Logische Programmierung I

Logisches Paradigma

IBM Watson KI

- 100GB Text als Fakten
- Wörterbücher
- Enzyklopädien
    - Wikipedia
    - IMDB
    - …
- Bibel
- …

- 6-8 Sekunden Ausführungszeit
- Linguistischer Präprozessor DeepQA

https://www.youtube.com/watch?v=WFR3lOm_xhE

Ein kleines Rätsel…

Fünf Autos stehen nebeneinander auf einem Parkplatz. Jedes hat eine andere Farbe, ein anderes Fabrikat und kommt aus verschiedenen Städten. Die Besitzer haben jeweils einen unterschiedlichen Beruf und in jedem Auto liegt eine unterschiedliche CD.

- Der Ferrari ist rot.
- Dem Lehrer gehört das silbrige Auto.
- Im VW liegt eine Madonna-CD.
- Der BMW kommt aus München und steht neben dem blauen Auto.
- Das Auto aus Hamburg steht neben dem braunen Auto.
- Der Metzger hat eine Abba-CD in seinem Auto.
- Das Auto mit der Beatles-CD steht neben dem Auto des Lehrers.
- Das Auto aus Köln gehört dem Notar.
- Neben dem blauen Auto steht ein Smart.
- Der Ford gehört dem Schreiner.
- Das grüne Auto kommt aus Hamburg.
- Neben dem Auto aus Berlin steht das Auto des Bäckers.
- Das Auto mit der Eminem-CD ist das vierte auf dem Parkplatz.
- Neben dem Auto aus Stuttgart steht kein BMW.

Welche Person besitzt die Heino-CD ?

Fakten und Regeln

- Fakten und deren Regeln (auch Beziehungen/Zusammenhänge) als Wissensbasis

- Durch Auflösen dieser Regeln, Generierung neuer Fakten/ Regeln => neues Wissen

- Rückführung zahlreicher Probleme auf eine Anzahl von Fakten/Regeln und deren Auflösung:
  - Sudoku
  - Data Mining
  - Gesetze
  - …

Beispiel: Sudoku

3	4		8	2	6		7	1
		8				9		
7	6			9			4	3
	8		1		2		3	
	3						9	
	7		9		4		1	
8	2			4			5	9
		7				3		
4	1		3	8	9		6	2

Fakten: vorhandene Zahlen im Sudoku Feld

Regeln: Regeln des Sudoku-Spiels

Beispiel: Sudoku

3	4		8	2	6	**5**	7	1
		8				9		
7	6			9			4	3
	8		1		2		3	
	3						9	
	7		9		4		1	
8	2			4			5	9
		7				3		
4	1		3	8	9		6	2

Regel 0: Es kommen nur die natürlichen Zahlen von 1-9 vor

Regel 1: Jede Zahl kann in jeder Zeile nur ein mal vorkommen

→ 5 und 9 fehlen

Regel 2: Jede Zahl kann in jeder Spalte nur ein mal vorkommen

→ 1,2,4,5,6,7,8 fehlen

→ **Neuer Fakt: 5 an [6,0]**

Übersicht

Prozedural	Objekt-orientierung	Funktional	**Logisch**	Parallel
Prozeduren	*Objekte und OOP*	*Funktionen*	*Aussagenlogik*	*Parallelität*
Datentypen und Zeiger	*Fünf Konzepte der OOP*	*Funktionale Konzepte*	***Resolution und Unifikation***	*Modelle*
Speicher-verwaltung	*Anwendung der Konzepte*	*Streams & Lazy Evaluation*	*Regelbasiertes Programmieren*	*Mechanismen*
				Analyse
C	*Java*	*Scala*	***Prolog***	*Java*

Anwendung

Prolog

Dr.-Ing Christian Braune

Prolog - Programming in Logic

- Entwickelt 1971 von Alain Colmerauer

- Syntax definiert in ISO/IEC 13211-1
  - nichtsdestotrotz zahlreiche Varianten verfügbar
  - Verwendung von SWI-Prolog: http://www.swi-prolog.org/

- Definition von Fakten und Regeln

- Anfragen zur Auflösung von Regeln und somit Generierung neuer Fakten

Fakt:
```
informatikstudent(aron).
```
Regel:
```
inVorlesung(pgp, X):-informatikstudent(X).
```
Frage:
```
?- inVorlesung(pgp, aron).
```

Prolog – Konventionen I

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron).
```

Alle Aussagen werden mit einem Punkt beendet (.)

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron).
```

Atomare Ausdrücke

- Bezeichner (klein): `aron, pgp, …`
- Zeichenketten: `'Hello World!'`
- Zahlen: `1337, 3.14`

Dr.-Ing Christian Braune

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron).
```

Variablen

- Bezeichner (Groß): `X, Y, Variable`
- Bezeichner(_ Prefix): `_x1, _x2`

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron).
```

Fakten

- Klauseln ohne rechte Seite (oder Regeln ohne Vorbedingung):
  - `informatikstudent(aron).`
  - `informatikstudent(X).`
  - `mensch(aron).`
  - `verheiratet(eva, adam).`

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron).
```

Regeln/Axiome

- Klauseln mit rechter Seite (Vorbedingung) und linker Seite (Produktion)
- Rechte und linke Seite werden durch `:-` getrennt
  - `inVorlesung(pgp, X) :- informatikstudent(X).`
  - `verheiratet(X, Y) :- ehemann(X, Y).`
  - `verheiratet(X, Y) :- ehefrau(X, Y).`

