# Funktionales Programmieren
# (Praktische Informatik 3)

Berthold Hoffmann
Studiengang Informatik
Universität Bremen

Winter 03/04

# Vorlesung vom 26.01.2004: Logisches Programmieren— Einführung und Grundlagen

# Inhalt

- Regelbasiert = Funktional $\vee$ Logisch
- Einführende Beispiele: `grandma` und `rev`
- Einige logische Grundlagen
  - Unifikation
  - SLD-Resolution
  - Tiefensuche

# Regelbasiertes Programmieren

- Funktionales Programmieren
  - Regeln sind Gleichungen (gerichtet)
  - Regeln definieren Funktionen
  - Reduktion durch verzögerte Auswertung
  - Beispiel: Listenumkehr (endrekursiv)

```
rev  []    a = a
rev (h:t) a = rev t (h:a)
```

  - Reduktion

`rev [1,2] []` $\leadsto$ `rev [2] [1]` $\leadsto$ `rev [] [2,1]` $\leadsto$ `[2,1]`

- Logisches Programmieren
  - Regeln sind Hornklausel, sie definieren Prädikate (Relationen)
  - Reduktion durch SLD-Resolution
  - **Beispiel**: Listenumkehr

    ```
    rev([],A,A).
    rev([H|T],A,R) :- rev(T,[H|A],R).
    ```

  - Reduktion

    `rev([1,2],[],R)` $\rightsquigarrow$ `rev([2],[1],R)` $\rightsquigarrow$ `rev([],[2,1],R)`
    $\rightsquigarrow$ yes, `R=[2,1]`

  - Aber: im Allgemeinen mehrdeutige Egebnisse!
  - Und: vielseitigere Fragen sind möglich! zum Beispiel:
    `rev(X,Y,[3,2,1])`

# Beispiele

# Die Windsors

- Abstammung als Prädikat (Relation)
  - Einfache Fakten

```
mother(queenMum,elizabethII).
mother(elizabethII,charles).
mother(elizabethII,edward).
mother(elizabethII,andrew).
mother(elizabethII,anne).
father(charles,william).
father(charles,harry).
mother(diana,william).
mother(diana,harry).
```

- abgeleitete Prädikate
  - Klauseln

```
parent(P,C)     :- father(P,C);
                   mother(P,C).
grandma(GM,GC) :- mother(GM,P), parent(P,GC).
```

  - $P$ `:-` $Q$ bedeutet "$P$, wenn $Q$" (Umkehr-Implikation "$\Leftarrow$").
  - $P$ `;` $Q$ bedeutet "$P$ oder $Q$" (Disjunktion).
  - $P$ `,` $Q$ bedeutet "$P$ und $Q$" (Konjunktion) .
- Konkret
  - "$X$ Elternteil von $Y$, wenn $X$ Mutter von $Y$, oder $X$ Vater von $Y$."
  - "$X$ Großmutter von $Y$,
    wenn $X$ Mutter von $Z$, und $Z$ Elternteil von $Y$ ist."

# Fragen ans Programm (Datenbasis)

- Einfache Fragen

  `grandma(elizabethII,william)`

  $\rightsquigarrow$ `mother(elizabethII,Z), parent(Z,william)`

  $\rightsquigarrow$ `parent(charles,william)`

  $\rightsquigarrow$ `father(charles,william)` $\rightsquigarrow$ yes

- "Beweis" der Frage
  - Durchsuchen der Datenbasis (von oben nach unten)
  - dabei Belegen der Variablen
  - yes wenn alle Fragen beantwortet werden können
  - no sonst

`grandma(anne,william)`

$\rightsquigarrow$ `mother(anne,Z), parent(Z,william)`

$\rightsquigarrow$ no

- Fragen nach allen Enkeln

`grandma(elizabethII,X)`

$\rightsquigarrow$ `mother(elizabethII,Z), parent(Z,X)` (1)

$\rightsquigarrow$ `parent(charles,X)` (2)

