 6.1 bedeutet das $E^- = \emptyset$. Weiterhin muß gelten:*

$$x \to y, x \to z \in \Gamma \quad \Rightarrow \quad y = z$$

Es handelt sich bei Taxonomischen Hierarchien also nach der Graphenterminologie um einen Wald von Bäumen.

Definition 6.30 (Tangled Hierarchy) *Eine* Tangled Hierarchy *ist ein homogen nichtmonotones bipolares Netzwerk $\Gamma = \langle I, K, P, E^+, E^- \rangle$*[6]*, wobei*

(i) I ist eine Menge von Individuen.

(ii) K ist eine Menge von Klassen.

(iii) P ist eine Menge von Eigenschaften.

(iv) $E^+ \subseteq (I \cup K) \times (K \cup P)$ ist eine Menge von positiven Kanten.

[6] Hier wurde die bei der Definition des Begriffs Netzwerk eingeführte Alternative für $L = \{+, -\}$ benutzt.

(v) $E^- \subseteq (I \cup K) \times P$ *ist eine Menge von negativen Kanten.*

Bemerkung: Sei Φ eine Erweiterung einer Tangled Hierarchy. Dann gilt für einen Pfad $\sigma = x_0 \to \cdots \to x_{n-1}$, daß die Knoten x_1 bis x_{n-2} vom Typ K sind.

Tangled Hierarchies erlauben im Gegensatz zu taxonomischen Hierarchien die mehrfache Vererbung. Betrachtet man die in Abschnitt Abschnitt 4.8.3 auf Seite 146 angegebene Übersetzung einer Frame Hierarchie der Expertensystem-Shell **DONALD-III** in ein Semantisches Netzwerk, so wird deutlich, daß es sich bei dem entstehenden Netzwerk um eine Tangled Hierarchy handelt. Der Grund dafür ist, daß negative Kanten nur zu Eigenschaften erlaubt sind, und nicht auch zwischen Klassen. Die Definition semantischer Strukturen zu solchen Netzwerken findet sich in [Krishnaprasad, 1989, S. 72ff].

In **DONALD-III** wird die zu einem Individuum oder einer Teilklasse spezifizierte Vererbungsinformation strikt von links nach rechts abgearbeitet. Dies impliziert eine Präferenzreihenfolge unter den von diesem Token ausgehenden Kanten. Netzwerke mit solchen Prioritätsinformationen werden im nächsten Abschnitt noch genauer betrachtet.

6.8 Netzwerke mit Präferenzen

Netzwerke mit Präferenzen *(Preferential Networks)* spezifizieren auf den zu einem Token gehörenden Kanten eine Präferenzreihenfolge, die bewertet, wie spezifisch die Information ist, die entlang einer Kante vererbt wird.

Da neben ererbter auch direkt spezifizierte Information bei der Erzeugung eines neuen Token vorliegt, die sich in jedem Fall bei der Vererbung aus mehreren Quellen durchsetzen soll, ergeben sich für die Ordungsrelation aber gewisse Bedingungen, die erfüllt sein müssen. Ebenso ist die Klassen – Teilklassen Beziehung, die durch die Topologie des Netzwerks gegeben ist, zu berücksichtigen. Insgesamt liegen eine Reihe von Einschränkungen vor, so daß die Ordnung nicht willkürlich gewählt werden kann. Die lokal für jedes Token spezifizierte Ordnung muß unter der globalen topologischen Bedingungen des Netzwerks fortsetzbarn sein.

Die Existenz einer solchen Ordnung kann nur ein Algorithmus sicherstellen, der in den Vererbungsmechanismus einbezogen und integriert ist. Diese automatische Integration steht für die in **DONALD-III** verwendete Implementation noch aus. Bei den Hierarchien, die während der Konstruktion des Expertensystems **INVEST** benötigt wurden, konnte die Existenz einer Ordung mit den geforderten Eigenschaften jedoch durch entsprechende Kodierung von Hand hergestellt, und damit die Korrektheit der Vererbung sichergestellt werden.

Kapitel 7

Komplexitätsbetrachtung

In diesem Kapitel werden einerseits die neuesten bekannten Resultate zur Komplexität der Familie der direkten pfadbasierten Vererbungsmethoden vorgestellt und andererseits im Vergleich dazu die Komplexität der bei der Implementation von **DONALD-III** verwendeten Verfahren bestimmt. Außerdem wird über die experimentellen Erfahrungen mit einer Implementation der von Touretzky in TMOIS vorgeschlagenen Vererbungsmethode in der Programmiersprache Prolog berichtet, die im Rahmen dieser Arbeit vorgenommen wurde.

Die folgenden Abschnitte geben eine kurze Einführung in die benutzten Begriffe der Komplexitätstheorie von Algorithmen und Problemen.

7.1 Probleme und Algorithmen

Ein *Problem* ist eine allgemeine Fragestellung, die normalerweise mehrere Parameter oder freie Variablen enthält, deren Werte nicht spezifiziert sind.

Ein Problem wird definiert durch:

- Eine allgemeine Beschreibung aller Parameter.
- Eine Aussage darüber, welche Eigenschaften eine Antwort (oder *Lösung*) erfüllen muß.

Eine *Instanz* eines Problems ergibt sich durch das Festlegen spezieller Werte für alle Problemparameter.

Ein *Algorithmus* ist eine allgemeine, schrittweise Prozedur oder Rechenvorschrift zur Lösung von Problemen. Ein Algorithmus *löst* ein Problem Π, falls dieser Algorithmus auf jede Instanz I von Π angewandt werden kann und in jedem Fall eine Lösung für I produziert.

Es ist nun von Interesse, den *effizientesten* Algorithmus zur Lösung eines Problems zu finden. Läuft ein Algorithmus auf einem Computer ab, so hängt der Begriff der Effizienz vom Verbrauch an allen zur Verfügung stehenden Betriebsmitteln ab. Das wichtigste Betriebsmittel ist dabei die Rechenzeit, da dieser Faktor in den meisten Fällen darüber entscheidet, ob ein Algorithmus auch in der Praxis einzusetzen ist.

Der Zeitbedarf eines Algorithmus wird am prägnantesten in Termen einer einzigen Variablen dargestellt, und zwar der *Größe* des Problems. Die Größe eines Problems ist dabei ein Maß für die Menge an Eingabedaten, die zur Festlegung einer Probleminstanz erforderlich sind.

Um den Zeitbedarf eines Algorithmus mathematisch exakt zu analysieren, ist es erforderlich, die Eingabe für den Computer in einer endlichen Folge von Symbolen eines festen endlichen Alphabets zu kodieren.

Die *Eingabelänge* einer Instanz I eines Problems Π ist definiert als die Anzahl der Symbole in der Problembeschreibung von I, die sich durch Anwendung des Kodierungsschemas für Π ergibt.

Die *Zeitkomplexitätsfunktion* für einen Algorithmus drückt seinen Zeitbedarf als längsten Zeitraum aus, den der Algorithmus zur Lösung einer Probleminstanz benötigt. Dabei wird jede mögliche Eingabelänge betrachtet. Diese Funktion ist insofern nicht wohldefiniert, als sowohl das gewählte Kodierungsschema als auch der konkret vorliegende Computer[1] nicht spezifiziert sind.

[1] Genauer spricht man von der Mächtigkeit des gewählten Berechnungsmodells.

Wie sich zeigt, spielen diese beiden Faktoren jedoch eine gar nicht so gravierende Rolle bei der Klassifikation von Algorithmen nach dem Grad ihrer Zeitkomplexität.

Betrachtet man unterschiedliche Zeitkomplexitätsfunktionen, hängt die Einteilung in die beiden Klassen *effizient genug* und *zu ineffizient* von der ganz konkreten Situation ab. Es gibt jedoch ein einfaches Kriterium, das die Abstraktion von dem konkreten Problem erlaubt. Man spricht von dem Unterschied zwischen polynomialem und exponentiellem Zeitverhalten.

Definition 7.1 *Für $n \in I\!N_0^+$ und $f : I\!N_0^+ \rightarrow I\!R$ sagt man „f ist von der Ordnung g“, falls es eine Konstante $c > 0$ gibt, so daß*

$$|f(n)| \leq c|g(n)| \quad \textit{für alle} \quad n \geq 0.$$

Mit dieser Definition enthält die Klasse der Algorithmen mit *polynomialem* Zeitbedarf alle Algorithmen, deren Zeitkomplexität von der Ordnung p ist, wobei $p(n)$ ein beliebiges Polynom in der Eingabelänge n ist. Alle anderen Algorithmen, für die keine solche Zeitschranke angegeben werden kann, werden als *exponentielle* Algorithmen bezeichnet. In diesem Sinn ist auch z.B. die Funktion $n^{\log n}$ exponentiell und man sollte so eigentlich besser von nichtpolynomial sprechen. Beispiele für den Zeitbedarf bei unterschiedlichem n und für einige polynomiale und exponentielle Abhängigkeiten finden sich z.B in [Garey and Johnson, 1979].

Das viel stärkere Wachstumsverhalten der exponentiellen Funktionen legt die Einteilung in „gute“ und „schwer handhabbare“ Algorithmen nahe. Im folgenden werden diese nicht gutartigen Algorithmen, wie im englischen Sprachraum üblich, als *intractable* bezeichnet. Exponentielle Algorithmen sind oft nur Variationen eines vollständigen, systematischen Probierverfahrens *(Exhaustive Search)*. Polynomiale Algorithmen erfordern dahingegen meistens tiefere Einsichten in die Struktur des Problems. Probleme ohne polynomiale Lösungsalgorithmen sind also noch nicht als endgültig gelöst zu betrachten.

Nach obiger Definition von *intractable* kann man durch Angabe eines polynomialen Algorithmus zeigen, daß ein Algorithmus *nicht* intractable ist. Die

Klasse von Algorithmen, für die noch keine polynomialen Algorithmen bekannt sind, wurde zuerst von Alan Turing noch weiter betrachtet.

Turing zeigte, daß es Probleme gibt, die *so* schwierig sind, daß sie sogar unentscheidbar sind. Sein klassischer Beweis ist die Unmöglichkeit der Spezifikation eines Algorithmus, der zu einem beliebigen, gegebenem Computerprogramm und zugehöriger beliebiger Eingabe entscheidet, ob das Programm mit dieser Eingabe irgendwann anhält oder endlos weiter läuft.

Für eine Reihe weiterer Probleme wurde die Unentscheidbarkeit gezeigt. Als Beispiel soll Hilberts zehntes Problem genügen. Diese Probleme bezeichnet man als „intractable im strengen Sinn“.

7.2 Entscheidungsprobleme

Zur Vereinfachung der Komplexitätsbetrachtung werden zwei zusätzliche Einschränkungen gemacht. Erstens genügt es, Probleme zu betrachten, bei denen die Länge des Lösungsausdrucks polynomial von der Eingabelänge abhängt. Probleme, die diese Bedingung nicht erfüllen, müssen als schlecht gestellt verworfen werden, da die Lösungen eventuell mehr Informationen enthalten, als jemals weiter verarbeitet werden können. Zweitens reicht es aus, nur *Entscheidungsprobleme* zu betrachten. Ein Entscheidungsproblem Π besteht aus einer Menge D_Π von Instanzen und einer Teilmenge $J_\Pi \subseteq D_\Pi$ von *Ja*-Instanzen. Diese Einschränkung erfordert im allgemeinen nur eine Umformulierung des Problems.

Sei z.B. ein Graph G mit einer Bewertungsfunktion $v(p)$ für alle Pfade p durch G gegeben. Statt nun nach einem unter v minimalen Pfad p zu suchen, stellt man die Frage, ob es zu gegebenem $B \in I\!R$ einen Pfad p gibt, so daß $v(p) \leq B$.

Die Möglichkeit, einen Algorithmus auf unterschiedlich mächtigen realen Computermodellen ablaufen zu lassen, kann bei der Entscheidung über die Gutartigkeit ignoriert werden, da sich alle diese Modelle in polynomialer Zeit gegenseitig simulieren lassen und polynomiale Komplexität auch nach polynomialer Simulation noch polynomial bleibt.

7.3 Die Klasse P von Problemen

Statt mit einem realen Computermodell beschäftigt man sich in der Komplexitättheorie mit einem abstrakten Berechnungsmodell, das durch die sogenannte Deterministische Turing Maschine (DTM) charakterisisiert wird.