```
inVorlesung(pgp, X):-informatikstudent(X).
```

Regeln/Axiome

- Entsprechen Hornklauseln (Head ← Body)
  - $H \leftarrow B_1, B_2, \dots, B_n$
    - Wenn alle B_i $\forall i: 1 \leq i \leq n$ wahr, dann H ebenfalls war
  - `,` = logische Konjunktion
  - `;` = logische Disjunktion
  - `not` = logische Negation

  - `verheiratet(X,Y) :- ehefrau(X,Y);ehemann(X,Y).`

Prolog – Konventionen VII

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron)
```

Anfragen

- `?-` impliziert eine Anfrage
- Anfrage ohne Variablen geben `true` oder `false` zurück
- Anfrage mit Variablen geben mögliche Belegung(en) oder `false` zurück

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, aron)
```

`true`

Ergebnis

```
informatikstudent(aron).
inVorlesung(pgp, X):-informatikstudent(X).

?- inVorlesung(pgp, Y)
```

?

Wie funktioniert ein Prolog Compiler?

Unifikation und Resolution

Aussagenlogische Resolution

Eine Klausel R heißt Resolvente der Klauseln C_1 und C_2, wenn es einen atomaren Satz A gibt, mit $A \in C_1$ and $(\neg A) \in C_2$ und

$$R = (C_1 \setminus \{A\}) \cup (C_2 \setminus \{\neg A\}).$$

Algorithmus (Resolution)

Gegeben: *Menge $\mathcal{S}$ von Klauseln.*

1. *Bilde (falls möglich) aus zwei Klauseln von $\mathcal{S}$ eine Resolvente und füge sie zu $\mathcal{S}$ hinzu.*
2. *Wiederhole Schritt 1 solange als möglich.*

Falls die leere Klausel $\Box$ zu $\mathcal{S}$ hinzugefügt wurde, ist die ursprüngliche Satzmenge $\mathcal{S}$ nicht wt-erfüllbar, andernfalls wt-erfüllbar.

Beispiel einer Resolution

Gilt folgende Folgerung?

$$\{B \vee C \vee B\ ,\ C \to \neg D\ ,\ (A \vee D) \wedge (B \to \neg D)\} \overset{?}{\models}_T A$$

Umgeformt in KNF:

$$\{B \vee C \vee B\ ,\ \neg C \vee \neg D\ ,\ (A \vee D) \wedge (\neg B \vee \neg D)\} \overset{?}{\models}_T A$$

Wir erhalten folgende Menge von Klauseln:

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

Anwenden des Resolutionsverfahrens:

$$\dfrac{\dfrac{\{A, D\} \quad \{\neg A\}}{\{D\}} \qquad \dfrac{\dfrac{\{B, C\} \quad \{\neg C, \neg D\}}{\{B, \neg D\}} \qquad \{\neg B, \neg D\}}{\{\neg D\}}}{\square}$$

Die Klauselmenge ist unerfüllbar, und die Folgerung ist gültig.

Beispiel einer Resolution

Gilt folgende Folgerung ?

$$\{(B \vee C \vee B) \wedge (A \vee C \vee D), (\neg B \vee \neg D)\} \overset{?}{\models}_T A$$

Wir erhalten folgende Menge von Klauseln:

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

Anwenden des Resolutionsverfahrens:

$$\dfrac{\dfrac{\{A, C, D\} \quad \{\neg A\}}{\{C, D\}} \qquad \dfrac{\{B, C\} \quad \{\neg B, \neg D\}}{\{C, \neg D\}}}{C}$$

Es sind jetzt nur noch Obermengen der bereits abgeleiteten Klauseln ableitbar (d.h. die Klauselmenge ist saturiert).
Die Klauselmenge ist daher erfüllbar, und die Folgerung ist nicht gültig.

Korrektheit und Vollständigkeit

Theorem

Der Resolutionsalgorithmus ist korrekt und vollständig, d. h. für eine gegebene Menge $\mathcal{S}$ von Klauseln lässt sich durch den Resolutionsalgorithmus genau dann die leere Klausel $\Box$ herleiten, wenn $\mathcal{S}$ nicht wt-erfüllbar ist.

Damit erhalten wir ein alternatives korrektes und vollständiges Beweisverfahren für die tautologische Folgerung:

$$\mathcal{J} \models_T S$$

gilt genau dann, wenn mittels Resolution die leere Klausel $\Box$ aus der Klauselmenge für $\mathcal{J} \cup \{\neg S\}$ hergeleitet werden kann.

Definition 7.1.1. Eine Klausel ist eine endliche Menge von Literalen.

Beispiele:

$$C_1 = \{A, B, C\}$$
$$C_2 = \{\neg A, D\}$$
$$C_3 = \emptyset \quad \text{(auch bezeichnet mit } \square\text{)}$$

Jede Menge $\mathcal{J}$ von Sätzen in KNF kann durch eine äquivalente Menge $\mathcal{S}$ von Klauseln ersetzt werden, dabei entspricht jede Disjunktion eines Satzes aus $\mathcal{J}$ einer Klausel.

Definition 7.1.2. Eine Klausel R heißt Resolvente der Klauseln C_1 und C_2, wenn es einen atomaren Satz A gibt, mit $A \in C_1$ and $(\neg A) \in C_2$ und

$$R = (C_1 \setminus \{A\}) \cup (C_2 \setminus \{\neg A\}).$$

Anfragen an Wissensbasis I

