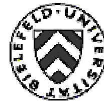
Universität Bielefeld

2 Logik und Inferenz

4. Vorlesung: Theorembeweisen. Logisches und Regelbasiertes Programmieren

Methoden der Künstlichen Intelligenz

Bernhard Jung WS 2001/2002



Universität Bielefeld

Übersicht

- Theorembeweisen
- Logisches Programmieren
- Regelbasiertes Programmieren

Universität Bielefeld

Wdh: Logische Wissensbasen

Wissensbasis (WB) besteht aus prädikatenlogischen Formeln, die Fakten und Regeln der Welt repräsentieren.

Formeln in kanonischer, quantorenfreier Form dargestellt:

- **z.B. *implicit quantifier form*; siehe VL 3, Charniak & McDermott**
- **z.B. konjunktive Normalform (KNF): WB als Menge von Klauseln (Klausel = Menge von disjunktiv verknüpften Literalen)**

Wieso quantorenfreie Form?

- **ermöglicht effiziente Inferenzverfahren, basierend auf Unfikation (anstelle etwa der ineffizienten universellen Einsetzung)**

Wie in quantorenfreie Form bringen? (Details siehe VL 3)

- **Existenzquantoren durch Skolemisierung eliminieren, existenzquantifizierte Variablen werden dabei durch Skolemfunktion ersetzt**
- **Verbleibende Variablen sind allquantifiziert, Allquantor wird weggelassen**

Universität Bielefeld

Wdh: Inferenzregeln und Theorembeweisen

Deduktive Inferenzregeln

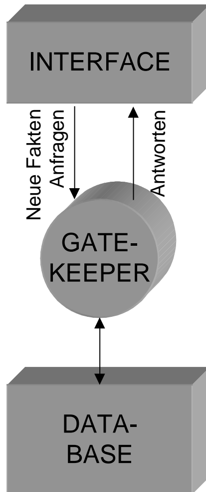
- **Ableitung einer Formel (oder auch mehrerer Formeln), die logisch aus gegebener Menge von Formeln (den "Axiomen") folgt.**
- **Betrachtete Inferenzregeln: Modus Ponens, universelle Einsetzung, Generalisierter Modus Ponens, Substitution, Resolution**

Theorembeweisen

- **Wiederholte Anwendung von (deduktiven) Inferenzregeln um Wahrheit einer bestimmten Formel (eines "Theorems") zu zeigen**
- **z.B. Modus Ponens + universelle Einsetzung; ineffizient**
- **z.B. Modus Ponens mit Unifikation ("Generalisierter Modus Ponens"); effizienter, aber unvollständig**
- **z.B. Resolution mit Unifikation; (widerspruchs-)vollständig**

Universität Bielefeld

Wdh: Design wissensbasierter Systeme



Einfachstes Design:

DATABASE (die Wissensbasis) hat Datenstrukturen in Form standardisierter prädikatenlogischen Formeln.

GATEKEEPER kann nur die gewöhnlichsten *Inferenzregeln* (Schlußregeln) des Prädikatenkalküls ausführen.

Basiskommandos zur Benutzung der DATABASE durch GATEKEEPER:

`assert`	**fügt jeweils eine Proposition in die DATABASE ein.**
`retract`	**nimmt Propositionen wieder heraus ("zieht sie zurück")**
`query`	**gibt eine Frage-Formel (Frage-Muster) vor und versucht, darauf passende Antwort-Formeln aus den assertierten Formeln zu *deduzieren*.**

Universität Bielefeld

Vorwärts- & Rückwärtsverkettung

Forward Chaining

Aus p' und (if p q)
inferiere q'

wobei p mit p' unifiziert
mit MGU θ und $q' = q\theta$

- **Inferenz zur Assertion Time**
- **"Die Assertion resolviert mit der Implikation."**

Backward Chaining

(if p q) sei assertiert.

Wenn nach q' gefragt wird (als goal)
und q, q' haben MGU θ
wird $p' = p\theta$ als subgoal aufgeworfen.

- **Inferenz zur Query Time**
- **"Das goal resolviert mit der Implikation."**

(Voraussetzung: die Formeln in DATABASE sind standardisiert.)