Eine DTM besteht aus einem zweiseitig unendlichem Band von Feldern, die mit den ganzen Zahlen beschriftet sind, und einem Schreib- und Lesekopf, der Symbole von den Feldern des Bandes lesen und neue Symbole in die Felder plazieren kann. Kontrolliert wird das ganze von einer Steuereinheit, die aufgrund des momentanen Zustands des Bandes und des internen Zustands der Steuereinheit die nächste durchzuführende Aktion bestimmt.

Ein *Programm* für eine DTM besteht also aus drei Teilen:

(i) Eine endliche Menge Γ von Bandsymbolen, mit einer Teilmenge von Σ Eingabesymbolen und einem Leersymbol $b \in \Gamma \setminus \Sigma$.

(ii) Eine endliche Menge Q von Zuständen mit einem Startzustand q_0 und zwei Endzuständen q_J und q_N.

(iii) Eine Übergangsfunktion $\delta : (Q \setminus \{q_J, q_N\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

Die Maschine wird gestartet, indem die Eingabe als Zeichenkette über den obigen Symbolen kodiert und dann auf das Band geschrieben wird. Alle anderen Felder des Bandes erhalten das Symbol b. Die Steuereinheit wird in Zustand q_0 versetzt, der Schreib- und Lesekopf auf Feld 1 positioniert. Die Berechnung läuft jetzt schrittweise ab. Wenn der momentane Zustand q einer der Endzustände ist, so terminiert die Berechnung mit der Antwort „Ja“ (q_J) bzw. „Nein“ (q_N). Anderenfalls wird das Feld unter dem Lesekopf gelesen und die Übergangsfunktion δ bestimmt aus dieser Information einen neuen Zustand q', ein Symbol, das auf die momentane Position des Bandes zu schreiben ist und die Richtung, in der die Maschine den Kopf dann um eine Position verschieben soll.

Ein solches Programm für eine DTM kann als Spezifikation eines Algorithmus verstanden werden. Die Entsprechung zwischen den obigen Entscheidungsproblemen und einem Programm sollte klar sein: Das Programm M löst ein auf dem Band in Schema e kodiertes Entscheidungsproblem Π, wenn es für alle Eingaben anhält. Alternativ sagt man auch: Es erkennt alle Sätze der Sprache $L_M = L[\Pi, e]$.

Die Zeitkomplexität eines Algorithmus kann jetzt formal definiert werden als die Anzahl der Schritte, die das DTM Programm M bei der Eingabe x benötigt, bis es in einen Haltezustand gerät. Der Zeitbedarf $T_M(n)$ bezeichnet dann die maximale Anzahl von Schritten, die für ein Problem x mit Länge n benötigt wird. Ein Programm M heißt ein *polynomiales* DTM Programm, falls es ein Polynom p gibt, so daß $T_M(n) \leq p(n)$ für alle $n > 0$. Mit diesen Hilfsmitteln kann jetzt die Klasse **P** von Sprachen oder Algorithmen definiert werden, die in polynomialer Zeit lösbar sind:

$$\mathbf{P} := \{L : \text{ Es gibt ein polynomiales DTM Programm mit } L = L_M\}$$

Es gibt jedoch eine theoretische Maschine, die durch ihr nichtdeterministisches Verhalten grundlegend umfassender und mächtiger als eine DTM ist.

7.4 Die Klasse NP von Problemen

Die deterministische Turing Maschine DTM kann durch Einführung von Nichtdeterminismus verallgemeinert werden. Eine solche NDTM ist etwas komplizierter aufgebaut. Eine Nichtdeterministische TM hat dieselbe Struktur wie eine DTM, nur ist sie zusätzlich mit einem *Ratemodul* ausgestattet. Dieses Ratemodul hat einen weiteren Schreibkopf, der die Symbole auf dem Band ändern kann.

Ein NDTM Programm wird exakt identisch zu einem DTM Programm spezifiziert. Der einzige Unterschied besteht in der Ausführung eines solchen Programms, die in zwei Phasen abläuft.

- Die erste Phase ist die *Ratephase*. Der Eingabestring x der Länge n wird mit Symbolen auf den Feldern 1 bis n kodiert. Der deterministische Schreib- und Lesekopf der Maschine wird auf Feld 1 positioniert.

Der Schreibkopf des Ratemoduls steht auf Feld -1 und die Steuereinheit ist inaktiv. Das Ratemodul bewegt jetzt seinen Schreibkopf in negativer Richtung über das Band, wobei eine endliche Folge von Symbolen geschrieben wird. Dann hält das Ratemodul an, wird inaktiv und übergibt die Kontrolle an die Steuereinheit in Zustand q_0. Das Ratemodul schreibt eine vollkommen willkürliche Folge von Symbolen auf das Band.

- Die zweite Phase läuft ohne weitere Mitwirkung exakt so ab, wie in einer deterministischen TM. Die Berechnung endet, wenn die Steuereinheit einen der Haltezustände erreicht. Wenn sie im Zustand q_J anhält, spricht man auch von einer *akzeptierenden* Berechnung. Alle anderen Berechnungen, ob sie überhaupt einen Haltzustand erreichen oder nicht, werden als *nichtakzeptierend* bezeichnet.

Jedes NDTM Programm M liefert also zu einer festen Eingabe x eine unendliche Zahl von möglichen Berechnungen, die sich in der zufälligen Wahl des Ratemoduls unterscheiden. Eine NDTM *akzeptiert* nun die Eingabe x, wenn *mindestens eine* dieser Berechnungen akzeptiert wird.

Die Zeit, die ein NDTM Programm M benötigt, um die Eingabe x zu akzeptieren wird als Minimum der Summe der Zeiten der beiden Phasen über alle akzeptierenden Berechnungen definiert.

Ein Programm M heißt dann ein *polynomiales* NDTM Programm, falls es ein Polynom p gibt, so daß $T_M(n) \leq p(n)$ für alle $n > 0$. Mit diesen Hilfsmitteln kann jetzt die Klasse **NP** von Sprachen oder Algorithmen definiert werden, die in nichtdeterministisch polynomialer Zeit lösbar sind:

$$\mathbf{NP} := \{L : \text{ Es gibt ein polynomiales NDTM Programm mit } L = L_M\}$$

Die größere Mächtigkeit einer NDTM verglichen mit einer DTM wird offensichtlich, wenn man sich die Minimierung der Zeit über alle akzeptierenden Berechnungen vor Augen führt. Durch diese Minimierung kann man davon ausgehen, daß das Ratemodul nicht nur vollkommen willkürliche Symbolfolgen auf das Band schreibt, sondern insbesondere auch eine Kodierung der Lösung des Problems. Da sich zu Beginn der zweiten Phase einer Berechnung die Lösung bereits kodiert auf dem negative Teil des Bandes befinden kann, ist die Aufgabe des deterministischen Teils natürlich einfacher. Statt der Konstruktion einer Lösung reduziert sich die Aufgabe auf die Überprüfung der Korrektheit einer *richtig geratenen* Lösung.

7.5 Vergleich der Klassen P und NP

Die erste fundamentale Aussage ist $\mathbf{NP} \supseteq \mathbf{P}$. Zum Beweis konstruiert man zu einem gegebenen deterministischen TM Programm ein neues nichtdeterministisches Programm. Erst einmal kann das DTM Programm direkt als zweite Phase des NDTM Programms verwendet werden. Modifiziert man das Programm noch so, daß alle geratenen Symbole auf der negativen Bandseite ignoriert werden, so erhält man zu einem DTM Programm ein äquivalentes NDTM Programm.

Wie oben bereits diskutiert, erscheint eine NDTM intuitiv mächtiger als eine DTM. Diese Tatsache legt die Vermutung nahe, daß es sich bei der Beziehung zwischen **P** und **NP** um eine *echte* Teilmengenrelation handelt. Es konnte trotz aller Anstrengungen *keine* allgemeine Methode gefunden werde, aus einem gegebenen nichtdeterministisch polynomialen Algorithmus einen äquivalenten deterministisch polynomialen Algorithmus zu konstruieren. Ein Beweis dieser Tatsache steht jedoch noch aus und ist aufgrund der langen vergeblichen Bemühungen sehr unwahrscheinlich.

Es gibt allerdings ein wichtiges Resultat zur Simulation einer nichtdeterministischen Maschine durch eine deterministische:

Theorem 7.1 *Für jedes Problem $\Pi \in \mathbf{P}$ gibt es ein Polynom p, so daß Π durch einen deterministischen Algorithmus mit Zeitkomplexität $O(2^{p(n)})$ gelöst werden kann.*

Beweis: Betrachte einen nichtdeterministisch polynomialen Algorithmus A zur Lösung von Π mit Zeitkomplexität $q(n)$, wobei q ein Polynom in n ist. Für jede akzeptierende Berechnung gibt es nun eine geratene Symbolfolge mit Länge kleiner als $q(n)$, so daß die zweite Phase des NDTM Programms M

nach weniger als $q(n)$ Schritten die Berechnung akzeptiert. Die Anzahl der erforderlichen Rateversuche ist also beschränkt durch $k^{q(n)}$, wobei k die Anzahl der möglichen Symbole des Bandalphabets ist. Die simulierende DTM M' benutzt jetzt für alle Rateversuche die deterministische zweite Phase von M, um zu überprüfen, ob A die Berechnung akzeptiert hätte. Die Zeitkomplexität der zweiten Phase von M' ist $q(n)$ und die Anzahl der Versuche ist $k^{q(n)}$. Zusammen ergibt sich eine Zeitkomplexität von $q(n) \cdot k^{q(n)}$, die sich durch $O(2^{p(n)})$ mit einem geeignet gewählten Polynom p abschätzen läßt. ∎

Nichtdeterministisch polynomiale Algorithmen lassen sich also in exponentieller Zeitkomplexität durch deterministische Algorithmen simulieren. Deshalb ist die Behauptung gerechtfertigt, daß ein Problem noch nicht vollständig gelöst ist, wenn nur Algorithmen mit exponentieller Komplexität bekannt sind. Diese Algorithmen entsprechen im allgemeinen nur der Aufzählung aller denkbaren Lösungen, die dann mit einem polynomialen Algorithmus auf Korrektheit geprüft werden. Man spricht auch von dem *Generate and Test* Verfahren.

Da der Beweis von $\mathbf{P} \neq \mathbf{NP}$ bis jetzt nicht gelang, konzentriert man sich für ein betrachtetes Problem Π auf Resultate der Form „Falls $\mathbf{P} \neq \mathbf{NP}$, dann ist $\Pi \in \mathbf{NP} \setminus \mathbf{P}$." Man betrachtet zwei Probleme als polynomial äquivalent, wenn sie sich wechselseitig durch eine Transformation mit polynomialer Zeitkomplexität in einander überführen lassen. Ein Problem Π heißt dann **NP**-*vollständig*, wenn $\Pi \in \mathbf{NP}$ und sich alle anderen Probleme aus **NP** polynomial in Π transformieren lassen. Die **NP**-*vollständigen* Probleme sind in diesem Sinne die *schwersten* Probleme.

Ein **NP**-vollständiges Problem Π hat also die Eigenschaft, daß die Existenz einer effizienten Lösung für Π die Existenz einer solchen für alle anderen Probleme in **NP** impliziert.

Ein Entscheidungsproblem Π, ob es nun nichtdeterministisch polynomial ist oder auch nicht, zu dem man ein **NP**-vollständiges Problem polynomial transformieren kann, hat die Eigenschaft, daß es nicht in polynomialer Zeit gelöst werden kann, solange nicht $\mathbf{P} = \mathbf{NP}$. Ein solches Problem wird auch als **NP**-*schwer* bezeichnet. Zum Beweis dieser Eigenschaft genügt also die Angabe einer polynomialen Reduktion ausgehend von einem bekannten **NP**-vollständigen Problem. Diese Tatsache wird in dem Beweis zu Theorem 7.2 ausgenutzt.

In der weiteren Betrachtung wird stillschweigend angenommen, daß $\mathbf{P} \neq \mathbf{NP}$ gilt, um die Diskussion über Algorithmen zu vereinfachen. So kann vermieden werden, Sätze immer wieder mit der Bedingung „Unter der Annahme, daß $\mathbf{P} \neq \mathbf{NP}$ gilt, ...“ zu beginnen.

7.6 Die Komplexität der direkten Vererbung

Grundlage der direkten pfadbasierten Methoden zur Analyse von Wissensrepräsentation ist das Vererbungsproblem aus Definition 6.28.

