

# Deskriptive Programmierung

## Operationelle Semantik von Prolog

## Motivation: Beobachtung einiger nicht so schöner (nicht so „logischer“?) Effekte

```
direct(frankfurt,san_francisco).  
direct(frankfurt,chicago).  
direct(san_francisco,honolulu).  
direct(honolulu,maui).  
  
connection(X, Y) :- direct(X, Y).  
connection(X, Y) :- direct(X, Z), connection(Z, Y).
```

```
?- connection(frankfurt,maui).  
true.  
  
?- connection(san_francisco,X).  
X = honolulu ;  
X = maui ;  
false.  
  
?- connection(maui,X).  
false.
```

## Motivation: Beobachtung einiger nicht so schöner (nicht so „logischer“?) Effekte

```
direct(frankfurt,san_francisco) .  
direct(frankfurt,chicago) .  
direct(san_francisco,honolulu) .  
direct(honolulu,maui) .  
  
connection(X, Y) :- connection(X, Z), direct(Z, Y) .  
connection(X, Y) :- direct(X, Y) .
```

```
?- connection(frankfurt,maui) .  
ERROR: Out of local stack
```

- Offenbar sind die impliziten logischen Operationen nicht kommutativ.
- Hinter der Programmausführung steckt also mehr als die rein logische Lesart.

## Etwas subtiler...

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .  
  
sub(X,Y,Z) :- add(Z,Y,X) .
```

```
?- sub(N,M,s(0)) .  
N = s(M) ;  
false.
```



```
add(X,0,X) .  
add(X,s(Y),s(Z)) :- add(X,Y,Z) .  
  
sub(X,Y,Z) :- add(Z,Y,X) .
```

```
?- sub(s(s(0)),s(0),N) .  
N = s(0) ;  
false.  
  
?- sub(N,M,s(0)) .  
N = s(0) ,  
M = 0 ;  
N = s(s(0)) ,  
M = s(0) ;
```

Die Wahl/Behandlung der Reihenfolge von Argumenten in Definitionen beeinflusst also die Qualität der Ergebnisse.

...

## ... und (daher) manchmal weniger Flexibilität als gewünscht

Die schön deskriptive Lösung:

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).  
  
mult(0,_,0).  
mult(s(X),Y,Z) :- mult(X,Y,U),add(U,Y,Z).
```

funktioniert sehr gut für eine Reihe von Anfragemustern:

```
?- mult(s(s(0)),s(s(s(0))),N).  
N = s(s(s(s(s(0))))) .  
  
?- mult(s(s(0)),N,s(s(s(s(0))))) .  
N = s(s(0)) ;  
false.
```

Man sagt, **mult** unterstützt die „Aufrufmodi“ **mult(+X,+Y,?Z)** und **mult(+X,?Y,+Z)**

Aber es gibt auch „Ausreißer“:

```
?- mult(N,M,s(s(s(s(0))))) .  
N = s(0) ,  
M = s(s(s(s(0)))) ;  
N = s(s(0)) ,  
M = s(s(0)) ;  
abort
```

... aber nicht **mult(?X,?Y,+Z)**.

**sonst Endlossuche**

## ... und (daher) manchmal weniger Flexibilität als gewünscht

Hingegen bei nur der Addition:

```
add(0,X,X).  
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

hatte das ja gut geklappt:

```
?- add(N,M,s(s(s(0)))).  
N = 0,  
M = s(s(s(0))) ;  
N = s(0),  
M = s(s(0)) ;  
N = s(s(0)),  
M = s(0) ;  
N = s(s(s(0))),  
M = 0 ;  
false.
```

In der Tat unterstützt **add**  
alle Aufrufmodi, sogar  
**add(?X,?Y,?Z)**.

1. Warum der Unterschied?
2. Was kann man tun, um **mult** auch so funktionieren zu lassen?

## Außerdem Vorsicht bei Verwendung/Positionierung negativer Aussagen nötig

Und, nun wird es ganz „komisch“:

```
loves(vincent,mia) .  
loves(marsellus,mia) .  
loves(mia,vincent) .  
  
jealous(X,Y) :- loves(X,Z) , loves(Y,Z) , X \= Y.
```



kleine Änderung

```
...  
  
jealous(X,Y) :- X \= Y, loves(X,Z) , loves(Y,Z) .
```

```
?- jealous(marsellus,X) .  
false.  
  