$\rightsquigarrow$ `father(charles,X)`

$\rightsquigarrow$ yes, `X = william;`

$\rightsquigarrow$ yes, `X = harry` $\rightsquigarrow$ no

  - Finden einer Belegung für `X`
  - Auf Wunsch (mit `;`) weitere Belegungen

- Fragen nach allen Großmüttern von William

```
grandma(X,william)
⇝ mother(X,Z), father(Z,william)
⇝ ··· ⇝
⇝ father(charles,william)
⇝ yes, X = elizabethII ⇝ no
```

  - "⇝ ···" verschweigt Fehlschläge
  - Nur eine Antwort (Datenbasis ist unvollständig)

- Aufzählung alle Großmütter-Enkel-Paare

```
grandma(X,Y)
⇝ ··· yes, X = queenMum, Y=charles
⇝ ···
⇝ ··· yes, X = elizabethII, Y=harry
⇝ ··· no
```

# Beispiel: Listen umkehren

- Erinnerung: So sah's in Haskell aus:

```
rev    []  a = a
rev (h:t) a = rev t (h:a)
```

- In PROLOG geht's ganz ähnlich:

```
rev(  [] ,A,A).
rev([H|T],A,R) :- rev(T,[H|A],R).
```

- Verwendungen von `rev`
  - Test: `rev([],[],[])` $\rightsquigarrow$ yes. `rev([X],[],[X])` $\rightsquigarrow$ yes.
    `rev([X],[],[Y])` $\rightsquigarrow$ yes, `X=Y` `rev([X],[],[])` $\rightsquigarrow$ no.
  - Funktional: `rev([1,···,12][],R)` $\rightsquigarrow$ yes. `R=[12,···,1]`

- Umkehr-funktional:

```
rev(Q,[],[1,2])
⇝ rev(T,[H],[1,2])                    [Q=[H|T], A= [], R=[1,2]]
⇝ rev(T',[H',H])),[1,2])    [T=[H'|T'], A'=[H], R=[1,2]]
⇝ yes,                          [A''=[H'|H|T'], H=2, H'=1, T'=[]]
Q = [H|T] = [2,H',T'] = [2,1] ;
⇝ ⊥
```

- Aufzählung der gesamten Relation

  `rev(Q,[],R)` $\rightsquigarrow$ yes. `Q=[], R= []`

  $\rightsquigarrow$ yes. `Q=[X], R= [X]`

  $\rightsquigarrow$ yes. `Q=[X,Y], R= [Y,X]`

  $\rightsquigarrow \ldots$

- Aufzählung der gesamten Relation

  `rev(Q,[],R)` $\rightsquigarrow$ yes. `Q=[], R= []`

  $\rightsquigarrow$ yes. `Q=[X], R= [X]`

  $\rightsquigarrow$ yes. `Q=[X,Y], R= [Y,X]`

  $\rightsquigarrow \ldots$

- "Beweisen" von Eigenschaften der Relation

  `rev(X,[],Y),rev(Y,[],X)` $\rightsquigarrow$ yes. `Q=[], R= []`

  $\rightsquigarrow$ yes. `Q=[X], R= [X]`

  $\rightsquigarrow$ yes. `Q=[X,Y], R= [Y,X]`

  $\rightsquigarrow \ldots$

# Einige Grundlagen des Logischen Programmierens

# Prädikatenlogik (Syntax)

- Formeln über Termen (Ausdrücken)
  $\forall X : P \mid \exists X : P \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \Rightarrow Q \mid \cdots$
  **Beispiel:**
  $\forall L, L', L'' : \texttt{rev}(L, [], L') \wedge \texttt{rev}(L', [], L'') \rightarrow L = L''$
- Einschränkungen für logisches Programmieren
  - alle Funktionssymbole sind frei (Konstruktorfunktionen)
  - andere Funktionen werden als Prädikate definiert (siehe `rev`)
  - Formeln sind positive Hornklauseln
    $\forall X_1, \ldots, X_n : p(t_{1,1}, \ldots t_{1,n_1}) \wedge \cdots \wedge p(t_{k,1}, \ldots t_{k,n_k}) \Rightarrow q(t_1, \ldots t_k)$
    - ▷ Keine Negation oder existenzielle Quantifizierung
    - ▷ alle Variablen sind all-quantifiziert

