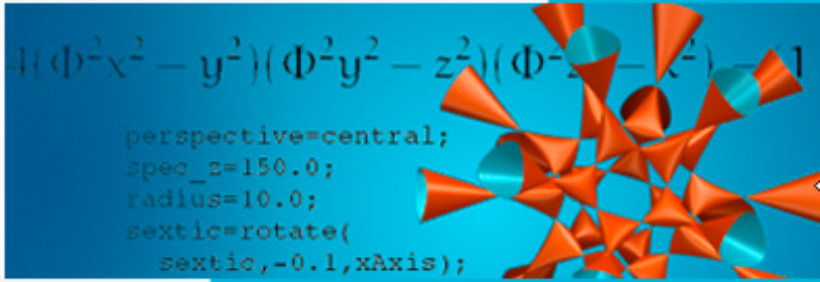
Philipps Universität Marburg

Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch
(Programmiersprachen und –werkzeuge)
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[The art of Prolog: 3.3, 3.4; Learn Prolog Now!: 5.1 – 5.3]

Komposition von Prolog-Programmen

- Verschmelzung der prozeduralen und deklarativen Sichtweise

- Top-Down Entwurf von Logikprogrammen

Prozedural vs. Deklarativ

- Programmverständnis ist typischerweise prozedural
- Außerhalb von Computerprogrammen ist unser Denken eher deklarativ
  - Definitionen von Bedeutung
  - Beschreibungen

Entwurfsrezept Logik-Programme

- Vermischen von prozeduraler und deklarativer Sicht
  - Prozedural aufbauen
    - Konstruktion fällt so leichter
    - Fokus auf einen Anwendungsfall
  - Deklarativ interpretieren (Kontrolle)
    - Welche andere Nutzungsmöglichkeiten gibt es?
    - Macht das Programm allgemein Sinn?

Entwurfsrezept: Fallstudie

- Entfernen eines Elements aus einer Liste
- Prozedur: `delete (L1, X, L2)`
- Zwei Eingabe-Parameter
  - Die ursprüngliche Liste (L1)
  - Das zu entfernende Element (X)
- Ein Ausgabe-Parameter
  - Die Ergebnisliste (L2)
- Bedeutung: Alle Grundinstanzen, in denen L2 die Liste L1 ohne alle X ist
- Beispielanwendung:

```
?- delete([a, b, c, b], b, L2).
```

  - Wobei die Antwort `L2 = [a, c]` ist

Prozedurales Denken

Entwurfsrezept: Fallstudie

- Komposition der rekursiven Prozedur
- Strukturelle Rekursion über L1 (Eingabe-Parameter)
  - Wir schreiben für L1: `[X | Xs]`
- Zwei rekursive Fälle
  - Der Kopf X ist das zu entfernende Element
  - Der Kopf X ist nicht das zu entfernende Element
- Ein Basisfall

Prozedurales Denken

Entwurfsrezept: Fallstudie

- Fall 1: Der Kopf X ist das zu entfernende Element
  - Ergebnis: rekursives Entfernen des Elements aus dem Rest der Eingabeliste
  - `delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).`
- Fall 2: Der Kopf X ist nicht das zu entfernende Element
  - Ergebnis: Liste mit X als Kopf und Rest wie oben
  - ```
    delete([X|Xs], Z, [X|Ys]) :-
           X =\= Z, delete(Xs,Z,Ys).
    ```
- Basisfall:
  - Die leere Liste enthält keine Elemente mehr
  - `delete([], X, []).`

Entwurfsrezept: Fallstudie
    ```

Entwurfsrezept: Fallstudie

- ```
  delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
  ```
  - “Die Löschung von X aus [X|Xs] ist Ys, wenn die Löschung von X aus Xs Ys ist“
  - Geteilte Variable X drückt aus, dass der Kopf gleich dem zu löschenden Element ist

- ```
  delete([X|Xs], Z, [X|Ys]) :-
      X =\= Z, delete(Xs,Z,Ys).
  ```
  - “Die Löschung von Z aus [X|Xs] ist [X|Ys], wenn Z von X verschieden ist und die Löschung von Z aus Xs Ys ist.“
  - Verschiedenheit von X und Z muss explizit spezifiziert werden

Entwurfsrezept: Fallstudie

Entwurfsrezept Logik-Programme

- Top-Down Ansatz mit schrittweise Verfeinerung
  - Formulierung des allgemeinen Problems
  - Zerlegen in Teil-Probleme

Entwurfsrezept: Fallbeispiel

- Permutation-sort
  - Finden einer sortierten Permutation

- `sort(Xs, Ys)`: Ys enthält alle Elemente von Xs in aufsteigender Reihenfolge