?- jealous(X,_).  
false.  
  
?- jealous(X,Y) .  
false.
```

Hingegen hatten wir vor  
der kleinen Änderung  
hier jeweils sinnvolle  
Ergebnisse gekriegt!

Um all diesen Phänomenen nachzugehen, müssen wir uns mit dem konkreten Prolog-Ausführungsmechanismus beschäftigen.

„Zutaten“ für diese Diskussion der operationellen Semantik, im folgenden betrachtet:

1. Unifikation
2. Resolution
3. Ableitungsbäume

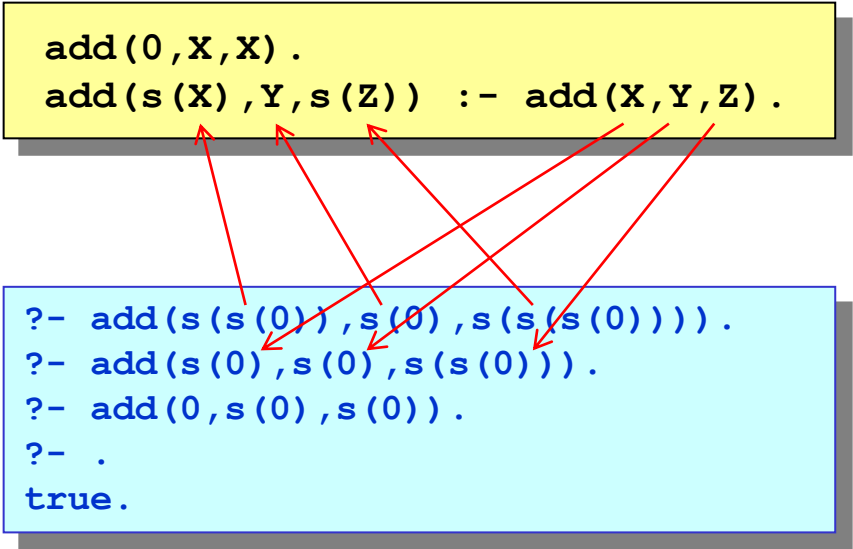


# Deskriptive Programmierung

**Unifikation**

## Analogie zu Haskell: Pattern Matching

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .
```



```
?- add(s(s(0)),s(0),s(s(s(0)))) .  
?- add(s(0),s(0),s(s(0))) .  
?- add(0,s(0),s(0)) .  
?- .  
true.
```

## Aber was ist mit „Ausgabevariablen“?

```
add(0,X,X) .  
add(s(X),Y,s(Z)) :- add(X,Y,Z) .
```

?



```
?- add(s(s(0)),s(0),N) .
```

Gleichheit „ $=$ “ als zweistelliges Prolog-Prädikat, das eine Menge leistet:

- Vergleiche auf Grundtermen (Terme ohne Variablen) durchführen, zum Beispiel:

$s(0) = s(0) \Rightarrow \text{true}$   
 $s(0) = s(s(0)) \Rightarrow \text{false}$

- Bindung von Variablen akzeptieren, zum Beispiel:

$N=0 \Rightarrow \text{true}$   
 $N=s(U) \Rightarrow \text{true}$   
 $s(0)=N \Rightarrow \text{true}$   
 $M=V \Rightarrow \text{true}$

- strukturell matchen und binden, zum Beispiel:

$s(s(0)) = s(V) \Rightarrow V=s(0)$   
 $s(U) = s(0) \Rightarrow U=0$

- Bindungen „aufsammeln“/verknüpfen, zum Beispiel:

$N=s(V) , M=V \Rightarrow N=s(M)$

## Gleichheit von Termen (1)

- Überprüfung der Gleichheit von Grundtermen:

<code>europa = europa ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(fritz,mueller) ?</code>	<code>yes</code>
<code>person(fritz,mueller) = person(mueller,fritz) ?</code>	<code>no</code>
<code>5 = 2 ?</code>	<code>no</code>
<code>5 = 2 + 3 ?</code>	<code>no</code>
<code>2 + 3 = +(2, 3) ?</code>	<code>yes</code>

⇒ Gleichheit von Termen bedeutet **strukturelle** Gleichheit.