```
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
divers(quinn).
divers(kaya).
vater_von(steffen, fritz).
vater_von(fritz, karin).
vater_von(steffen, lisa).
vater_von(paul, maria).
mutter_von(karin, maria).
mutter_von(kaya, paul).

elternteil_von(X, Kind)
   :- vater_von(X, Kind).
elternteil_von(X, Kind)
   :- mutter_von(X, Kind).
```

```
?-maennlich(steffen).
```


Dr.-Ing Christian Braune

Match von Fakten und Regeln

Match

- Vergleichen von Anfragen mit Fakten und Regeln aus der Wissensbasis
- Beispiel:
  - Fakt: `C`
  - Query: `?-C` liefert `true`
- Sequentieller Vergleich von Fakten/Regeln und Abbruch beim ersten Treffer (vgl. Pattern Matching)
- **Annahme:** geschlossenen Wissensbasis
  - „Was nicht da ist, existiert auch nicht“

Dr.-Ing Christian Braune

Anfragen an Wissensbasis II

```
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
divers(quinn).
divers(kaya).
vater_von(steffen, fritz).
vater_von(fritz, karin).
vater_von(steffen, lisa).
vater_von(paul, maria).
mutter_von(karin, maria).
mutter_von(kaya, paul).

elternteil_von(X, Kind)
    :- vater_von(X, Kind).
elternteil_von(X, Kind)
    :- mutter_von(X, Kind).
```

```
?-maennlich(steffen).

→ maennlich(paul) =
  maennlich(steffen)?
      false!

→ maennlich(fritz) =
  maennlich(steffen)?
      false!

→ maennlich(steffen)=
  maennlich(steffen)?
      true!
← true
```


Dr.-Ing Christian Braune

Anfragen an Wissensbasis III

```
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
divers(quinn).
divers(kaya).
vater_von(steffen, fritz).
vater_von(fritz, karin).
vater_von(steffen, lisa).
vater_von(paul, maria).
mutter_von(karin, maria).
mutter_von(kaya, paul).

elternteil_von(X, Kind)
   :- vater_von(X, Kind).
elternteil_von(X, Kind)
   :- mutter_von(X, Kind).
```

```
?- elternteil_von(fritz, Y).
```

- Aussagenlogik – Wir untersuchen, ob Sätze wahr oder falsch sind bzw. ob sie aus einfacheren Sätzen zusammengesetzt sind
  - Beispiele:

    Lukas ist ein einmaliger Name.

    Es regnet blaue Tiger und das Mensaessen schmeckt heute besonders gut.

- Prädikatenlogik – Wir untersuchen die Struktur von Sätzen und erlauben Variablen. Einsetzen von Werten in diese Variablen erzeugt wahre oder falsche Aussagen:
  - Beispiele:

    X ist ein einmaliger Name.

    Es regnet Y und das Mensaessen schmeckt heute besonders Z.

- Terme wie `maennlich(steffen).` können noch als aussagenlogische Sätze aufgefasst werden

- Andere Terme nicht mehr:
  - `elternteil_von(fritz, Y).`

- Diese Aussage können wir auch umschreiben als: *"Wenn ein Y existiert, für das `elternteil_von(fritz, Y).` eine wahre Aussage ist, welchen Wert hat dann Y?"*

- Prolog hilft uns dabei beide Teilfragen effizient zu beantworten!

Dr.-Ing Christian Braune

- Notwendigkeit des Findens möglicher Belegungen
- Unifikation als Hilfsmittel
  - Finden von Substitutions-/Ersetzungsschritten von zwei prädikatenlogischen Termen, sodass beide Terme gleich werden
  - Liste an Substitutionsschritten = Unifikator
- Beispiel:
  - $A_1 = h(X, f(a), Y)$
  - $A_2 = h(a, Z, b)$
  - Ersetze wie folgt: $[X \leftarrow a; Y \leftarrow b; Z \leftarrow f(a)] \Rightarrow \sigma$ (Unifikator)
  - $\sigma(A_1) = \sigma(A_2) = h(a, f(a), b)$
- Allgemeinster Unifikator = Unifikator mit den wenigsten Substitutionsschritten

Unifizierbarkeit

Zwei prädikatenlogische Terme s und t sind unifizierbar, wenn:

1. s und t den gleichen Wert haben
2. s eine Variable und t ein Ausdruck ist, indem s nicht vorkommt
3. s und t beide Variablen sind, oder
4. s und t komplexe Terme sind, wobei
   - die Funktoren gleich sind, und
   - die Funktoren die gleiche Stelligkeit besitzen, und
   - die Argumente der Funktoren aus s und t paarweise unifizierbar sind

Funktor?

- Ein Funktor ist eine strukturerhaltende Abbildung zwischen zwei Kategorien.
- Seien C, D Kategorien. Ein (kovarianter) Funktor ist eine Abbildung $F\colon\ C \to D$, die:
  - Objekte auf Objekte abbildet: $F\colon\ Ob(C) \to Ob(D)$
  - Morphismen auf Morphismen abbildet: seien X, Y, Z Objekte in C, dann gilt ..., so dass
    - $F(id_X) = id_{F(X)}$
    - $F(g \circ f) = F(g) \circ F(f)$ für alle Morphismen $f\colon X \to Y$ und $g\colon Y \to Z$.

[Quelle: Wikipedia]

Funktoren

- Funktoren sind für logische Programmierung das, was Funktionen höherer Ordnung für funktionale Programmierung sind.

- Funktoren sind Terme
- Funktoren können Terme als Parameter haben
- Funktoren können Terme als Rückgabewert haben

- Schreibweise: `inVorlesung/2`

  - `inVorlesung` ist ein binärer (zweistelliger) Funktor

Anfragen an Wissensbasis IV