Für die in Touretzkys Arbeit [Touretzky, 1986] definierte Form der Vererbung in Semantischen Netzwerken gibt es *keinen* polynomialen Algorithmus, der als Eingabe ein azyklisches Netzwerk Γ nimmt und daraus eine Konklusionsmenge von Γ berechnet.

Der Beweis dieser Tatsache beruht auf der polynomialen Transformation eines bekannten **NP**-vollständigen Problems in ein Vererbungsproblem. Als **NP**-vollständiges Problem dient dabei ein Entscheidungsproblem mit Namen *Pfad mit verbotenen Paaren* (PMVP) aus der Graphentheorie. Dieses von Gabow, Maheshwari und Osterweil formulierte Problem, ist in [Gabow *et al.*, 1976] beschrieben und wird dort zum Zweck der automatischen Erzeugung von Testpfaden durch Programme benutzt.

Definition 7.2 (Pfade mit verbotenen Paaren) *Sei $G = \langle V, E\rangle$ ein gerichteter Graph, mit zwei Knoten $s, t \in V$. Gegeben sei außerdem eine Menge $C = \{(a_1, b_1), \ldots, (a_n, b_n)\}$ von Paaren von Knoten aus V. Das Problem PMVP ist nun die Beantwortung der folgenden Frage: Gibt es einen Pfad von s nach t, der höchstens einen Punkt aus jedem Paar in C enthält?*

Wie in [Gabow *et al.*, 1976] gezeigt wird, ist das obige Entscheidungsproblem **NP**-vollständig, selbst wenn man nur den eingeschränkten Fall eines azyklischen Graphen betrachtet und zusätzlich verlangt, daß alle Paare aus C disjunkt sind. Ohne Beschränkung der Allgemeinheit kann man auch annehmen, daß es für jedes verbotene Paar (a_i, b_i) keinen Pfad von b_i nach a_i in G gibt.

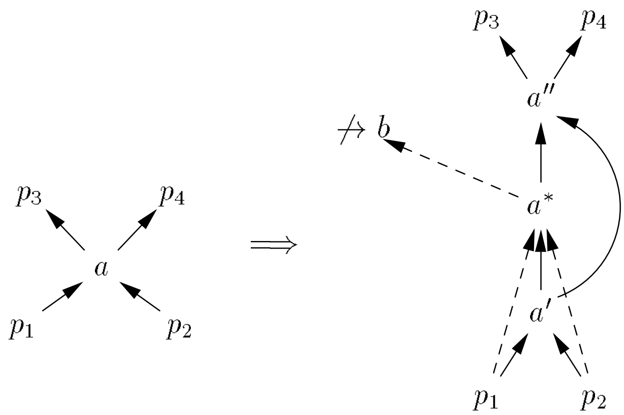


Abbildung 7.1: Konstruktion zu dem Beweis von Theorem 7.2

Wäre das nicht der Fall, so könnte man gegebenenfalls die Rollen von a_i und b_i vertauschen.

Nach diesen Vorbereitungen kann das folgende Theorem bewiesen werden, wobei der in Abbildung 7.1 dargestellten Konstruktion entscheidende Bedeutung zukommt.

Theorem 7.2 (Selman & Levesque, 1990) *Das in der Definition 6.28 betrachtete Vererbungsproblem für on-path, downward, credulous Inheritance ist NP-schwer.*

Beweis: Zu der oben betrachteten Instanz des Problems PMVP wird im ersten Schritt ein azyklisches Netzwerk Γ konstruiert. Jede Kante von G wird zu einer positiven Kante von Γ. Für jeden Knoten a, der als erstes Element eines verboteten Pfades in C auftritt, wird a durch die in Abbildung 7.1 dargestelle Konstuktion ersetzt. Die Nachbarn von a sind durch die vier Knoten p_1 bis p_4 dargestellt. Im Verlauf der Konstruktion wird a in die drei neuen Knoten a', a^* und a'' übersetzt. Zusätzlich werden vier neue Kanten definiert und für jede nach a führende Kante des ursprünglichen G eine neue negative Kante nach a^*. Da G azyklisch ist und für die verbotenen Pfade keine Pfade von b_i nach a_i führen, ist auch das neu entstandene Netzwerk wieder azyklisch. Es gibt nun genau dann einen Pfad von s nach t in G, der höchstens ein Element von jedem verbotenen Paar enthält, wenn es Konklusion der Form $s \to t$ in einer *on-path downward credulous* Erweiterung von Γ gibt. Den Bemerkungen auf S. 222 zufolge ist das Problem der Berechnung einer Erweiterung also **NP**-schwer, da es gelungen ist, ein **NP**-vollständiges Problem polynomial[2] in einen azyklischen Graphen zu transformieren. ■

Die direkte Konsequenz von Theorem 7.2 ist, daß die von Touretzky in [Touretzky, 1986, S. 101ff.] vorgeschlagene Konditionierung eines Netzwerks nicht effizient durchgeführt werden kann. Diese Konditionierung soll den Einsatz von *Shortest Path* Algorithmen erlauben, die im allgemeinen ja keine korrekte Implementation der Vererbung zulassen. Auf kürzesten Wegen beruhende Algorithmen können auf einer Parallel Marker Propagation Machine[3] (PMPM) in $O(d)$ Zeiteinheiten durchgeführt werden. d ist dabei ein Maß für die Tiefe des Netzwerks und kann durch $O(\log n)$ abgeschätzt werden, wobei n die Zahl der Knoten des Netzwerks ist. Diese Abschätzung gilt natürlich nur so lange, wie das Netzwerk nicht entartet[4] ist.

Selman und Levesque konnten weiterhin zeigen, daß das in Theorem 7.2 benutzte Netzwerk nicht mehrdeutig ist. Ebenso hängt die in dem Beweis verwendete Konstruktion nicht entscheidend von der Tatsache ab, daß es sich um die on-path credulous Methode zur Vererbung handelt. Vielmehr ist die für die Komplexität wesentliche Eigenschaft die *Richtung*, in der Pfade aus vorhandenen Pfaden konstruiert werden. Den Unterschieden zwischen Definition 6.17 und Definition 6.18 auf Seite 198 kommt also bezüglich der Aneinanderkettung von Pfaden große Bedeutung zu. Diese schlägt sich in der Tatsache nieder, daß für die Aufwärtskonstruktion ein **NP**-schweres Problem entsteht, während für die Abwärtskonstruktion polynomiale Algorithmen vorliegen.

Um ein intuitives Verständis für diesen grundsätzlichen Effizienzunterschied zu gewinnen, betrachtet man die Vorgehensweise der polynomialen *upward* Algorithmen. Diese konstruieren iterativ eine Konklusionsmenge. Addiert man zu den bereits abgeleiteten Pfaden einen weiteren, so ist die entscheidende

[2] Es werden zu jedem Knoten a mit n eintreffenden Kanten genau zwei neue Knoten definiert und $n+4$ neue Kanten.

[3] Vergleiche die ausführliche Diskussion von NETL in Abschnitt 3.4 auf Seite 94 und die Betrachtung in Abschnitt 6.1 auf Seite 187.

[4] Wenn das Netzwerk z.B. aus einer langen linearen Kette von Knoten besteht.

Aufgabe, festzustellen, ob der neue Pfad durch die bereits vorhandenen verhindert *(preempted)* wird. Um zu entscheiden, ob ein Pfad verhindert wird, müssen alle intermediären Token betrachtet werden. Im ungünstigsten Fall ist dazu der Rückgriff auf *alle* Pfade der zugrundeliegenden grounded Extension nötig. Da eine solche Erweiterung eine exponentielle Größe haben kann, ergibt sich ein potentielles Effizienzproblem. Betrachtet man speziell die upward Methode, so ist nur erforderlich, die Pfade in der *momentan* iterativ konstruierten Erweiterung zu betrachten. Diese lassen sich aber alle in polynomialer Zeit wieder aus bereits konstruierten Erweiterungsstufen ableiten. Damit ist die Suche nach intermediären Token für die Aufwärtskonstruktion von Pfaden effizient durchführbar.

In [Horty *et al.*, 1987b, S. 30ff.] ist eine PMPM Algorithmus beschrieben, der eine *off-path skeptische upward* Erweiterung eines Netzwerks Γ berechnet. Genauer gesagt wird die Komplexität einer Abfrage der folgenden Form betrachtet: „Wird eine gegebene Konklusion $x \to y$ oder $x \not\to y$ von einem Pfad aus einer Erweiterung unterstützt?“ Dieses Problem wird mit query(x, y) bezeichnet und erscheint auf den ersten Blick einfacher als die Berechnung einer Konklusionsmenge. Die Konklusionsmenge eines Netzwerks mit n Token[5] enthält maximal $n(n-1)$ Elemente und man kann so durch $n(n-1)$ Anfragen eine Konklusionsmenge bestimmen. Das ist nur ein polynomialer Unterschied, der in dieser Betrachtung ignoriert werden kann.

Horty gibt für die Laufzeit seines Algorithmus zur Beantwortung einer solchen Anfrage query(x, y) folgende Schranke an:

$$O(\max(d(x), D(x, y) + [C(x, y) \cdot d(x, y)]))$$

Dabei ist $d(x)$ das Maximum der Längen aller kürzesten Wege mit dem Token x als Anfangspunkt und einem anderen Token z des Netzwerks als Endpunkt. $C(x, y)$ ist die Anzahl der zueinander in Konflikt stehenden Pfade. In der ersten Phase des Algorithmus wird das Netzwerk auf eine kleinere Teilmenge $\Gamma^{x,y}$ von Γ reduziert, die für das Ergebnis der Abfrage relevant ist. $D(x, y)$ ist dann die Entsprechung von $d(x)$ in $\Gamma^{x,y}$. In realistischen Beispielen kommt dem Faktor $d(x)$ kein signifikanter Einfluß zu und auch die Zahl der widersprochenen Pfade ist relativ gering. In diesem Fall nähert sich die Laufzeit des Algorithmus der unteren parallel möglichen Schranke $(D(x, y))$.

[5] Die Token eines Netzwerks entsprechen den Knoten eines Graphen (siehe Definition 6.6).

Simuliert man die parallel arbeitende PMP Maschine auf einem einzelnen Prozessor, so muß diese Zeit noch mit der Anzahl n der Token multipliziert werden. Für weitere Details wird auf [Horty *et al.*, 1987a] und [Horty *et al.*, 1987b] verwiesen. Selman und Levesque konnten zusätzlich zeigen, daß auch für alle weiteren Vererbungsstrategien, die der upward Methode folgen, polynomiale Algorithmen möglich sind, und zwar unabhängig von der Wahl on-path / off-path und skeptisch / credulous.

7.7 Experimentelle Erfahrungen

Im Rahmen der Untersuchungen von Netzwerken wurde auch eine Implementation des von Touretzky vorgeschlagenen Vererbungssystems in der Programmiersprache Prolog erstellt. Diese Implementation sollte ursprünglich zur experimentellen Generierung von Netzwerken und ihren Erweiterungen eingesetzt werden. Insbesondere war geplant, die Folgen der in allen theoretischen Arbeiten vorgenommen Einschränkung auf azyklische Netzwerke kennenzulernen. Es zeigte sich aber, daß dieses experimentelle Vorgehen nicht realisierbar war.

Erfahrungen mit einem kommerziellen Prolog System (BIM-Prolog), das ansonsten als effiziente und stabile Implementation bekannt ist, zeigten, daß selbst auf einer mit 32 Megabyte realem Hauptspeicher (64 Megabyte virtuell) ausgestatteten Maschine experimentelle Untersuchungen für Netzwerke mit mehr als acht Token nicht mehr sinnvoll waren. Prolog Systeme arbeiten typischerweise mit einer Reihe von Stacks, die offensichtlich in irgendeiner Weise überliefen und letztendlich zum Zusammenbruch des Systems führten. Auch die mögliche Parametrisierung der Stackgrößen brachte keine Verbesserung dieses Verhaltens.

Im Licht der obigen Komplexitätresultate ist dieses Fehlschlagen nicht weiter verwunderlich, da Touretzkys Vererbungssystem **NP**-schwer ist, und damit exponentielle Laufzeiten zu erwarten sind.

7.8 Vererbung in DONALD-III

Das in **DONALD-III** verwendete Vererbungsschema muß effizient implementierbar sein, damit es seiner Rolle als *Information Retrieval* System einer Expertensystem-Shell gerecht werden kann.

Im letzten Kapitel wurde bereits die Einordnung von **DONALD** als Mitglied einer Klasse von Systemen mit speziellen Eigenschaften durchgeführt. Die Hierarchie von Frames konnte in ein Vererbungsnetzwerk vom Typ einer *Tangled Hierarchy* mit Präferenzrelation transformiert werden. In solchen Systemen ist es möglich, die Vererbungsrelation in Form einer transitiven Hülle zu berechnen.

