TECHNISCHE UNIVERSITÄT DRESDEN

PROGRAMMIERUNG

ÜBUNG 8: LOGIKPROGRAMMIERUNG MIT PROLOG^{-}

Eric Kunze

eric.kunze@tu-dresden.de

TU Dresden, 01. Juni 2022

letzte Änderung:
01.06.2022, 11:08

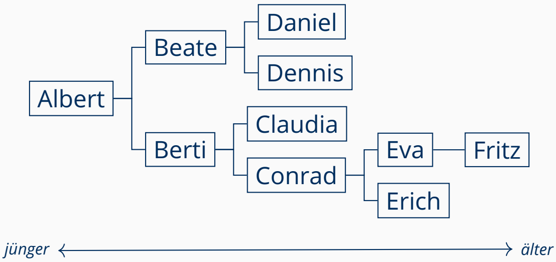
INHALT

1. Funktionale Programmierung
    1.1 Einführung in Haskell: Listen
    1.2 Algebraische Datentypen
    1.3 Funktionen höherer Ordnung
    1.4 Typpolymorphie & Unifikation
    1.5 Beweis von Programmeigenschaften
    1.6 λ–Kalkül
2. **Logikprogrammierung**
3. Implementierung einer imperativen Programmiersprache
4. Verifikation von Programmeigenschaften
5. H_0 – ein einfacher Kern von Haskell

Logikprogrammierung und Prolog

EIN EINFÜHRENDES BEISPIEL

Wir betrachten den folgenden Familienstammbaum:



Nun wollen wir die Verwandtschaftsbeziehungen abbilden. Dafür brauchen wir vor allem Geschlechter und Eltern-Kind-Beziehung(en). Dazu sei

$$\textbf{Fam} = \{albert, beate, berti, daniel, dennis, claudia, conrad, eva, erich, fritz\}$$

die Menge aller Familienmitglieder.

Wir betrachten hier den stark vereinfachten Fall und nehmen eine binäre Geschlechtsvorstellung an.

PRÄDIKATENLOGIK

- *Variablen*, z.B. $X \rightsquigarrow$ kann verschiedene Werte annehmen
  $\hookrightarrow$ Großbuchstaben
- *Konstruktoren*, z.B. s als „Nachfolger“
  $\hookrightarrow$ Kleinbuchstaben
- *Prädikate*, z.B. $male \rightsquigarrow$ Wahrheit hängt von Argumenten ab
  - $male(albert) = \text{true}$, aber $male(beate) = \text{false}$
  - $parent(claudia, berti) = \text{true}$

Prädikate kodieren Relationen passender Stelligkeit, d.h. beispielsweise

$$\mathbf{M} = \{X \in \mathbf{Fam} : male(X) = \text{true}\}$$

$$\mathbf{P} = \{(X, Y) \in \mathbf{Fam} \times \mathbf{Fam} : parent(X, Y) = \text{true}\}$$

Problem: Woher wissen wir, was wahr und was falsch ist?

FAKTEN & REGELN

Ziel: wahre Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

- Albert ist männlich: $male(albert) = \text{true}$
- Claudia ist ein Elternteil von Berti: $parent(claudia, berti) = \text{true}$

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Vater ist männliches Elternteil:
  $parent(X, Y) \wedge male(X) \implies father(X, Y)$

EINFÜHRUNG IN PROLOG

- Französisch: programmation en logique
- hier: Teilsprache Prolog^{-}
- **Interpreter:** *swipl*

  `https://www.swi-prolog.org/download/stable`
  - Nutzung wie üblich im Terminal
  - `swipl <filename>` startet die interaktive Session
- **Online-Editor & Interpreter:** `https://swish.swi-prolog.org/`

EINFÜHRUNG IN PROLOG

- Französisch: programmation en logique
- hier: Teilsprache Prolog^{-}
- **Interpreter:** *swipl*

  `https://www.swi-prolog.org/download/stable`
  - Nutzung wie üblich im Terminal
  - `swipl <filename>` startet die interaktive Session
- **Online-Editor & Interpreter:** `https://swish.swi-prolog.org/`

- Prolog-Programme bestehen aus **Fakten** und **Regeln**.