```
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
divers(quinn).
divers(kaya).
vater_von(steffen, fritz).
vater_von(fritz, karin).
vater_von(steffen, lisa).
vater_von(paul, maria).
mutter_von(karin, maria).
mutter_von(kaya, paul).

elternteil_von(X, Kind)
    :- vater_von(X, Kind).
elternteil_von(X, Kind)
    :- mutter_von(X, Kind).
```

```
?- elternteil_von(fritz, Y).

Unifizierbar?
A1: elternteil_von(fritz, Y)
A2: elternteil_von(X, Kind)
    :- vater_von(X, Kind).
```

Ja mit σ_1 = [X← fritz, Kind←Y]

```
→ vater_von(fritz, Y)
Unifizierbar?
A1: vater_von(fritz, Y)
A2: vater_von(steffen, fritz)
        nicht unifizierbar!!
Unifizierbar?
→ vater_von(fritz, Y) =
  vater_von(fritz, karin)?
        unifizierbar
```

Ja mit σ_2 = [Y←karin]

```
        → Y = karin
```

Beobachtungen

- Speicherung der Ersetzungen im Unifikator
- mehrere unterschiedliche Ersetzungen möglich
- nicht-deterministische Auswahl der Regel zum Unifizieren (sehr ineffizient)
- daher zur Anfragenbewältigung via SLD-Resolution

SLD-Resolution

- **S**elektionsfunktion
  - Auswahl passender Literale zur Resolventenbildung
  - In Prolog Tiefensuche (!)
- **L**ineare Resolution
  - Resolventenbildung zwischen neuer Klausel und der zuletzt erzeugten Resolvente
- **D**efinite Klauseln
  - Hornklausen mit genau einem nicht-negativen Literal
  - Fakten/Regeln ohne Negation sind definite Klauseln, da:

$$\mathtt{A :- A_1, A_2, \ldots, A_n}$$

$$\leftrightarrow A_1 \land A_2 \land \ \ldots \land A_n \rightarrow A$$
$$\leftrightarrow \neg A_1 \lor \neg A_2 \lor \ \ldots \lor \neg A_n \lor A$$

Idee von Prolog

- Darstellung der Wissensbasis als definite Klauseln bzw. Faktenklauseln
- Negiere die Anfrage (Gegenbeweis)
- Führe schrittweise die Resolventenbildung im Abgleich mit der Wissensbasis durch
- Schlussfolgerung von `false?` Dann Anfrage erfolgreich
- Speicherung der Substitutionen im Unifikator

SLD-Resolution - Definite Klauseln

```
maennlich(paul).
maennlich(fritz).
maennlich(steffen).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
divers(quinn).
divers(kaya).
vater_von(steffen, fritz).
vater_von(fritz, karin).
vater_von(steffen, lisa).
vater_von(paul, maria).
mutter_von(karin, maria).
mutter_von(kaya, paul).

elternteil_von(X, Kind)
   :- vater_von(X, Kind).
elternteil_von(X, Kind)
   :- mutter_von(X, Kind).
```

```
{{maennlich(paul)},
 {maennlich(fritz)},
 {maennlich(steffen)},
 {weiblich(karin)},
 {weiblich(lisa)},
 {weiblich(maria)},
 {divers(quinn)},
 {divers(kaya)},
 {vater_von(steffen, fritz)},
 {vater_von(fritz, karin)},
 {vater_von(steffen, lisa)},
 {vater_von(paul, maria)},
 {mutter_von(karin, maria)},
 {mutter_von(kaya, paul)},
 {elternteil_von(X, Kind),
  ¬vater_von(X, Kind)},
 {elternteil_von(X, Kind),
  ¬mutter_von(X, Kind)}}
```

Notation: {A,B} := A v B

Dr.-Ing Christian Braune

SLD-Resolution - Anfrage negieren

- `?- elternteil_von(fritz, karin).`
  ➔ `elternteil_von(fritz, karin):- true.`
  ➔ `{¬elternteil_von(fritz, karin)}`

- dann Resolventenbildung bis
  - leere Menge erreicht (Widerspruch zum Gegenbeweis also Erfolg der Anfrage) oder
  - bis keine Resolventenbildung mehr möglich (Misserfolg)

Notation: {A,B} := A v B

SLD-Resolution – Beispiel I

{{maennlich(paul)}, {maennlich(fritz)}, {maennlich(steffen)}, {weiblich(karin)}, {weiblich(lisa)}, {weiblich(maria)}, {divers(quinn)}, {divers(kaya)} {vater_von(steffen, fritz)}, {vater_von(fritz, karin)}, {vater_von(steffen, lisa)}, {vater_von(paul, maria)}, {mutter_von(karin, maria)}, {mutter_von(kaya, paul)}, {¬ vater_von(X, Kind), elternteil_von(X, Kind)}, {¬ mutter_von(X, Kind), elternteil_von(X, Kind)}}

{ **¬elternteil_von(fritz, karin)**}

{¬vater_von(X, Kind), **elternteil_von(X, Kind)**}

σ = [X←fritz, Kind←karin]

{ **¬vater_von(fritz, karin)**}

{**vater_von(fritz, karin)**}

□

Leere Menge ←→ false

➔ Anfrage erfolgreich!

Notation: {A,B} := A v B

SLD-Resolution – Beispiel II

{{maennlich(paul)}, {maennlich(fritz)}, {maennlich(steffen)}, {weiblich(karin)}, {weiblich(lisa)}, {weiblich(maria)}, {divers(quinn)}, {divers(kaya)} {vater_von(steffen, fritz)}, {vater_von(fritz, karin)}, {vater_von(steffen, lisa)}, {vater_von(paul, maria)}, {mutter_von(karin, maria)}, {mutter_von(kaya, paul)}, {¬ vater_von(X, Kind), elternteil_von(X, Kind)}, {¬ mutter_von(X, Kind), elternteil_von(X, Kind)}}

{ **¬elternteil_von(karin, paul)** }

{¬vater_von(X, Kind), **elternteil_von(X, Kind)**}

σ = [X←karin, Kind←paul]

{ **¬vater_von(karin, paul)** }

□

Probiere nächste Regel

{ **¬elternteil_von(karin, paul)** }

{¬mutter_von(X, Kind), **elternteil_von(X, Kind)**}

σ = [X←karin, Kind←paul]

{ **¬mutter_von(karin, paul)** }

□

Keine weitere Regel - Misserfolg

SLD-Resolution – Beispiel III