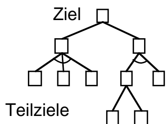
Universität Bielefeld

Rückwärtsverkettung – Idee

Die meisten Inferenzen in einem deduktiven System werden zur Query Time vorgenommen.

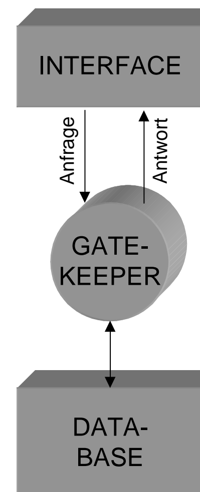
- **GATEKEEPER wird eine Implikation (if p q) zwar assertieren, aber nichts damit anstellen, bis eine Frage (query, goal) der Form q' gestellt wird.**
  Dann wird subquery (subgoal) p' gestellt.
- **Das Stellen von subgoals, sub-subgoals etc. wird durch backward chaining (Rückwärtsverkettung) bewerkstelligt.**
- **Der Basismechanismus des Backward Chaining bezieht sich auf *nichtkonjunktive goals:* Verkettung von Implikationen (rückwärts).**
- **Als Suchstruktur bei *konjunktiven goals* von der Form (if (and $p1$ $p2$) q) werden Goal Trees eingesetzt.**

Goal Trees
(Zielbäume)



Universität Bielefeld

Goal-Formeln kennzeichnen:



Verwende `(Show:formula)` zur Kennzeichnung von goal-Formeln.

Z.B. goal	`(Show: (color tweety yellow))`
bzw. subgoal	`(Show: (inst tweety canary))`
bei Regel	`(if (inst ?x canary)(color ?x yellow))`

- **`Show:` ist kein logischer Junktor! (manchmal Pseudo-Junktor genannt)**
- **hängt aber zusammen mit dem Junktor `not`**

Wenn eine Anfrage (query) an DATABASE gestellt wird, muß die entsprechende Formel gekennzeichnet werden, um sie von den "beliefs" (assertierten Formeln) unterscheiden zu können.

`Show:` *versus* `not`

`(Show: (color tweety yellow))` **(goal)**
kann als Versuch verstanden werden,
`(color tweety yellow)` ***durch Widerspruch zu beweisen*:**

<u>Logisch gesehen</u> ist `Show:` gleichbedeutend mit `not` (die entsprechenden Ausdrücke werden in gleicher Weise skolemisiert)

<u>pragmatisch gesehen</u> entspricht `Show:` einer Annahme (probeweise gemachte Assertion)

`(not (color tweety yellow))`
wird probeweise assertiert – *als Annahme;*

schon vorhanden seien folgende "geglaubte" Assertionen:

`(inst tweety canary)`

`(if (inst ?x canary)(color ?x yellow))`
gleichbedeutend:
`(or (not(inst ?x canary))(color ?x yellow))`

Das negierte goal resolviert mit der Implikation zu
`(not (inst tweety canary))`

Widerspruch! **Also gilt das ursprüngliche goal.**

Goal-Formeln skolemisieren:

Für goals werden die Konventionen für das Skolemisieren umgedreht (entsprechend dem Verfahren bei not):

`(Show: (exists (y) (color y yellow)))`
wird skolemisiert als
`(Show: (color ?y yellow))`

`(Show: (forall (y) (color y yellow)))`
wird skolemisiert als
`(Show: (color sk-8 yellow))`

intuitiv: Wenn es für ein gewisses *neues* `sk-8` gezeigt werden kann, kann es für beliebige Instanzen gezeigt werden.

Zentral: Antwortsubstitutionen

<u>Allgemein:</u>

Gegeben das goal $(\mathtt{Show:}q')$ mit $(\mathtt{if}\ p\ q)$ in DATABASE und q, q' haben MGU θ .

Produziere subgoal $(\mathtt{Show:}p\theta)$ und falls das eine Antwort ψ hat, ist $\theta \cup \psi$ eine Antwort auf das ursprüngliche goal.

