

PROGRAMMIERUNG

ÜBUNG 8: LOGIKPROGRAMMIERUNG MIT PROLOG

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 01. Juni 2022

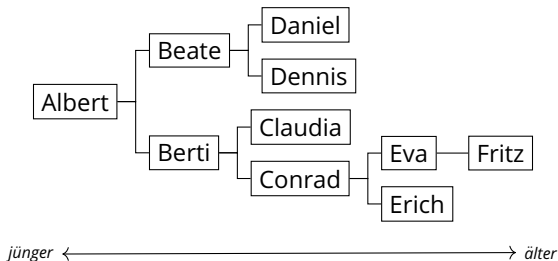
letzte Änderung:
01.06.2022, 11:08

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

Fam = {*albert*, *beate*, *berti*, *daniel*, *dennis*, *claudia*, *conrad*, *eva*, *erich*, *fritz*}
die Menge aller Familienmitglieder.

- ▶ *Variablen*, z.B. $X \rightsquigarrow$ kann verschiedene Werte annehmen
↔ Großbuchstaben
- ▶ *Konstruktoren*, z.B. s als „Nachfolger“
↔ 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?

Ziel: wahre Aussagen beschreiben

Fakten: unabhängig von anderen Zuständen immer wahr

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

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/`
-

- ▶ 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:

```
1 | male(albert).  
2 | male(berti).  
3 | male(conrad).
```

```
4 | male(dennis).  
5 | male(daniel).  
6 | male(erich).  
7 | male(fritz).
```

- ▶ Elternbeziehungen:

```
9 | parent(berti,albert).  
10 | parent(beate,albert).  
11 |  
12 | parent(conrad,berti).  
13 | parent(claudia,berti).
```

```
15 | parent(erich,conrad).  
16 | parent(eva,conrad).  
17 |  
18 | parent(dennis,beate).  
19 | parent(daniel,beate).  
20 |  
21 | parent(fritz,eva).
```


PROLOG: REGELN

Ziel: *wahre* Aussagen beschreiben

Regeln: Abhängigkeit von einem oder mehreren anderen Fakten

- ▶ Geschlecht weiblich:

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

- ▶ Prädikat *father*:

```
25 | 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*.

```
27 | ancestor(X,Y) :- parent(X,Y).
```

```
28 | 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.`

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) $G = (?- L_1, \dots, L_n)$

SLD-Resolution:

- ▶ wähle ein L_i aus
- ▶ es gibt eine Regel $C = (M_0 :- M_1, \dots, M_m)$,
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, \dots, M_m unter Anwendung von σ — formal:

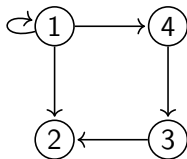
$$G' = (?- \tilde{\sigma}(L_1), \dots, \tilde{\sigma}(L_{i-1}), \tilde{\sigma}(M_1), \dots, \tilde{\sigma}(M_m), \tilde{\sigma}(L_{i+1}), \dots, \tilde{\sigma}(L_n))$$

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



```
1 edge(1,1).
```

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

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

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

```
5 edge(4,3).
```

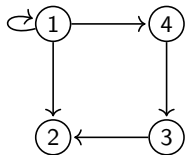
```
7 path(U, U).
```

```
8 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 ,

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

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

```
1 nat(0).  
2 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. $\langle n + 1 \rangle \rightsquigarrow \langle n \rangle$.

- ▶ Haskell-Rekursion: $n \rightsquigarrow n - 1$
- ▶ Prolog-Rekursion: $n + 1 \rightsquigarrow n$

ARITHMETIK IN PROLOG: FUNKTIONEN

Funktionen müssen als Relationen dargestellt werden, d.h.

statt $f : \mathbb{N} \rightarrow \mathbb{N}$ kodieren wir $\mathbf{F} = \{(x, y) \in \mathbb{N} \times \mathbb{N} : f(x) = y\}$
 $f(x) = y$

Beispiel: Summe zweier natürlicher Zahlen

rekursive Idee:

$$\begin{array}{ll} 0 + y = y & \Leftarrow y \in \mathbb{N} & \text{(Basisfall)} \\ (x + 1) + y = s + 1 & \Leftarrow x + y = s & \text{(Rekursionsfall)} \end{array}$$

Haskell:

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

Prolog:

```
1
2 sum(0, Y, Y) :- nat(Y).
3 sum(s(X), Y, s(S))
4     :- sum(X, Y, S).
```

AUFGABE 2 – TEIL (A)

```
1 nat(0).  
2 nat(s(X)) :- nat(X).  
3  
4 sum(0, Y, Y) :- nat(Y).  
5 sum(s(X), Y, s(S)) :- sum(X, Y, S).
```

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

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

AUFGABE 2 – TEIL (B)

```
1 nat(0).
2 nat(s(X)) :- nat(X).
3
4 sum(0, Y, Y) :- nat(Y).
5 sum(s(X), Y, s(S)) :- sum(X, Y, S).
6
7 even(0).
8 even(s(s(N))) :- even(N).
```

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

```
10 div2(0, 0).
11 div2(s(0), 0).
12 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
```

```

1 nat(0).
2 nat(s(X)) :- nat(X).
3
4 sum(0, Y, Y) :- nat(Y).
5 sum(s(X), Y, s(S)) :- sum(X, Y, S).

```

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

```

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

```

```

17 div(0, M, 0) :- lt(0, M).
18 div(N, M, 0) :- lt(N, M).
19 div(N, M, s(Q)) :- lt(0, M), sum(M, V, N),
20                       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

```