```
{{maennlich(paul)}, {maennlich(fritz)}, {maennlich(steffen)},
{weiblich(karin)}, {weiblich(lisa)}, {weiblich(maria)},
{divers(quinn)}, {divers(kaya)} {vater_von(steffen, fritz)},
{vater_von(fritz, karin)}, {vater_von(steffen, lisa)},
{vater_von(paul, maria)}, {mutter_von(karin, maria)}, {mutter_von(kaya,
paul)}, {¬ vater_von(X, Kind), elternteil_von(X, Kind)},
{¬ mutter_von(X, Kind), elternteil_von(X, Kind)}}
```

{ **¬elternteil_von(fritz, Y)**}   {¬vater_von(X, Kind), **elternteil_von(X, Kind)**}

σ_1 = [X←fritz, Kind←Y]

{**¬vater_von(fritz, Y)**}   {**vater_von(fritz, karin)**}

σ_2 = [Y←karin]

□

← Y = karin

Suche weitere Belegung…

SLD-Resolution – Beispiel III

```
{{maennlich(paul)}, {maennlich(fritz)}, {maennlich(steffen)},
{weiblich(karin)}, {weiblich(lisa)}, {weiblich(maria)},
{divers(quinn)}, {divers(kaya)} {vater_von(steffen, fritz)},
{vater_von(fritz, karin)}, {vater_von(steffen, lisa)},
{vater_von(paul, maria)}, {mutter_von(karin, maria)}, {mutter_von(kaya,
paul)}, {¬ vater_von(X, Kind), elternteil_von(X, Kind)},
{¬ mutter_von(X, Kind), elternteil_von(X, Kind)}}
```

{ **¬elternteil_von(fritz, Y)**} — {¬vater_von(X, Kind), **elternteil_von(X, Kind)**}

σ_1 = [X←fritz, Kind←Y]

{**¬vater_von(fritz, Y)**} □

← keine weitere Regel auf diesem Zweig anwendbar

{ **¬elternteil_von(fritz, Y)**} — {¬mutter_von(X, Kind), **elternteil_von(X, Kind)**}

σ_2 = [X←fritz, Kind←Y]

{**¬mutter_von(fritz, Y)**} □

← keine weitere Regel anwendbar

Dr.-Ing Christian Braune

SLD-Resolution – Auswahl von Regeln

- Zwei mögliche Heuristiken:

  - Breitensuche:
    - Wende alle anwendbaren Regeln an
    - Wenn kein Gegenbeispiel gefunden, mache mit Ergebnissen weiter

  - Tiefensuche (Prolog):
    - Wende die erste Regel solange an, bis Gegenbeispiel gefunden
    - Wenn kein Gegenbeispiel gefunden, gehe eine Anwendung zurück und wende die nächste Regel an

      → Backtracking

Auswahl von Regeln: Breitensuche

```
1. p(X,Z) :- p(X,Y),q(Y,Z).
2. p(X,X).
3. q(0,1).
```

➡

```
{p(X,Z),¬p(X,Y),¬q(Y,Z)}
{p(X,X)}
{q(0,1)}
```

Anfrage ?-`p(1,1).`

{¬p(1,1)}

Klausel 1:
σ=[X←1, Z ← 1]

{¬p(1,Y), ¬q(Y,1)}

Klausel 2:
σ=[X←1]

□

➔ Leere Menge
➔ Aussage wahr

, = Konjunktion   , = Disjunktion

Auswahl von Regeln: Tiefensuche