Der entscheidende Teil der Vererbung in **DONALD-III** ist in Abbildung 7.2 dargestellt. Es gibt noch eine große Zahl weiterer Funktionen, die in konstanter Zeit ablaufen und hier nicht weiter betrachtet werden müssen.

Zur Abschätzung der Komplexität der Erzeugung einer Instanz I wird von folgender Situation ausgegangen:

(i) Die betrachtete Hierarchie Γ von Frames habe die Tiefe d.

(ii) Die direkt spezifizierten Oberklassen von I seien $K(I)$.

(iii) Die Anzahl der Slots einer Klasse oder Teilklasse $C \in \Gamma$ sei durch die Funktion $s(C)$ gegeben. Es gebe eine Schranke L für die maximale Anzahl von Slots einer Klasse.

(iv) Die Vererbungsinformation aller Vorgänger von I wurde bereits berechnet. Die Menge der Vorgänger von I werde mit $V(I)$ bezeichnet.

Bei der Erzeugung der Instanz und der Berechnung der vererbten Information wird nun die Funktion `create-frame` einmal aufgerufen. Diese ruft im wesentlichen die Funktion `parse-views` aus Abbildung 7.2 auf, deren Laufzeit zu bestimmen ist. Argumente für `parse-views` bildet die Liste der spezifizierten Vorgänger in Präferenzreihenfolge. Den zeitaufwendigen Teil von `parse-views` bilden die Bindung der Variablen `own-views` und `all-views`.

Die Schleife zur Initialisierung von `own-views` wird genau $|K(I)|$ mal durchlaufen und enthält jeweils einen Aufruf von `parse-view` für jedes Element

```
(defun parse-views (view-source-list frame)
  (let* ((own-views
          (repeat with view-names = nil
                  for view-source in view-source-list
                  do (assert (not (memq (second view-source) view-names))
                             () "View ~S yet encountered"
                             (second view-source))
                  do (push (second view-source) view-names)
                  collect (parse-view view-source
                                      (fetch-view-no-error frame
                                         (second view-source)))))
         (all-views (collect-from-own own-views))
         (frame-name (frame-name frame)))
    (dolist (view all-views)
            (setf (view-upward view) frame)
            (do-all-slots (slot view)
                          (setf (slot-upward slot) view)))
    (etypecase frame
               (instance
                (dolist (view all-views)
                        (pushnew frame-name
                                 (class-instances (view-definition view))))
                (setf (instance-views frame) all-views))
               (subclass
                (dolist (view all-views)
                        (pushnew frame-name
                                 (class-subclasses
                                  (view-definition view)))))
                (setf (subclass-views frame) all-views))))


(defun view-alpha-sort (view1 view2)
  (string< (symbol-name (view-class view1))
           (symbol-name (view-class view2))))

(defun collect-from-own (own-views)
  (let ((inherited-views
         (repeat for recent-view in own-views
                 for subclass = (view-definition recent-view)
                 when (and (typep subclass 'subclass)
                           (subclass-views subclass))
                 append (and (typep subclass 'subclass)
                             (subclass-views subclass)))))
    (repeat with all = own-views
            for inherited in inherited-views
            do (repeat for own in all
                       when (eq (view-class own) (view-class inherited))
                       do (view-merge own inherited)
                       and return t
                       finally (push (copy-view inherited) all))
            finally (return (sort all 'view-alpha-sort)))))
```

Abbildung 7.2: Der wesentliche Teil der Vererbung in **DONALD-III**

$k \in K(I)$. `parse-view` ist nicht mehr in der Abbildung enthalten, benötigt aber $O(s(k))$ Operationen, da sie alle Slots von k betrachten muß. Da die maximale Anzahl von Eigenschaften einer beliebigen Klasse durch die Konstante L beschränkt ist, kann der Aufwand für `own-views` durch $c_1 \cdot L \cdot |K(I)|$ abgeschätzt werden.

Die Variable `own-views` wird als Resultat der Funktion `collect-from-own` initialisiert. Die Funktion `collect-from-own` enthält wiederum eine Schleife zur Initialisierung von `inherited-views`. Alle in der Schleife aufgerufenen Funktionen haben konstante Laufzeit, deshalb liefert die Schleife ebenso einen durch $c_2 \cdot |K(I)|$) bestimmten Anteil zur Laufzeit. Die von ihr erzeugte Liste hat eine Länge $l \leq |V(I)|$. Die Hauptschleife der Funktion `collect-from-own` ist eine Doppelschleife über `own-views` und `inherited-views`. Innerhalb der Schleife wird die Funktion `view-merge` aufgerufen, deren Laufzeit wieder nur von der obigen Konstante L abhängt. Insgesamt ergibt sich für `collect-from-own` eine durch $c_2 \cdot |K(I)| + c_3 \cdot L \cdot l \cdot |K(I)| \leq c_2 \cdot |K(I)| + c_3 \cdot L \cdot |V(I)| \cdot |K(I)|$ beschränkte Laufzeit.

Da nach Definition $|K(I)| \leq |V(I)|$ läßt sich die Gesamtlaufzeit mit geeignet gewählten Konstanten weiter nach oben abschätzen:

$$c_1 \cdot L \cdot |K(I)| + c_2 \cdot |V(I)| + c_3 \cdot L \cdot |V(I)| \cdot |V(I)| \leq c \cdot L \cdot |V(I)|^2$$

Insgesamt ergibt sich für die Erzeugung einer Instanz eine quadratisch von der Zahl $n := |V(I)|$ der Vorgänger abhängende Laufzeit, wenn man L wirklich als Konstante betrachten kann[6]. Diese Effizienz von $O(n^2)$ konnte aber nur erreicht werden, indem die Vererbungsinformation aller Vorgänger bereits zum Zeitpunkt ihrer Erzeugung vollständig berechnet wurde. Unterläßt man diese Optimierung, die natürlich auf der anderen Seite den Nachteil erhöhten Speicherplatzbedarfs hat, so erhöht sich der Laufzeitbedarf um einen weiteren Faktor der Größenordnung n und man erhält dann eine Laufzeit von $O(n^3)$. In **DONALD-III** war der Mehrbedarf an Speicher als Preis für die erzielte höhere Geschwindigkeit leicht zu verschmerzen, da die Implementation sicherstellt, daß komplexe Slotwerte niemals wirklich kopiert werden. Es werden immer nur zusätzliche Zeiger *(Pointer)* verwaltet.

[6] Es ist zu erwarten, daß die Erfüllung dieser Annahme in realistischen Wissensbasen immer garantiert ist.

Die Komplexitätsbetrachtung von **DONALD-III** bildete den Schluß dieses Kapitels. Im letzten Kapitel sollen noch einige zusammenfassende Bemerkungen folgen.

Kapitel 8

Schlußbetrachtung

In diesem letzten Kapitel soll die geleistete Arbeit noch einmal kurz zusammengefaßt werden. Den Abschluß bildet ein Ausblick auf Lösungsansätze für Konzepte, die noch nicht vollständig befriedigend umgesetzt werden konnten. Ebenso wird die Richtung weiterer theoretischer Forschungsarbeiten auf dem Gebiet der Wissensrepräsentation in Semantischen Netzwerken mit direkten pfadbasierten Methoden betrachtet.

8.1 Zusammenfassung

Die Arbeit gliedert sich in einen praktischen und einen theoretischen Teil. Den Ausgangspunkt bildete die praktische Arbeit der Implementation eines Wissensbasierten Systems zur Wertpapieranlageberatung. Aufbauend auf den Ergebnissen von [Heuer *et al.*, 1988] wurde das Wissensbasierte System **INVEST** und die zugrunde liegende Expertensystem-Shell **DONALD** weiter verbessert und vervollständigt.

Die folgenden Eigenschaften heben die aktuelle Version **DONALD-III** von vergleichbaren Systemen ab:

- Die Ausnutzung hierarchischer Strukturen dominiert die Wissensrepräsentation mit Frames. Die Vererbung von Informationen innerhalb dieser heterogen Struktur geschieht unter verschiedenen orthogonalen Aspekten gleichzeitig.
- **DONALD** ist in der Lage, nicht nur die klassischen Diagnoseprobleme zu lösen, die bisher Haupteinsatzgebiet für Wissensbasierte Systeme waren, sondern auch kompliziertere Aufgaben zu bewältigen, bei denen es erforderlich ist, eine Lösung aus primitiveren Bausteinen zu *synthetisieren*.
- Zur Verwaltung von Hypothesen und Vorschlägen für die weitere Vorgehensweise bei der Lösung eines Problems steht ein heuristischer Ansatz zur Verfügung, der auf dynamisch anpaßbarer Abstraktionsstufe den Ablauf des Systems steuert.
- Der Einsatz von Produktionsregeln wurde weitgehend vermieden. Stattdessen kommt ein komplexer gewichteter Mustervergleich zum Einsatz, der auch der Betrachtungsweise von menschlichen Experten nahe zu kommen scheint.
- Falls für spezielle Anwendungen auf Produktionsregeln doch nicht verzichtet werden kann, steht auch ein Interpretierer für Produktionsregeln zur Verfügung, der in das Gesamtkonzept der Frame-basierten Wissensrepräsentation integriert ist.
- Die Konstruktion von Lösungen geschieht auf inkrementelle Weise. Dabei werden parallel unterschiedliche Lösungsansätze verfolgt und immer wieder bewertet.
- Die Verwaltung der Problemdaten erlaubt es, einmal eingegebene Daten und auch getroffene Entscheidungen rückgängig zu machen. Dies wird durch eine nichtmonotone Verwaltung der Hypothesen ermöglicht.

Im Verlauf des **DONALD** Projektes wurde jedoch immer klarer, daß das Vorgehen oft von pragmatischen Gesichtspunkten geprägt worden war. Es waren *ad hoc* Entscheidungen gefällt worden, ohne weitere theoretisch detailliertere Überlegungen anzustellen. Ein solches Vorgehen birgt natürlich die Gefahr von Fehlentscheidungen in sich, war aber auf Grund des experimentellen Charakters des Systems unvermeidlich. Da die Entwicklung der Expertensystem-Shell mehr oder minder gleichzeitig mit der Erweiterung der Anwendung **INVEST** voranschritt, war es oft erforderlich, bestimmte neue Konzepte möglichst schnell in einer Art *Rapid Prototyping* zumindest ansatzweise zu implementieren. Bei diesem Vorgehen war die Verwendung der Programmiersprache LISP von großem Vorteil.

Der theoretische Teil bietet neben der Einordnung des in **DONALD** verwendeten Wissensrepräsentationsformalimus in die Ergebnisse der aktuellen Forschung auf diesem Gebiet und der Erweiterung der ursprünglichen Ideen von Touretzky [Touretzky, 1986] auch Komplexitäts- und Laufzeitabschätzungen für verschiedene Algorithmen zur Vererbung von Eigenschaften innerhalb eines Semantischen Netzwerks.

Insgesamt stellt **DONALD-III** die Integration eines soliden und mathematisch fundierten Wissensrepräsentationsformalimus mit polynomialer Laufzeit und einem heuristischen Ansatz zur Darstellung von nichtmonotonem Schließen in Expertensystemen dar.

8.2 Ausblick

Obwohl die Kombination aus **DONALD** und **INVEST** auch in der Praxis ihre Leistungsfähigkeit unter Beweis stellen konnte, steht ihrem weiteren Einsatz insbesondere die Verwendung von LISP als Implementationssprache entgegen. Diese Sprache steht auf den im Bankbereich typisch verwendeten Rechnern nicht zur Verfügung und es fehlt so die notwendige Einbindung in die klassische EDV-Landschaft solcher Unternehmen. Es wäre also eine Reimplementation auf der Basis einer Programmiersprache wie **C** erforderlich gewesen.

Weiterhin wäre eine Effizienzsteigerung des integrierten Interpretierers für Produktionsregeln wünschenswert. Mit den bisherigen Methoden der Verwaltung von Strukturen und Netzwerken der Wissensbasis ist der direkte Umgang mit dem System recht kompliziert. Hier fehlt ein guter Spezialeditor, der auch mit einem graphischen Benutzerinterface aufwarten kann.

