Arbeitsblätter zur Lehrveranstaltung

Programmierparadigmen der Künstlichen Intelligenz

des Studiengangs

Wirtschaftsinformatik

an der

Technischen Universität Ilmenau

apl. Prof. Dr.-Ing. habil. Rainer Knauf

Inhaltsverzeichnis

1 Einführung

1.1 Teilgebiete der Künstlichen Intelligenz

Einige Teilgebiete[1] der Künstlichen Intelligenz (KI) sind

- **Wissensrepräsentation,**
- **maschinelles Beweisen (deduktive Inferenz),**
- Sprachverarbeitung,
- Bildverarbeitung,
- Spielprogramme,
- algorithmisches Lernen (induktive und analoge Inferenz),
- Robotik,
- **KI–Sprachen** und
- **Wissensbasierte Systeme**.

Letzteres ist ein „Kerngebiet“ der KI; es findet in der akademischen Forschung und der industriellen Praxis die meiste Beachtung und Anwendung. Die meisten der anderen Gebiete entwickelten sich

- aufgrund der Tatsache, dass sie Grundlagen (z.B. Wissensrepräsentation, Deduktion, induktive und analoge Inferenz) bzw. Werkzeuge (z.B. KI–Sprachen) liefern oder
- aufgrund der inhaltlichen Verwandtschaft (Anwendung von Methoden der Wissensverarbeitung wie z.B. bei Spielprogrammen) oder
- aufgrund der Tatsache, dass es sich um sinnvolle Erweiterungen bzw. Anwendungen Wissensbasierter Systeme handelt (z.B. Sprachverarbeitung, Bildverarbeitung) oder
- (umgekehrt) aufgrund der Tatsache, dass Wissensbasierte Systeme wesentliche Komponenten beitragen (wie etwa in der Robotik),

zu Teilgebieten der KI.

Der Leitgedanke der KI besteht in der „Mechanisierung von Denkprozessen“, d.h. der Anwendung von Methoden der Wissensverarbeitung. **G.W. LEIBNIZ** (wahrscheinlich der Schöpfer dieser Idee) nannte die dazu notwendigen Komponenten

- „**lingua characteristica**“ (heute würde man dies „*Wissensdarstellungssprache*“ oder schlicht „formale Wissensdarstellung“ nennen) und
- „**calculus ratiocinator**“ (ein „*Wissensverarbeitungskalkül*“, der aus einer Menge von Formulierungen in der lingua characteristica nach vorgegebenen Verarbeitungsmechanismen neue Formulierungen der lingua characteristica erzeugen kann).

Diese beiden Begriffe stehen im Mittelpunkt der Lehrveranstaltung und werden durch formale Konzepte untersetzt.

[1] Die **fett gedruckten** Teilgebiete sind die in der Lehrveranstaltung betrachteten.

1.2 Evolution der Programmiersprachen bis hin zu den KI–Sprachen

Eine **Programmiersprache** ist eine zum Abfassen (Formulieren) von Computerprogrammen geschaffene formale Sprache (DIN 44300, Teil 4). Man unterscheidet Sprachen der

- 1. Generation: Maschinensprachen
- 2. Generation: maschinenorientierte Sprachen (Assemblersprachen)
- 3. Generation: problemorientierte Sprachen (höhere Programmiersprachen)
- 4. Generation: datenorientierte Sprachen (Datenbank–Abfragesprachen, Software–Entwicklungswerkzeuge)
- 5. Generation: wissensorientierte Sprachen (KI–Sprachen)

Je nach Verwendungszweck sind gegenwärtig alle Generationen anzutreffen.

Analog zu den natürlichen Sprachen unterscheidet man bei Programmiersprachen

- Syntax

  Die Syntax ist eine formale Beschreibung „erlaubter“ Zeichenreihen, die durch eine Grammatik G definiert sind. $G = [A, V, R, S]$ mit

  - einem Alphabet (= Vorrat sprachbildender Zeichen = Menge terminaler Symbole) A,
  - einem Vokabular (= Vorrat sprachbeschreibender und sprachbildender Zeichen = Menge terminaler und nichtterminaler Zeichen) V,
  - einer Menge von Grammatikregeln $R = \{[u, v] : u \in (V^* \setminus A^*), v \in V^*\}$ und
  - einem Satzsymbol S mit $S \in (V \setminus A)$.

  Möglichkeiten der Syntaxbeschreibung sind

  - Aufzählung aller erlaubten Zeichenreihen (Sätze),
  - Aufzählung der Mengen A, V und R und Angabe des Satzsymbols S,
  - Syntaxdiagramme und
  - die Backus–Naur–Form.

- Semantik

  Sie beschreibt die Beziehung (Zuordnung) zwischen den sprachbildenden Symbolen und ihrer Bedeutung; sie ist eine Abbildung zwischen der „Symbolwelt“ der Sprache und einer realen Welt.

1.2.1 Maschinensprachen (1. Generation)

. . . sind interne Sprachen des Prozessors und sind

- nur übertragbar auf Maschinen mit gleichem Prozessor und gleich beschalteter Peripherie,
- schwer lesbar,
- wenig übersichtlich und
- fehleranfällig.

Programme dieser Sprachen haben einen sehr hohen Erstellungs– und Änderungsaufwand. Eine Anwendung gibt es heute bestenfalls noch bei Geräterechnern (, für die niemand einen Assembler baut . . .).

1.2.2 Maschinenorientierte Sprachen (2. Generation)

- Portabilität ähnlich Maschinensprachen (bis auf u.U. symbolische Adressen)
- Maschinenbefehle durch Mnemonics abgekürzt
- häufig wiederkehrende Befehlsfolgen können u.U. zu Macros zusammengefaßt werden
- hoher Erstellungs– und Änderungsaufwand
- Anwendungen:
  - Systemsoftware (Betriebssystem, Standardsoftware)
  - NC–Maschinen
  - laufzeit– und speicherplatzkritische Anwendungen
- Werkzeug zur Übersetzung: Assembler

1.2.3 Problemorientierte Sprachen (3. Generation)

- auf einen Anwendungsbereich ausgerichtet
- weitestgehend unabhängig vom Rechnertyp
- geringer Programmier– und Entwicklungsaufwand
- leicht erlernbar
- portabel
- weniger laufzeit– und speichereffizient als Assemblersprachen (schlechtere „Hardware–Ausnutzung“ als Preis für höhere Portabilität)
- Werkzeuge zur Übersetzung: Compiler, Interpreter
- Beispiele:
  - 1954: FORTRAN (**FOR**mula **TRAN**slation), math.-techn. Anwendungen
  - 1959: ALGOL (**ALGO**rithmic **L**anguage), math. Anwendungen
  - 1961: COBOL (**CO**mmon **B**usiness **O**riented **L**anguage), wirtschaftl. Anwendungen
  - 1963/64: BASIC (**B**eginners **A**ll purpose **S**ymbolic **I**nstruction **C**ode), für Ausbildungszwecke, sehr einfach zu lernen, an FORTRAN angelehnt
  - 1965: PL/1 (**P**rogramming **L**anguage # **1**), Kombination von FORTRAN und COBOL, sowohl für math.-techn. als auch für kommerziell–administrative Anwendungen
  - 1969: PASCAL, auf strukturierte Programmierung ausgerichtet, deshalb in der Ausbildung weit verbreitet
- **Normen** sollen die Portabilität sicherstellen. Deutsche Normen sind z.B. DIN 66026 für ALGOL, DIN 66027 für FORTRAN, DIN 66028 für COBOL, DIN 66255 für PL/1, DIN 66256 für PASCAL und DIN 66284 für BASIC.
- Weniger bekannte Sprachen der 3. Generation sind:
  - ADA, für rechnerintegrierte Systeme, militär. Anwendungen

- APC, besonders gut für Matrix–Operationen, graphische Anwendungen, Planung
- PEARL für Echtzeit– und Prozeßdatenverarbeitung
- RPG, eine Formularsprache zur Verarbeitung von Listen und Datein
- MODULA, eine Weiterentwicklung von PASCAL, modulorientiert, unterstützt Parallelverarbeitung
- C, „assemblernahe“ und trotzdem höhere Programmiersprache, unterstützt strukturierte Programmierung, starke Verbreitung durch UNIX
- FORTH für Robotersteuerungen

1.2.4 Datenorientierte Sprachen (4. Generation)

Aus vertriebspolitischen Gründen gibt es keine einheitliche Begriffswelt dafür. Es werden verschiedene Dinge darunter verstanden:

1. Endbenutzersprachen
   - hervorgegangen aus Datenbank–Abfragesprachen
   - sollen Endbenutzer (= „Nicht–Programmierer“ in die Lage versetzen, ihr Problem selbst zu formulieren
   - Beispiel: SQL (**S**tructured **Q**uery **L**anguage) für Datenbank–Abfrage und einfache Verarbeitungs–Fuktionen
2. Softwareentwicklungs–Werkzeuge
   - unterstützen die professionelle Entwicklung von Anwendungs–Software
   - Generierung von Quellcode (der 3. Generation) durch bloße Parametervorgabe[2]

1.2.5 Wissensorientierte Sprachen (5. Generation)

Die Sprachen der 1.-3. Generation sind **prozedural**er Natur, d.h. sie beschreiben das „Wie“ der Problemlösung durch eine Folge von Anweisungen. Sprachen der 5. Generation hingegen sind **deklarativ**er Natur, d.h. sie beschreiben lediglich das Problem, das „Was“, ohne Anweisungen zur Lösungsfindung zu geben.[3]

Etwa ein Ausschnitt aus einer Suchanfrage in einer Datenbank könnte auf **prozedural**e Weise wie folgt beschrieben werden:

1. *Nimm einen Mitarbeiter*
2. *Prüfe, ob weiblich*
3. *falls JA, auslesen*
4. *falls NEIN, nicht berücksichtigen*
5. *Nimm den nächsten Mitarbeiter*

Das gleiche Problem könnte man **deklarativ** etwa so beschreiben:

Finde alle weiblichen Mitarbeiter

Beispiele für Sprachen der 5. Generation sind:

[2] Diese (relativ alte) Idee wird heute unter dem Schlagwort „Komponententechnologie“ behandelt.
[3] Bei der 4. Generation streitet sich die Fachwelt, ob diese Sprachen prozedural oder deklarativ sind.

- LISP (**LIS**t **P**rocessing)
  - 1958 am MIT entwickelt
  - arbeitet ausschließlich mit Listen (das gilt sowohl für Daten als auch für Prozeduren)
  - die (nahezu) einzige Methode, Schleifen zu programmieren, ist die Rekursion – damit verlieren die Schleifen ihr „prozedurales Outfit“
- LOGO ist eine vereinfachte Version von LISP für Lernzwecke
- PROLOG (**PRO**gramming in **LOG**ic)
  - 1971 in Marseille entwickelt
  - Programm = Menge von Aussagen + Hypothese[4]
  - PROLOG–System = deduktive Hülle eines Schlussfolgerungsoperators

2 Mathematisch–logische Grundlagen

2.1 Einfache Aussagen

Gegeben sei eine Menge von Individuensymbolen und für jede natürliche Zahl n eine Menge n-stelliger Funktionssymbole und eine Menge n-stelliger Prädikatensymbole.

1. Jedes Individuensymbol ist ein variablenfreier **Term**.
2. Wenn $c_1, \ldots, c_n$ variablenfreie Terme sind und f ein n-stelliges Funktionssymbol ist, so ist $f(c_1, \ldots, c_n)$ ein variablenfreier Term.
3. Weitere variablenfreie Terme gibt es nicht.

Wenn $t_1, \ldots, t_n$ variablenfreie Terme sind und p ein n-stelliges Prädikatensymbol ist, so ist $p(t_1, \ldots, t_n)$ eine **einfache Aussage**.

2.2 Prädikate, Funktionen, Interpretation

Ein n-stelliges **Prädikat** bildet aus der Menge aller n-Tupel von Objekten eines Objektbereiches I eindeutig in die Menge der Wahrheitswerte ab: $I^n \longrightarrow \{wahr, falsch\}$.
Eine n-stellige **Funktion** bildet aus der Menge aller n-Tupel von Objekten eines Objektbereiches I eindeutig in den Objektbereich ab: $I^n \longrightarrow I$.
Eine **Interpretation** ist eine Abbildung aus der „Symbolwelt“ des PK1 in eine reale Welt:

Individuensymbole	$\longrightarrow$	*Objekte*
Prädikatensymbole	$\longrightarrow$	*Prädikate*
Funktionssymbole	$\longrightarrow$	*Funktionen*

[4] . . . von der man gern wissen möchte, ob sie aus der Aussagenmenge folgt.

2.3 Zusammengesetzte Aussagen

Wenn A eine Aussage ist, so ist auch $\neg A$ (die Negation von A) eine (**zusammengesetzte**) **Aussage**.
Wenn A_1 und A_2 Aussagen sind, so sind

- $(A_1 \wedge A_2)$ (die Konjunktion von A_1 und A_2),
- $(A_1 \vee A_2)$ (die Disjunktion von A_1 und A_2),
- $(A_1 \rightarrow A_2)$ (die Implikation von A_1 und A_2) und
- $(A_1 \leftrightarrow A_2)$ (die Äquivalenz von A_1 und A_2)

(zusammengesetzte) Aussagen.
Der Wahrheitswert zusammengesetzter Aussagen ist aus den Wahrheitswerten seiner Komponenten wie folgt ermittelbar:

A_1	A_2	$\neg A_1$	$A_1 \wedge A_2$	$A_1 \vee A_2$	$A_1 \rightarrow A_2$	$A_1 \leftrightarrow A_2$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

2.4 Variablen und Quantifizierungen

Wenn $A(c)$ eine Aussage ist und $A(X)$ aus $A(c)$ entsteht, indem c überall durch X ersetzt wird, so sind auch $\forall X A(X)$ und $\exists X A(X)$ Aussagen. $\forall$ heißt **Allquantor**, $\exists$ heißt **Existenzquantor** und X heißt (all- bzw. existenzquantifizierte) **Variable**.
Nicht quantifizierte Variablen heißen **freie Variablen**.
Für endliche Individuenbereiche $I = \{c_1, \ldots, c_n\}$ gilt:

$$\forall X A(X) \quad \text{ist äquivalent zu} \quad \bigwedge_{i=1}^{n} A(c_i)$$

$$\exists X A(X) \quad \text{ist äquivalent zu} \quad \bigvee_{i=1}^{n} A(c_i)$$

2.5 Terme und Ausdrücke

1. Jede Variable ist ein **Term**.
2. Jedes Individuensymbol ist ein Term.
3. Wenn $t_1, \ldots, t_n$ Terme sind und f ein n-stelliges Funktionssymbol ist, so ist $f(t_1, \ldots, t_n)$ ein (strukturierter) Term.
4. Weitere Terme gibt es nicht.

1. Wenn $t_1, \ldots, t_n$ Terme sind und p ein n-stelliges Prädikatensymbol ist, so ist $p(t_1, \ldots, t_n)$ ein (atomarer) **Ausdruck** (eine Atomformel).
2. Wenn A ein Ausdruck ist, so ist auch $\neg A$ ein Ausdruck.

3. Wenn A_1 und A_2 Ausdrücke sind, so sind auch $(A_1 \wedge A_2)$, $(A_1 \vee A_2)$, $(A_1 \to A_2)$ und $(A_1 \leftrightarrow A_2)$ Ausdrücke.

4. Wenn A ein Ausdruck und X eine Variable ist, so sind auch $\forall X A(X)$ und $\exists X A(X)$ Ausdrücke. A heißt Wirkungsbereich des Quantors von X.