Terme werden vor einem Vergleich nicht „ausgewertet“!

- Überprüfung der Gleichheit von Termen mit Variablen:

```
person(fritz, Nachname, datum(27, 11, 2007))  
    = person(fritz, mueller, datum(27, MM, 2007)) ?
```

- Für eine Variable darf jeder beliebige Term eingesetzt werden:
  - insbesondere **mueller** für **Nachname** und **11** für **MM**
  - Nach dieser Ersetzung sind beide Terme gleich.

## Gleichheit von Termen (3)

Welche Variablen müssen wie ersetzt werden, um die Terme anzugleichen?

```
datum(1, 4, 1985) = datum(1, 4, Jahr) ?  
datum(Tag, Monat, 1985) = datum(1, 4, Jahr) ?  
a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, g(H, i, K))) ?  
X = Y + 1 ?  
[[the, Y]|Z] = [[X, dog], [is, here]] ?
```

Zur Erinnerung, Listensyntax:

```
[1,2,a] = [1|[2,a]] = [1,2|[a]] = [1,2|. (a,[])] = . (1,. (2,. (a,[])))
```

Und was ist mit:

```
p(X) = p(q(X)) ?
```

„occurs check“ (siehe später)

Einige weitere (problematische) Fälle:

```
loves(vincent, X) = loves(X, mia) ?  
loves(marcellus, mia) = loves(X, X) ?  
a(b, C, d(e, F, g(h, i, J))) = a(B, c, d(E, f, p(H, i, K))) ?  
p(b, b) = p(X) ?  
...
```



### Substitution:

- Ersetzen von Variablen durch andere Variablen oder andere Formen von Termen (Konstanten, Strukturen, ...)
- Abbildung, die jedem Term eindeutig einen neuen Term zuordnet, wobei sich der neue vom alten Term nur durch die Ersetzung von Variablen unterscheidet.

- Notation:

$$U = \{\text{Nachname} / \text{mueller}, \text{MM} / 11\}$$

- Die Substitution  $U$  verändert nur die Variablen **Nachname** und **MM**, alles andere bleibt unverändert!
- $U(\text{person}(\text{fritz}, \text{Nachname}, \text{datum}(27, 11, 2007)))$   
   $= \text{person}(\text{fritz}, \text{mueller}, \text{datum}(27, 11, 2007))$

- Unifikator:

- Substitution, die zwei Terme „gleichmacht“.
- z.B., Anwendung der Substitution  $U = \{ \text{Nachname}/\text{mueller}, \text{MM}/11 \}$  :

$$\begin{aligned} &U(\text{person}(\text{fritz}, \text{Nachname}, \text{datum}(27, 11, 2007))) \\ &== U(\text{person}(\text{fritz}, \text{mueller}, \text{datum}(27, \text{MM}, 2007))) \end{aligned}$$

- allgemeinster Unifikator:

- Unifikator, der möglichst viele Variablen unverändert lässt.
- Beispiel:  $\text{datum}(\text{TT}, \text{MM}, 2007)$  und  $\text{datum}(\text{T}, 11, \text{J})$

- $U_1 = \{ \text{TT}/27, \text{T}/27, \text{MM}/11, \text{J}/2007 \}$  


- $U_2 = \{ \text{TT}/\text{T}, \text{MM}/11, \text{J}/2007 \}$  

- Prolog sucht immer einen allgemeinsten Unifikator.