```
1. p(X,Z) :- p(X,Y),q(Y,Z).
2. p(X,X).
3. q(0,1).
```

$\Rightarrow$

```
{p(X,Z),¬p(X,Y),¬q(Y,Z)}
{p(X,X)}
{q(0,1)}
```

Anfrage ?-`p(1,1).`

{¬ p(1,1)}

Klausel 1:
$\sigma = [X \leftarrow 1,\ Z \leftarrow 1]$

{**¬p(1,Y)**, **¬q(Y,1)**}

Klausel 1:
$\sigma = [X \leftarrow 1,\ Z \leftarrow Y]$

{**¬p(1,Y)**, **¬q(Y,Y)**, **¬q(Y,1)**}

...

Terminiert nicht!

Breiten- vs. Tiefensuche

- Breitensuche:
  - ➢ Je höher die Verschachtelungstiefe bis zum ersten Ergebnis, desto länger dauert die Suche
  - ➢ exponentielles Wachstum des Speicherbedarfs
- Tiefensuche:
  - Möglichkeit, Ergebnis nach wenigen Schritten zu finden
  - aber: Gefahr der Endlosrekursion

- aus Performanzgründen in SWI-Prolog Tiefensuche

Daher wichtig:
Reihenfolge der Regeln in Prolog beachten

Backtracking

- Prolog arbeitet Fakten und Regeln von oben nach unten ab
- Schlägt eine Lösungssuche fehl, Anwendung der nächsten möglichen Regel

- Backtracking = Fortsetzung der Lösungssuche beginnend am letzten Alternativpunkt
- alle ab diesem Punkt substituierten Variablen werden wieder frei

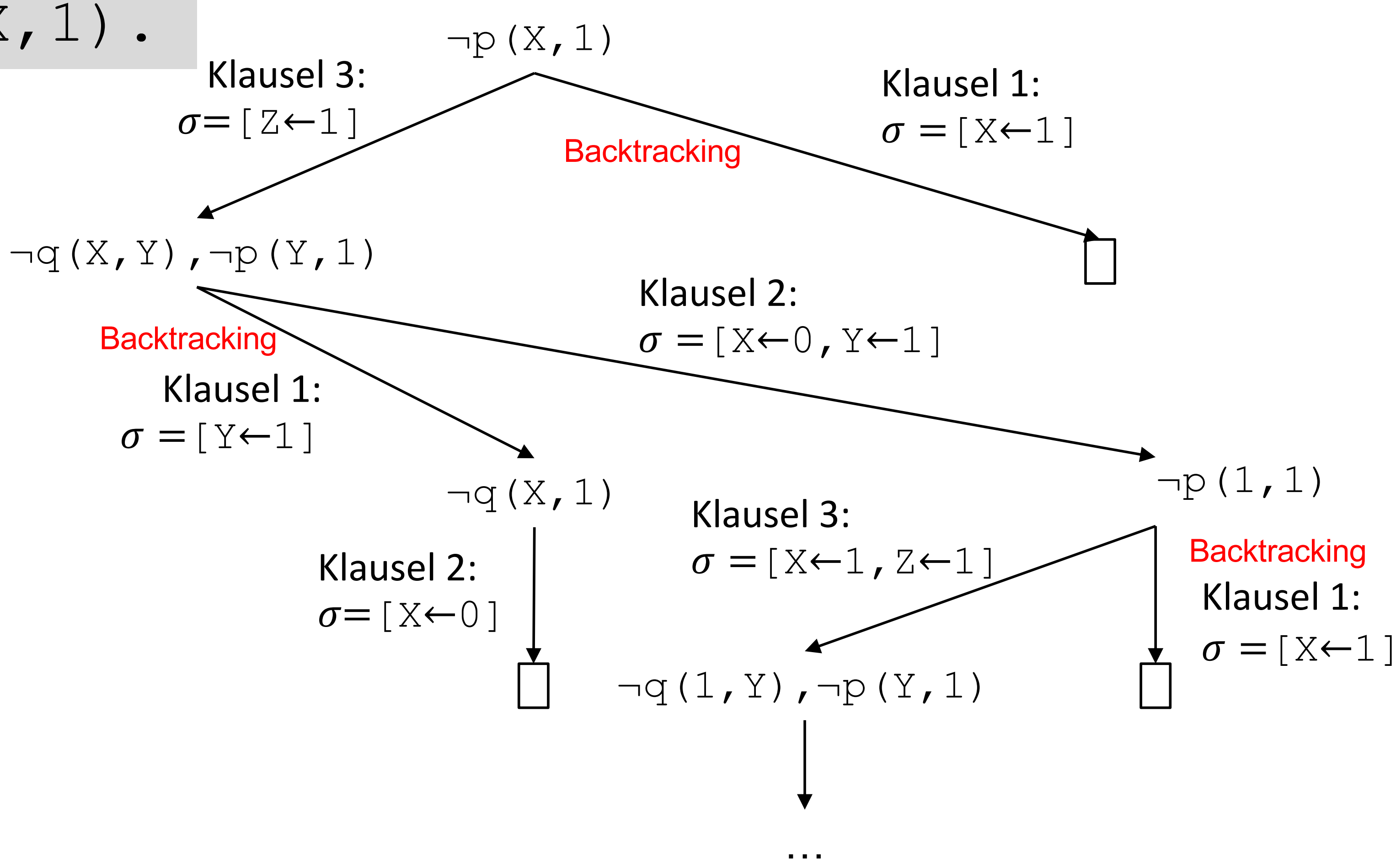
→ Systematisches „Ausprobieren“

Backtracking - Beispiel