5. Weitere Ausdrücke gibt es nicht.

Zusammenhang Ausdruck — Aussage

Ein Ausdruck ohne freie Variable heißt Aussage.

Vereinbarung zur Verkürzung der Notation von Ausdrücken

1. Außenklammern können weggelassen werden.

2. Die Stärke der Bindung von Quantoren und Junktoren ist wie folgt priorisiert: $\forall$, $\exists$ und $\neg$ binden am stärksten; danach folgen (in der angegebenen Reihenfolge) $\wedge$, $\vee$, $\to$ und $\leftrightarrow$.

3. Ketten von Konjunktionen oder Disjunktionen gelten als von links geklammert.

2.6 Allgemeingültigkeit, Kontradiktorizität und Äquivalenz von Aussagen

Eine Aussage A heißt **allgemeingültig** ($ag \quad A$), falls sie für jede Interpretation wahr ist.
Eine Aussage A heißt **kontradiktorisch** ($kt \quad A$), gdw. $ag \quad \neg A$.
Zwei Aussagen A_1 und A_2 heißen **äquivalent** ($A_1 \equiv A_2$), gdw. $ag(A_1 \leftrightarrow A_2)$.

Wichtige äquivalente Umformungen

(1)	$\neg\neg A$	$\equiv$	A
(2)	$\neg(A \wedge B)$	$\equiv$	$\neg A \vee \neg B$
(3)	$\neg(A \vee B)$	$\equiv$	$\neg A \wedge \neg B$
(4)	$\neg(A \to B)$	$\equiv$	$A \wedge \neg B$
(5)	$\neg(A \leftrightarrow B)$	$\equiv$	$(A \wedge \neg B) \vee (\neg A \wedge B)$
(6)	$\neg\forall X A(X)$	$\equiv$	$\exists X \neg A(X)$
(7)	$\neg\exists X A(X)$	$\equiv$	$\forall X \neg A(X)$
(8)	$\nabla X A(X) \circ B$	$\equiv$	$\nabla X(A(X) \circ B)$
		mit	$\nabla \in \{\forall, \exists\}$ und $\circ \in \{\wedge, \vee\}$
		und	wenn X nicht in B vorkommt
	Ausnahmen (von (8)):		
(8.1)	$\forall X A(X) \wedge \forall X B(X)$	$\equiv$	$\forall X(A(X) \wedge B(X))$
(8.2)	$\exists X A(X) \vee \exists X B(X)$	$\equiv$	$\exists X(A(X) \vee B(X))$
(9)	$A \to B$	$\equiv$	$\neg A \vee B$
(10)	$A \to \nabla X B(X)$	$\equiv$	$\nabla X(A \to B(X))$
(11)	$\forall X A(X) \to B$	$\equiv$	$\exists X(A(X) \to B)$
(12)	$\exists X A(X) \to B$	$\equiv$	$\forall X(A(X) \to B)$

$$A \wedge (B \vee C) \quad \equiv \quad (A \wedge B) \vee (A \wedge C) \tag{13}$$

$$A \vee (B \wedge C) \quad \equiv \quad (A \vee B) \wedge (A \vee C) \tag{14}$$

2.7 Folgern

Sei M eine Menge von Aussagen, A eine Aussage. A **folgt aus** M ($M \models A$), falls jede Interpretation, die zugleich alle Elemente von M wahr macht (jedes Modell von M), auch A wahr macht.
Für endliche Mengen von Aussagen $M = \{A_1, \ldots, A_n\}$ bedeutet das:
$M \models A$, gdw.

$$ag(\bigwedge_{i=1}^{n} A_i \rightarrow A)$$

bzw. (was wegen $\neg(\bigwedge_{i=1}^{n} A_i \rightarrow A) \equiv \bigwedge_{i=1}^{n} A_i \wedge \neg A$ dasselbe ist)

$$kt(\bigwedge_{i=1}^{n} A_i \wedge \neg A).$$

2.8 HORN - Klauseln

HORN-Klauseln sind Ausdrücke der Form

$$\forall X_1 \ldots X_n(\underbrace{A}_{Klauselkopf} \leftarrow \underbrace{\bigwedge_{i=1}^{m} A_i}_{Klauselkörper}),$$

wobei A und die A_i quantorfreie Atomformeln ohne freie Variablen sind.

Varianten / Spezialfälle

1. **Regeln** *(vollständige HORN-Klauseln)*

$$\forall X_1 \ldots \forall X_n(A(X_1, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{m} A_i(X_1, \ldots, X_n))$$

2. **Fakten** *(HORN-Klauseln mit leerem Klauselkörper)*

$$\forall X_1 \ldots \forall X_n(A(X_1, \ldots, X_n) \leftarrow true)$$

3. **Fragen** *(HORN-Klauseln mit leerem Klauselkopf)*

$$\forall X_1 \ldots \forall X_n(false \leftarrow \bigwedge_{i=1}^{m} A_i(X_1, \ldots, X_n))$$

4. **leere HORN-Klauseln**

$$false \leftarrow true$$

2.9 Resolutionsmethode und deren Hintereinanderanwendung

Sei $\{K_1, \ldots, K_n\}$ eine Menge von Fakten und Regeln (kurz: Klauseln),

$$H \equiv \bigwedge_{i=1}^{m} H_i$$

eine Frage (eine Hypothese).
Eine der Klauseln K_j sei

$$A \leftarrow \bigwedge_{k=1}^{p} B_k \quad ,$$

wobei A und die B_k Atomformeln sind und alle auftretenden Variablen bezüglich der ganzen Klausel allquantifiziert sind.
Es gebe Termeinsetzungen ϑ_1 und ϑ_2 in die Variablen von A und eines der H_i (etwa H_l mit $1 \leq l \leq m$), so dass $\vartheta_1(A) \equiv \vartheta_2(H_l)$.

$$M \equiv \bigwedge_{i=1}^{n} K_i \wedge \neg H$$

ist kontradiktorisch ($kt \quad M$), wenn M nach Ersetzen von H in M durch

$$(\bigwedge_{i=1}^{l-1} \vartheta_2(H_i)) \wedge (\bigwedge_{k=1}^{p} \vartheta_1(B_k)) \wedge (\bigwedge_{i=l+1}^{m} \vartheta_2(H_i))$$

noch immer kontradiktorisch ist.

Satz von ROBINSON

Eine Klausel„menge“ M ist kontradiktorisch ($kt \quad M$), gdw. durch wiederholte Anwendung der Resolutionsmethode in endlich vielen Schritten die Hypothese durch die leere Klausel (Symbol: $\square$) ersetzt werden kann.

Unifikation *(=„Finden von Termeinsetzungen ϑ_1 und ϑ_2“)*

Zwei **Atomformeln** $p_1(t_{11}, \ldots, t_{1n})$ und $p_2(t_{21}, \ldots, t_{2m})$ **sind unifizierbar**, gdw. sie

- die gleichen Prädikatensymbole aufweisen ($p_1 \equiv p_2$),
- die gleiche Stelligkeit aufweisen ($n = m$) und
- die Terme t_{1i} und t_{2i} jeweils miteinander unifizierbar sind.

Die **Unifizierbarkeit zweier Terme** t_1 und t_2 richtet sich nach deren Sorte:

1. t_1 **und** t_2 **sind Konstanten (Individuensymbole)**:
   Die Unifikation ist erfolgreich, gdw. t_1 und t_2 identisch sind ($t_1 = t_2$).
2. $t_1 = f_1(t_{11}, \ldots, t_{1n})$ **und** $t_2 = f_2(t_{21}, \ldots, t_{2m})$ **sind strukturierte Terme**:
   Die Unifikation ist erfolgreich, wenn
   - die Funktionssymbole identisch sind ($f_1 = f_2$),
   - die Stelligkeiten gleich ist ($n = m$) und

- die t_{1i} und t_{2i} jeweils miteinander unifizierbar sind.

3. t_1 **ist Variable und** t_2 **ist Konstante oder strukturierter Term**:
   Die Unifikation ist erfolgreich, wenn t_1 nicht in t_2 enthalten ist. t_1 wird durch t_2 ersetzt (t_1 wird *instantiiert*).

4. t_1 **und** t_2 **sind Variablen**:
   Die Unifikation ist erfolgreich. Die Variablen werden gleichgesetzt ($t_2 := t_1$ bzw. $t_1 := t_2$).

2.10 Literaturhinweise

Asser,G. *Einführung in die mathematische Logik. Teil 2: Prädikatenkalkül der ersten Stufe.* Leipzig: Teubner Verlagsgesellschaft, 1982

Bibel,W. *Deduktion – Automatisierung der Logik.* München: Oldenbourg, 1992

Genesereth,M.R.; Nilsson,N.J. *Logische Grundlagen der Künstlichen Intelligenz.* Braunschweig, Wiesbaden: Vieweg, 1989

Goltz,H.-J.; Herre,H. *Mathematische Grundlagen der logischen Programmierung.* In: Informatik Informationen Reporte. 3(1987)5

Schmitt *Theorie der Logischen Programmierung.* Berlin u.a.: Springer, 1992

3 Einführung in die Logische Programmierung mit PROLOG

3.1 Einordnung des logischen Programmierparadigmas

Programmierparadigmen:

- imperativ (prozedural)
  Beschreibung von Problemlösungsalgorithmen (WIE?)
  - „rein“ imperativ
    Beschreibung von auf Daten zugreifenden Algorithmen, z.B. in FORTRAN, ALGOL, PL/1, PASCAL, FORTH, C, MODULA-2, ...
  - objektorientiert
    Datenelemente und die sie verarbeitenden Algorithmen (Methoden) bilden ein Objekt einer Klassenhierarchie, z.B. in SMALLTALK, C++, TURBO-PASCAL 6.0, ...
- deskriptiv (deklarativ)
  „kalkülisierte“ Beschreibung von Problemen (WAS?)
  - funktional, *z.B. in LISP*
  - **logisch,** *z.B. in PROLOG*

Problembeschreibung

Notation von Aussagen über das Problemgebiet (den Diskursbereich*) mit Hilfe prädikatenlogischer Ausdrücke*

Programmabarbeitung

- durch mustergesteuerte Prozeduraufrufe
  Anhand eines „aktuellen Zustands“ (eines Ziels*) wird eine Prozedur gesucht, die das Problem durch „kleinere“ Teilprobleme ersetzt. Es wird systematisch eine Folge solcher Ersetzungsschritte generiert, bei der das „leere“ Ziel entsteht.*

- bei Bedarf auch durch explizit vorzugebende Steuerinformation
  Dieses Suchverfahren kann zielgerichtet beeinflußt werden.

3.2 Syntax

Syntax von Termen

Termsorte		Syntax	Beispiele
Konstante	Name	Zeichenfolge, beginnend mit Kleinbuchstaben, die Buchstaben, Ziffern und Unterstriche enthalten kann	`otto_1`, `rainer`
		beliebige Zeichenfolge, die in “ ... “ eingeschlossen ist	`\IST NAME\`, `\hofnarr@kanzleramt.de\`
		Zeichenfolge aus Sonderzeichen	`%&&$#`
	Zahl	Ziffernfolge, ggf. mit Vorzeichen, Dezimalpunkt und Exponentendarstellung	`3`, `-5`, `1001`, `3.6E-12`
Variable	allgemein	Zeichenfolge, beginnend mit einem Großbuchstaben oder einem Unterstrich, die Buchstaben, Ziffern und Unterstriche enthalten kann	`X`, `Eingabe`, `_alter`
	anonym	Unterstrich	`_`
strukturierter Term	allgemein	$< fkt\_symb > (< term >, \ldots, < term >)$	`hauptstadt(brd)`, `freund(frau(uwe))`
	Liste	leere Liste:	`[ ]`
		$[< term > \mid < restliste >]$	`[mueller|[mayer|_]]`
		$[< term >, < term >, \ldots, < term >]$	`[mueller, mayer, schulze]`

Syntax von Klauseln

Sorte	Syntax	Beispiele
Fakt	$< praed\_symb > (< term >, \ldots, < term >).$	`liefert(xy_ag,motor,opel).`

Regel	$< praed\_symb > (< term >, \ldots, < term >) :-$
	$< praed\_symb > (< term >, \ldots, < term >),$
	$\ldots\ ,$
	$< praed\_symb > (< term >, \ldots, < term >).$
		`konkurrenten(Fa1,Fa2) :-`
		`liefert(Fa1,Prod,_),`
		`liefert(Fa2,Prod,_).`

Frage	$?- < praed\_symb > (< term >, \ldots, < term >),$ $\ldots\ ,$ $< praed\_symb > (< term >, \ldots, < term >).$	`?- konkurrenten(ibm,Wer),` `liefert(Wer,_,ibm).`

BACKUS-NAUR – Form

$$< PROLOG - Programm > \quad ::= \quad < Wissensbasis >< Hypothese >$$

$$
\begin{array}{rcl}
< Wissensbasis > & ::= & < Klausel > | < Klausel >< Wissensbasis > \\
< Klausel > & ::= & < Fakt > | < Regel > \\
< Fakt > & ::= & < Atomformel > . \\
< Atomformel > & ::= & < Praedikatensymbol > (< Termfolge >) \\
< Praedikatensymbol > & ::= & < Name > \\
< Name > & ::= & < Kleinbuchstabe > | < Kleinbuchstabe >< Restname > | \\
 & & \text{“} < Zeichenfolge > \text{“} | < Sonderzeichenfolge > \\
< Restname > & ::= & < Kleinbuchstabe > | < Ziffer > | \_ | \\
 & & < Kleinbuchstabe >< Restname > | \\
 & & < Ziffer >< Restname > | \_ < Restname > \\
< Zeichenfolge > & ::= & < Zeichen > | < Zeichen >< Zeichenfolge > \\
< Zeichen > & ::= & < Buchstabe > | < Ziffer > | < Sonderzeichen > \\
< Buchstabe > & ::= & < Kleinbuchstabe > | < Großbuchstabe > \\
< Kleinbuchstabe > & ::= & \mathbf{a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z} \\
< Großbuchstanbe > & ::= & \mathbf{A\ |B\ |C\ |D\ |E\ |F\ |G\ |H\ |I\ |J\ |K\ |L\ |M\ |N\ |O\ |P\ |Q\ |R\ |} \\
 & & \mathbf{S\ |T\ |U\ |V\ |W\ |X\ |Y\ |Z} \\
< Ziffer > & ::= & \mathbf{0\ |1\ |2\ |3\ |4\ |5\ |6\ |7\ |8\ |9} \\
< Sonderzeichen > & ::= & +\ |\text{-}\ |*\ |/\ |\backslash|\hat{}\ |<|>|=\ |\text{'}\ |\tilde{}\ |\text{:}\ |.\ |\#\ |@\ |\$\ |\&\ |\%\ |\S \\
< Sonderzeichenfolge > & ::= & < Sonderzeichen > | \\
 & & < Sonderzeichen >< Sonderzeichenfolge > \\
< Termfolge > & ::= & \varepsilon | < Term > | < Term > , < Termfolge > \\
< Term > & ::= & < Konstante > | < Variable > | < strukturierterTerm > \\
< Konstante > & ::= & < Name > | < Zahl > \\
< Zahl > & ::= & < Ziffer > | < Ziffer >< Zahl > \\
< Variable > & ::= & < Großbuchstabe > | < Großbuchstabe >< Restvariable > | \_ | \\
 & & \_ < Restvariable > \\
< Restvariable > & ::= & < Buchstabe > | < Ziffer > | \_ | \\
 & & < Buchstabe >< Restvariable > | \\
 & & < Ziffer >< Restvariable > | \_ < Restvariable > \\
< strukturierterTerm > & ::= & < Funktionssymbol > (< Termfolge >) | < Liste > \\
< Funktionssymbol > & ::= & < Name > \\
< Liste > & ::= & [< Termfolge >] | [< Term > | < Liste >] \\
< Regel > & ::= & < Atomformel > \text{:-} < Atomformelfolge > . \\
< Atomformelfolge > & ::= & < Atomformel > | < Atomformel > , < Atomformelfolge > \\
< Hypothese > & ::= & \text{?-} < Atomformelfolge > .
\end{array}
$$

3.3 Prolog aus logischer Sicht

Die Menge aller zu einem Sachverhalt (einem Diskursbereich) formulierten Fakten und Regeln heißt **Wissensbasis**.