Die theoretischen Untersuchungen bieten mit dem neuen Trend zur Entwicklung direkter pfadbasierter Vererbung in Verbindung mit präferenzgesteuerter Vererbung die Möglichkeit, auch in der Praxis einsetzbare Systeme auf ein mathematisch orientiertes Fundament zu stellen. Während die erste im Rahmen dieser Arbeit implementierte Version einer pfadbasierten Vererbung noch nicht über ausreichende Effizienz verfügte und exponentielle Komplexität hatte, sind die theoretischen Ursachen für die auftretende Ineffizienz inzwischen bekannt und können durch geringfügig modifizierte Definitionen der Vererbung vermieden werden. Die Grundbausteine einer Common LISP Implementation dieses Ansatzes bietet [Horty *et al.*, 1987b]. Die dort geschilderten Ideen konnten auch in LISP umgesetzt werden, aber es fehlt noch eine wirkliche Integration in **DONALD**.

Ein weiterer interessanter Aspekt findet sich in [Thomason and Aronis, 1990]. Ihre Arbeit enthält den Ansatz, ein effizientes Vererbungssystem mit seiner eingeschränkten Ausdrucksfähigkeit[1] durch einen echten Theorembeweiser zu ergänzen und so ein hybrides System zu schaffen.

[1] Zumindest aus Sicht der Prädikatenlogik vermißt man in typischen Vererbungssystemen die Möglichkeit, Konjunktionen und Disjunktionen darzustellen.

Anhang A

Ein Beispieldialog

Dieser Anhang zeigt eine komplette Konsultation von **DONALD-III**, bzw. **INVEST**. Das Beispiel demonstriert das nichtmonotone Schlußfolgern, den schrittweisen Aufbau der Lösung, die Möglichkeit, sich Fragen und Lösungen erläutern zu lassen und anderes mehr. (Text hinter einem Semikolon wurde nachträglich als Erläuterung hinzugefügt.)

```
Wie heißt der Kunde?
> Kunde-1                                ; Text hinter dem ">" sind
                                         ; Eingaben des Benutzers.
        **** Rangliste ****

No. Bestätigt  S  Hypothese              ; S steht für "Status".
 1.  100%      +  Wertpapier             ; "+" ("-") heißt: diese
                                         ; Hypothese wird momentan
                                         ; geglaubt (verworfen).
```

```
Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste              (8)  ändere Eintrag
  (2)  weiter                 (9)  warum
  (3)  bestätigt             (10)  warum nicht
  (4)  Vorschläge            (11)  bisherige Antworten
  (5)  Kandidaten            (12)  neue Beratung
  (6)  bestätige             (13)  Ende
  (7)  verwerfe

> 2
--> weiter                            ; fahre in der Konsulta-
                                      ; tion fort.

Es folgt das Füllen des Aspekts --Anleger-- für Kunde-1:

Wie lange möchten Sie Ihr Geld anlegen?
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht    (3)  mittel
  (2)  lang                   (4)  kurz

> k                                   ; als Antwort reicht eine
--> kurz                              ; Nummer oder ein eindeu-
                                      ; tiges Anfangsstück.

Wie hoch ist der Betrag, den Sie anlegen wollen?
Bitte wählen Sie aus:
  (1)  regelmäßig   (3)  mittel
  (2)  groß         (4)  klein

> k
--> klein

Zu welchem Zweck möchten Sie Ihr Geld anlegen?
Bitte wählen Sie aus:
  (1)  Spekulation            (3)  Vermögensansammlung
```

```
  (2)  Vermögensstreuung

> s
--> Spekulation

Haben Sie schon eigene Vorstellungen bzgl. der Anlageform?
Bitte wählen Sie aus:
  (1)  nein          (3)  Investment    (5)  Aktien
  (2)  Optionen      (4)  Zinspapiere

> o
--> Optionen

Wie hoch ist Ihre Renditeerwartung?
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht    (3)  normal
  (2)  überdurchschnittlich

> ?                                      ; Gib weitere Erläute-
                                         ; rungen.

Bei der Angabe der Renditeerwartung sollte man
nicht vergessen, daß Rendite und Risiko stets
einander bedingen.  Eine höhere Rendite muß
immer mit  Nachteilen bei der  Liquidität oder
beim Risiko erkauft werden.

Wie hoch ist Ihre Renditeerwartung?
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht    (3)  normal
  (2)  überdurchschnittlich

> ü
--> überdurchschnittlich

Wie groß ist Ihre Risikobereitschaft?
```

```
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht     (3)  normal
  (2)  überdurchschnittlich    (4)  klein

> ü
--> überdurchschnittlich

Muß das Geld während der Anlagedauer verfügbar sein?
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht     (3)  ja
  (2)  nein

> ?

Verfügbar sind Geldanlagen  im weitesten Sinne im-
mer, da man  schlimmstenfalls  die Anlage beleihen
kann.  Verfügbar soll hier heißen,  daß die Anlage
liquidierbar  ist  unter (teilweisem) Verzicht auf
die angestrebten  Anlageziele,  aber ohne  zusätz-
liche Kosten wie Kreditzinsen.

Muß das Geld während der Anlagedauer verfügbar sein?
Bitte wählen Sie aus:
  (1)  ich weiß noch nicht     (3)  ja
  (2)  nein

> 3
--> ja

Müssen steuerliche Aspekte schon jetzt beachtet werden?
Bitte wählen Sie aus:
  (1)  nein   (2)  ja

> 1
--> nein
                                      ; nach einer Runde werden
```

```
      **** Rangliste ****          ; diese Hypothesen ge-
                                   ; glaubt bzw. verworfen.
No. Bestätigt  S  Hypothese
 1.  100%      +  ausländisch
 2.   96%      +  Option/Optionsscheine
 3.   72%      +  Aktien
 4.   55%         Investmentfonds
 5.  -22%      -  Rentenpapier
 6.  -41%      -  kursneutral
 7.  -76%      -  Immobilienfonds


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste            (8)  ändere Eintrag
  (2)  weiter               (9)  warum
  (3)  bestätigt           (10)  warum nicht
  (4)  Vorschläge          (11)  bisherige Antworten
  (5)  Kandidaten          (12)  neue Beratung
  (6)  bestätige           (13)  Ende
  (7)  verwerfe

> 9
--> warum                          ; erkläre mir, warum
                                   ; Du glaubst, daß ...
Warum welche Hypothese?
Bitte wählen Sie aus:
  (1)  ausländisch            (5)  Rentenpapier
  (2)  Option/Optionsscheine  (6)  kursneutral
  (3)  Aktien                 (7)  Immobilienfonds
  (4)  Investmentfonds        (8)  Wertpapier

> 3                                ; ... Aktien in Frage
--> Aktien                         ; kommen.

-------------------------------------------------
```

```
Name:    Aktien
Konfidenz: 72.6%
--------------------------------------------------
--------------------------------------------------

Die Antwort auf die wichtige  Frage:
Wie lange möchten Sie Ihr Geld anlegen?
war: kurz
Das spricht sehr für  den Vorschlag
"Aktien"
(750 Punkte)
--------------------------------------------------
Soll ich weitere Gründe nennen? (J oder N) Ja
--------------------------------------------------

Die Antwort auf die  Frage:
Zu welchem Zweck möchten Sie Ihr Geld anlegen?
war: Spekulation
Das spricht absolut für  den Vorschlag
"Aktien"
(1000 Punkte)
--------------------------------------------------
Soll ich weitere Gründe nennen? (J oder N) Nein


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste             (8)  ändere Eintrag
  (2)  weiter                (9)  warum
  (3)  bestätigt            (10)  warum nicht
  (4)  Vorschläge           (11)  bisherige Antworten
  (5)  Kandidaten           (12)  neue Beratung
  (6)  bestätige            (13)  Ende
  (7)  verwerfe

> 2
```

```
--> weiter

Es folgt das Füllen des Aspekts --AKT-Anleger-- für Kunde-1:

In welchem Umfang (in DM) besitzen Sie schon Aktien?
Bitte wählen Sie aus:
  (1)  keine               (3)  zw. 10 u. 50 Tsd.
  (2)  unter 10 Tsd.       (4)  über 50 Tsd.

> 3
--> zw. 10 u. 50 Tsd.

Besitzen Sie schon Aktien ausländischer AG's?
Bitte wählen Sie aus:
  (1)  ja     (2)  nein

> n
--> nein

Was ist/wäre Ihr Ziel bei der Aktienanlage?
Bitte wählen Sie aus:
  (1)  schneller begrenzter Kursgewinn
  (2)  mittelfr. bedeutender Kursgewinn
  (3)  Vermögensstreuung

> 1
--> schneller begrenzter Kursgewinn

Wieviel Zeit haben Sie, sich um Kurse etc.
zu kümmern?
Bitte wählen Sie aus:
  (1)  keine Zeit   (2)  wenig Zeit   (3)  viel Zeit

> w
--> wenig Zeit
```

```
Wie groß ist Ihre Risikofreudigkeit bzgl. Aktien?
Bitte wählen Sie aus:
  (1)  klein    (2)  mittel   (3)  groß

> g
--> groß

Es folgt das Füllen des Aspekts --OPT-Anleger-- für Kunde-1:

Wie schätzen Sie die Kursentwicklung für die
nächste Zeit ein?
Bitte wählen Sie aus:
  (1)  fallend           (3)  steigend
  (2)  konstant          (4)  ich weiß nicht

> 1
--> fallend

Wie groß ist Ihre Risikobereitschaft
bei Optionen oder Optionsscheinen?
Bitte wählen Sie aus:
  (1)  klein    (2)  mittel   (3)  groß

> g
--> groß

Wie lange soll die Option bestehen?
Bitte wählen Sie aus:
  (1)  wenige Monate   (2)  wenige Jahre

> 1
--> wenige Monate

        **** Rangliste ****

No. Bestätigt  S  Hypothese
```

```
 1.  100%     +  Option
 2.   45%        Standardwert
 3.   24%        Turnaround
 4.   20%        Optionsschein
 5.   18%     -  ausl. Aktien
 6.    3%     -  Wachstumswert


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste              (8)  ändere Eintrag
  (2)  weiter                 (9)  warum
  (3)  bestätigt             (10)  warum nicht
  (4)  Vorschläge            (11)  bisherige Antworten
  (5)  Kandidaten            (12)  neue Beratung
  (6)  bestätige             (13)  Ende
  (7)  verwerfe

> vor
--> Vorschläge

(instance proposal-1                  ; Die bestätigten Hypo-
   (view proposal)                    ; thesen "ausländisch"
   (view wertpapier with              ; und "Option/Options-
        (herkunft = ausland)          ; schein" haben Views zu
        (risiko = enorm))             ; diesem Proposal beige-
   (view börsengängigeswertpapier)    ; tragen.
   (view terminpapier))


Soll ich mehr zeigen? (J oder N) Ja

(instance proposal-2                  ; Die bestätigten Hypo-
   (view proposal)                    ; thesen "ausländisch"
   (view wertpapier with              ; und "Aktien" haben
        (herkunft = ausland)          ; Views zu diesem Propo-
```

```
        (nominalwert = 50)            ; sal beigetragen.
        (börsengängig = ja)
        (deckungsstockfähig = nein)
        (lombardfähig = nein)
        (mündelsicher = nein)
        (absicherung = sonstiges))
   (view aktie)
   (view börsengängigeswertpapier))

Soll ich mehr zeigen? (J oder N) Ja


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste               (8)  ändere Eintrag
  (2)  weiter                  (9)  warum
  (3)  bestätigt              (10)  warum nicht
  (4)  Vorschläge             (11)  bisherige Antworten
  (5)  Kandidaten             (12)  neue Beratung
  (6)  bestätige              (13)  Ende
  (7)  verwerfe

> 10
--> warum nicht                       ; erkläre mir, warum
                                      ; Du glaubst, daß ...
Warum nicht welche Hypothese?
Bitte wählen Sie aus:
  (1)  Option                   (8)  Option/Optionsscheine
  (2)  Standardwert             (9)  ausländisch
  (3)  Turnaround              (10)  Wertpapier
  (4)  Optionsschein           (11)  Investmentfonds
  (5)  ausl. Aktien            (12)  Immobilienfonds
  (6)  Wachstumswert           (13)  kursneutral
  (7)  Aktien                  (14)  Rentenpapier

> 4
```

```
--> Optionsschein                      ; ... Optionsscheine
                                       ; nicht in Frage kommen.

--------------------------------------------------
Name:    Optionsschein
Konfidenz: 20.8%
--------------------------------------------------
--------------------------------------------------

Die Antwort auf die entscheidene  Frage:
Wie schätzen Sie die Kursentwicklung für die
nächste Zeit ein?
war: fallend
Das spricht absolut gegen  den Vorschlag
"Optionsschein"
(-1000 Punkte)
--------------------------------------------------
Soll ich weitere Gründe nennen? (J oder N) Nein


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste              (8)  ändere Eintrag
  (2)  weiter                 (9)  warum
  (3)  bestätigt             (10)  warum nicht
  (4)  Vorschläge            (11)  bisherige Antworten
  (5)  Kandidaten            (12)  neue Beratung
  (6)  bestätige             (13)  Ende
  (7)  verwerfe

> 10                                     ; ausl. Aktien sahen
--> warum nicht                          ; nach der ersten Runde
                                         ; doch vielversprechend
Warum nicht welche Hypothese?            ; aus.
Bitte wählen Sie aus:
  (1)  Option                   (8)  Option/Optionsscheine
```

```
  (2)  Standardwert              (9)  ausländisch
  (3)  Turnaround               (10)  Wertpapier
  (4)  Optionsschein            (11)  Investmentfonds
  (5)  ausl. Aktien             (12)  Immobilienfonds
  (6)  Wachstumswert            (13)  kursneutral
  (7)  Aktien                   (14)  Rentenpapier

> 5
--> ausl. Aktien
--------------------------------------------------
Name:    ausl. Aktien
Konfidenz: 18.6%
--------------------------------------------------
--------------------------------------------------

Die Antwort auf die entscheidene  Frage:
Was ist/wäre Ihr Ziel bei der Aktienanlage?
war: schneller begrenzter Kursgewinn
Das spricht sehr gegen  den Vorschlag
"ausl. Aktien"
(-800 Punkte)
--------------------------------------------------
Soll ich weitere Gründe nennen? (J oder N) Ja


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste               (8)  ändere Eintrag
  (2)  weiter                  (9)  warum
  (3)  bestätigt              (10)  warum nicht
  (4)  Vorschläge             (11)  bisherige Antworten
  (5)  Kandidaten             (12)  neue Beratung
  (6)  bestätige              (13)  Ende
  (7)  verwerfe

> 2
```

```
--> weiter

        **** Rangliste ****             ; die Rangliste bleibt
                                        ; leer: das Ende der
No. Bestätigt S  Hypothese              ; Hypothesenhierarchie
                                        ; ist erreicht.

Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste            (8)  ändere Eintrag
  (2)  weiter               (9)  warum
  (3)  bestätigt           (10)  warum nicht
  (4)  Vorschläge          (11)  bisherige Antworten
  (5)  Kandidaten          (12)  neue Beratung
  (6)  bestätige           (13)  Ende
  (7)  verwerfe

> vo
--> Vorschläge

(instance proposal-1                  ; Die Hypothesen "aus-
   (view proposal)                    ; ländisch" und "Aktien"
   (view wertpapier with              ; wurden revidiert. Die
        (herkunft = inland)           ; bestätigten Hypothesen
        (risiko = enorm)              ; "Option" und "Option/Op-
        (börsengängig = ja))          ; tionsschein" haben Views
   (view börsengängigeswertpapier)    ; zu diesem Proposal bei-
   (view terminpapier)                ; getragen.
   (view optionen))

Soll ich mehr zeigen? (J oder N) Ja


Konsultation>
Bitte wählen Sie aus:
  (1)  Rangliste            (8)  ändere Eintrag
```

```
  (2)  weiter               (9)  warum
  (3)  bestätigt           (10)  warum nicht
  (4)  Vorschläge          (11)  bisherige Antworten
  (5)  Kandidaten          (12)  neue Beratung
  (6)  bestätige           (13)  Ende
  (7)  verwerfe

> 13
--> Ende

Konsultation beendet
```