- Statements werden mit `.` abgeschlossen.
- Variablen beginnen mit Großbuchstaben.

- **UND**-Operator: `,`
- **ODER**-Operator: `;`
- Negation: `not`

PROLOG: FAKTEN

Ziel: *wahre* Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

- männliche Familienmitglieder:

PROLOG: FAKTEN

Ziel: *wahre* Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

- männliche Familienmitglieder:

```
male(albert).
male(berti).
male(conrad).
```

```
male(dennis).
male(daniel).
male(erich).
male(fritz).
```

PROLOG: FAKTEN

Ziel: *wahre* Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

- männliche Familienmitglieder:

```
male(albert).
male(berti).
male(conrad).
```

```
male(dennis).
male(daniel).
male(erich).
male(fritz).
```

- Elternbeziehungen:

PROLOG: FAKTEN

Ziel: *wahre* Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

- männliche Familienmitglieder:

```
male(albert).
male(berti).
male(conrad).
```

```
male(dennis).
male(daniel).
male(erich).
male(fritz).
```

- Elternbeziehungen:

```
parent(berti,albert).
parent(beate,albert).

parent(conrad,berti).
parent(claudia,berti).
```

```
parent(erich,conrad).
parent(eva,conrad).

parent(dennis,beate).
parent(daniel,beate).

parent(fritz,eva).
```

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

- Prädikat *father*:

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

- Prädikat *father*:

```
father(X,Y) :- parent(X,Y), male(X).
```

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

- Prädikat *father*:

```
father(X,Y) :- parent(X,Y), male(X).
```

- Prädikat *ancestor* — Wir suchen eine Regel, um zu prüfen, ob X ein Vorfahre von Y ist.

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

- Prädikat *father*:

```
father(X,Y) :- parent(X,Y), male(X).
```

- Prädikat *ancestor* — Wir suchen eine Regel, um zu prüfen, ob X ein Vorfahre von Y ist. Ein Elternteil ist immer auch ein Vorfahre; die Vorfahren eines Elternteils von Y sind wiederum Vorfahren von Y.

PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- Geschlecht weiblich:

```
female(X) :- not(male(X)).
```