Prinzip

- Formulierung einer Menge $M = \{K_1, \ldots, K_n\}$ von Fakten und Regeln (kurz „Klauseln“), d.h. einer Wissensbasis
- Formulierung einer Hypothese (Frage, Ziel) $H \equiv H_1 \wedge \ldots \wedge H_n$ (die H_i heißen **Teilziele**)

- zu zeigen: $M \models H$, d.h. $kt(\bigwedge_{i=1}^{n} \wedge \neg H)$
- dies wird gezeigt durch die Resolutionsmethode und deren Hintereinanderanwendung nach ROBINSON

Suchalgorithmus für eine Folge von Resolutionsschritten

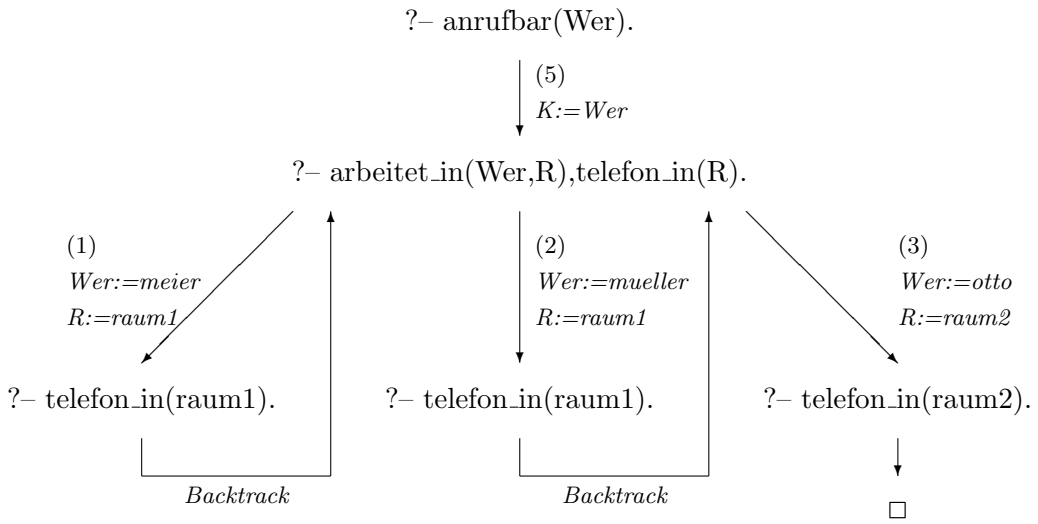
Tiefensuche mit Backtrack:

- Die Teilziele eines aktuellen Ziels werden **von links nach rechts** bearbeitet.
- Die Wissensbasis wird **von oben nach unten** nach einem mit dem 1. Teilziel unifizierbaren Klauselkopf durchsucht.
- Gibt es genau eine derartige Klausel, so wird das 1. Teilziel durch den Klauselkörper ersetzt (unter Einsetzen der Variablenersetzungen, d.h. des Unifikators).
- Gibt es mehrere derartige Klauseln, so wird ein sog. Entscheidungspunkt (place marker) eingerichtet, der momentane Bearbeitungszustand (akt. Ziel und Variablenbelegungen) im sog. Backtrack-Keller gekellert und zunächst die (in der Wissensbasis) am weitesten oben stehende Klausel angewandt, d.h. das 1. Teilziel durch deren Klauselkörper ersetzt.
- Gibt es keine derartige Klausel (mehr), so erfolgt ein Backtrack, d.h. der Algorithmus setzt zum letzten Entscheidungspunkt zurück und beginnt, ausgehend vom dort gekellerten Zustand, mit der nächsten alternativ anwendbaren Klausel.
- Das Verfahren bricht ab, wenn
  - als aktuelles Ziel die leere Klausel entsteht (Prolog's Antwort: „ja“ bzw. eine konstruktive Lösung) oder
  - das Ziel nicht leer ist, keine Klausel (mehr) anwendbar ist und kein Entscheidungspunkt (mehr) eingerichtet ist (Prolog's Antwort: „nein“).

Veranschaulichung des Suchalgorithmus' durch einen Suchbaum:
ODER-Baum der Abarbeitung, dessen Knoten das jeweils aktuelle Ziel symbolisieren und dessen gerichtete Kanten jeweils einen Resolutionsschritt symbolisieren. Die Kanten sind markiert

- mit der Nummer derjenigen Klausel, mit deren Kopf das 1. Teilziel unifiziert wurde und
- mit dem Unifikator, d.h. der Variablenersetzung.

Beispiel:
Wissensbasis:
(1) `arbeitet_in(meier,raum1).`
(2) `arbeitet_in(mueller,raum1).`
(3) `arbeitet_in(otto,raum2).`
(4) `telefon_in(raum2).`
(5) `anrufbar(K) :- arbeitet_in(K,R),telefon_in(R).`
Ziel: `?- anrufbar(Wer).`
Suchbaum:



Prolog's Antwort (konstruktive Lösung): „Wer := otto"

3.4 Literaturhinweise

3.4.1 Lehrbücher

Belli,F. *Einführung in die logische Programmierung mit Prolog.* Mannheim, Wien, Zürich: Bibliograph. Institut, 1986

Bothe,K.; Stojanow,S. *Praktische Prolog – Programmierung.* Berlin, München: Verlag Technik, 1991

Bratko,I. *Prolog.* Bonn: Addison Wesley, 1987

Clocksin,W.F.; Mellish,C.S. *Programming in Prolog.* Berlin, Heidelberg, New York: Springer, 1981,1987

Geske,U. *Programmieren mit Prolog.* Berlin: Akademie-Verlag, 1988

Kleine Büning,H.; Schmitgen,S. *Prolog.* Stuttgart:B.G.Teubner,1986

Knauf,R. *Logische Programmierung und Wissensbasierte Systeme – eine Einführung.* Aachen: Shaker-Verlag, ISBN 3-86111-310-4, 1993

3.4.2 Produkte und Anwendungen

Antoniou,G. *Turbo Prolog.* Düsseldorf: Data Becker, 1987

Böhringer,B.; Chiopris,C.; Futo,I. *Wissensbasierte Systeme mit Prolog.* Bonn: Addison-Wesley, 1988

Bradbury,A.; Woodward,R. *Turbo Prolog-Begleitbuch.* Maidenhead: McGraw-Hill, 1990

Dietrich,W. *Turbo Prolog.* Bonn: Addison-Wesley, 1988

Fahrion,R. *Wirtschaftsinformatik – Grundlagen und Anwendungen.* Heidelberg: Physika–Verlag, 1989

Gabriel,R. *Wissensbasierte Systeme in der betrieblichen Praxis.* Maidenhead: McGraw-Hill,1991 (Prolog für Wirtschaftsinformatik)

Grothaus,H.; Gust,H. *Turbo Prolog.* Würzburg: Vogel-Buchverlag,1987

Hanus,H. *Problemlösen mit Prolog.* Stuttgart: B.G. Teubner, 1987

Janson *Die Programmiersprache Turbo Prolog.* München:Franzis,1988

Kinnebrok,W. *Handbuch Turbo Prolog.* München: Oldenbourg, 1990

Kinnebrok,W. *Professionelles Programmieren mit Turbo Prolog.* München: Oldenbourg, 1988

Kinnebrok,W. *Turbo Prolog.* München: Oldenbourg, 1990

Lehner,Ch. *Prolog und Linguistik.* München: Oldenbourg, 1992

Schildt,M. *Professioneles Turbo Prolog.* Hamburg: McGraw-Hill,1988

Schnupp,P. *Prolog - Einführung in die Programmierpraxis.* München, Wien: Hanser, 1986 (IF/Prolog)

Schnupp,P.; Höß,K. *TerminalBuch Prolog.* München: Oldenbourg, 1989 (u.a. für Analyse u. Generierung formaler Sprachen)

Weiskamp,K.; Hengl,T. *KI-Programmierung mit Turbo Prolog.* Maidenhead: McGraw-Hill, 1989

4 Noch 'ne KI–Sprache: LISP

4.1 Einführung in LISP

LISP ist ein Kürzel für **LIS**t **P**rocessor; die vorwiegend mit ihr durchgeführte Tätigkeit ist die Verarbeitung von Listen. Wichtige Überlegungen, von denen man bei der Entwicklung von LISP ausging, entstammen der Algorithmentheorie. Aus der Algorithmentheorie weiß man, dass es bei Funktionen

$$f : \mathbb{N}^n \to \mathbb{N}$$

auch nicht berechenbare Funktionen gibt (was leicht mit Hilfe der sog. CANTOR'schen Diagonalmethode bewiesen werden kann) und dass die Klasse der berechenbaren Funktionen exakt durch die Klasse der partiell rekursiven Funktionen abgedeckt wird.[5] Eine Sprache wie LISP, bei der die „Verschachtelung" von Funktionen und die Rekursion das wesentliche Programmierprinzip bilden, sollte somit in der Lage sein, alles „Berechenbare" auch „formulierbar" zu machen.

Wichtigstes Sprachelement ist die **Funktion**. Funktionen werden (wie alle Programm– und Datenstrukturen in LISP) in Form von Listen notiert:

$$(OPERATOR \quad OPERAND_1 \quad OPERAND_2 \ldots)$$

Das erste Element dieser Liste ist der Name der Funktion und die folgenden Elemente sind die Argumente der Funktion.

Nachdem sich ein LISP–System mit einem Prompt gemeldet hat, wird eine sog. **Read–Eval–Print – Loop** aufgerufen, die zyklisch

[5] Was partiell rekursive Funktionen sind, ist vielleicht aus der Veranstaltung „Theoretische Informatik" bekannt: Es gibt einen Satz von Grundfunktionen (Nullfunktion, Projektion, Nachfolgerfunktion), die per Definition partiell rekursiv sind und die partielle Rekursivität komplizierterer Funktionen wird durch „ineinander Einsetzen" partiell–rekursiver Funktionen und/oder durch rekursiven Abbau eines Arguments gezeigt.

- LISP–Ausdrücke einliest,
- diese auswertet (i.allg. heißt das, den Funktionswert zu ermitteln) und
- das Ergebnis der Auswertung ausgibt.

Für LISP untypisch (und im „strengen“ LISP verboten) sind globale Variablen. Eine „wahrhaftig“ strukturierte Programmierung ist streng genommen nur gänzlich ohne globale Variable möglich. Nur so können Softwaremoduln beliebig konfiguriert werden, ohne dass andere als die im Funktionsaufruf angegebenen Schnittstellen beachtet werden müssen. Die Einhaltung der im Funktionsaufruf angegebenen Schnittstellen ist zudem syntaktisch (d.h. durch einen Compiler oder Interpreter) überprüfbar.

Um Funktionen aufeinander aufbauend verwenden zu können, muß LISP über eine Möglichkeit verfügen, Funktionen zu definieren, mit einem Namen zu belegen und über diesen Namen aufzurufen. Der sog. **Lambdakalkül** und die rekursive Darstellung von Funktionen sind zwei wichtige Hilfsmittel dafür.

Der Lambdakalkül ist nichts anderes als eine Vorschrift, wie Funktionen definiert werden. Er spiegelt sich in einer speziellen LISP–Struktur wider, mit der Funktionen definiert werden, mit einem Namen belegt werden und in der angegeben wird, welches die freien Variablen (Operanden) der Funktion sind.[6]

4.2 LISP–Ausdrücke

4.2.1 Syntax

Literale Atome

Literale Atome sind Folgen von Buchstaben und Ziffern, deren erstes Zeichen ein Buchstabe ist.

Die interne Repräsentation literaler Atome zeigt die Abbildung 1.

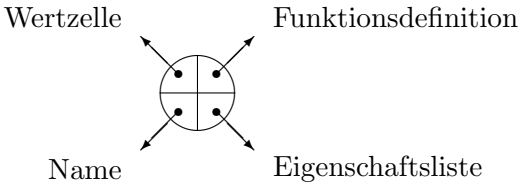


Abbildung 1: Interne Repräsentation eines literalen Atoms

Die Atome `NIL` und `T` (true) sind spezielle literale Atome, deren Bedeutung noch zu besprechen sein wird.

Numerische Atome

Numerische Atome sind ganze Zahlen und Gleitkommazahlen.

[6] Aus der Sicht traditioneller (prozeduraler) Programmiersprachen sind sämtliche Operanden Wertparameter; sie können und dürfen (aus den oben genannten Gründen) niemals durch den Funktionsaufruf überschrieben werden. Der einzige Referenzparameter (der hier eigentlich kein Parameter ist) ist der Funktionswert.

Symbolische Ausdrücke

1. Literale Atome sind symbolische Ausdrücke.
2. Numerische Atome (Zahlen) sind symbolische Ausdrücke.
3. Wenn X und Y symbolische Ausdrücke sind, so ist das gepunktete Paar (X . Y) ein symbolischer Ausdruck.
4. Weitere symbolische Ausdrücke gibt es nicht.

Die interne Repräsentation eines gepunkteten Paares zeigt die Abbildung 2.

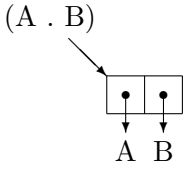


Abbildung 2: Interne Repräsentation eines gepunkteten Paares

Listen

1. `NIL` ist die „leere Liste“; sie kann auch als () notiert werden.
2. Gepunktete Paare der Form

   $(Element_1 \text{ . } (Element_2 \text{ . } (\cdots (Element_N \text{ . } \texttt{NIL}) \cdots)))$

   heißen **Liste** und können auch (vereinfacht) als

   $(Element_1 \quad Element_2 \quad \cdots \quad Element_N)$

   notiert werden.

Eine nichtleere Liste ist ein Spezialfall eines gepunkteten Paares; eine leere Liste ist ein Spezialfall eines literalen Atoms.

Die interne Repräsentation einiger Listen zeigt die Abbildung 3.

4.2.2 Auswertung von LISP–Ausdrücken

Auswerteregeln:

1. Der Wert eines numerischen Atoms (einer Zahl) ist die Zahl selbst.
2. Der Wert eines literalen Atoms ist das Objekt, welches bei der Auswertung des durch die Wertzelle referenzierten Ausdrucks entsteht.
3. Der Wert eines Ausdrucks

   $(OPERATOR \quad OPERAND_1 \quad \ldots \quad OPERAND_N)$

   ist derjenige Funktionswert, der entsteht, wenn man

   (a) die Operanden $OPERAND_1$, $\ldots OPERAND_N$ von links nach rechts (nach denselben Regeln) auswertet und dann

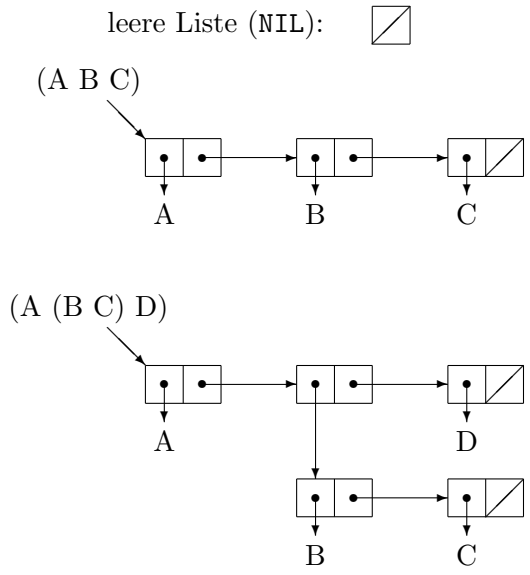


Abbildung 3: Interne Repräsentation einiger Listen

(b) auf die Ergebnisse $WERT_1, \ldots, WERT_N$ dieser Auswertung $OPERATOR$ anwendet.