„theta“ *„psi“*

Beispiel: Assertiert seien
(i) `(inst tweety canary)`
(ii) `(if (inst ?x canary)(color ?x yellow))`

`(Show: (color ?y yellow))` **als goal**

resolviert mit (ii) zum subgoal
`(Show: (inst ?y canary))` θ = {x = ?y}
dies unifiziert mit (i) zu
`(inst tweety canary)` ψ = {y = tweety}
und damit bestätigt sich das ursprüngliche goal zu
`(color tweety yellow)` $\theta \cup \psi$ = {x = tweety}

D.h. die Frage *"Existiert etwas, das gelb ist?"*
führt zu der Antwort *"Ja,* `tweety` *ist gelb":* {x = tweety}

Antwortsubstitution

Goal Trees für subgoal-Suche

Konjunktives Goal: **Finde Antworten für ein Konjunkt, die auch Antworten für die anderen Konjunkte sind.**
z.B. `(Show: (and respect(?x,?y) older(?x,?y) ) )`

Algorithmus: Theorembeweiser ("Deductive Retriever")

---> Goal Trees
(UND-ODER Bäume, in welchen Knoten Goals darstellen)

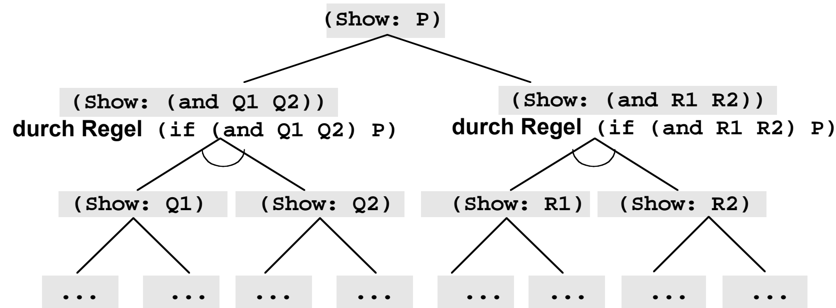
Bild siehe nächste Folie

Die UND-Knoten eines Goal Trees enthalten *constraints:* Randbedingungen an die Lösungen für jede ihrer Komponenten.

Hier sind das die folgenden: Die Variablenbindungen für gleich benannte Variablen in den Teillösungen müssen identisch sein!

Universität Bielefeld

Theorembeweiser-Goal Tree



UND-Knoten: alle Teilziele müssen gezeigt werden
Constraints: Variablenbindungen nicht widersprüchlich

ODER-Knoten: (mindestens) ein Teilziel muß gezeigt werden

Universität Bielefeld

Theorembeweisen: Bemerkungen

J.A. Robinson, 1996

Algorithmen zum Theorembeweisen sind oft unvollständig (existierende Beweise werden nicht immer gefunden)!

Versuche, einen Theorembeweiser vollständig(er) zu machen, werden i.a. teuer mit Effizienzverlusten bezahlt.

Es gibt etliche Varianten des Resolutionsverfahrens und speziellen Strategien dafür, die im Gebiet "Automatisches Beweisen" untersucht werden*.

Sie alle gehen auf das allgemeine Resolutionsverfahren ("complete resolution") von ROBINSON (1965) zurück.

Theorembeweisen z.B. Grundlage von PROLOG.

Universität Bielefeld

Übersicht

- Theorembeweisen
- Logisches Programmieren
- Regelbasiertes Programmieren

Universität Bielefeld

Programmierstile

Die zugrundeliegenden Verarbeitungsmodelle (Programmiersprache + ausführende Maschine) sind sehr verschieden.