- Prädikat *father*:

```
father(X,Y) :- parent(X,Y), male(X).
```

- Prädikat *ancestor* — Wir suchen eine Regel, um zu prüfen, ob X ein Vorfahre von Y ist. Ein Elternteil ist immer auch ein Vorfahre; die Vorfahren eines Elternteils von Y sind wiederum Vorfahren von Y.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(Z,Y), ancestor(X,Z).
```

ARBEITEN MIT SWIPL — ANFRAGEN

Nun möchten wir Programme auch ausführen. Aus Logik-Sicht ist die Ausführung eine Anfrage (*query*): wir wollen wissen, ob ein Fakt gilt oder nicht (bzw. ob er gültig gemacht werden kann). Diesen Fakt nennen wir das Ziel (*goal*).

- Ist Albert männlich?
- Anfrage: `?- male(albert).`
- Antwort: `true.`

ARBEITEN MIT SWIPL — ANFRAGEN

Nun möchten wir Programme auch ausführen. Aus Logik-Sicht ist die Ausführung eine Anfrage (*query*): wir wollen wissen, ob ein Fakt gilt oder nicht (bzw. ob er gültig gemacht werden kann). Diesen Fakt nennen wir das Ziel (*goal*).

- Ist Albert männlich?
- Anfrage: `?- male(albert).`
- Antwort: `true.`

Im Allgemeinen gibt es kein I/O. Wir können das aber „simulieren“, indem wir Variablen nutzen.

- Welche Personen sind männlich?
- Anfrage: `?- male(X).`
- Anzeigen mehrerer Lösungen in `swipl` durch `;`

Die Belegung einer solchen Variable lässt sich mittel SLD-Refutation unter Nutzung von Unifikationen ermitteln.

SLD-REFUTATIONEN

Ziel: zeige Gültigkeit einer Anfrage (eines Goals) $G = (\texttt{?-}\ L_1, \ldots, L_n)$

SLD-Resolution:

- wähle ein L_i aus
- es gibt eine Regel $C = (M_0 \texttt{:-}\ M_1, \ldots, M_m)$,
  wobei C und G keine gemeinsamen Variablen haben
- σ sei der allgemeinste Unifikator von L_i und M_0

Dann: ersetze L_i durch $M_1, \ldots, M_m$ unter Anwendung von σ — formal:

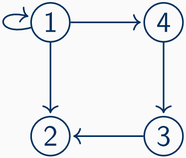
$$G' = \Big(\texttt{?-}\ \tilde{\sigma}(L_1), \ldots \tilde{\sigma}(L_{i-1}), \tilde{\sigma}(M_1), \ldots, \tilde{\sigma}(M_m), \tilde{\sigma}(L_{i+1}), \ldots, \tilde{\sigma}(L_n)\Big)$$

G' heißt *Resolvente* von G und C unter σ.

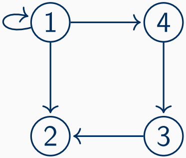
- **SLD-Ableitung** (derivation): Folge von SLD-Resolutionen
- **SLD-Refutation** (refutation): endliche Folge von SLD-Resolutionen
  mit dem leeren Goal `?-.` als Ende

Aufgabe 1

AUFGABE 1

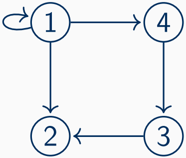


AUFGABE 1




```
edge(1,1).
edge(1,4).
edge(1,2).
edge(3,2).
edge(4,3).
```

AUFGABE 1




```
edge(1,1).
edge(1,4).
edge(1,2).
edge(3,2).
edge(4,3).
```

```
path(U, U).
path(U, W) :- edge(U, V), path(V, W).
```

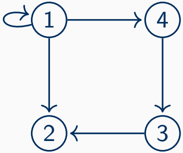
AUFGABE 1

Hinweis: Die Zeilenangaben in den Refutationen können von denen in der Übung abweichen.

```
        ?- path(4,X).
{X=4}   ?- .                        % 7

        ?- path(4,X).
        ?- edge(4,W), path(W,X).    % 8
{W=3}   ?- path(3,X).               % 5
{X=3}   ?- .                        % 6

        ?- path(4,X).
        ?- edge(4,W), path(W,X).    % 8
{W=3}   ?- path(3,X).               % 5
        ?- edge(3,U), path(U,X).    % 8
{U=2}   ?- path(2,X).               % 4
{X=2}   ?- .                        % 7
```



Aufgabe 2

ARITHMETIK IN PROLOG: NATÜRLICHE ZAHLEN

natürliche Zahlen: Unärkodierung mit Konstruktor `s`,

$$\langle \mathtt{n} \rangle := \underbrace{s(\cdots s}_{\mathtt{n}\text{ mal}}(0))$$

und einstelliges Prädikat `nat`, das über klassische Rekursion definiert ist:

```
nat(0).
nat(s(X)) :- nat(X).
```

ARITHMETIK IN PROLOG: NATÜRLICHE ZAHLEN

natürliche Zahlen: Unärkodierung mit Konstruktor `s`,

$$\langle \mathtt{n} \rangle := \underbrace{s(\cdots s}_{\mathtt{n}\text{ mal}}(0))$$

und einstelliges Prädikat `nat`, das über klassische Rekursion definiert ist:

```
nat(0).
nat(s(X)) :- nat(X).
```

Anmerkung 1: Prolog weiß dabei nicht, dass es sich um Zahlen handelt. Vielmehr sind alle Zeichen nur Symbole; die Zahl `0` wird zum Beispiel als Konstante im Sinne der Prädikatenlogik angesehen.

ARITHMETIK IN PROLOG: NATÜRLICHE ZAHLEN

natürliche Zahlen: Unärkodierung mit Konstruktor `s`,

$$\langle \mathtt{n} \rangle := \underbrace{s(\cdots s}_{\mathtt{n}\text{ mal}}(0))$$

und einstelliges Prädikat `nat`, das über klassische Rekursion definiert ist:

```
nat(0).
nat(s(X)) :- nat(X).
```

Anmerkung 1: Prolog weiß dabei nicht, dass es sich um Zahlen handelt. Vielmehr sind alle Zeichen nur Symbole; die Zahl `0` wird zum Beispiel als Konstante im Sinne der Prädikatenlogik angesehen.

Anmerkung 2: Durch die Unärkodierung müssen wir bei rekursiven Funktionen ein wenig umdenken. Die Zahl $\langle n \rangle$ lässt sich in einer Rekursion nicht auf $\langle n-1 \rangle$ reduzieren. Stattdessen fügen wir vorher ein `s` ein, um es anschließend wieder zu entfernen, d.h. $\langle n+1 \rangle \leadsto \langle n \rangle$.

- Haskell-Rekursion: $n \leadsto n-1$
- Prolog-Rekursion: $n+1 \leadsto n$

ARITHMETIK IN PROLOG: FUNKTIONEN

Funktionen müssen als Relationen dargestellt werden, d.h.

statt $\begin{array}{r} f : \mathbb{N} \to \mathbb{N} \\ f(x) = y \end{array}$ kodieren wir $\mathbf{F} = \{(x, y) \in \mathbb{N} \times \mathbb{N} : f(x) = y\}$

ARITHMETIK IN PROLOG: FUNKTIONEN

Funktionen müssen als Relationen dargestellt werden, d.h.

statt $\begin{array}{c} f : \mathbb{N} \to \mathbb{N} \\ f(x) = y \end{array}$ kodieren wir $\mathbf{F} = \{(x, y) \in \mathbb{N} \times \mathbb{N} : f(x) = y\}$

Beispiel: Summe zweier natürlicher Zahlen

rekursive Idee:

$$\begin{aligned} 0 + y &= y &\quad \Longleftarrow \quad & y \in \mathbb{N} && \text{(Basisfall)} \\ (x+1) + y &= s + 1 &\quad \Longleftarrow \quad & x + y = s && \text{(Rekursionsfall)} \end{aligned}$$

ARITHMETIK IN PROLOG: FUNKTIONEN

Funktionen müssen als Relationen dargestellt werden, d.h.

statt $\begin{array}{c} f : \mathbb{N} \to \mathbb{N} \\ f(x) = y \end{array}$ kodieren wir $\mathbf{F} = \{(x, y) \in \mathbb{N} \times \mathbb{N} : f(x) = y\}$

Beispiel: Summe zweier natürlicher Zahlen

rekursive Idee:

$$0 + y = y \quad \Longleftarrow \quad y \in \mathbb{N} \qquad \text{(Basisfall)}$$

$$(x + 1) + y = s + 1 \quad \Longleftarrow \quad x + y = s \qquad \text{(Rekursionsfall)}$$

Haskell:

```
sum :: Int -> Int -> Int
sum 0 y = y
sum x y = 1 + sum (x-1)  y
```

Prolog:

```

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S))
            :- sum(X, Y, S).
```

AUFGABE 2 – TEIL (A)

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

Gesucht: Prädikat `even`, dass alle natürlichen Zahlen enthält

AUFGABE 2 – TEIL (A)

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

Gesucht: Prädikat `even`, dass alle natürlichen Zahlen enthält