4. Der Wert eines Ausdrucks

   (`SETQ` $OPERAND_1$ $OPERAND_2$)

   ist der Wert des Operanden $OPERAND_2$.

   Die Auswertung von `SETQ` („Setquote“) hat zudem den Nebeneffekt, daß der Wert von $OPERAND_2$ in die Wertzelle von $OPERAND_1$ geschrieben wird. $OPERAND_1$ muss ein literales Atom sein und wird nicht ausgewertet; $OPERAND_2$ kann ein beliebiger Ausdruck sein.

5. Der Wert des Ausdrucks

   (`QUOTE` $OPERAND$)

   ist gleich dem unausgewerteten Operanden $OPERAND$, d.h. seinem Namen bzw. des aus Namen gebildeten Ausdrucks.

4.3 Elementare LISP–Funktionen

4.3.1 Zugriffsfunktionen

Um auf einzelne Elemente einer Liste zugreifen zu können, gibt es in LISP zwei eingebaute Funktionen:

1. Der Wert von (`CAR` $LISTE$) ist das erste Element der Liste $LISTE$.

2. Der Wert von (`CDR` $LISTE$) ist die **Liste** der restlichen (außer dem ersten) Elemente der Liste $LISTE$ bzw. `NIL`, falls diese leer ist.

Erscheint das Argument von `CAR` als gequotete Liste, so liefert `CAR` den **Namen** des ersten Elements als Funktionswert; ansonsten den **Wert** des ersten Elements.

Erscheint das Argument von `CDR` als gequotete Liste, so liefert `CDR` die Liste der **Namen** der restlichen (außer dem ersten) Elemente als Funktionswert; ansonsten die Liste der **Werte** der restlichen Elemente.

Die interne Repräsentation von `CAR` und `CDR` durch Zeiger wird durch die Abbildung 4 veranschaulicht.

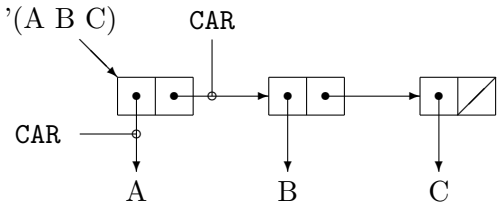


Abbildung 4: Interne Repräsentation von `CAR` und `CDR` einer Liste

Übliche Schreibweisen für Verschachtelungen von `CAR` und `CDR` sind u.a.:

`(CAAR LISTE)`	steht z.B. für	`(CAR (CAR LISTE))`
`(CDAR LISTE)`	steht z.B. für	`(CDR (CAR LISTE))`
`(CADR LISTE)`	steht z.B. für	`(CAR (CDR LISTE))`
`(CADAR LISTE)`	steht z.B. für	`(CAR (CDR (CAR LISTE)))`
`(CAADR LISTE)`	steht z.B. für	`(CAR (CAR (CDR LISTE)))`
`(CADAAR LISTE)`	steht z.B. für	`(CAR (CDR (CAR (CAR LISTE))))`

4.3.2 Konstruktionsfunktionen

Konstruktionsfunktionen dienen dazu, gepunktete Paare (bzw. als deren Spezialfall Listen) aus symbolischen Ausdrücken zu bilden:

1. `(CONS` $OPERAND_1$ $OPERAND_2$`)`
   bildet das gepunktete Paar ($OPERAND_1$. $OPERAND_2$).

2. `(LIST` $OPERAND_1$ $OPERAND_2$ ... $OPERAND_N$`)`
   bildet die Liste ($OPERAND_1$ $OPERAND_2$ ... $OPERAND_N$).

4.3.3 Typprädikate

Prädikate sind ein Spezialfall von Funktionen; sie haben einen Wertebereich mit nur zwei Elementen: `T` (für *true*, *wahr*) und `NIL` (für *false*, *falsch*). Typprädikate gestatten es, den Typ ihres Arguments zu erkennen:

1. Der Wert von `(ATOM` $OPERAND$`)` ist `T` (*true*), gdw. der Wert von $OPERAND$ ein (literales oder numerisches) Atom ist.

2. Der Wert von `(NUMBERP` $OPERAND$`)` ist `T` (*true*), gdw. der Wert von $OPERAND$ ein numerisches Atom (eine Zahl) ist.

3. Der Wert von `(LISTP` $OPERAND$`)` ist `T` (*true*), gdw. der Wert von $OPERAND$ eine nichtleere Liste ist.

4. Der Wert von `(NULL` $OPERAND$`)` ist `T` (*true*), gdw. der Wert von $OPERAND$ `NIL` ist.

Weitere Typprädikate testen Listen auf Gleichheit. Hierbei müssen jedoch

- Listen mit gleicher Struktur und
- Listen, die in ihrer rechnerinternen Repräsentation identisch sind

voneinander unterschieden werden. Die Funktionen zur Testung von Listen auf Gleichheit sind folgende:

1. Der Wert von `(EQUAL` $LISTE\_1$ $LISTE\_2$`)` ist `T` (*true*), gdw. $LISTE\_1$ und $LISTE\_2$ strukturell und wertmäßig gleich sind.
2. Der Wert von `(EQL` $LISTE\_1$ $LISTE\_2$`)` ist `T` (*true*), gdw. $LISTE\_1$ und $LISTE\_2$ physisch gleich sind.

4.4 Definition von Funktionen

1. Durch

   `(LAMBDA` $PARAMETERLISTE$ $AUSDRUCK$`)`

   wird die in $AUSDRUCK$ beschriebene Funktion auf die $PARAMETERLISTE$ angewandt.

   $AUSDRUCK$ kann ein LISP–Ausdruck oder eine Liste mehrerer (nacheinander auszuwertender) LISP–Ausdrücke sein. Der Wert des letzten ausgewerteten LISP–Ausdrucks wird als Funktionswert des LAMBDA–Ausdrucks zurückgegeben.

   Das Atom `LAMBDA` zeigt an, dass der gesamte LAMBDA–Ausdruck als Funktion der hinter ihm stehenden Argumente (die im LAMBDA–Ausdruck einen symbolischen Namen bekommen) zu interpretieren ist.

2. Mit

   `(DEFUN` $ATOM$ $PARAMETERLISTE$ $AUSDRUCK$`)`

   kann man einer Funktion einen Namen geben (der in dem Parameter $ATOM$ stehen muß), unter dem sie zukünftig aufrufbar ist. $AUSDRUCK$ beschreibt die Funktion und nimmt dabei auf die symbolischen Parameter der $PARAMETERLISTE$ bezug.

3. Soll der Name und die Definition einer Funktion erst durch die Auswertung eines LISP-Ausdrucks entstehen[7], so erfolgt der Funktionsaufruf mit

   `(FUNCALL` $ATOM$ $ARGUMENTE$`)`

   Der Wert von $ATOM$ ist hierbei ein kompletter LAMBDA–Ausdruck.

4. Der Aufruf einer Funktion über ihren Namen kann (auch) mit Hilfe von

   `(APPLY` $FUNKTIONSNAME$ $ARGUMENTE$`)`

   erfolgen, wobei (im Gegensatz zum Direktaufruf mit $FUNKTIONSNAME$ als Operator) $FUNKTIONSNAME$ auch ein (noch auszuwertender) Ausdruck sein darf.

[7] Dies ist z.B. dann der Fall, wenn diese Angaben in einer „dynamischen Wissensbasis“ stehen, die erst während des Programmlaufs eingelesen werden.

Durch solche Funktionen wie `FUNCALL` und `APPLY` ist eine Art der Programmierung möglich, die es in keiner anderen Sprache gibt: Funktionsnamen können auch Variablen sein, die

- in Falle von `FUNCALL` beliebig gesetzt werden können, so dass ein und dieselbe Bezeichnung nacheinander für verschiedene Funktionen verwendet werden kann oder
- im Falle von `APPLY` der Name der aufzurufenden Funktion durch eine Auswertung (also ggf. einen weiteren Funktionsaufruf) erst „errechnet“ wird.

4.5 Komplexere LISP–Funktionen

4.5.1 Verarbeitung von Listen

Einige der hier aufgelisteten Funktionen verändern auch ihre Argumente physisch; in diesem Falle ist das in der nachfolgenden Beschreibung besonders vermerkt.

1. Der Wert von

   `(APPEND` $LISTE\_1$ $LISTE\_2$`)`

   ist diejenige Liste, die sich durch das Anhängen der Liste $LISTE\_2$ an die Liste $LISTE\_1$ ergibt.

2. Der Wert von

   `(REVERSE` $LISTE$`)`

   ist diejenige Liste, die sich durch das Umkehren der Reihenfolge der Elemente aus der Liste $LISTE$ entsteht.

3. Der Wert von

   `(DELETE` $ELEMENT$ $LISTE$`)`

   ist diejenige Liste, die sich aus der Liste $LISTE$ ergibt, indem alle vorkommenden Elemente $ELEMENT$ aus ihr entfernt werden bzw. `NIL`, falls $ELEMENT$ nicht in $LISTE$ vorkommt.

   Diese Veränderung wird auch physisch ausgeführt, d.h. die interne Darstellung von $LISTE$ wird verändert.

4. Der Wert von

   `(MEMBER` $ELEMENT$ $LISTE$`)`

   ist diejenige Liste, die sich durch das Streichen des ersten vorkommenden Elements $ELEMENT$ aus der Liste $LISTE$ ergibt bzw. `NIL`, falls $ELEMENT$ nicht in der Liste vorkommt.

5. Der Wert von

   `(RPLACA` $LISTE$ $NEUES\_ELEMENT$`)`

   (replace car) ist diejenige Liste, die sich aus der Liste $LISTE$ ergibt, wenn deren erste Element durch das Element $NEUES\_ELEMENT$ ausgetauscht wird.

   Diese Veränderung wird auch physisch ausgeführt, d.h. die interne Darstellung von $LISTE$ wird verändert.

6. Der Wert von

   `(RPLACD` $LISTE$ $NEUE\_RESTLISTE$`)`

   (replace cdr) ist diejenige Liste, die sich aus der Liste $LISTE$ ergibt, wenn deren Restliste durch die Liste $NEUE\_RESTLISTE$ ersetzt wird.

   Diese Veränderung wird auch physisch ausgeführt, d.h. die interne Darstellung von $LISTE$ wird verändert.

Die Anwendung einer die physische Struktur einer Liste verändernden Funktion führt dazu, dass die gelöschten Elemente nicht mehr dereferenzierbar sind. Um das berlaufen des Speichers durch derartigen „Datenmüll“ zu verhindern, verfügen LISP–Systeme über eine dynamische Speicherplatzverwaltung, die regelmäßig eine sog. „garbage collection“ (Müllsammlung) durchführt, um diesen Speicherplatz wieder zurückzugewinnen.

4.5.2 Bedingte Anweisungen

Bedingte Anweisungen, die in traditionellen Programmiersprachen gewöhnlich mit IF– oder CASE– Anweisungen realisiert werden, werden in LISP durch die Funktion `COND` realisiert:

Auswerteregel 6

6. Zur Berechnung des Wertes des Ausdrucks

$$\begin{array}{ll}(\texttt{COND} & (PRAEDIKAT_1 \quad AUSDRUCK_{11} \ldots AUSDRUCK_{1N_1}) \\ & (PRAEDIKAT_2 \quad AUSDRUCK_{21} \ldots AUSDRUCK_{2N_2}) \\ & \ldots \\ & (PRAEDIKAT_M \quad AUSDRUCK_{M1} \ldots AUSDRUCK_{MN_M}))\end{array}$$

   werden die Prädikate $PRAEDIKAT_1 \ldots PRAEDIKAT_M$ in dieser Reihenfolge solange ausgewertet, bis ein Prädikat einen von `NIL` verschiedenen Wert liefert.

   Wird ein solches Prädikat gefunden, so werden die in derselben Liste stehenden Ausdrücke der Reihe nach ausgewertet.

   Der Wert des `COND`–Ausdrucks ist der Wert des letzten dabei ausgewerteten Ausdrucks. Steht hinter dem ersten „erfolgreichen“ Prädikat kein Ausdruck, so ist der Wert des Prädikats der Funktionswert. Ist keines der Prädikate „erfolgreich“, so ist `NIL` der Funktionswert des `COND`–Ausdrucks.

4.5.3 LET – Konstruktionen

Für die Definition von Funktionen ist es mitunter zweckmäßig, lokale Variable zu verwenden, um Mehrfachauswertungen von LISP–Ausdrücken zu vermeiden. Dies gelingt mit Hilfe der sog. `LET`–Konstruktion

$$\begin{array}{ll}\texttt{(LET ((}VARIABLE_1 & AUSDRUCK_1\texttt{)} \\ \texttt{(}VARIABLE_2 & AUSDRUCK_2\texttt{)} \\ \ldots & \\ \texttt{(}VARIABLE_N & AUSDRUCK_N\texttt{))} \\ LET\_KOERPER\ \texttt{)} & \end{array}$$

Als $VARIABLE_1 \ldots VARIABLE_N$ können beliebige literale Atome verwendet werden. Sie erhalten die durch die Auswertung der jeweils dahinter stehenden Ausdrücke ihre Werte und können im LET–Körper anstelle dieses Ausdrucks verwendet werden. Auf diese Weise werden die hinter den Variablen stehenden Ausdrücke nur ein Mal ausgewertet.

4.6 Ein Beispiel: Tiefe–zuerst–Suche in Graphen

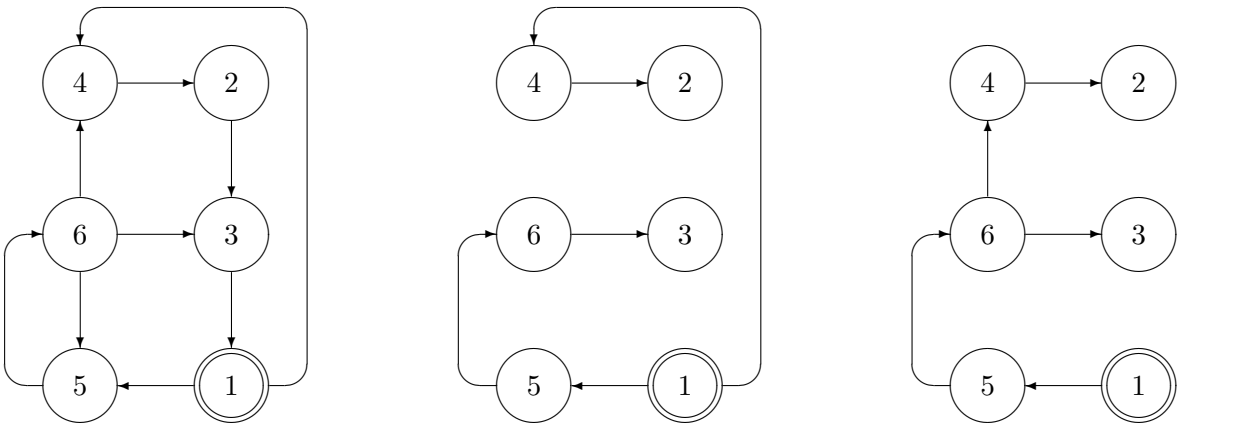
Typische Probleme, mit denen sich die KI beschäftigt, sind solche, die sich auf Suchprobleme in gerichteten Graphen zurückführen lassen, z.B. das folgende:

gegeben: - ein gerichteter Graph G
- ein Startknoten S innerhalb von G
gesucht: - ein Erreichbarkeitsbaum von G mit der Wurzel S

Was ein Erreichbarkeitsbaum ist und wie man einen solchen bestimmen kann, wird im folgenden erläutert:

> Es sind alle Knoten des Graphen G zu bestimmen, die auf Pfaden von einem gegebenen Startknoten S erreicht werden können.
>
> Ein Erreichbarkeitsbaum ist ein Teilgraph von G,
>
> - der alle erreichbaren Knoten enthält und
> - in dem es von S zu jedem (anderen) erreichbaren Knoten genau einen Pfad gibt.