- **Prozeduraler Programmierstil**
  Schrittweise ausgeführte Folgen von Anweisungen (Prozeduren) modifizieren Daten in einem Speicher
- **Objektorientierter Programmierstil**
  Berechnung durch Kommunikation: Austausch von Nachrichten zwischen aktiven unabhängigen Objekten
- **Funktionaler Programmierstil**
  Berechnung durch Auswertung von Funktionen (Abbildung von Definitionsbereich in Wertebereich)
- **Logikorientierter Programmierstil**
  Deklarative Notierung von Problem und Lösungs"wissen" als logische Formeln, Berechnung durch Theorembeweisen
- **Regelbasierter Programmierstil**
  Berechnung durch Regelanwendung und deren Vorwärts- oder Rückwärtsverkettung

Universität Bielefeld

Idee logischer Programmierung

- **Probleme werden als logische Aussagen formuliert (Theoreme, die man "anfragt").**
- **Das zur Lösung eines Problems benötigte Wissen wird ebenfalls in Form logischer Aussagen formuliert (Axiome, die man "assertiert").**
- **Die Lösung eines Problems besteht im Finden eines logischen Beweises für das angefragte Theorem.**

Idee: Das Finden des Beweises wird einem automatischen Theorembeweiser übertragen, der den Beweis (sofern existent) konstruieren kann.

Damit reduziert sich das Problemlösen auf ein Formulierungsproblem.

(Idealvorstellung; nicht in vollem Umfang erreicht!)

Universität Bielefeld

Theorembeweiser

- Grundlage logikorientierter Programmierung
- basieren in vielen Fällen auf dem Resolutionsverfahren:
  - allgemeineres Schlußverfahren als Modus ponens
  - arbeitet auf Formeln in Klauseldarstellung
- Resolutionsbeweiser sind zwar allgemein langsam, für Hornklauseln in der Aussagenlogik jedoch sehr effizient
- sind auf Klauseln der Prädikatenlogik (PL) erweiterbar und haben dort für Hornklauseln immer noch Effizienzvorteile

⇨ PROLOG

Universität Bielefeld

Von PL-Klauseln zu „Regeln“

$\forall x_1 \ldots \forall x_s \; (L_1 \vee \ldots \vee L_m)$ **L_j: Literale (Atome oder negierte Atome)**

⬇

$\forall x_1 \ldots \forall x_s \; (A_1 \vee \ldots \vee A_k \vee \neg C_1 \vee \ldots \vee \neg C_n)$

⬇

$\forall x_1 \ldots \forall x_s \; (A_1 \vee \ldots \vee A_k \text{ <- } C_1 \;\&\; \ldots \;\&\; C_n)$ $\& \cong \wedge$

⬇

Kurzschreibweise als „Regel“: $(\underbrace{A_1, \ldots, A_k}_{\text{Disjunktion von Atomen}} \text{ <- } \underbrace{C_1, \ldots, C_n}_{\text{Konjunktion von Atomen}})$

Universität Bielefeld

PROLOG: Hornklauseln!

Beschränkung auf Klauseln mit höchstens einem nicht-negierten Atom:

$\forall x_1 \ldots \forall x_s \; (A \vee \neg C_1 \vee \ldots \vee \neg C_n)$

⬇

$\forall x_1 \ldots \forall x_s \; (A \text{ <- } C_1 \;\&\; \ldots \;\&\; C_n)$

⬇

$(A \text{ <- } C_1, \ldots, C_n)$

Prozedurale Semantik: Um A zu zeigen, müssen die Cj gezeigt werden.

Axiome formulieren in PROLOG

Eine Definition von "Großvater":

Für alle x, für alle y, für alle z gilt:
x ist Großvater von y, wenn
x Vater von z ist und z Vater von y.

oder ganz formal:

$\forall x \forall y \forall z$ (grossvater x y)
$\leftarrow$ (vater x z)(vater z y)

Die entsprechende Definition "über die Mutter":

$\forall x \forall y \forall z$ (grossvater x y)
$\leftarrow$ (vater x z)(mutter z y)

*** "klassische" Clocksin/Mellish–Syntax:**

```
   grossvater(X Y)
:- vater(X Z),vater(Z Y)
```

In PROLOG gelten alle Variablen als implizit all-quantifiziert (die prozedurale Semantik erlaubt Ersetzungen durch beliebige in der Wissensbasis vorh. Terme); Quantoren werden weggelassen:

```
  (grossvater x y)
← (vater x z),(vater z y)
```

```
  (grossvater x y)
← (vater x z),(mutter z y)
```

Seien jetzt diese beiden Implikationen (Regeln) assertiert, zusammen mit folgenden Fakten:

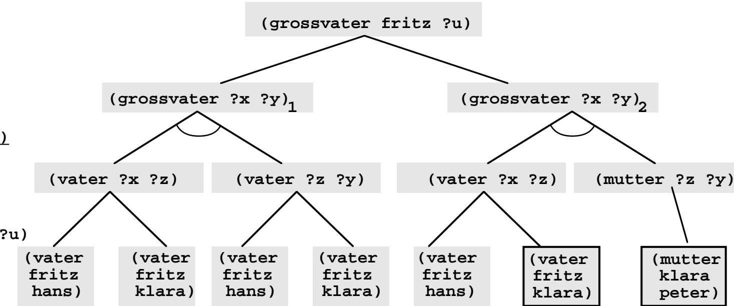
```
(vater fritz hans)
(vater fritz klara)
(mutter klara peter)
```

Zeige: Ist Fritz der Großvater von jemand?

```
>(grossvater fritz u)?
>(grossvater fritz peter)
```


Goal-Tree Suche in PROLOG

```
show:     (grossvater fritz ?u)
show:     (vater fritz ?z)
success:  (vater fritz hans)
show:     (vater hans ?y)
redo:     (vater hans ?y)
fail:     (vater hans ?y)
redo:     (vater fritz ?z)
success:  (vater fritz klara)
show:     (vater klara ?y)
redo:     (vater klara ?y)
fail:     (vater klara ?y)
redo:     (grossvater fritz ?u)
show:     (vater fritz ?z)
success:  (vater fritz hans)
show:     (mutter hans ?y)
fail:     (mutter hans ?y)
redo:     (vater fritz ?z)
success:  (vater fritz klara)
show:     (mutter klara ?y)
success:  (mutter klara peter)
success:  (grossvater fritz peter)
```



FRAGE: Was sind hier die Constraints an den UND-Knoten?

– **daß die Variablenbindungen gleich benannter Variablen in den Teillösungen auch tatsächlich gleich sind**

Übersicht

- Theorembeweisen
- Logisches Programmieren
- Regelbasiertes Programmieren

Regelbasiertes Programmieren

- ♦ **einer der Hauptansätze für die Erstellung größerer Systeme; Grundlage von Expertensystemen**
- ♦ **angebracht, wo Wissen über die Zusammenhänge zwischen Aufgabensituation und Aufgabenlösungen verfügbar ist**

Grundidee: jede Regel entspricht einem Stück Wissen des Typs

If <circumstances> then <do action, or conclude something>

"Was ist zu tun, wenn ..." "Was ist anzunehmen, wenn ..."

Produktionsregeln

„Condition-Action Statements“

$C_1 \; C_2 \; \ldots \; C_n \rightarrow A_1 \; A_2 \; \ldots \; A_m$

Wenn C_1 und C_2 ... und C_n gelten, dann führe A_1 und A_2 ... und A_m aus.

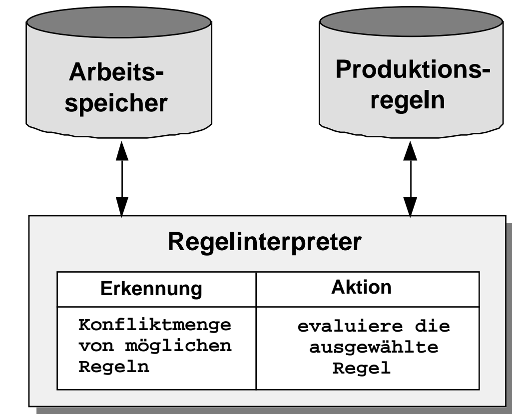
„führe aus“:

- nimm an, daß gilt (Implikation)
- ändere „Weltzustand“ (Aktion)

Aufbau eines Produktionssystems

Regeln sind modular ⇒ leichte Modifizierung einzelner Regeln möglich

Aufbau der Regeln entspricht einzelnen Wissenselementen der Domäne („Regelwissen“)