# Prädikatenlogik (Semantik)

- Modelle
  - Eine Interpretation $\mathcal{M}$ der Funktions- und Prädikatssymbole als Funktionen und Prädikate
  - eine Formelmenge $T$ (Theorie) muss gelten
  - $\mathcal{M}$ erfüllt $T$: $\mathcal{M} \models \mathcal{T}$
  - Tautologien, erfüllbare und unerfüllbare Formeln
- Allgemein: Folgern und Schließen
  - Nachweisen der Äquivalenz durch Anwendung von syntaktischen Transformationsregeln
    **Beispiele**: $\neg(A \vee B) \equiv (\neg A \wedge \neg B)$

- Resolution: Rückwärtsschließen mit Hornklauseln
  Frage: $F_1 \wedge \cdots \wedge F_n$, Klausel $Q \Leftarrow P_1 \wedge \cdots \wedge P_n$ mit $Q = F_1$

$$\frac{F_1 \wedge \cdots \wedge F_n \qquad Q \Leftarrow P_1 \wedge \cdots \wedge P_k}{P_1 \wedge \cdots \wedge P_k \wedge F_2 \wedge \cdots \wedge F_n}$$

- Tiefensuche: Verwende die immer letzte Resolvente
- Selektionsregel: Resolviere die erste Frage von links,
  Versuche die Klauseln von oben nach unten.
- Fahre fort, bis die Resolvente leer, oder kein Schritt möglich ist.
- Backtracking: Bei Mißerfolg einige Schritte rückgängig machen und einen noch nicht versuchten Schritt versuchen.
- Bei Erfolg erhält man einen Beweis in umgekehrter Richtung.

# Unifikation

- Die Literale $F_1$ und $Q$ sind nicht gleich
  Sie müssen gleich gemacht werden (Unifikation)
  Dabei werden Variablen durch Terme ersetzt (Substitution)

$$\frac{F_1 \wedge \cdots \wedge F_n \qquad Q \Leftarrow P_1 \wedge \cdots \wedge P_k}{(P_1 \wedge \cdots \wedge P_k \wedge F_2 \wedge \cdots \wedge F_n)\sigma}$$

- **Beispiele**:
  - $F_1 = \mathtt{rev}(\mathtt{Q}, [], [1,2])$
  - $Q = \mathtt{rev}([\mathtt{H}|\mathtt{T}], \mathtt{A}, \mathtt{R})$
  - Unifikator: $\sigma = \{\mathtt{Q} = [\mathtt{H}|\mathtt{T}], \mathtt{A} = [], \mathtt{R} = [1,2]\}$
  - Resolutionsschritt mit Unifikation

$$\frac{\mathtt{rev}(\mathtt{Q}, [], [1,2]) \qquad \mathtt{rev}([\mathtt{H}|\mathtt{T}], \mathtt{A}, \mathtt{R}) \Leftarrow \mathtt{rev}(\mathtt{T}, [\mathtt{H}|\mathtt{A}], \mathtt{R})}{(\mathtt{rev}(\mathtt{T}, [\mathtt{H}|[]], [1,2]))}$$

  - Unifikation ist syntaktisch, auf textuelle Gleichheit
  - Variablen der Literale müssen disjunkt sein
  - Substitutionen dürfen nicht zyklisch sein (occur check)

# Zusammenfassung

- Regelbasiertes Programmieren
- Einführung in logisches Programmieren
  - Einfache Terme (die Windsors)
  - Listen (`rev`)
- Einige Grundlagen des logischen Programmierens
  - Resolution
  - Unifikation