```
even(0).
even(s(s(N))) :- even(N).
```

AUFGABE 2 – TEIL (B)

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).

even(0).
even(s(s(N))) :- even(N).
```

Gesucht: Relation `div2` mit $(\langle n \rangle, \langle \lfloor \frac{n}{2} \rfloor \rangle)$

AUFGABE 2 – TEIL (B)

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).

even(0).
even(s(s(N))) :- even(N).
```

Gesucht: Relation `div2` mit $(\langle n \rangle, \langle \lfloor \frac{n}{2} \rfloor \rangle)$

```
div2(0, 0).
div2(s(0), 0).
div2(s(s(N)), s(M)) :- div2(N, M).
```

AUFGABE 2 –TEIL (C)

```
10 div2(0, 0).
11 div2(s(0), 0).
12 div2(s(s(N)), s(M)) :- div2(N, M).
```

gesucht: SLD-Refutation für `?- div2(<3>, <1>).`

```
?- div2(<3>, <1>).
?- div2(<1>, 0).     % 12
?- .                 % 11
```

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

Gesucht: Relation `div` mit $(\langle \mathtt{n} \rangle, \langle \mathtt{m} \rangle, \langle \lfloor \frac{\mathtt{n}}{\mathtt{m}} \rfloor \rangle)$

AUFGABE 2 – TEIL (D) *ZUSATZ*

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

Gesucht: Relation `div` mit $(\langle \mathtt{n} \rangle, \langle \mathtt{m} \rangle, \langle \lfloor \frac{\mathtt{n}}{\mathtt{m}} \rfloor \rangle)$

```
lt(0, s(M)) :- nat(M).
lt(s(N), s(M)) :- lt(N, M).
```

AUFGABE 2 – TEIL (D) *ZUSATZ*

```
nat(0).
nat(s(X)) :- nat(X).

sum(0, Y, Y) :- nat(Y).
sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

Gesucht: Relation `div` mit $(\langle \mathtt{n} \rangle, \langle \mathtt{m} \rangle, \langle \lfloor \frac{\mathtt{n}}{\mathtt{m}} \rfloor \rangle)$

```
lt(0, s(M)) :- nat(M).
lt(s(N), s(M)) :- lt(N, M).
```

```
div(0, M, 0) :- lt(0, M).
div(N, M, 0) :- lt(N, M).
div(N, M, s(Q)) :- lt(0, M), sum(M, V, N),
                   div(V, M, Q).
```

```
          ?- div(⟨3⟩, ⟨2⟩, ⟨1⟩)
          ?- lt(⟨0⟩, ⟨2⟩) , sum(⟨2⟩, V1, ⟨3⟩) , div(V1, ⟨2⟩, ⟨0⟩)   % 19
          ?- nat(⟨1⟩) , sum(⟨2⟩, V1, ⟨3⟩) , div(V1, ⟨2⟩, ⟨0⟩)       % 14
          ?- nat(⟨0⟩) , sum(⟨2⟩, V1, ⟨3⟩) , div(V1, ⟨2⟩, ⟨0⟩)       % 2
          ?- sum(⟨2⟩, V1, ⟨3⟩) , div(V1, ⟨2⟩, ⟨0⟩).                 % 1
          ?-* sum(⟨0⟩, V1, ⟨1⟩) , div(V1, ⟨2⟩, ⟨0⟩).                % 4
{V1=⟨1⟩}  ?- nat(⟨1⟩) , div(⟨1⟩, ⟨2⟩, ⟨0⟩).                          % 3
          ?- nat(⟨0⟩) , div(⟨1⟩, ⟨2⟩, ⟨0⟩).                          % 2
          ?- div(⟨1⟩, ⟨2⟩, ⟨0⟩).                                     % 1
          ?- lt(⟨1⟩, ⟨2⟩).                                            % 18
          ?- lt(⟨0⟩, ⟨1⟩).                                            % 15
          ?- nat(⟨0⟩).                                                % 14
          ?- .                                                        % 1
```