Anhang B

Literaturverzeichnis

[Abelson and Sussman, 1985] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[Aikins, 1983] Janice S. Aikins. Prototypical knowledge for expert systems. *Artificial Intelligence*, 20(2):163–210, 1983.

[Aikins, 1984] Janice S. Aikins. A representation scheme using both frames and rules. In B. Buchanan and E. Shortliffe, editors, *Rule Based Expert Systems*, pages 424–440. Addison Wesley, Reading, MA, 1984.

[Allen, 1978] J. Allen. *Anatomy of LISP*. McGraw-Hill, New York, 1978.

[Alty and Coombs, 1984] J. L. Alty and M. J. Coombs. *Expert Systems*. NCC Publications, The National Computing Centre Ltd., Manchester, England, 1984.

[Bachem, 1986] J. Bachem. EVA — Ein Expertensystem zur Vermögensanlageberatung. *GMD-Spiegel*, (2), 1986.

[Ballard and Brown, 1982] Dana H. Ballard and Christopher M. Brown. *Computer Vision.* Prentice Hall, Englewood Cliffs, New Jersey, 1982.

[Baltes, 1985] H. Baltes. GABI — Ein wissensbasiertes Geldanlageberatungsprogramm. Memo 3, Sonderforschungsbereich 314, Universität des Saarlandes, Saarbrücken, November 1985.

[Barr and Feigenbaum, 1981a] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 1. Morgan Kaufmann Publishers, Los Altos, CA, 1981.

[Barr and Feigenbaum, 1981b] Avron Barr and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 2. Morgan Kaufmann Publishers, Los Altos, CA, 1981.

[Bauer and Radig, 1985] W. Bauer and B. Radig, Herausgeber. *GI Kongreß 1985: Wissensbasierte Systeme*, Informatik-Fachberichte 112. Springer, 1985.

[Berkley and Bobrow, 1964] Edmund C. Berkley and Daniel G. Bobrow. *The Programming Language LISP: Its Operation and Applications.* MIT Press, 1964.

[Bobrow and Collins, 1975] Daniel G. Bobrow and A. M. Collins, editors. *Representation and Understanding: Studies in Cognitive Science.* Academic Press, New York, 1975.

[Bobrow and Hayes, 1985] Daniel G. Bobrow and Patrick J. Hayes. Artificial intelligence – where are we? *Artificial Intelligence*, 24:375–415, 1985.

[Bobrow and Stefik, 1981] Daniel G. Bobrow and M. Stefik. The LOOPS manual. Technical Report KB-VLSI-81-13, XEROX Palo Alto Research Center, 1981.

[Bobrow and Winograd, 1977] Daniel G. Bobrow and Terry Winograd. An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1):3–46, 1977.

[Bobrow and Winograd, 1979] Daniel G. Bobrow and Terry Winograd. KRL: Another perspective. *Cognitive Science*, 3(1):29–42, 1979.

[Bobrow, 1980] Daniel G. Bobrow. Special issue on nonmonotonic logic. *Artificial Intelligence*, 13:1–174, 1980.

[Boden, 1977] Margaret Boden. *Artificial Intelligence and Natural Man*. Harvester Press Limited, 1977.

[Bossu and Siegel, 1985] G. Bossu and P. Siegel. Saturation, non-monotonic reasoning and the closed world assumption. *Artificial Intelligence*, 25:13–63, 1985.

[Brachman and Levesque, 1985] Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*. Morgan Kaufmann Publishers, Los Altos, CA, 1985.

[Brachman and Schmolze, 1985] Ronald J. Brachman and J. G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[Brachman, 1979] Ronald J. Brachman. On the epistemological status of semantic networks. In N. V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 3–50. Academic Press, New York, 1979.

[Brodie *et al.*, 1984] M. Brodie, J. Mylopoulos, and J. W. Schmidt, editors. *Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*. Springer, 1984.

[Brownston *et al.*, 1985] Lee Brownston, Robert Farell, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS 5*. Addison-Wesley, Reading, MA, 1985.

[Buchanan and Shortliffe, 1984] B. Buchanan and E. Shortliffe, editors. *Rule Based Expert Systems*. Addison Wesley, Reading, MA, 1984.

[Buchanan *et al.*, 1983] B. G. Buchanan, D. Barstow, R.Bechtel, J. Bennet, W. Clansey, and C. Kulikowski. Constructing an expert system. In Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat, editors, *Building Expert Systems*, chapter 5. Addison-Wesley, Reading, MA, 1983.

[Budde *et al.*, 1984] R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllinghoven. Approaches to rapid prototyping. In *Proceedings of the Working Conference on Prototyping 1983*, Berlin, 1984. Springer.

[Charniak and McDermott, 1985] Eugene Charniak and Drew V. McDermott. *Introduction to Artificial Intelligence.* Addison-Wesley, Reading, MA, 1985.

[Charniak *et al.*, 1980] Eugene Charniak, Christopher K. Riesbeck, and Drew V. McDermott. *Artificial Intelligence Programming.* Lawrence Erlbaum Associates, New Jersey, 1980.

[Christaller *et al.*, 1989] Thomas Christaller, Franco DiPrimio, and Angi Voss, editors. *Die KI-Werkbank BABYLON: Eine offene und portable Entwicklungsumgebung für Expertensysteme.* Addison-Wesley, Bonn, 1989.

[Clancey, 1985] W. J. Clancey. Heuristic classification. *Artificial Intelligence*, 27:289–350, 1985.

[Clarkson, 1963] G. P. E. Clarkson. A model of the trust investment process. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought.* McGraw-Hill, 1963. (reprinted from: *A Simulation of Trust Investment.* Prentice Hall, Englewood Cliffs, NJ 1961).

[Clocksin and Mellish, 1990] William F. Clocksin and Christopher S. Mellish. *Programmieren in Prolog.* Symbolic Computation. Springer, Berlin, third edition, 1990.

[Codd, 1970] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), June 1970. (reprinted volume 6 number 1, Januar 1983).

[Coelho and Cotta, 1988] Helder Coelho and Jose Carlos Cotta. *Prolog by Example: How to Learn, Teach and Use It.* Springer, Berlin, 1988.

[Cohen and Feigenbaum, 1982a] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 3. Morgan Kaufmann Publishers, Los Altos, CA, 1982.

[Cohen and Feigenbaum, 1982b] Paul R. Cohen and Edward A. Feigenbaum. *The Handbook of Artificial Intelligence*, volume 4. Morgan Kaufmann Publishers, Los Altos, CA, 1982.

[Cooper, 1983] Doug Cooper. *STANDARD PASCAL User Reference Manual.* W.W. Norton & Company, New York, 1983.

[Cremers und Becks, 1987] A. B. Cremers und K.-H. Becks, Herausgeber. *Anwender-Forum Expertensysteme Proceedings*, Universität Dortmund, Lehrstuhl für Informationssysteme, 1987.

[Darnell and Margolis, 1988] Peter A. Darnell and Philip E. Margolis. *Software Engineering in C.* Springer Books on Professional Computing. Springer, New York, 1988.

[Date, 1984] C. J. Date. *A Guide to DB2.* Addison-Wesley, Reading, MA, 1984.

[Davis and Lenat, 1980] Randall Davis and Douglas B. Lenat, editors. *Knowledge Based Systems in Artificial Intelligence.* McGraw-Hill, 1980.

[Davis, 1980] M. Davis. The mathematics of non-monotonic reasoning. *Artificial Intelligence*, 13:73–80, 1980.

[de Kleer and Williams, 1987] Johan de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[de Kleer, 1986a] Johan de Kleer. An assumption based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[de Kleer, 1986b] Johan de Kleer. Extending the TMS. *Artificial Intelligence*, 28:163–196, 1986.

[de Kleer, 1986c] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28:197–224, 1986.

[Deering *et al.*, 1981] Michael Deering, Joseph Faletti, and Robert Wilensky. PEARL: A package for efficient access to representations in LISP. In *Proc. of the 7^{th} IJCAI*, pages 930–932, Vancouver, Canada, 1981.

[Deering *et al.*, 1982] Michael Deering, Joseph Faletti, and Robert Wilensky. Using the PEARL AI package. Memo 19, University of California in Berkley, Berkley, CA, 1982.

[Dehio and Kieser, 1983] P. Dehio and A. Kieser. *Die Gestaltung von Entscheidungsunterstützungssystemen.* Angewandte Informatik 9, 1983.

[Delgrande and Jackson, 1991] James P. Delgrande and W. Ken Jackson. Default logic revisited. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 118–127, Cambridge, MA, April 1991. Morgan Kaufmann Publishers.

[Delgrande, 1988] James P. Delgrande. An approach to default reasoning based on a first-order conditional logic: Revised report. *Artificial Intelligence*, 36:63–90, 1988.

[Dewhurst and Stark, 1990] Stephen C. Dewhurst and Kathy S. Stark. *Programmieren in C++.* Carl Hanser Verlag, München – Wien, 1990.