Die Abbildung 5 zeigt einen Beispielgraphen und zwei verschiedene Erreichbarkeitsbäume dieses Graphen vom Startknoten 1.



Beispielgraph *ein Erreichbarkeitsbaum* *noch ein Erreichbarkeitsbaum*

Abbildung 5: Ein gerichteter Graph und Erreichbarkeitsbäume vom Startknoten 1

Eine mögliche Repräsentation des Beispielgraphen ist eine Liste von Listen, wobei jedes Element für eine Kante steht:

```
(SETQ BSPGRAPH ((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)(6 4)))
```

Ein wichtiger Schritt bei der Bestimmung eines Erreichbarkeitsbaumes ist die Bestimmung der ersten (bzw. nächsten) Kante des Graphen, die von einem gegebenen Knoten ausgeht:

```
(DEFUN NAECHSTE_KANTE (GRAPH STARTKNOTEN)
       (COND ((NULL GRAPH) NIL)
             ((EQUAL STARTKNOTEN (CAAR GRAPH)) GRAPH)
             (T (NAECHSTE_KANTE (CDR GRAPH) STARTKNOTEN))))
```

Die Funktion `NAECHSTE_KANTE`

- gibt `NIL` zurück, falls der Graph leer ist,
- gibt den gesamten Graphen zurück, falls gleich die erste in der Liste stehende Kante vom gegebenen Knoten ausgeht und
- ruft sich selber wieder auf mit dem um die erste Kante reduzierten Graphen in allen anderen Fällen, was zum Funktionswert
  - `NIL` führt, wenn keine Kante im Graphen vom gegebenen Knoten ausgeht bzw.
  - demjenigen Teilgraphen führt, der mit der ersten gefundenen vom Startknoten ausgehenden Kante beginnt.

Für eine so gefundene Kante (‚die die erste Kante des durch `NAECHSTE_KANTE` gelieferten Teilgraphen ist,) muß nun überprüft werden, ob sie schon in einem (bis dahin entstandenen) Erreichbarkeitsbaum eingetragen ist. Der Erreichbarkeitsbaum wird fast genauso repräsentiert wie der Graph; es wird jedoch noch eine zusätzliche (dummy-) Kante `('ANF` *WURZELKNOTEN*`)` darin aufgenommen, die die Wurzel des Baumes kennzeichnet. Auf diese Weise wird sichergestellt, dass jeder im Erreichbarkeitsbaum enthaltene Knoten auch Endknoten irgendeiner Kante ist. Das Suchen eines gegebenen Knotens im Baum wird dadurch recht einfach:

```
(DEFUN ENTHALTEN (KNOTEN BAUM)
       (COND ((NULL BAUM) NIL)
             ((EQUAL KNOTEN (CADAR BAUM)) T)
             (T (ENTHALTEN KNOTEN (CDR BAUM)))))
```

Ein in den Erreichbarkeitsbaum aufzunehmende neue Kante kann nun mit Hilfe der Funktionen `NAECHSTE_KANTE` und `ENTHALTEN` gesucht werden:

```
(DEFUN NEUE_KANTE (GRAPH STARTKNOTEN BAUM)
       (COND ((NULL GRAPH) NIL)
             (T ( LET ((RESTGRAPH (NAECHSTE_KANTE GRAPH STARTKNOTEN)))
                  (COND ((ENTHALTEN (CADAR RESTGRAPH) BAUM)
                         (NEUE_KANTE (CDR RESTGRAPH) STARTKNOTEN BAUM)
                        (T RESTGRAPH))))))
```

Die Funktion `NEUE_KANTE` sucht in `GRAPH` die erste Kante, die mit `STARTKNOTEN` beginnt und noch nicht in `BAUM` enthalten ist. Für den Fall, dass eine aufzunehmende Kante gefunden wurde, liefert sie den Restgraphen (ab der aufgenommenen Kante) als Funktionswert; anderenfalls `NIL`.

Um eine Tiefe–zuerst–Strategie für die Konstruktion eines Erreichbarkeitsbaumes zu realisieren, muss ein Stack angelegt werden, in dem verbleibende mögliche Lösungsschritte abgelegt werden, wenn mehrere Schritte möglich sind. In diesem Stack werden „Restgraphen“ abgelegt[8],

[8] Natürlich werden nicht die „Restgraphen“ selbst, sondern Zeiger darauf abgelegt, so dass jedes Element des Stacks nur 2 Zeiger (einen auf den „Restgraphen“ und einen auf das nächste Stackelement) beherbergt.

in denen weiter zu suchen ist, falls die Suche nach weiteren Lösungsschritten im aktuellen Zustand fehlschlägt.

Eine Funktion `SUCHSCHRITT`, die in einem (verbleibenden) Teilgraphen ab einem Startknoten eine in einen gegebenen Baum aufzunehmende Kante sucht und dabei

- rekursiv im restlichen Teilgraphen weitersucht, falls die erste Kante des Teilgraphen nicht aufgenommen werden kann (etwa weil sie schon drin ist) bzw.
- auf die im Stack verbleibenden Lösungsvarianten zugreift, falls keine im Teilgraphen enthaltene Kante in den Baum aufgenommen werden kann

kann wie folgt definiert werden:

```
(DEFUN SUCHSCHRITT (GRAPH TEILGRAPH STARTKNOTEN STACK BAUM)
       (LET ((RESTGRAPH (NEUE_KANTE TEILGRAPH STARTKNOTEN BAUM)))
            (COND ((NULL RESTGRAPH)
                        (COND ((NULL STACK) BAUM)
                              (T (SUCHSCHRITT
                                    GRAPH
                                    (CDAR STACK)
                                    (CADAAR STACK)
                                    (CDR STACK)
                                    BAUM))))
                  (T (SUCHSCHRITT
                        GRAPH
                        GRAPH
                        (CADAR RESTGRAPH)
                        (CONS RESTGRAPH STACK)
                        (CONS (CAR RESTGRAPH) BAUM))))))
```

Um einen Erreichbarkeitsbaum zu konstruieren, muss diese Funktion mit der initialen Argumentbelegung

`GRAPH` = gegebener Graph
`TEILGRAPH` = gegebener Graph
`STARTKNOTEN` = Wurzelknoten des zu konstruierenden Baumes (= WURZ)
`STACK` = `NIL`
`BAUM` = `'((ANF` WURZ`))`

aufgerufen werden. Um diesen Aufruf zu vereinfachen, kann man sich sich noch eine Funktion `ERREICHBARKEITSBAUM` definieren:

```
(DEFUN ERREICHBARKEITSBAUM (GRAPH WURZEL)
       (SUCHSCHRITT GRAPH GRAPH WURZEL NIL (LIST (LIST 'ANF WURZEL))))
```

Um die Arbeitsweise der Funktion `SUCHSCHRITT` zu verstehen, empfielt es sich, sie für dieses Aufrufbeispiel Schritt für Schritt nachzuvollziehen und für jeden rekursiven Aufruf die aktuellen Aufrufparameter zu notieren und den sich ergebenden Restgraphen `RESTGRAPH` zu notieren.

Und so sieht der Aufruf für den in Abbildung 5 gezeigten Beispielgraphen aus, bei dem als Lösung der über „*noch ein Erreichbarkeitsbaum*“ stehende Graph entsteht:

```
MeinLisp>(SETQ BSPGRAPH ((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)
                                                        (6 4)))
((2 3)(1 5)(3 1)(1 4)(5 6)(4 2)(6 5)(6 3)(6 4)))
MeinLisp> (ERREICHBARKEITSBAUM BSPGRAPH 1)
((4 2)(6 4)(6 3)(5 6)(1 5)(ANF 1))
```

Zum Vergleich sei hier ein PROLOG–Programm angegeben, welches dasselbe Problem nach exakt der gleichen Strategie löst. Die Namen der Funktionen treten hier als Prädikatennamen auf und der Funktionswert

- dokumentiert sich bei dem Prädikat `enthalten` im Erfolg (das entspricht dem Funktionswert *true* bzw. `T`) bzw. Fehlschlagen (das entspricht dem Funktionswert $false$ bzw `NIL`) des Prädikats
- tritt bei allen anderen Prädikaten als letztes Argument, welches dann (einziger) Ausgabeparameter ist, auf.[9]

```
naechste_kante([[S,Z]|RestGr],S,[[S,Z]|RestGr]) :- !.
naechste_kante([_|RestGr],S,Gr) :- naechste_kante(RestGr,S,Gr).

enthalten(K,[[_,K]|_]) :- !.
enthalten(K,[_|R]) :- enthalten(K,R).

neue_kante(Gr,S,Baum,NeueKante) :-
        naechste_kante(Gr,S,RestGr),
        knoten_ausw(RestGr,S,Baum,NeueKante).

knoten_ausw([[S,Z]|Rest],S,Baum,NeueKante) :-
        enthalten(Z,Baum), !,
        neue_kante(Rest,S,Baum,NeueKante).
knoten_ausw(RestGr,_,_,_,RestGr).

suchschritt(Gr,TeilGr,S,Stack,Baum,NBaum) :-
        neue_kante(TeilGr,S,Baum,RestGr),
        restgr_ausw(RestGr,Gr,TeilGr,S,Stack,Baum,NBaum).

restgr_ausw([ ],_,_,_,[ ],Baum,Baum) :- !.
restgr_ausw([ ],Gr,TeilGr,S,[[[S,Z]|R]|Rest],Baum,NBaum) :- !,
        suchschritt(Gr,R,Z,Rest,Baum,NBaum).
restgr_ausw([[S,Z]|Rest],Gr,TeilGr,S,St,Baum,NBaum) :-
        suchschritt(Gr,Gr,Z,[[[S,Z]|Rest]|St],[[S,Z]|Baum],NBaum).

erreichbarkeitsbaum(Gr,Wurz,Baum) :-
        suchschritt(Gr,Gr,Wurz,[ ],[[anf,Wurz]],Baum).
```

[9] Bezüglich der Ausdrucksfähigkeit der „Wissensdarstellung“ kann PROLOG

- einerseits als Spezialfall von LISP aufgefaßt werden, da es nur zwei „Funktionswerte“ bei der Auswertung von Prädikaten kennt (*true* und $false$);
- andererseits kann es auch als Verallgemeinerung von LISP aufgefaßt werden, da jedes Argument auch Rückgabeparameter sein kann, so das durch ein Prädikat (bei erfolgreicher Auswertung, d.h. mit *true* als Ergebnis) auch mehrere Werte (und nicht nur **der** Funktionswert) „errechnet“ werden können.

Bezüglich der „Wissensverarbeitung“ ist PROLOG spezieller, denn es hat einen (nur wenig manipulierbaren) eingebauten Verarbeitungsmechanismus, der jedoch – und das wiederum ist ein Vorteil – nicht mehr explizit programmiert werden braucht.

Der Aufruf dieses Programms erfolgt in PROLOG durch

```
?- erreichbarkeitsbaum([[2,3],[1,5],[3,1],[1,4],[5,6],[4,2],
                                              [6,5],[6,3],[6,4]],1,X).
X = [[4,2],[6,4],[6,3],[5,6],[1,5],[anf 1]]
```

4.7 Literaturhinweise

Görz, Günther (Hrsg.) *Einführung in die künstliche Intelligenz.* Bonn u.a.: Addison Wesley, 1993

Lunze, Jan; Schwarz, Wolfgang *Künstliche Intelligenz – Einführung und technische Anwendungen.* Berlin: Verlag Technik, 1990

5 Weitere Teigebiete der KI im Überblick

5.1 Erweiterungen des Prädikatenkalküls der ersten Stufe

Der PK1 ist die ausdrucksfähigste Wissensdarstellung, über welcher man noch vollständig inferieren kann. Zwar gibt es Erweiterungen des PK1 mit höherer Ausdrucksfähigkeit (etwa Prädikatenkalküle höherer Ordnung, bei denen z.B. auch Prädikatensymbole Gegenstand der Quantifizierung durch $\forall$ oder $\exists$ sein können); es gilt jedoch:

1. Ein PKn ist für $n > 1$ nicht mehr vollständig.
2. Für jede widerspruchsfreie Erweiterung des PK1 gilt:
   - Die Erweiterung ist äquivalent zum PK1, d.h. es handelt sich nicht um eine „echte“ Erweiterung oder
   - die Erweiterung macht die Logik unvollständig.

Trotz alledem werden in der praktischen Wissensverarbeitung einige „unechte“ Erweiterungen benutzt. Obwohl diese die Ausdrucksfähigkeit nicht verbessern, bringen sie in der Praxis dadurch ein Gewinn, daß sie ein Stück mehr „Semantik“ der prädikatenlogisch beschriebenen „Welt“ syntaktisch kontrollierbar machen.

5.1.1 Prädikatenlogik mit Gleichheit

Diese Erweiterung besteht in der Einführung eines speziellen (infix notierten) Prädikats $\equiv/2$, welches als (semantische) Gleichheit zweier (syntaktisch verschiedener) Elemente des Individuenbereiches interpretiert wird.

Die Gleichheit von Termen impliziert auch die Gleichheit derjenigen Strukturen, in denen die als „gleich“ definierte Terme vorkommen und die ansonsten (sogar) syntaktisch gleich sind:

1. $t \equiv t$ gilt für jeden Term t.
2. Für alle n-stelligen Funktionssymbole f und alle Terme t_i und t'_i gilt:

$$(f(t_1, \ldots, t_n) \equiv f(t'_1, \ldots, t'_n)) \quad \text{, wenn} \quad (t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n)$$

3. Für alle n-stelligen Prädikatensymbole p[10] und alle Terme t_i und t'_i gilt:

$$(p(t_1, \ldots, t_n) \rightarrow p(t'_1, \ldots, t'_n)) \quad \text{, wenn} \quad (t_1 \equiv t'_1 \wedge \ldots \wedge t_n \equiv t'_n)$$

[10] Für $n = 2$ kann p auch das Gleichheitsprädikat selbst sein.

Die semantische Gleichheit syntaktisch verschiedener Terme muss dann auch bei der Inferenz berücksichtigt werden. Es mu die Mglichkeit geschaffen werden, Klauseln auch dann anzuwenden, wenn die entsprechenden Terme zwar nicht miteinander unifizierbar sind, wohl aber nach der Ersetzung derselben durch (semantisch) gleiche Terme die Unifizierbarkeit gegeben ist. Diese zusätzliche Sorte von Ableitungsschritten wird durch die *Paramodulationsregel* definiert, die für die Verarbeitung von HORN–Klauseln wie folgt aussieht:

Sei $\{K_1, \ldots, K_n\}$ eine Menge von Fakten und Regeln (kurz: Klauseln),

$$H \equiv \bigwedge_{i=1}^{m} H_i$$

eine Frage (eine Hypothese).