```
1. p(X,X).
2. q(0,1).
3. p(X,Z) :- q(X,Y),p(Y,Z).
```

```
{p(X,X)}
{q(0,1)}
{p(X,Z),¬q(X,Y),¬p(Y,Z)}
```

Anfrage ?-`p(X,1).`



Systematisch zum Prolog-Programm I

1. Anlegen einer Wissensbasis/-datenbank

2. Anlegen von Beziehungsregeln

   - Spezialfälle vor generellen Fällen

   - „Generiere und Teste“-Ansatz

Dr.-Ing Christian Braune

Systematisch zum Prolog-Programm II

1. Anlegen einer „Wissensbasis“ bzw. Datenbank

```
king('Edward VII').
king('Edward VIII').
queen('Elizabeth II').
…
```

- Sehr aufwändig:
- Einführung von Listen:

```
kingList(['King Edward VII',
        'King Edward VIII','King George V',…]).
queenList(['Elizabeth II',
          'Victoria', 'Freddie Mercury', ...]).
```

Listen - Initialisierung

- Liste mit (vier festen) Personennamen
  - `[max, paula, susi, tom]`
- Listen können auch Variablen enthalten für Elemente, die noch bestimmt werden müssen
- Liste mit vier noch unbekannten Personennamen
  - `[P1, P2, P3, P4]`
- Liste mit (zwei) bekannten und (zwei) noch offenen Namen
  - `[max, FreundinVonMax, susi, FreundVonSusi]`

Listen: Head und Tail

- spezielle Notation erlaubt, Listen mit variablen Listenresten darzustellen
  - Syntax: `[Kopf | Rest]`
  - Beispiel: head, tail

```
head([H|_], H).
tail([], []).
tail([_|R], R).
```

```
?-head([max, paula, susi, tom], X).
X=max.
?-tail([max, paula, susi, tom], X).
X = [paula, susi, tom].
?-head([max], X).
X = max.
?-tail([max], X).
X = [].
```

Listen: weitere Beispiele

- allgemeinere Darstellungen Notation für variable Listenreste
  - Syntax: `[Elem1, Elem2, ... , ElemN | Rest]`
  - Semantik: unifizierbar mit einer Liste mit N Elementen, die jeweils mit den Variablen oder Konstanten `Elem1, Elem2, ... , ElemN` unifizieren, und einer (evtl. leeren) Restliste beliebiger Länge

- Beispiel: „die ersten Drei“

```
threeHeads([H1,H2,H3|_],H1,H2,H3).
```

```
?- threeHeads([max,paula,susi| Weitere],A,B,C).
   A = max,
   B = paula,
   C = susi.
```

Listen: weitere Beispiele

- eine Liste mit mind. vier und evtl. weiteren Elementen

```
?- threeHeads([Elem1,Elem2,Elem3,Elem4|Rest],A,B,C).
   A = Elem1,
   B = Elem2,
   C = Elem3.
```


Dr.-Ing Christian Braune

Systematisch zum Prolog-Programm III

2. Anlegen von Beziehungsregeln

- Durch Verknüpfung von einfachen Regeln komplexere Strukturen abbildbar

- Beispiel:

```
grosseltern(X,Y) :- eltern(X,Z), eltern(Z,Y).
```

Systematisch Beziehungsregeln erstellen I

Ein Beispiel:

Zwei Listen sollen durch eine Funktion `append` zu einer dritten Liste zusammengefügt werden

Wie sieht eine dazugehörige Regel aus?

Dr.-Ing Christian Braune

Beziehungsregeln – Zentrale Idee

Jede Funktion z=F(x,y,...) kann auch als Relation R(z,x,y) aufgefasst werden

Systematisch Beziehungsregeln erstellen II

- Erster Schritt: **Was soll die Regel prinzipiell tun?**

- Im Beispiel:
  - Wann liegt zwischen drei beliebigen endlichen Listen `List1`, `List2` und `List3` die (dreistellige) Relation `append` vor?

  - die Relation `append` soll dabei

    - **genau dann** zwischen `List1`, `List2` und `List3` vorliegen,
    - **wenn** List3 sich durch Aneinanderhängen von `List1` und `List2` ergibt

Systematisch Beziehungsregeln erstellen III

- Zweiter Schritt: **Spezialfälle ermitteln**

- Im Beispiel:
  - falls `List1` leer, dann liegt Relation `append` für beliebige Listen `List2` gleich `List3` ist

  - als Prolog-Fakt: `append([], List, List).`

  - Beachte: Verwendung der selben Variable innerhalb der Klausel stellt Unifizierbarkeit dar
    - Alternative: `append([], List2, List3):- List2 = List3.`

Systematisch Beziehungsregeln erstellen IV

- Dritter Schritt: **Normalfall definieren**

- Im Beispiel:
  - falls `[H1|R1]` nicht leer (d.h. setzt sich aus Kopf `H1` und beliebigen Rest `R1` zusammen),
  - dann liegt Relation `append` zwischen
    - `[H1|R1]`
    - einer beliebigen Liste `List2` und
    - einer Liste aus `H1` als Kopf und `Z` als Rest
  - vor, wenn `append` zwischen `R1`, `List2` und `Z` vorliegt

  - In Prolog: `append([H1|R1], List2, [H1|Z]) :- append(R1, List2, Z).`

  - Warum?
    - Rekursives Entfernen eines Elementes aus `List1` und `List3`, bis `List1` leer ist
    - Der Rest in `List3` muss dann gleich der zweiten Liste ein (Spezialfall)

append - Zusammenfassung

- Regel `append` folgendenermaßen definiert:

```
append([], List, List).
append([H1|R1], List2, [H1|Z]) :- append(R1, List2, Z).
```

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


Dr.-Ing Christian Braune

- Listen in Kombination mit Rekursion erlaubt in fast allen Fällen das effiziente Beschreiben von Zusammenhängen

- Andere Anwendungen:
  - Schneller Aufbau einer Wissensbasis
    - `member(Element,List).`
    - `member(2,[1,2,3]).` ergibt `true`
  - Kombination von Attributen zu einem Objekt
    - `house(Owner,Pet,Car,Nationality).`
    - Gerade in Kombination mit der unassigned Variable sehr sinnvoll

Beispiel `reverse` - I

Weiteres Beispiel:

Eine Relation `reverse`, die eine Liste umkehrt.

```
?- reverse([a,b,1],[1,b,a]).
```

- Erster Schritt: **Aufstellen einiger elementarer Fälle**

  - `reverse([],[]).`
  - `reverse([X],[X]).`
  - `reverse([X,Y],[Y,X]).`
  - `reverse([X,Y,Z],[Z,Y,X]).`

- Zweiter Schritt: **Allgemeines Vorgehen ableiten**

  - Wir nehmen das vorderste Element und hängen es hinten an eine neue Liste

  - Regel für den allgemeinen Fall lautet also
    - `reverse([Head|Rest], Result) :- …`

  - Was wissen wir über `Result`?
    - `Head` muss als erstes Element nun an letzter Stelle stehen
    - `append(...,[Head],Result)`

  - Aber woran?

Beispiel `reverse` - IV

- An den Rest der Liste, nachdem er umgedreht wurde
  - `reverse(Rest,`**`RevRest`**`),`
    `append(`**`RevRest`**`,[Head],Result).`
  - Komplettes Programm

```
reverse([],[]).
reverse([Head|Rest],Result) :-
        reverse(Rest,RevRest),append(RevRest,[Head],Result).
```

```
?- reverse([1,2,3,4],X).
X=[4,3,2,1].
```

Beispiel `palindromic` - I

Ein letztes Beispiel:

Eine Liste soll dahingehend überprüft werden, ob sie ein Palindrom ist

```
?- palindromic([a,b,c,c,b,a]).
```

Beispiel `palindromic` - II

- Zerlegen der Liste in zwei Teile
  - `append(Part1,Part2,List)`

- Vergleich der beiden Teile
  - `reverse(Part1,Part2)`

```
palindromicEven(List) :- append(Part1,Part2,List),
                         reverse(Part1,Part2).
```

- Aber: gilt bisher nur von Listen mit gerade Länge

→ Einführung eines beliebigen mittleren Elements

```
palindromicOdd(List) :- append(Part1,[_|Part2],List),
                        reverse(Part1,Part2).
```

„Generiere und Teste“ - Ansatz

```
palindromic(List) :- append(Part1,[_|Part2],List),
                     reverse(Part1,Part2).
```

- Erster Schritt: Generiere alle möglichen Lösungen
  - `append` definiert zwei beliebige Teillisten als List
- Zweiter Schritt: Überprüfe (Teste), ob die generierte Lösung stimmt
  - `reverse` testet, ob die zweite Liste die umgedrehte erste Liste ist
- allgemeiner Lösungsansatz „Generiere und Teste“
- nicht performant, aber in Prolog häufig sehr sinnvoll

Zusammenfassung

- Prolog ist eine Logische Programmiersprache
- Programmieren in Prolog erfordert ‘Denken in Relationen’

- Die gesamte Funktionsweise ist auf Relationen zwischen atomaren Ausdrücken und Fakten ausgelegt
- Durch Kombination lassen sich komplexe Beziehungen beschreiben

- Auswertung des Programms basiert auf 2 Mechanismen:
  - Unifikation
  - Resolution

- Prolog geht von der Korrektheit des Programmes aus
  - Keine Fehlerbehandlung bei falschen Klauseln/inkonsistenten Wissensbasen/Endlosschleifen

Zusammenfassung Syntax

- Atomare Ausdrücke (**klein** beginnende Wörter, Zahlen, Zeichenketten)
- Variablen (**groß** beginnende Wörter, _ als Präfix)
  - Gültigkeitsbereich ist immer die Klausel, in der sie verwendet wird
  - keine Typendeklaration; Typüberprüfung nur zur Laufzeit
  - Instanziierung durch Unification
  - Spezielle Variable: „_“ – unassigned Variable
- Klauseln aus der Zusammensetzung von Atomen und Variablen
  - Fakten (Klauseln ohne Vorbedingungen)
  - Regeln (Klauseln mit Vorbedingungen)
- Listen (zusammengesetztes Element aus Atomen, Variablen und Klauseln)
- Queries
  - Wenn variablenfreie Abfrage, gibt true oder false zurück
  - Wenn Abfrage mit Variable, gibt Belegungen der Variable zurück

- **Def. Unifikation**
- Mit Hilfe von Unifizierung aus einem Fakt und einer Regel einen neuen Fakt erzeugen
- Beispiel:

Regel 1: $C(X) \leftarrow A(X)$
Regel 2: $A(value)$

Unifikation: $C(value)$

- Unifikation „generiert" neues Wissen durch Unifizierung