[Doyle, 1979a] Jon Doyle. A truth maintenance system. Memo 521, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1979.

[Doyle, 1979b] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[Doyle, 1983] Jon Doyle. The ins and outs of reason maintenance. In *Proc. of the 8^{th} IJCAI*, pages 349–351, Karlsruhe, FRG, 1983.

[Dreyfus, 1985a] Hubert L. Dreyfus. *Die Grenzen künstlicher Intelligenz: Was Computer nicht können.* Athenäum Verlag GmbH, Königstein, Taunus, 1985. (engl. Originialtitel: What Computers Can't Do — The Limits of Artificial Intelligence. 1979).

[Dreyfus, 1985b] Hubert L. Dreyfus. From micro-worlds to knowledge representation: AI at an impasse. In *Readings in Knowledge Representation*, pages 71–93. Morgan Kaufmann Publishers, Los Altos, CA, 1985.

[Enderton, 1972] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Etherington and Reiter, 1983] D. W. Etherington and Ray Reiter. On inheritance hierarchies with exceptions. In *Proc. of AAAI-83*, pages 104–108, Washington, DC, 1983.

[Etherington, 1986] D. W. Etherington. *Reasoning with Incomplete Information: Investigations of Non-monotonic Reasoning.* PhD thesis, University of British Columbia, Vancouver, Canada, 1986.

[Etherington, 1987] D. W. Etherington. Formalizing non-monotonic reasoning systems. *Artificial Intelligence*, 31:41–85, 1987.

[Fahlman, 1979] Scott E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge.* MIT Press, Cambridge, MA, 1979.

[Feigenbaum and Feldman, 1963] Edward A. Feigenbaum and Julian Feldman, editors. *Computers and Thought.* McGraw-Hill, 1963.

[Feigenbaum and McCorduck, 1983] Edward A. Feigenbaum and Pamela McCorduck. *The Fifth Generation.* Addison-Wesley, Reading, MA, 1983.

[Fikes and Kehler, 1985] Richard Fikes and Tom Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, September 1985.

[Findler, 1979] N. V. Findler, editor. *Associative Networks: Representation and Use of Knowledge by Computers.* Academic Press, New York, 1979.

[Foderaro, 1979] John K. Foderaro. *The FRANZ LISP Manual.* University of California, Berkley, 1979.

[Forgy, 1982] C. L. Forgy. Rete: A fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence*, 19:17–38, 1982.

[Gabbay, 1982] Dov M. Gabbay. Intuitionistic basis for non-monotonic logics. In D. W. Loveland, editor, *Springer Lecture Notes in Computer Science*, number 138, pages 260–273, 1982.

[Gabow *et al.*, 1976] Harold N. Gabow, Shachindra N. Maheshwari, and Leon J. Osterweil. On two problems in the generation of program test paths. *IEEE Transactions on Software Engineering*, SE-2(3), September 1976.

[Garey and Johnson, 1979] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[Gelfond, 1986] M. Gelfond. On stratified autoepistemic theories. Technical report, Department of Computer Science, University of Texas, El Paso, 1986.

[Gelfond, 1987] M. Gelfond, editor. *Readings in Nonmonotonic Reasoning.* Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Genesereth and Ginsberg, 1985] Michael R. Genesereth and M. L. Ginsberg. Logic programming. *Communications of the ACM*, 28:933–941, 1985.

[Genesereth and Nilsson, 1987] Michael R. Genesereth and N. J. Nilsson. *Logical Foundations of Artificial Intelligence.* Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Ginsberg, 1987a] M. L. Ginsberg. Multi-valued logics I: Formal description. Technical Report 87-4, Logic Group, Stanford University, Stanford, CA, 1987.

[Ginsberg, 1987b] M. L. Ginsberg. Multi-valued logics II: Inference. Technical Report 87-5, Logic Group, Stanford University, Stanford, CA, 1987.

[Ginsberg, 1987c] M. L. Ginsberg. Multi-valued logics III: Applications. Technical Report 87-6, Logic Group, Stanford University, Stanford, CA, 1987.

[Goldberg and Robinson, 1983] Adele Goldberg and David Robinson. *Smalltalk-80.* Addison-Wesley, Reading, MA, 1983.

[Hamann, 1982] Christian-Michael Hamann. *Einführung in das Programmieren in LISP.* Walter de Gruyter & Co., 1982.

[Harbison and Jr., 1991] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice Hall, Englewood Cliffs, NJ, third edition, 1991.

[Harmon *et al.*, 1989] Paul Harmon, Rex Maus, and William Morrissey. *Expertensysteme: Werkzeuge und Anwendungen.* R. Oldenbourg Verlag, München – Wien, 1989.

[Hayes *et al.*, 1982] J. E. Hayes, Donald Mitchie, and Y-H. Pao, editors. *Machine Intelligence*, volume 10. Ellis Horwood Limited, 1982.

[Hayes-Roth *et al.*, 1983] Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems.* Addison-Wesley, Reading, MA, 1983.

[Hayes, 1979] Patrick J. Hayes. The logic of frames. In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter & Co., Berlin, 1979.

[Hendrix, 1979] G. G. Hendrix. Encoding knowledge in partitioned networks. In N. V. Findler, editor, *Associative Networks: Representation and Use of Knowledge by Computers*, pages 51–92. Academic Press, New York, 1979.

[Heuer *et al.*, 1988] Stephan Heuer, Ulrich Koch, and Colin Cryer. Invest: An expert system for financial investments. *IEEE Expert, Intelligent Systems and Their Applications*, 3(2), Summer 1988.

[Heuer, 1987] Stephan Heuer. Implementation einer Frame-basierten Wissensrepräsentation für Expertenssysteme. Diplomarbeit, Institut für Numerische und instrumentelle Mathematik der Westfälischen Wilhelms-Universität zu Münster, Münster, NRW, März 1987.

[Hewitt, 1977] Hewitt. *Viewing Control Structures as Patterns of Passing Messages.* Artificial Intelligence Volume 8, 1977.

[Hillis, 1985] W. D. Hillis. *The Connection Machine.* MIT Press, Cambridge, MA, 1985.

[Horty and Thomason, 1990] John F. Horty and Richmond H. Thomason. Boolean extensions of inheritance networks. In *Proc. of AAAI-90*, pages 633–639, Boston, MA, 1990.

[Horty *et al.*, 1987a] John F. Horty, Richmond H. Thomason, and David S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. In *Proc. of AAAI-87*, pages 358–363, Seattle, WA, 1987.

[Horty *et al.*, 1987b] John F. Horty, Richmond H. Thomason, and David S. Touretzky. A skeptical theory of inheritance in nonmonotonic semantic networks. Technical Report CMU-CS-87-175, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, October 1987.

[Horty, 1991] John F. Horty. Some direct theories of nonmonotonic inheritance. Draft of an article, University of Maryland, Institute for Advanced Computer Studies, College Park, MD, 1991.

[Humpert and Holley, 1988] B. Humpert and P. Holley. Expert systems in finance planning. *Expert Systems*, 5(2):78–101, 1988.

[Jackson, 1990] Peter Jackson. *Introduction to Expert Systems*. Addison-Wesley, Reading, MA, second edition, 1990.

[Jensen and Wirth, 1978] K. Jensen and N. Wirth. *PASCAL User Manual and Report*. Springer, Berlin - Heidelberg - New York, 1978.

[Johnson, 1983] P. E. Johnson. What kind of expert should a system be? *The Journal of Medicine and Philosophy*, 8:77–97, 1983.

[Joslin *et al.*, 1986] J. L. Joslin, G. Lorant, and R. E. C. Wegner. The multi-asset allocation model: A financial planning framework for long-term investment management. *Tax Management Financial Planning Journal*, 2(21):523–531, 1986.

[Kernighan and Ritchie, 1988] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, second edition, 1988.

[Klein and Methlie, 1990] M. Klein and L. Methlie. *Expert Systems in Finance*. Addison-Wesley, Reading, MA, 1990.

[Koch und Heuer, 1986] Ulrich Koch und Stephan Heuer. Die Expertensystem-Shell DONALD. Preprints Angewandte Mathematik und Informatik 8/86 I, Institut für Numerische und instrumentelle Mathematik der Westfälischen Wilhelms-Universität zu Münster, Münster, NRW, November 1986.

[Koch und Heuer, 1987] Ulrich Koch und Stephan Heuer. Die Expertensystem-Shell DONALD. In A. B. Cremers und K.-H. Becks, Herausgeber, *Anwender-Forum Expertensysteme Proceedings*, pages 115–125, Universität Dortmund, Lehrstuhl für Informationssysteme, 1987.

[Koch, 1989a] Ulrich Koch. *Implementation einer musterorientierten Expertensystem-Shell und Kalküle für eine intuitionistische Modallogik als*

Basis des nicht-monotonen Schließens. Dissertation, Institut für Numerische und instrumentelle Mathematik der Westfälischen Wilhelms-Universität zu Münster, Münster, NRW, Dezember 1989.

[Koch, 1989b] Ulrich Koch. INVEST: Ein Expertensystem zur Wertpapieranlageberatung. In Johannes Kuntner, Herausgeber, *Expertensysteme im Bankbereich*, volume 2 of *Schriftenreihe der Österreichischen Gesellschaft für Artificial Intelligence*, pages 61–75, Wien, 1989. Eigenverlag ÖGAI.

[Kowalski and Sergot, 1986] Robert A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[Krasner, 1984] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, Reading, MA, 1984.

[Krishnaprasad, 1989] Thirunaray Krishnaprasad. *The Semantics of Inheritance Systems.* PhD thesis, State University of New York at Stony Brook, December 1989.

[Kuntner, 1989] Johannes Kuntner, Herausgeber. *Expertensysteme im Bankbereich*, volume 2 of *Schriftenreihe der Österreichischen Gesellschaft für Artificial Intelligence*, Wien, 1989. Eigenverlag ÖGAI.

[Kunz *et al.*, 1978] J. C. Kunz, R. J. Fallat, D. H. McClung, J. J. Osborn, R. A. Votteri, H. P. Nii, J. S. Aikins, L. M. Fagan, and E. A. Feigenbaum. A physiological rules-based system for interpreting pulmonary function test results. Report HPP-78-19, Heuristic Programming Project, Computer Science Department, Stanford University, 1978.

[Lamarre, 1991] Philippe Lamarre. S4 as the conditional logic of nonmonotonicity. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 357–367, Cambridge, MA, April 1991. Morgan Kaufmann Publishers.

[Ledgard, 1980] Henry Ledgard. *ADA: An Introduction [Part I]; Ada Reference Manual [Part II].* Springer, New York, 1980.

[Lehnert, 1977] H. J. Lehnert. A critical perspective on KRL. *Cognitive Science*, 3(1):1–28, 1977.

[Lifschitz, 1985] V. Lifschitz. Closed-world databases and circumscription. *Artificial Intelligence*, 27:229–235, 1985.

[Lifschitz, 1986] V. Lifschitz. On the satisfiabilty of circumscription. *Artificial Intelligence*, 28:17–27, 1986.

[Lukaszewicz, 1990] Witold Lukaszewicz. *Non-monotonic Reasoning: Formalizations of Commonsense Reasoning.* Ellis Horwood Limited, New York, 1990.

[Marr, 1982] David Marr. *Vision.* W. H. Freeman and Company, San Francisco, CA, 1982.

[McCarthy *et al.*, 1962] John McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Evin. *LISP 1.5 Programmers Manual.* MIT Press, Cambridge, MA, 1962.

[McCarthy, 1960] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 7:184–195, 1960.

[McCarthy, 1980] John McCarthy. Circumscription – a form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39,171–172, 1980.

[McCarthy, 1985] John McCarthy. Programs with common sense. In *Readings in Knowledge Representation*, pages 299–307. Morgan Kaufmann Publishers, Los Altos, CA, 1985.

[McCarthy, 1986] John McCarthy. Applications of circumscription to formalizing common sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[McDermott and Doyle, 1980] Drew V. McDermott and Jon Doyle. Non-monotonic logic I. *Artificial Intelligence*, 13:41–72, 1980.

[McDermott, 1982a] Drew V. McDermott. Non-monotonic logic II. *Artificial Intelligence*, 29:33–57, 1982.

[McDermott, 1982b] J. McDermott. R1: A rule-based configurer of computer systems. *Artificial Intelligence*, 19:39–88, 1982.