Eine der Klauseln K_j sei

$$(s \equiv t) \leftarrow \bigwedge_{k=1}^{p} B_k \quad ,$$

wobei die B_k Atomformeln sind und alle auftretenden Variablen bezüglich der ganzen Klausel allquantifiziert sind.

Es gebe Termeinsetzungen ϑ_1 und ϑ_2 in die Variablen von s und einem in einem der H_i (etwa H_l mit $1 \leq l \leq m$) vorkommenden Term r, so daß $\vartheta_1(s) \equiv \vartheta_2(r)$.

$$M \equiv \bigwedge_{i=1}^{n} K_i \wedge \neg H$$

ist kontradiktorisch ($kt\ M$), wenn M nach Ersetzen von H in M durch

$$(\bigwedge_{i=1}^{l-1} \vartheta_2(H_i)) \wedge \vartheta_2(H'_l) \wedge (\bigwedge_{k=1}^{p} \vartheta_1(B_k)) \wedge (\bigwedge_{i=l+1}^{m} \vartheta_2(H_i))$$

noch immer kontradiktorisch ist.

H'_l entsteht hierbei aus H_l durch Ersetzen von r durch t.

Desweiteren muss bei der Prädikatenlogik mit Gleichheit zur vollständigen Resolution zu jeder Wissensbasis die Klausel

$$\forall X(X \equiv X)$$

hinzugefügt werden.

5.1.2 Prädikatenlogik mit Sorten

Jedem Term wird eine Sorte oder (im Falle strukturierter Terme) ein Tupel von Sorten zugeordnet:

1. Variablen und Konstanten haben eine Sorte S, die als Index mitgeführt wird, z.B. X_S, c_S usw.

2. Jedem n-stelligen Funktionssymbol f ist ein $(n+1)$-Tupel von Sorten $[S_1, \ldots, S_n, S_{n+1}]$ zugeordnet. Es werden nur Terme der Gestalt $f(t_1, \ldots, t_n)$ zugelassen, bei denen t_i von der Sorte S_i ist; die Sorte des gesamten strukturierten Terms $f(t_1, \ldots, t_n)$ ist dann S_{n+1}.
3. Jedem n-stelligen Prädikat p ist ein n-Tupel $[S_1, \ldots, S_n]$ von Sorten zugeordnet und es sind nur Atomformeln $p(t_1, \ldots, t_n)$ zugelassen, bei denen t_i von der Sorte S_i ist.

Daß Sorten keine *echte* Erweiterung sind, wird dadurch offensichtlich, dass man für jede Sorte S ein 1-stelliges Prädikat p_S einführen kann und auf diese Weise dasselbe auch ohne Sorten ausdrücken kann. $p_S(X)$ bildet X auf wahr ab, gdw. X von der Sorte S ist. Dadurch wird z.B.

$$\forall X_S \Phi(X) \qquad \text{zu} \qquad \forall X(p_S(X) \rightarrow \Phi(X))$$

und

$$\exists X_S \Phi(X) \qquad \text{zu} \qquad \exists X(p_S(X) \wedge \Phi(X))$$

Bei HORN–Klauseln bedeutet dies eine Erweiterung des Klauselkörpers um die entsprechenden 1-stelligen Sortenprädikate.

5.2 Inferenzmethoden

Grundlage für das Inferieren ist **Metawissen**, z.B.

1. Wissen über Verallgemeinerungsmöglichkeiten
2. Wissen über Möglichkeiten des analogen Schließens
3. Wissen über Integritätsbedingungen einer Wissensbasis
4. Wissen über Eigenschaften von Aussagen
5. Wissen über Verfahren des sukzessiven Ableitens zur Simulation des Folgerns

Noch eine Stufe höher (die man vielleicht *Meta–Metawissen* bezeichnen könnte) könnte man das für das Ableiten nötige Wissen über

- Eigenschaften von Ableitungsverfahren wie z.B. Vollständigkeit und Korrektheit sowie
- das Verhältnis zwischen Folgern und Ableiten

einordnen.

Vollständig ist ein Ableitungsverfahren genau dann, wenn alle Aussagen H, die aus einer Menge von Aussagen $\{A_1, \ldots, A_n\}$ *folgen*, nach diesem Ableitungsverfahren auch *ableitbar* sind.

Korrekt ist ein Ableitungsverfahren genau dann, wenn alle Aussagen H, die aus einer Menge von Aussagen $\{A_1, \ldots, A_n\}$ *ableitbar* sind, aus dieser auch *folgen*.

Das Verhältnis zwischen Folgern und Ableiten läßt sich demnach wie folgt formulieren: $\{A_1, \ldots, A_n\} \models H$, wenn [*gdw.*] durch ein korrektes [*und vollständiges*] Ableitungsverfahren gezeigt werden kann , daß $\{A_1, \ldots, A_n\} \vdash H$.

5.2.1 Deduktion

Das Ziel der Deduktion ist es, genau das **Folgern** zu simulieren.

Hülleneigenschaften der Folgerungsrelation

Sei A die Menge aller (denkbaren) Aussagen; $P(A)$ die Potenzmenge davon, d.h. die Menge aller Teilmengen von A. Der Folgerungsbegriff definiert eine Relation Fl zwischen Elementen aus $P(A)$. Zur Relation Fl gehören alle Paare $[M, Fl(M)]$ mit $M, Fl(M) \in P(A)$, bei denen $Fl(M)$ die Menge der aus M folgerbaren Aussagen ist:

$$Fl(M) := \{H | M \models H\}$$

Es gelten folgende Sätze:

1. **Satz der Einbettung**: $M \subseteq Fl(M)$
   Jede in M enthaltene Aussage folgt auch aus M.

2. **Satz der Monotonie**: $M_1 \subseteq M_2 \longrightarrow Fl(M_1) \subseteq Fl(M_2)$
   Wenn eine Menge M_1 Teilmenge einer Menge M_2 ist, dann ist auch die Menge der aus M_1 folgerbaren Aussagen eine Teilmenge der aus M_2 folgerbaren Aussagen.

3. **Satz der Abgeschlossenheit**: $Fl(Fl(M)) \subseteq Fl(M)$
   Hat man für eine gegebene Menge M von Aussagen $Fl(M)$ gebildet, so führt das nochmalige Bilden der daraus folgerbaren Aussagen nicht zu einer Erweiterung gegenüber $Fl(M)$.

4. **Endlichkeitssatz**:
   $M \models H \longrightarrow \exists M^*((M^* \subseteq M) \wedge (M^* \models H) \wedge (card(M^*) < \infty))$
   Wenn $M \models H$, so gibt es stets eine endliche Teilmenge M^* von M, so daß $M^* \models H$.

5. **Ableitungstheorem**: $(M \models (H_1 \rightarrow H_2)) \longrightarrow (M \cup \{H_1\} \models H_2)$
   Wenn $M \models (H_1 \rightarrow H_2)$, dann $M \cup \{H_1\} \models H_2$.

6. **Deduktionstheorem**: $(M \cup \{H_1\} \models H_2) \longrightarrow (M \models (H_1 \rightarrow H_2))$
   Wenn $M \cup \{H_1\} \models H_2$, dann $M \models (H_1 \rightarrow H_2)$.

7. **Unvollständigkeitssatz** (zugunsten der Verständlichkeit etwas „schlampig“ formuliert):

   Für jede widerspruchsfreie Menge M von Aussagen gilt: Es gibt eine Aussage H derart, daß weder $M \models H$ noch $M \models \neg H$.

Grundlage für die Realisierbarkeit korrekter und vollständiger Ableitungsverfahren „$\vdash$“ ist der GÖDELsche **Vollständigkeitssatz**:

> Es gibt einen Algorithmus, der die aus einer gegebenen Menge von Aussagen folgerbaren Aussagen aufzählen kann.

Wenn es diesen gibt, muß es natürlich erst recht ein algorithmisierbares Verfahren geben, welches von einer gegebenen Hypothese H entscheidet, ob sie aus einer Menge von Aussagen M folgt. Zwei derartige Verfahren sind

- die Resolutionsmethode und deren Hintereinanderanwendung nach ROBINSON „$\vdash_*$“ und
- das natürliche Schließen nach GENTZEN „$\vdash_{NI}$“.

Die Resolutionsmethode und deren Hintereinanderanwendung nach ROBINSON „$\vdash_*$“

Wie sich dieses Verfahren für HORN–Klauseln darstellt, dürfte noch in Erinnerung sein:

Sei $\{K_1, \ldots, K_n\}$ eine Menge von Fakten und Regeln,

$$H \equiv \bigwedge_{i=1}^{m} H_i$$

eine Frage (eine Hypothese).

Eine der Klauseln K_j sei

$$A \leftarrow \bigwedge_{k=1}^{p} B_k \quad ,$$

wobei A und die B_k Atomformeln sind und alle auftretenden Variablen bezüglich der ganzen Klausel allquantifiziert sind.

Es gebe Termeinsetzungen (Substitutionen) ϑ_1 und ϑ_2 in die Variablen von A und eines der H_i (etwa H_l mit $1 \leq l \leq m$), so daß $\vartheta_1(A) \equiv \vartheta_2(H_l)$.

$$M \equiv \bigwedge_{i=1}^{n} K_i \wedge \neg H$$

ist kontradiktorisch ($kt \quad M$), wenn M nach Ersetzen von H in M durch

$$\underbrace{(\bigwedge_{1=1}^{l-1} \vartheta_2(H_i))}_{\text{die ersten } l-1 \text{ Teilhypothesen}} \wedge \underbrace{(\bigwedge_{k=1}^{p} \vartheta_1(B_k))}_{\text{Einsetzen des Körpers der Klausel statt } H_1} \wedge \underbrace{(\bigwedge_{i=l+1}^{m} \vartheta_2(H_i))}_{\text{restliche Teilhypothesen}}$$

noch immer kontradiktorisch ist.

Einige Begriffe hieraus sollen jetzt jedoch etwas genauer behandelt werden.

Substitution:

Eine **Substitution** ϑ ist eine Abbildung von einer endlichen Menge X von Variablen in die Menge der Terme. Sie wird i.allg. notiert als Menge von Paaren:

$$\vartheta = \{[x,t] | x \in X, t = \vartheta(x)\}$$

Substitution in strukturierten Termen:

$$\vartheta(f(Y,Z)) = f(\vartheta(Y), \vartheta(Z))$$

Hintereinanderausführung von Substitutionen „$\circ$“:

$$\sigma \circ \vartheta(t) = \sigma(\vartheta(t))$$

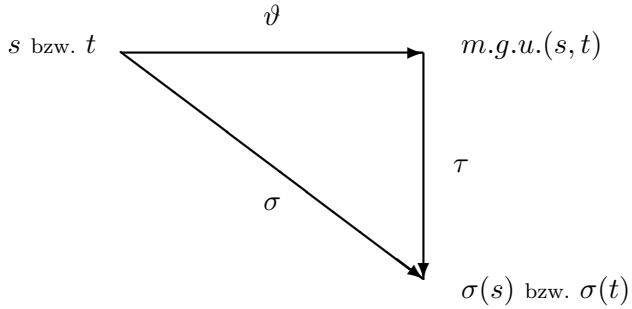
Substitutionen, die zwei Terme (syntaktisch) identisch machen (und nur solche sind für die Resolutionsmethode von Interesse), unifizieren diese bzw. heißen **Unifikator**:

ϑ **unifiziert** zwei Terme s und t (oder: ist **Unifikator** von s und t), falls $\vartheta(s) = \vartheta(t)$ ist.

Sinnvollerweise sucht man bei der Resolutionsmethode nicht irgendeinen Unifikator, sondern den **allgemeinsten Unifikator** (*most general unifier*):

Eine Substitution ϑ heißt **allgemeinster Unifikator** zweier Terme s und t, d.h. $\vartheta = m.g.u.(s,t)$, wenn

1. ϑ ein Unifikator von s und t ist und
2. für jeden Unifikator σ von s und t eine Substitution τ existiert, so daß $\sigma = \tau \circ \vartheta$ ist:



Um einen Algorithmus angeben zu können, welcher den allgemeinsten Unifikator zweier gegebener Terme s und t ermittelt, benötigen wir zunächst den Begriff der **Unterscheidungsterme**, die wie folgt gebildet werden:

Lese s und t simultan von links nach rechts. Man nehme die erste Stelle, an der sich s und t unterscheiden und nenne $\hat{s}$ und $\hat{t}$ diejenigen Teilterme von s bzw. t, die an dieser Stelle beginnen. $\hat{s}$ und $\hat{t}$ sind die Unterscheidungsterme von s und t.

Ein Algorithmus, der für zwei Terme s und t entscheidet, ob sie unifizierbar sind und im positiven Falle den allgemeinsten Unifikator liefert, ist der folgende:

Algorithmus vom allgemeinsten Unifikator:
Setze $i := 0$, $\vartheta_i := id$ (identische Substitution[11]), $s_i := s$ und $t_i := t$.
$\odot$ Teste, ob s_i und t_i syntaktisch übereinstimmen.
Falls ja, setze $\vartheta := \vartheta_i$. ϑ ist allgemeinster Unifikator. *stop*
Falls nein, bilde die Unterscheidungsterme $\hat{s}_i$ und $\hat{t}_i$ von s_i und t_i.
Teste, ob $\hat{s}_i$ und $\hat{t}_i$ beides keine Variablen sind.
Falls ja, so sind s und t nicht unifizierbar.[12] *stop*
Falls nein (sei etwa $\hat{s}_i$ eine Variable), so teste, ob $\hat{t}_i$ die Variable $\hat{s}_i$ enthält.
Falls ja, so sind s und t nicht unifizierbar. *stop*
Falls nein, setze
$\vartheta' := \vartheta_i \cup \{\vartheta_i(\hat{s}_i, \hat{t}_i)\}$[13]
$s' := \vartheta'(s_i)$ und $t' := \vartheta'(t_i)$
$i := i + 1$
$\vartheta_i := \vartheta'$ und $s_i := s'$ und $t_i := t'$
und gehe nach $\odot$.

[11] D.h. jede Variable wird durch sich selbst ersetzt.
[12] Wenn es sich um zwei strukturierte Terme handelt, dann haben sie verschiedene Funktionssymbole (sonst wären es nicht die *Unterscheidungs*terme) und sind schon deshalb nicht unifizierbar.
[13] D.h. ersetze auf allen rechten Seiten im Unifikator ϑ_i die Variable $\hat{s}_i$ durch den Term $\hat{t}_i$.

Mit der Resolutionsmethode und der (darin enthaltenen) Unifikation werden Resolutionsschritte vollzogen, deren Ziel der **Satz von ROBINSON** benennt:

$$M = \bigwedge_{i=1}^{n} K_i \wedge \neg H$$

ist kontradiktorisch ($kt \quad M$), gdw. durch wiederholte Anwendung der Resolutionsmethode schließlich (in endlich vielen Schritten) die Hypothese durch die (kontradiktorische) leere Klausel $false \leftarrow true$ (Symbol: $\square$) ersetzt werden kann.