- ```
  sort(Xs,Ys) :-
      permutation(Xs,Ys), ordered(Ys).
  ```

- Teilprobleme:
  - Was bedeutet, dass Ys eine Permutation von Xs ist?
  - Was bedeutet, dass Ys geordnet ist?
  ```

Entwurfsrezept: Fallbeispiel

- `ordered([X]).`
- `ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).`

Später mehr zu Vergleichen und Arithmetik

Entwurfsrezept: Fallbeispiel

- Permutation
  - Nicht-deterministische Auswahl eines Elements
  - Voranstellen dieses Elements an Ergebnis-Liste
  - Rest der Ergebnis-Liste: Permutation des Rests der Eingabe-Liste (ohne das ausgewählte Element)

```
permutation(Xs, [Z|Zs]) :-
    select(Z, Xs, Ys), permutation(Ys, Zs).
permutation([], []).
```

- Auswahl
  - Das erste Vorkommen eines Elements aus der Liste entfernen

```
select(X, [X|Xs], Xs).
select(X, [Y|Ys], [Y|Zs]) :- select (X, Ys, Zs).
```

Vergleiche in Prolog

(Tu)Prolog Notation

• $x < y$	`X < Y.`
• $x \leq y$	`X =< Y.`
• $x = y$	`X =:= Y.`
• $x \neq y$	`X =\= Y.`
• $x \geq y$	`X >= Y.`
• $x > y$	`X >Y.`

```
?- 2< 4.
yes
```

```
?- 2 =< 4.
yes
?- 4 =< 4.
yes
```

```
?- 4=:=4.
yes
```

```
?- 4=\=5.
yes
?- 4=\=4.
no
```

Arithmetik in Prolog

	Prolog Notation
• $6 + 2 = 8$	`8 is 6+2.`
• $6 * 2 = 12$	`12 is 6*2.`
• $6 - 2 = 4$	`4 is 6-2.`
• $6 - 8 = -2$	`-2 is 6-8.`
• $6 \div 2 = 3$	`3 is 6/2.`
• $7 \div 2 = 3$	`3 is 7/2.`
• $7 \bmod 2 = 1$	`1 is mod(7,2).`

X is 6+2.

Arithmetik in Prolog

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
?- add_3_and_double(1,X).
```

Ja

```
X = 8
```

Lösung: `add_3_and_double(1,8)`

Arithmetik in Prolog

- Variablen auf der rechten Seite müssen instantiiert sein

```
add_3_and_double(X,Y) :- Y is (X+3)*2.
?-   add_3_and_double(1, A).
?-   add_3_and_double(A, 8).
```

OK. Die Variable X aus der Definition ist zu 1 instantiiert.

Fehler. Die Variable X aus der Definition ist nicht instantiiert.

- Variablen müssen zu Integern instantiiert sein

```
?- X = 3, X < 4.
?- X = b, X < 4.
```

OK.

Fehler

Arithmetik und Listen

- Rekursive Berechnung der Länge einer Liste
  - Basisfall:
    - `len([],0).`
  - Strukturelle Rekursion
    - `len([S|T],N) :- len(T,X), N is X+1.`

Variable S wird nur an einer Stelle genutzt. Wir können "_" als "anonyme Variable" verwenden:
`len([_|T],N) :- len(T,X), N is X+1.`

```
?- len([a,b,c,d,e,[a,b],g],X).
Ja
X = 7
```

Akkumulatoren in Prolog

- Berechnung der Länge einer Liste mit Akkumulator
- Akkumulator-Invariante
  - Anzahl der bisher besuchten Elemente
- Haupt-Prozedur
  - Aufruf der Hilfsfunktion mit passendem initialen Akkumulator
  - Zu beginn noch keine Elemente besucht: initialer Akkumulator ist 0
- `leng(List,Length) :- accLen(List,0,Length).`

Akkumulatoren in Prolog

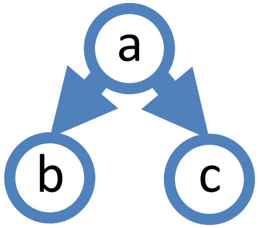
- Strukturelle Rekursion
  - `accLen([_|T],A,L) :- Anew is  A+1, accLen(T,Anew,L).`
  - Basisfall
    - `accLen([],A,A).`

Akkumulator. Enthält Länge der Liste.

Selber Variablenname für das “Resultat“.

Binäre Bäume in Prolog

- Datenstrukturen werden als Funktor repräsentiert
  - `tree(Element, Left, Right)`
  - Leerer Baum: Atom “void“



- Beispiel

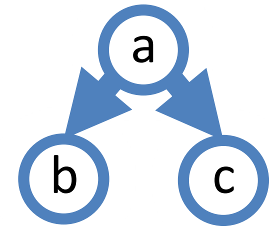
```
tree(a,tree(b,void,void),tree(c,void,void)).
```

- Typdefinition
  - Prozedur, die überprüft, ob ein Wert ein binärer Baum ist