[McDermott, 1982c] J. McDermott. XSEL: A computer sales person's assistant. In J. E. Hayes, Donald Mitchie, and Y-H. Pao, editors, *Machine Intelligence*, volume 10, pages 325–337. Ellis Horwood Limited, 1982.

[Mendelson, 1979] E. Mendelson. *An Introduction to Mathematical Logic.* Van Nostrand, New Jersey, 1979.

[Merritt, 1989] D. Merritt. *Building Expert Systems in Prolog.* Springer, New York, 1989.

[Metzing, 1979] D. Metzing, editor. *Frame Conceptions and Text Understanding.* Walter de Gruyter & Co., Berlin, 1979.

[Miller *et al.*, 1982] A. M. Miller, H. E. Pople, and J.D Myers. INTERNIST-I, an experimental computer-based diagnostic consultant for general internal medicine. *New England Journal of Medicine*, (307):468–476, 1982.

[Minsky, 1963] Marvin Minsky. Steps toward artificial intelligence. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought.* McGraw-Hill, 1963. (reprinted from: *Proceedings of The Institute of Radio Engineers.* January 1961, 49:8–30).

[Minsky, 1975] Marvin Minsky. A framework for representing knowledge. In Patrick Henry Winston, editor, *The Psychology of Computer Vision*, chapter 6, pages 211–277. McGraw-Hill, New York, 1975.

[Mitchie, 1984] Donald Mitchie, editor. *Introductory Readings in Expert Systems*, volume 1 of *Studies in Cybernetics.* Gordon and Breach Science Publishers, 1984.

[Moon and Weinreb, 1980] D. Moon and D. Weinreb. Flavors: Message passing in the lisp machine. Memo 602, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1980.

[Moore, 1985] R. C. Moore. Semantical considerations on nonmonotonic reasoning. *Artificial Intelligence*, 25:75–94, 1985.

[Mylopoulos and Levesque, 1983] J. Mylopoulos and H. J. Levesque. An overview of knowledge representation. In M. Brodie, J. Mylopoulos, and J. W. Schmidt, editors, *Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages*, pages 3–17. Springer, 1983.

[Newell and Simon, 1963] A. Newell and H. Simon. GPS, a program that simulates human thought. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought*. McGraw-Hill, 1963. (reprinted from: *Lernende Automaten*. München, R. Oldenbourg KG, 1961).

[Nilsson, 1980] Nils J. Nilsson. *Principles of Artificial Intelligence*. Morgan Kaufmann Publishers, Los Altos, CA, 1980.

[Perlis and Minker, 1986] D. Perlis and J. Minker. Completeness results for circumscription. *Artificial Intelligence*, 28:29–42, 1986.

[Puppe, 1990] Frank Puppe. *Einführung in Expertensysteme*. Addison-Wesley, Reading, MA, 1990.

[Quillian, 1967] M. R. Quillian. Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, 12:410–430, 1967.

[Quillian, 1968] M. R. Quillian. Semantic memory. In Marvin Minsky, editor, *Semantic Information Processing*, pages 227–270. MIT Press, 1968.

[Reiter and Criscuolo, 1981] Ray Reiter and G. Criscuolo. On interacting defaults. *Proc. of the* 7^{th} *IJCAI*, pages 270–276, 1981.

[Reiter, 1980] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Reiter, 1987] Ray Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[Rich, 1983] Elaine Rich. *Artificial Intelligence*. McGraw-Hill, 1983.

[Roberts and Goldstein, 1977a] R. B. Roberts and I. P. Goldstein. The FRL manual. Memo 409, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1977.

[Roberts and Goldstein, 1977b] R. B. Roberts and I. P. Goldstein. The FRL primer. Memo 408, MIT Artificial Intelligence Laboratory, Cambridge, MA, 1977.

[Robinson, 1983] J. A. Robinson. Theorem-proving on the computer. *Journal of the ACM*, 10:163–174, 1983.

[Robinson, 1985] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1985.

[Savory, 1985] Stuart E. Savory, editor. *Künstliche Intelligenz und Expertensysteme*, München – Wien, 1985. R. Oldenbourg Verlag.

[Schank and Abelson, 1977] Roger C. Schank and Robert P. Abelson. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates, New Jersey, 1977.

[Schank and Riesbeck, 1981] Roger C. Schank and Christopher K. Riesbeck. *Inside Computer Understanding.* Lawrence Erlbaum Associates, New Jersey, 1981.

[Selman and Levesque, 1991] Bart Selman and Hector J. Levesque. The tractability of path-based inheritance. In John F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, The Morgan Kaufmann Series in Representation and Reasoning, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[Shapiro, 1987a] Stuart C. Shapiro, editor. *Encyclopedia of Artificial Intelligence*, volume 1. John Wiley & Sons, New York, 1987.

[Shapiro, 1987b] Stuart C. Shapiro, editor. *Encyclopedia of Artificial Intelligence*, volume 2. John Wiley & Sons, New York, 1987.

[Sheperdsen, 1985] J. C. Sheperdsen. Negation as failure II. *Journal of Logic Programming*, 3:185–202, 1985.

[Shortliffe, 1976] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* Elsevier, New York, 1976.

[Smith and Clayton, 1984] David E. Smith and Jan E. Clayton. Another look at frames. In B. Buchanan and E. Shortliffe, editors, *Rule Based Expert Systems.* Addison Wesley, Reading, MA, 1984.

[Sowa, 1991] John F. Sowa, editor. *Principles of Semantic Networks: Explorations in the Representation of Knowledge.* The Morgan Kaufmann Series in Representation and Reasoning. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[Stansfield and Greenfield, 1987] J. L. Stansfield and N. R. Greenfield. PlanPower: a comprehensive financial planner. *IEEE Expert*, 2(3):51–60, 1987.

[Steele Jr., 1990] Guy L. Steele Jr. *Common LISP: The Language.* Digital Press, second edition, 1990.

[Stein, 1991] Lynn Andrea Stein. Extensions as possible worlds. In John F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, The Morgan Kaufmann Series in Representation and Reasoning, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, MA, 1986.

[Stoyan and Görz, 1984] Herbert Stoyan and Günter Görz. *LISP: Eine Einführung in die Programmierung.* Springer, Berlin, 1984.

[Stroustrup, 1986] Bjaren Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Tatar, 1987] Deborah G. Tatar. *A Programmer's Guide to COMMON LISP.* Digital Press, 1987.

[Thomason and Aronis, 1990] Richmond H. Thomason and John M. Aronis. Hybridizing nonmonotonic inheritance with theorem proving. Technical Report CMU-CS-90-174, University of Pittsburgh, Intelligent Systems Program, Pittsburgh, PA, October 1990.

[Thomason and Horty, 1988] Richmond H. Thomason and J. F. Horty. Mixing strict and defeasible inheritance. In *Proc. of AAAI-88*, pages 427–432, St. Paul, MN, 1988.

[Thomason and Touretzky, 1991] Richmond H. Thomason and Davis S. Touretzky. Inheritance theory and networks with roles. In John F. Sowa, editor, *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, The Morgan Kaufmann Series in Representation and Reasoning, San Mateo, CA, 1991. Morgan Kaufmann Publishers.

[Thomason *et al.*, 1986] Richmond H. Thomason, John F. Horty, and David S. Touretzky. A calculus for inheritance in monotonic semantic nets. Technical Report CMU-CS-138, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA, 1986.

[Thomason, 1989] Richmond H. Thomason. Completeness proofs for monotonic nets with relations and identity. In Zbigniew W. Ras, editor, *Proceedings of the Fourth International Symposium on Methodologies for Intelligent Systems*, Charlotte, North Carolina, October 1989.

[Thomason, 1990] Richmond H. Thomason. Fahlman's netl and inheritance theory. Technical report, Carnegie Mellon University, Intelligent Systems Program, Pittsburgh, PA, 1990.

[Tinhofer *et al.*, 1990] G. Tinhofer, E. Mayr, H. Noltemeier, and M. Syslo, editors. *Computational Graph Theory.* Springer, Wien, 1990.

[Touretzky and Thomason, 1988] David S. Touretzky and Richmond H. Thomason. Nonmonotonic inheritance and generic reflexives. In *Proc. of AAAI-88*, pages 433–438, St. Paul, MN, 1988.

[Touretzky and Thomason, 1990] David S. Touretzky and Richmond H. Thomason. An inference algorithm for networks that mix strict and defeasible inheritance. Draft paper, Carnegie Mellon University, University of Pittsburgh, 1990.

[Touretzky *et al.*, 1987] David S. Touretzky, Richmond H. Thomason, and John F. Horty. A clash of intuitions: the current state of nonmonotonic multiple inheritance systems. In *Proc. of the 10th IJCAI*, pages 476–482, Milan, Italy, 1987.

[Touretzky *et al.*, 1991] David S. Touretzky, Richmond H. Thomason, and John F. Horty. A skeptic's menagerie; conflictors, preemptors, reinstaters, and zombies in nonmonotonic inheritance. Draft paper, Carnegie Mellon University, University of Pittsburgh, University of Maryland, Pittsburgh, PA, 1991.

[Touretzky, 1984] David S. Touretzky. Implicit ordering of defaults in inheritance systems. *Proc. 5th Nat. Conf. on AI*, pages 322–325, 1984.

[Touretzky, 1986] David S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann Publishers, Los Altos, CA, 1986.

[Turing, 1963] Alan M. Turing. Computing machinery and intelligence. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought.* McGraw-Hill, 1963. (reprinted from: *Mind*, October 1950, 59:433-460).

[Urmi, 1976] Jaak Urmi. *INTERLISP/370 Reference Manual.* Department of Mathematics, Linkoeping University, Sweden, 1976.

[Waterman, 1986] Donald A. Waterman. *A Guide to Expert Systems.* Addison-Wesley, Reading, MA, 1986.

[Weyhrauch, 1980] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.

[Wilensky, 1986] Robert Wilensky. *Common LISPcraft.* W.W. Norton & Company, New York, 1986.

[Winograd, 1980] Terry Winograd. Extended inference modes in reasoning by computer systems. *Artificial Intelligence*, 13:5–26, 1980.

[Winston and Brown, 1979a] Patrick Henry Winston and R. H. Brown. *Artificial Intelligence: An MIT Perspective*, volume 1. MIT Press, 1979.

[Winston and Brown, 1979b] Patrick Henry Winston and R. H. Brown. *Artificial Intelligence: An MIT Perspective*, volume 2. MIT Press, 1979.

[Winston and Horn, 1983] Patrick Henry Winston and Berthold Horn. *LISP.* Addison-Wesley, Reading, MA, second edition, 1983.

[Winston, 1975] Patrick Henry Winston, editor. *The Psychology of Computer Vision.* McGraw-Hill, 1975.

[Winston, 1984] Patrick Henry Winston, editor. *Artificial Intelligence.* Addison-Wesley, Reading, MA, second edition, 1984.

[Witteveen, 1991] Cees Witteveen. Skeptical reason maintenance is tractable. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 570–581, Cambridge, MA, April 1991. Morgan Kaufmann Publishers.

[Woods, 1975] William A. Woods. What's in a link: Foundations for semantic networks. In D. G. Bobrow and A. M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82. Academic Press, New York, 1975.

[Yuasa and Hagiya, 1985] Taiichi Yuasa and Masami Hagiya. Kyoto Common Lisp Report. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1985.

Lebenslauf und Bildungsgang

Name	Stephan Heuer
Geburtstag, -ort	15.03.1960 in Albersloh
Familienstand	ledig
Eltern	Helmut Heuer Friederike Heuer, geb. Lückertz

Schulbildung

1966–1969	St. Josefs Grundschule in Münster
1969–1978	Gymnasium Paulinum in Münster
05.06.1978	Abitur

1978–1979	Wehrdienst in Bad Segeberg, Hamburg und Münster

Studium

WS 1979/80	Aufnahme des Studiums der Mathematik an der Westfälischen Wilhelms-Universität Münster
10.02.1983	Vordiplom in Mathematik
01.03.1984– 31.03.1987	Studentische Hilfskraft am Rechenzentrum und am Institut für Numerische und instrumentelle Mathematik der WWU Münster
01.09.1986	Beginn der Diplomarbeit
30.03.1987	Diplom in Mathematik mit Nebenfach Physik. Diplomarbeit über das Thema „Implementation einer Frame-basierten Wissensrepräsentation für Expertensysteme“
seit 01.04.1987	Wissenschaftl. Mitarbeiter am Institut für Numerische und instrumentelle Mathematik der WWU Münster Beginn der Dissertation