Da sich bekanntlich nicht jeder beliebige Ausdruck in eine Konjunktion von HORN–Klauseln umformen läßt, scheint eine Verallgemeinerung dieser Methode auf Ausdrücke in Klauselform sinnvoll:

Sei $\{K_1, \ldots, K_n\}$ eine Menge von Ausdrücken in Klauselform. Es gebe darin 2 variablenfremde[14] Klauseln K_k und K_l mit

$$\begin{aligned} K_k &\equiv \bigvee_{i=1}^{m_A} A_i \leftarrow \bigwedge_{i=1}^{m_B} B_i \equiv \bigvee_{i=1}^{m_A} A_i \vee \bigvee_{i=1}^{m_B} \neg B_i \\ K_l &\equiv \bigvee_{i=1}^{m_C} C_i \leftarrow \bigwedge_{i=1}^{m_D} D_i \equiv \bigvee_{i=1}^{m_C} C_i \vee \bigvee_{i=1}^{m_D} \neg D_i \end{aligned}$$

(wobei A_i, B_i, C_i und D_i Atomformeln sind) derart, daß eine der A_i (etwa A_s) und eine der D_i (etwa D_t) miteinander unifizierbar sind, d.h. $\vartheta(A_s) \equiv \vartheta(D_t)$ mit $\vartheta = m.g.u.(A_s, D_t)$.

$$M \equiv \bigwedge_{i=1}^{n} K_i$$

ist kontradiktorisch ($kt \quad M$), wenn M nach dem Hinzufügen von

$$\begin{aligned} K &\equiv \bigvee_{i=1}^{s-1} \vartheta(A_i) \vee \bigvee_{i=s+1}^{m_A} \vartheta(A_i) \vee \bigvee_{i=1}^{m_C} \vartheta(C_i) \leftarrow \\ &\quad \bigwedge_{i=1}^{m_B} \vartheta(B_i) \wedge \bigwedge_{i=1}^{t-1} \vartheta(D_i) \wedge \bigwedge_{i=t+1}^{m_D} \vartheta(D_i) \\ &\equiv \bigvee_{i=1}^{s-1} \vartheta(A_i) \vee \bigvee_{i=s+1}^{m_A} \vartheta(A_i) \vee \bigvee_{i=1}^{m_C} \vartheta(C_i) \vee \\ &\quad \bigvee_{i=1}^{m_B} \vartheta(\neg B_i) \vee \bigvee_{i=1}^{t-1} \vartheta(\neg D_i) \vee \bigvee_{i=t+1}^{m_D} \vartheta(\neg D_i) \end{aligned}$$

noch immer kontradiktorisch ist. K heißt *Resolvente* von K_k und K_l.

Allein durch die Resolutionsmethode ist das ROBINSON'sche Ableitungsverfahren bei Ausdrücken in Klauselform allerdings noch nicht vollständig; hier bedarf es noch einer sogenannten **Faktorenregel**:

[14] D.h. sie dürfen keine gleich benannten Variablen haben. Ggf. muß vor der Resolution eine entsprechende Variablenumbenennung vollzogen werden. Dies ist ohne weiteres möglich, da gleich benannte Variablen in verschiedenen Klauseln völlig unabhängig voneinander sind.

Sind in einer Klausel K zwei Atomformeln A_i und A_j innerhalb des Klauselkopfes oder –körpers untereinander unifizierbar mit $\vartheta = m.g.u.(A_i, A_j)$, so ist $\vartheta(K)$ eine Resolvente.[15]

Der Satz von ROBINSON müßte dann wie folgt verallgemeinert werden:

$$M = \bigwedge_{i=1}^{n} K_i$$

ist kontradiktorisch ($kt \quad M$), gdw. sich durch wiederholte Anwendung von Resolutionsmethode und Faktorenregel schließlich (in endlich vielen Schritten) die (kontradiktorische) leere Klausel $false \leftarrow true$ (Symbol: $\square$) ableiten läßt.

Da es durch dieses Verfahren sicher sehr viele Wege geben kann, aus eine Konjuktion von Ausdrücken in Klauselform die leere Klausel $\square$ abzuleiten, erscheint die Suche nach *einem* derartigen Weg kaum systematisierbar. Zwei Möglichkeiten, diese Suche effizient zu gestalten, sind:

1. das Löschen „überflüssiger“ Klauseln

   „Überflüssig“ sind

   (a) tautologische Klauseln, d.h. solche, in denen die gleiche Atomformel A sowohl im Klauselkopf als auch im Klauselkörper vorkommt.

   (b) Klauseln K_j, die von „allgemeineren“ Klauseln K_i *subsumiert* werden:

   K_i subsumiert K_j, falls

   i. durch ein und dieselbe Substitution ϑ jedes Literal aus K_i einem Literal aus K_j identisch wird und

   ii. K_i höchstens genauso viele Literale wie K_j hat.[16]

2. die Einschränkung der Regelanwendung auf bestimmte Fälle

   Es ist vernünftig, die Kontradiktorizität einer Klauselmenge M nicht in einer Teilmenge $M_1 \subset M$ zu suchen, die erfüllbar ist. Diesen Gedanken greift die *Stützmengenstrategie* (engl. *set of support*) auf:

   > Wenn $M = M_1 \cup M_2$ (mit $M_1 \cap M_2 = \emptyset$) eine Klauselmenge mit erfüllbarem M_1 ist, dann läßt die Stützmengenstrategie für M nur solche Resolventenbildungen zu, für die mindestens eine der beteiligten Klauseln in M_2 ist oder durch (ggf. mehrere) Resolventenbildungen aus M_2 gewonnen wurde.

   Demnach sollte man auch die Faktorenregel sinnvollerweise nur auf Klauseln aus M_2 oder daraus gewonnenen Klauseln anwenden.

Ein weiteres korrektes und vollständiges Ableitungsverfahren ist:

[15] Die dabei syntaktisch identisch werdenden Atomformeln im Klauselkopf oder –körper in der Resolvente werden dabei nur nur ein Mal notiert.

[16] Diese zweite Bedingung verhindert, daß eine Klausel die aus ihr durch die Anwendung der (gerade mühsam erarbeiteten und essentiell benötigten) Faktorenregel gewonnenen Klauseln (Faktoren) subsumiert.

Das natürliche Schließen nach GENTZEN „$\vdash_{NI}$“

Sei M eine Menge von Aussagen; H, H_1, H_2, H^*, A und B Aussagen. Es gilt

1. Einbettung:
   $M \vdash_{NI} H$, wenn $H \in M$.
2. (a) UND–Einführung:
   $M \vdash_{NI} (H_1 \wedge H_2)$, wenn $M \vdash_{NI} H_1$ und $M \vdash_{NI} H_2$.
   (b) UND–Beseitigung:
   $M \vdash_{NI} H_1$ und $M \vdash_{NI} H_2$, wenn $M \vdash_{NI} (H_1 \wedge H_2)$.
   (c) ODER–Einführung:
   $M \vdash_{NI} (H_1 \vee H_2)$, wenn $M \vdash_{NI} H_1$ oder $M \vdash_{NI} H_2$.
   (d) ODER–Beseitigung:
   $M \vdash_{NI} H$, wenn $M \vdash_{NI} (H_1 \vee H_2)$ und $M \cup \{H_1\} \vdash_{NI} H$ und $M \cup \{H_2\} \vdash_{NI} H$.
   (e) IMPLIKATIONs–Einführung:
   $M \vdash_{NI} (H_1 \rightarrow H_2)$, wenn $M \cup \{H_1\} \vdash_{NI} H_2$.
   (f) IMPLIKATIONs–Beseitigung:
   $M \vdash_{NI} H_2$, wenn $M \vdash_{NI} (H_1 \rightarrow H_2)$ und $M \vdash_{NI} H_1$.
   (g) ÄQUIVALENZ–Einführung:
   $M \vdash_{NI} (H_1 \leftrightarrow H_2)$, wenn $M \vdash_{NI} (H_1 \rightarrow H_2)$ und $M \vdash_{NI} (H_2 \rightarrow H_1)$.
   (h) ÄQUIVALENZ–Beseitigung:
   $M \vdash_{NI} (H_1 \rightarrow H_2)$ und $M \vdash_{NI} (H_2 \rightarrow H_1)$, wenn $M \vdash_{NI} (H_1 \leftrightarrow H_2)$.
   (i) NICHT–Einführung:
   $M \vdash_{NI} \neg H$, wenn $M \cup \{H\} \vdash_{NI} H^*$ und $M \cup \{H\} \vdash_{NI} \neg H^*$.
   (j) NICHT–Beseitigung:
   $M \vdash_{NI} H$, wenn $M \vdash_{NI} \neg\neg H$.
   (k) ALLQUANTOR–Einführung:
   $M \vdash_{NI} \forall X A(X)$, wenn $M \vdash_{NI}^{(a)} A(a)$.
   (l) ALLQUANTOR–Beseitigung:
   $M \vdash_{NI}^{(a)} A(a)$, wenn $M \vdash_{NI} \forall X A(X)$.
   (m) EXISTENZQUANTOR–Einführung:
   $M \vdash_{NI} \exists X A(X)$, wenn $M \vdash_{NI} A(a)$.
   (n) EXISTENZQUANTOR–Beseitigung:
   $M \vdash_{NI} B$, wenn $M \vdash_{NI} \exists X A(X)$ und $M \cup \{A(a)\} \vdash_{NI}^{(a)} B$.
3. $M \vdash_{NI} H$, gdw. das durch sukzessives Hintereinanderanwenden der Regeln gemäß 1. und 2. möglich ist.

$\vdash_{NI}^{(a)}$ ... „Die Konstante a wird in keinem der Ableitungsschritte benutzt“

Nachteilig an dem Verfahren nach GENTZEN ist, daß es kaum algorithmisierbar ist:

1. Es gibt keine Methode, den nächsten Ableitungsschritt *zielgerichtet* zu unternehmen.

   Es gibt keine Möglichkeit zu evaluieren, wie nahe man der Lösung ist.

   Da jedoch stets mindestens eine der Regeln anwendbar ist[17], hilft hier auch kein Backtrack–Mechanismus zwecks Weitersuche in der Breite, denn die Tiefensuche gerät nie in eine Sackgasse.

[17] Auch das ist eine interessante Eigenschaft des Ableitungsverfahrens, die gar nicht so selbstverständlich ist

2. Eine (auch denkbare) Breitensuche scheitert an der Größe des Suchraumes.

5.2.2 Induktion

Ziel der Induktion in der Wissensverarbeitung ist es,

- aus einer Menge elementarer wahrer Aussagen allgemeine Regeln abzuleiten bzw.
- aus allgemeinen Regeln und einem neuen Beispiel (Fakt) neue allgemeine Regeln zu generieren, die das Beispiel mit erfassen.

Auf der Ebene der Aussagenlogik (PK0) gibt es bereits massenhaft Anwendungen von Verfahren der Induktion[18]; auf prädikatenlogischem (PK1–) Niveau hat man hierfür erste Grundlagen geschaffen.

Induktionsverfahren über prädikatenlogischen Ausdrücken haben i.allg. zum Ziel, aus einer Menge variablenfreier Ausdrücke $\{A_1, \ldots, A_n\}$ einen allgemeinen variablenbehafteten Ausdruck B zu induzieren, so daß für alle A_i gilt:

$$ag(B \rightarrow A_i)$$

B ist dann eine logische Begründung für alle beobachteten Situationen A_i. Meist gibt es sehr viele Begündungen für die beobachteten Situationen A_i, die jedoch unterschiedlich interessant sein können. Hier ist i.allg. eine „beste“ Begründung gefragt:

> B heißt *bester induktiver Schluß* (oder einfachste Begründung) für $M = \{A_1, \ldots, A_n\}$, falls gilt:
>
> 1. B ist eine logische Begründung für M und
> 2. Jede andere logische Begründung C für M begründet auch B.

Dieser Sachverhalt kann bildlich wie in Abbildung 6 gezeigt werden, wobei die Pfeile tatsächlich als Implikationssymbole allgemeingültiger Aussagen interpretiert werden können.

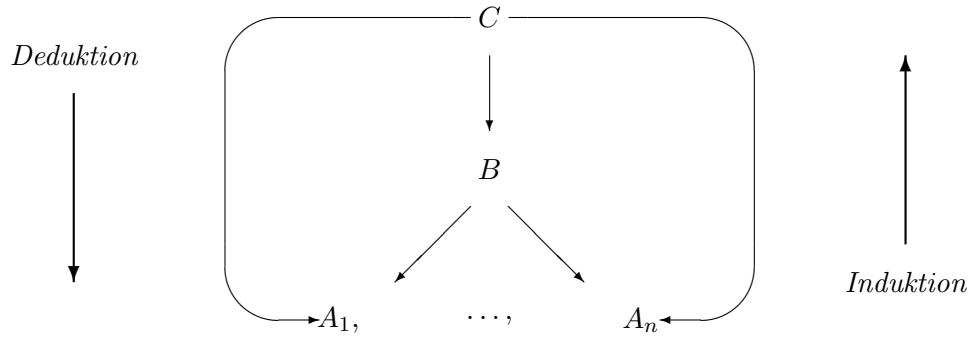


Abbildung 6: B ist der „beste induktive Schluß“ für $\{A_1, \ldots, A_n\}$

Ein (zu) einfacher Weg des Findens eines solchen „besten induktiven Schlusses“B wäre

$$B = \bigwedge_{i=1}^{n} A_i$$

Dies scheint jedoch kaum praktischen Anforderungen zu genügen und wird um so sinnloser, je größer n ist.

[18] . . . , so z.B. das auf der sog. „Entropie der Information“ beruhende „ID3–Verfahren“, welches ein optimales Regelwerk anhand von Beispielen generiert (und in einer anderen Lehrveranstaltung behandelt wird).

Solche „besten induktiven Schlüsse“ B lassen sich u.a. auch als die speziellsten gemeinsamen Muster aller Ausdrücke $\{A_1, \ldots, A_n\}$ auffassen.

Für dieses Verfahren müssen alle A_i Kombinationen (ganau genommen Konjunktionen) von Atomformeln sein, deren Variablen allquantifiziert sind. Der zu findende induktive Schluss B ist ein Ausdruck der Gestalt

$$\forall X_1 \ldots \forall X_m \Phi$$

wobei Φ keine Quantoren enthält.

Der „Trick“ dieses Verfahrens besteht darin, diejenigen Atomformeln, die das gleiche Prädikatensymbol aufweisen, durch die Angabe eines „gemeinsamen Musters“ zusammenzufassen. Dabei entstehen variablenbehaftete Atomformeln mit vorangestellten Allquantoren, die konjunktiv verknüpft werden. Durch Ausklammern der Allquantoren[19] entsteht dann Φ.

Das Problem reduziert sich somit auf ein sogenanntes *Antiunifikationsproblem*:

1. Für quantorfreie Ausdrücke A und B gilt $B \geq A$, genau dann, wenn $ag(B \rightarrow A)$.[20]

2. B heißt **Antiunifikation** einer Menge von Aussagen $\{A_1, \ldots, A_n\}$, falls $\forall i : B \geq A_i$.

3. Die **speziellste Antiunifikation** Φ von $\{A_i, \ldots, A_n\}$, ist diejenige Antiunifikation, für die gilt: Alle anderen Antiunifikationen C sind „allgemeiner“ als Φ, d.h. $C \geq \Phi$.

   Die Termersetzung ϑ, die Φ aus allen A_i erzeugt ($\vartheta(A_1) = \vartheta(A_2) = \ldots = \vartheta(A_n) = \Phi$), heißt **speziellster gemeinsamer Antiunifikator** (*most specific antiunifier*) von $\{A_1, \ldots, A_n\}$:

   $$\vartheta = m.s.a.(\{A_1, \ldots, A_n\})$$

Das Ziel der Antiunifikation ist demnach das Finden eines *speziellsten gemeinsamen Antiunifikators* einer Menge von Atomformeln $\{A_1, \ldots, A_n\}$.