```
/*binary_tree(Tree) :-
  Tree is a binary tree.*/
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
    binary_tree(Left), binary_tree(Right).
```

Binäre Bäume in Prolog

- Datenstrukturen werden als Funktor repräsentiert
  - `tree(Element, Left, Right)`
  - Leerer Baum: Atom “void“



- Beispiel

```
tree(a,tree(b,void,void),tree(c,void,void)).
```

- Typdefinition
  - Prozedur, die überprüft, ob ein Wert ein binärer Baum ist

```
/*binary_tree(Tree) :-
  Tree is a binary tree.*/
binary_tree(void).
binary_tree(tree(Element,Left,Right)) :-
    binary_tree(Left), binary_tree(Right).
```

doppelt-rekursiv

Binäre Bäume in Prolog

```
/* tree_ member (Element, Tree) ~
   Element is an element of the binary tree Tree. */
tree_member(X,tree(X,Left,Right)).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).

/* isotree( Tree1, Tree2) :-
   Tree1 and Tree2 are isomorphic binary trees.*/
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```

Binäre Bäume in Prolog

```
/* tree_ member (Element, Tree) ~
   Element is an element of the binary tree Tree. */
tree_member(X,t…t)).
tree_member(X,t…t)) :- tree_member(X,Left).
tree_member(X,tree(Y,Left,Right)) :- tree_member(X,Right).
```

doppelt-rekursiv

```
/* isotree( Tree1, Tree2) :-
   Tree1 and Tree2 are isomorphic binary trees.*/
isotree(void,void).
isotree(tree(X,Left1,Right1),tree(X,Left2…
    isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)) :-
    isotree(Left1,Right2), isotree(Right1,Left2).
```

doppelt-rekursiv

Doppelt-rekursive Datenstrukturen

- Prozeduren für doppelt-rekursive Datenstrukturen sind selbst doppelt-rekursiv

- Zwei alternative Manifestationen
  - Zwei unterschiedliche rekursive Fälle (wie bei tree_member)
  - Ein (oder mehr) rekursive Regeln mit je zwei rekursiven Anwendungen der Prozedur (wie bei isotree)

Unification

- Bisher:
  - Definition der Bedeutung von Prologprogrammen nur für
    - Grund-Ziele
    - Grund-Instanzen von Regeln
- Unification
  - Kern-Konzept von Logikprogrammierung
  - “Vereinheitlichung“ von Ausdrücken

Terminologie

- Gemeinsame Instanz
  - Ein Term *t* ist eine
  - Gemeinsame Instanz von zwei Termen t_1 und t_2
  - Wenn es zwei Substitutionen θ_1 und θ_2 gibt, sodass
  - $t = t_1\theta_1$ und $t = t_2\theta_2$

- Allgemeine Instanz
  - Ein Term *s* ist allgemeiner als ein Term *t*,
  - Wenn t eine Instanz von *s* ist,
  - Aber *s* keine Instanz von *t*

Terminologie

- Alphabetische Variante
  - Ein Term *s* ist eine alphabetische Variante von einem Term *t*,
  - Wenn *s* eine Instanz von *t* ist und
  - *t* eine Instanz von *s*
- Alphabetische Variante: “identisch bis auf Benennung“
  - Beispiel
  - `member(X,tree(Left,X,Right))`
  - `member(Y,tree(Left,Y,Z))`

Terminologie

- Unifikator
  - Eine Substitution, die
  - Zwei Terme identisch macht

- Jede gemeinsame Instanz entsteht durch einen Unifikator
- Jeder Unifikator erzeugt eine gemeinsame Instanz

- Beispiel
  - t_1 = `append([1,2,3], [3,4] ,List)`
  - t_2 = `append([X|Xs], Ys, [X|Zs])`

  - Unifikator: `{X=1,Xs=[2,3], Ys=[3,4], List=[1|Zs]}`
  - Gemeinsame Instanz: `([1, 2, 3], [3, 4], [1|Zs]).`

Kleinster gemeinsamer Unifikator

- “Kleinster gemeinsamer Unifikator“ oder “Allgemeinster Unifikator“ von zwei Termen
  - Ein Unifikator beider Terme
  - Der Unifikator mit der allgemeinsten gemeinsamen Instanz

- Wenn sich zwei Terme vereinheitlichen lassen, dann ist der kleinste gemeinsame Unifikator
  - eindeutig
  - bestimmbar

- Wird in vollständigem (abstrakten) Interpreter genutzt um gemeinsame Instanz von Ziel und Regelkopf zu finde.