Recognize-Act-Cycle in OPS5

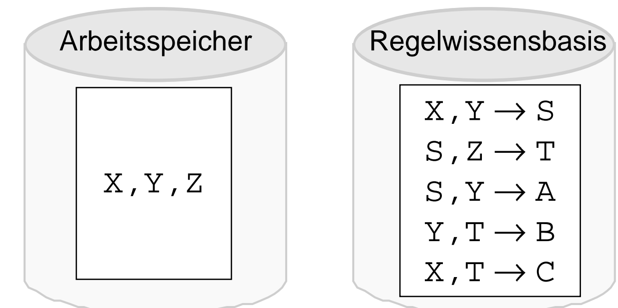
(analog in CLIPS)

1. „MATCH“: Prüfen sämtlicher Produktionsregeln gegen Working Memory Elements --> Konfliktmenge
2. „CONFLICT RESOLUTION“: Wähle eine Produktions-regel mit erfüllter Bedingungsseite aus. Bei leerer Konfliktmenge: Kontrolle an den Benutzer
3. „ACT“: Alle Aktionen der ausgewählten Produktionsregel ausführen
4. Falls HALT-Aktion: Kontrolle an den Benutzer; sonst: gehe zu 1.

Regelbasierte Problemlösung

- Abstrakt gesprochen besteht eine Problemlösung darin, durch Anwendung der Regeln ein bestimmtes Symbol (oder mehrere) in einem Arbeitsspeicher zu erzeugen.
- Beispiel. Sei gegeben:

Ziel:
Arbeits-speicher enthält C



Vorwärtsverkettender Interpreter

Komponenten:

- ◆ Datenbasis/Arbeitsspeicher
- ◆ Produktionsregeln

(1) DATA ← Ausgangsdatenbasis
(2) Until DATA erfüllt Terminierungskriterium do
(3) Begin
(4) Wähle eine anwendbare Regel R (deren Bedingungsteil durch DATA erfüllt ist)
(5) DATA ← Ergebnis der Anwendung des R-Aktionsteils auf DATA
(6) End

Lösung mit Vorwärtsverkettung

Voraussetzung über die Konfliktresolution:

- ◆ Ignoriere Regeln, die bereits im Arbeitsspeicher stehende Symbole nochmals eintragen würden
- ◆ Berücksichtige Regeln gemäß ihrer Reihenfolge im Regelspeicher

Arbeitsspeicher	angewendete Regel
X,Y,Z	X,Y → S
X,Y,Z,S	S,Z → T
X,Y,Z,S,T	S,Y → A
X,Y,Z,S,T,A	Y,T → B
X,Y,Z,S,T,A,B	X,T → C
X,Y,Z,S,T,A,B,C	

Lösung mit Rückwärtsverkettung

- ◆ nur solche Regeln werden ausgewählt, die das gesuchte Symbol in den Arbeitsspeicher (AS) schreiben
- ◆ sind sie (noch) nicht anwendbar → Teilziel: anwendbar machen

Arbeitsspeicher	Ziel: C
X,Y,Z	Regel: X,T → C
X,Y,Z	Teilziel 1: X (schon im AS)
X,Y,Z	Teilziel 2: T
X,Y,Z	Regel: S,Z → T
X,Y,Z	Teilziel 1: S
X,Y,Z	Regel: X,Y → S
X,Y,Z	Teilziel 1: X (im AS)
X,Y,Z,S	Teilziel 2: Y (im AS)
X,Y,Z,S,T,C	Teilziel 2: Z (im AS)

Programmierstile im Vergleich

Anweisungsbasierter Stil

- Programm = Sequenz von Befehlen und Abfragen
- Programmierer legt fest, was getan wird und in welcher Reihenfolge
- Kontrollfluß übersichtlich, aber starr

Regelbasierter Stil

- Programm = Menge von Regeln und Regelinterpreter
- festgelegt nur, was in einer Situation zu tun ist; Regel-interpreter bestimmt Abfolge
- Kontrollfluß flexibel, aber evtl. unübersichtlicher

Leseempfehlung heute:

- ◆ Charniak & McDermott, Kap. 6, 351-360; 437ff.

zu PROLOG:

- ◆ Görz, 2. Auflage: Kapitel 9.2 (nach Bedarf)