Das Problem des Findens des speziellsten Antiunifikators (auch most specific antiunifier $m.s.a.$ genannt) zweier Atomformeln wird durch das Verfahren, welches den $m.s.a.$ zweier Terme findet, mitgelöst, so daß die Angabe des letzteren hier genügt.

Ein Verfahren zur Ermittlung der speziellsten Antiunifikation 2er Terme

Man versteht Antiunfikation am besten als (potentiell nicht-deterministischen) Reduktionsprozeß.

Man gibt ein Regelsystem an. Dann ist dreierlei zu tun:

1. Man muß den Formalismus des Reduktionssystems mit der zu l
   ösenden Aufgabe in Beziehung setzen.

2. Man muß Termination nachweisen.

3. Man muß die Korrektheit der Ergebnisse – im jeweiligen Anwendungsszenario – nachweisen.

[19] Man denke daran, daß die Atomformelnn vorher variablenfremd zu machen sind!

[20] Sind A und B Konjunktionen von Atomformeln, dann bedeutet dies nichts anderes als „A ist allgemeiner als“ B. D.h. für jede Atomformel B_i in B gibt es in A eine Atomformel A_j und eine Substitution ϑ derart, daß $\vartheta(A_j) \equiv B_i$ ist.

Es seien zwei Terme t_1 und t_2 gegeben. In beiden Termen wird sukzessive die Antiunifikation ausgef
ührt, bis sie beide syntaktisch gleich sind. Die Stellen, an denen noch etwas zu tun ist, werden in einer Markierungsmenge M notiert, und zwar durch Positionen im Term. Die sukzessive aufgebauten Substitutionen legt man in Σ ab, wobei die linke Seite jeder Substitutionsregel nur einmal auftaucht, denn es wird ja nur in ein und demselben Term – der Antiunifikation – ersetzt. Deshalb enth
ält Σ Tripel.

Eine Ausgangssituation f
ür zwei Terme t_1 und t_2 wird formalisiert als

$$[\, t_1, t_2, \{\varepsilon\}, \emptyset \,]$$

und eine Zielsituation der Form

$$[\, t, t, \emptyset, \Sigma \,]$$

bedeutet, daß t als die speziellste Antiunifikation konstruiert wurde und Σ die beiden Substitutionen enth
ält, die erlauben, daraus t_1 bzw. t_2 zu erzeugen.

Das Regelsystem

Vor jeder Regel werden die Bedingungen der Regelanwendung angegeben.

R1 $p \in M,\ t_1.p \neq t_2.p,\ [x, t_1|_p, t_2|_p] \in \Sigma$

$$[\, t_1, t_2, M, \Sigma \,] \quad \Longrightarrow \quad [\, t_1[p \hookleftarrow x], t_2[p \hookleftarrow x], M \setminus \{p\}, \Sigma \,]$$

R2 $p \in M,\ t_1.p \neq t_2.p,\ \not\exists\, [x, t_1|_p, t_2|_p] \in \Sigma,\ x \notin \pi_1(\Sigma)$

$$[\, t_1, t_2, M, \Sigma \,] \quad \Longrightarrow \quad [\, t_1[p \hookleftarrow x], t_2[p \hookleftarrow x], M \setminus \{p\}, \Sigma \cup \{[x, t_1|_p, t_2|_p]\} \,]$$

R3 $p \in M,\ t_1.p = t_2.p,\ t_1|_p \neq t_2|_p,\ \alpha(t_1.p) = n$

$$[\, t_1, t_2, M, \Sigma \,] \quad \Longrightarrow \quad [\, t_1, t_2, M \setminus \{p\} \cup \{p1, \ldots, pn\}, \Sigma \,]$$

R4 $p \in M,\ t_1|_p = t_2|_p$

$$[\, t_1, t_2, M, \Sigma \,] \quad \Longrightarrow \quad [\, t_1, t_2, M \setminus \{p\}, \Sigma \,]$$

Ein paar umgangssprachliche Erkl
ärungen sollen zun
ächst dem Formalismus den Schrecken nehmen, bevor Details der Notation eingef
ührt werden.

zu **R1** An einer noch zu bearbeitenden Position in M stehen verschiedene Terme, die bereits schon einmal durch eine Variable x eliminiert worden sind. Dieselbe Variable kann hier benutzt werden. Diese Position ist dann erledigt; mehr ist nicht zu tun.

zu **R2** An einer noch zu bearbeitenden Position in M stehen verschiedene Terme, die in dieser Form noch nicht ersetzt worden sind. Es wird eine neue Variable x gew
ählt und f
ür diese Terme eingef
ügt. Entsprechend wird die Substitutionsliste erg
änzt und die Position gestrichen.

zu **R3** An einer noch zu bearbeitenden Position in M stehen Terme mit demselben Funktionssymbol. An dieser Position ist nichts zu
ändern, aber alle Teiltermpositionen werden in die Liste der zu bearbeiten Aufgaben aufgenommen.

zu **R4** An einer noch zu bearbeitenden Position p stehen jeweils dieselben Terme. Diese Position ist erledigt; mehr ist nicht zu tun.

Notationen

Positionen in Termen sind W
örter
über der Menge der nat
ürlichen Zahlen $I\!N$. Das leere Wort heißt ε.[21]

Erkl. Wenn t irgendein Term ist, der an der Position p den Teilterm $f(t_1, \ldots, t_n)$ hat, dann hat er an den Positionen $p1, \ldots pn$ die entsprechenden Teilterme $t_1, \ldots, t_n$.

Wenn p eine Position im Term t ist, dann bezeichnet $t.p$ den Operator an dieser Position im Term t und $t|_p$ den in t an der Position beginnenden Teilterm. Trivialerweise ist f
ür jeden Term t und die Wurzelposition ε stets $t = t|_\varepsilon$.

Bspl. Betrachten wir den Term $t = f(\sin(x+1), 1-\cos(\varphi))$. Das Funktionssymbol an der Position ε ist f und das Symbol an der Position 1 ist sin. Insgesamt haben wir:

$t.\varepsilon = f \quad t.1 = \sin \quad t.2 = - \quad t.11 = + \quad t.21 = 1 \quad t.22 = \cos \quad t.111 = x \quad t.112 = 1 \quad t.221 = \varphi$

Der Teilterm an der Position 1 ist $\sin(x+1)$ und der Teilterm an der Position 11 ist $x+1$. F
ür die Teilterme "auf der rechten Seite" des 2-stelligen Operators f haben wird insgesamt also:

$t|_2 = 1-\cos(\varphi) \quad t|_{21} = 1 \quad t|_{22} = \cos(\varphi) \quad t|_{221} = \varphi$

Wenn in einem Term t an einer Position p ein Term t' eingesetzt wird, so notieren wir das durch $t[p \hookleftarrow t']$.

Bspl. Betrachten wir den Term $t = f(\sin(x+1), 1-\cos(\varphi))$. Eine Ersetzung von z an der Position 1 wird notiert als $t[1 \hookleftarrow z]$ und liefert $t[1 \hookleftarrow z] = f(z, 1-\cos(\varphi))$.

Eigentlich geh
ört zur Beschreibung der Arit
ät eines Funktionssymbols nicht nur die Stelligkeit, sondern auch die Sortigkeit. Da die Arit
ät nur dort ins Spiel kommt, wo Sortenunterschiede nicht mehr auftreten k
önnen, werden diese ignoriert. α gibt zu einem Funktionssymbol dessen Stelligkeit an. F
ür Konstanten und Variablen ist dieser Wert 0.

[21] Achtung: Da diese Auffassung kein irgendwie geartetes n-
äres Zahlsystem voraussetzt, gibt es auch keine Konfusionen.

Mit π_1 wird die Projektion auf das erste Argument bezeichnet. So ist insbes. $\pi_1(M)$ die Menge aller Variablen, die in Tripeln aus M vorkommen.

Termination

Es ist fast trivial, die Termination dieses Reduktionssystems zu beweisen. Man macht das am besten nur
über M.

3 Regeln verkleinern M jeweils echt. Die einzige Regel, die M vergr
ößert, ist **R3**. Diese Regel f
ührt allerdings zu einer gegebenen Position nur l
ängere Positionen ein. Wegen der Endlichkeit der Ausgangsterme kann **R3** nur endlich oft angewendet werden. Das gen
ügt.

Beispiele

Ein erstes Beispiel

Gegeben seien die Terme

$$\begin{aligned} t_1 &= p(f(a,b), f(a,b), X, a) \\ t_2 &= p(f(a,X), g(a,b), b, Y) \end{aligned}$$

Die Ausgangssituation ist demnach

$$[p(f(a,b), f(a,b), X, a), p(f(a,X), g(a,b), b, Y), \{\varepsilon\}, \emptyset]$$

Die Regelanwendungen ergeben

Regel	t_1	t_2	M	Σ
	$p(f(a,b), f(a,b), X, a)$	$p(f(a,X), g(a,b), b, Y)$	$\{\varepsilon\}$	$\emptyset$
R3	$p(f(a,b), f(a,b), X, a)$	$p(f(a,X), g(a,b), b, Y)$	$\{1,2,3,4\}$	$\emptyset$
R3	$p(f(a,b), f(a,b), X, a)$	$p(f(a,X), g(a,b), b, Y)$	$\{11,12,2,3,4\}$	$\emptyset$
R4	$p(f(a,b), f(a,b), X, a)$	$p(f(a,X), g(a,b), b, Y)$	$\{12,2,3,4\}$	$\emptyset$
R2	$p(f(a,X_0), f(a,b), X, a)$	$p(f(a,X_0), g(a,b), b, Y)$	$\{2,3,4\}$	$\{[X_0,b,X]\}$
R2	$p(f(a,X_0), X_1, X, a)$	$p(f(a,X_0), X_1, b, Y)$	$\{3,4\}$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)]\}$
R2	$p(f(a,X_0), X_1, X_2, a)$	$p(f(a,X_0), X_1, X_2, Y)$	$\{4\}$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)],$ $[X_2,X,b]\}$
R2	$p(f(a,X_0), X_1, X_2, X_3)$	$p(f(a,X_0), X_1, X_2, X_3)$	$\emptyset$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)],$ $[X_2,X,b],$ $[X_3,a,Y]\}$

Noch ein (scheinbar ganz ähnliches) Beispiel

Gegeben seien die Terme

$$\begin{aligned} t_1 &= p(f(a,b), f(a,b), X, b) \\ t_2 &= p(f(a,X), g(a,b), b, X) \end{aligned}$$

Die Ausgangssituation ist demnach

$$[p(f(a,b),f(a,b),X,b),p(f(a,X),g(a,b),b,X),\{\varepsilon\},\emptyset]$$

Die Regelanwendungen ergeben

Regel	t_1	t_2	M	Σ
	$p(f(a,b),f(a,b),X,b)$	$p(f(a,X),g(a,b),b,X)$	$\{\varepsilon\}$	$\emptyset$
R3	$p(f(a,b),f(a,b),X,b)$	$p(f(a,X),g(a,b),b,X)$	$\{1,2,3,4\}$	$\emptyset$
R3	$p(f(a,b),f(a,b),X,b)$	$p(f(a,X),g(a,b),b,X)$	$\{11,12,2,3,4\}$	$\emptyset$
R4	$p(f(a,b),f(a,b),X,b)$	$p(f(a,X),g(a,b),b,X)$	$\{12,2,3,4\}$	$\emptyset$
R2	$p(f(a,X_0),f(a,b),X,b)$	$p(f(a,X_0),g(a,b),b,X)$	$\{2,3,4\}$	$\{[X_0,b,X]\}$
R2	$p(f(a,X_0),X_1,X,b)$	$p(f(a,X_0),X_1,b,X)$	$\{3,4\}$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)]\}$
R2	$p(f(a,X_0),X_1,X_2,b)$	$p(f(a,X_0),X_1,X_2,X)$	$\{4\}$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)],$ $[X_2,X,b]\}$
R1	$p(f(a,X_0),X_1,X_2,X_0)$	$p(f(a,X_0),X_1,X_2,X_0)$	$\emptyset$	$\{[X_0,b,X],$ $[X_1,f(a,b),g(a,b)],$ $[X_2,X,b]\}$

Das Problem, die Antiunifikation einer *Menge* von (i.allg. mehr als zwei) Atomformeln zu definieren, ist nun recht einfach zu lösen: Statt der Tripel in der Termersetzung Σ entstehen für eine n-elementige Menge $\{t_1,t_2,\ldots,t_n\}$ dann $n+1$-Tupel der Art

$$[Variable,Ersetzung\_in\_t_1,Ersetzung\_in\_t_2,\ldots,Ersetzung\_in\_t_n]$$

Wie die Vorbedingungen für die Anwendung der Regeln R1-R4 in diesem Falle zu formulieren sind, verbleibt dem Leser als bungsaufgabe![22]

5.2.3 Abduktion

Die dritte Methode des Schließens, die **Abduktion**, ist eine Schlußweise, bei der aus beobachteten Erscheinungen auf mögliche Ursachen geschlossen wird. Ein solcher Schluß erfolgt demnach umgekehrt zur natürlichen Kausalität.

Im einfachsten Fall stellt sich ein abduktiver Schluß wie folgt dar:

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

Dieser Schluß ist natürlich außerordentlich fragwürdig, denn es gilt mitnichten

$$\{A \to B, B\} \models A$$

Bei wissensverarbeitenden Diagnosesystemen ist eine solche Schlußweise jedoch an der Tagesordnung, denn dort wird von einer vorliegenden Symptomatik auf den vorliegenden Fehler geschlossen, obwohl die natürliche Kausalität genau entgegengesetzt orientiert ist.

Um die praktische Anwendung einer solch fragwürdigen Schlußweise zu legitimieren, sollten die Regeln möglichst so beschaffen sein, daß sie ein **vollständiges** und **eindeutiges** Regelsystem bilden. Für die Diagnoseproblematik würde das bedeuten, daß alle überhaupt denkbaren Diagnosen (vollständig) erfaßt werden und für eine gegebene Symptomatik (Konklusion) nur eine einzige Diagnose (eindeutig) als Prämisse in Frage kommt, d.h.

[22] Es ist zu beachten, dass es zur Anwendung von R1 und R2 genügt, dass *ein Paar* von Termen t_i und t_j existiert mit $t_i.p \neq t_j.p$. Für die Anwendung der Regeln R3 und R4 hingegen müssen natürlich *alle* Funktionssymbole an der Stelle p bzw. alle an der Stelle p beginnenden Teilterme identisch sein.

- alle denkbaren Prämissen des Diskursbereiches müssen in (mindestens) einer Regel vorkommen und
- die Konklusionen sämtlicher in einer Situation zu prüfenden Regeln müssen einander ausschließen, d.h. sie müssen paarweise disjunkt sein:

$$\forall i \forall j((i \neq j) \rightarrow ((Konklusion_i \wedge Konklusion_j) \equiv false))$$

Die Bezeichnungen „Prämissen“ und „Konklusionen“ beziehen sich hier auf die natürliche Kausalität. Je eklatanter diese Anforderungen verletzt werden, desto fragwürdiger wird dieser Schluß. Liegen beide Eigenschaften 100%ig vor, so kann die Implikation durch die Äquivalenz ersetzt werden. Da eine Äquivalenz nichts anderes als eine Implikation in beiden Richtungen ist, handelt es sich dann folglich um einen (formallogisch sauberen) deduktiven Schluß.