Eingabe: zwei Terme  $T_1$  und  $T_2$  (im allgemeinen mit ggfs. gemeinsamen Variablen)

Ausgabe: ein allgemeinsten Unifikator  $U$  für  $T_1$  und  $T_2$ ,  
falls  $T_1$  und  $T_2$  unifizierbar sind, ansonsten Fehlschlag

Methode:

1. Wenn  $T_1$  und  $T_2$  gleiche Konstanten oder Variablen sind,  
dann ist  $U = \emptyset$
  2. Wenn  $T_1$  eine Variable ist, die nicht in  $T_2$  vorkommt,  
dann ist  $U = \{T_1 / T_2\}$
  3. Wenn  $T_2$  eine Variable ist, die nicht in  $T_1$  vorkommt,  
dann ist  $U = \{T_2 / T_1\}$
- 
- „occurs check“

### Methode (Forts.):

4. Falls  $T_1 = f(T_{1,1}, \dots, T_{1,n})$  und  $T_2 = f(T_{2,1}, \dots, T_{2,n})$  Strukturen mit dem gleichen Funktor und der gleichen Anzahl von Komponenten sind, dann
  1. Finde einen allgemeinsten Unifikator  $U_1$  für  $T_{1,1}$  und  $T_{2,1}$
  2. Finde einen allgemeinsten Unifikator  $U_2$  für  $U_1(T_{1,2})$  und  $U_1(T_{2,2})$
  - ...
  - n. Finde einen allgemeinsten Unifikator  $U_n$  für  
 $U_{n-1}(\dots(U_1(T_{1,n}))\dots)$  und  $U_{n-1}(\dots(U_1(T_{2,n}))\dots)$

Falls alle diese Unifikatoren existieren, dann ist

$$U = U_n \circ U_{n-1} \circ \dots \circ U_1 \quad (\text{Komposition der Unifikatoren})$$

5. Sonst:  $T_1$  und  $T_2$  sind nicht unifizierbar.

`datum(1, 4, 1985) = datum(1, 4, Jahr) ?`

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator  $U_1$  für **1** und **1**  
 $\Rightarrow$  gleiche Konstanten, daher  $U_1 = \emptyset$
2. Finde einen allgemeinsten Unifikator  $U_2$  für  $U_1(\mathbf{4})$  und  $U_1(\mathbf{4})$   
 $\Rightarrow$  gleiche Konstanten, daher  $U_2 = \emptyset$
3. Finde einen allgemeinsten Unifikator  $U_3$  für  $U_2(U_1(\mathbf{1985}))$  und  $U_2(U_1(\mathbf{Jahr}))$   
 $\Rightarrow$  Konstante vs. Variable, daher  $U_3 = \{\mathbf{Jahr} / \mathbf{1985}\}$

Ein allgemeinsten Unifikator insgesamt ist:

$$U = U_3 \circ U_2 \circ U_1 = \{\mathbf{Jahr} / \mathbf{1985}\}$$

`loves(marcellus, mia) = loves(X, X) ?`

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator  $U_1$  für `marcellus` und `X`  
 $\Rightarrow$  Konstante vs. Variable, daher  $U_1 = \{X / \text{marcellus}\}$
2. Finde einen allgemeinsten Unifikator  $U_2$  für  $U_1(\text{mia})$  und  $U_1(X)$   
 $\Rightarrow$  **verschiedene** Konstanten, daher existiert  $U_2$  nicht!

Folglich existiert auch kein Unifikator für die Ausgangsterme!

$$d(\mathbf{E}, g(\mathbf{H}, \mathbf{J})) = d(\mathbf{F}, g(\mathbf{H}, \mathbf{E})) \quad ?$$

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator  $U_1$  für  $\mathbf{E}$  und  $\mathbf{F}$   
 $\Rightarrow$  verschiedene Variablen, daher  $U_1 = \{\mathbf{E}/\mathbf{F}\}$
2. Finde einen allgemeinsten Unifikator  $U_2$  für  $U_1(g(\mathbf{H}, \mathbf{J}))$  und  $U_1(g(\mathbf{H}, \mathbf{E}))$

$$g(\mathbf{H}, \mathbf{J}) = g(\mathbf{H}, \mathbf{F}) \quad ?$$

$\Rightarrow$  Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

- Finde einen allgemeinsten Unifikator  $U_{2,1}$  für  $\mathbf{H}$  und  $\mathbf{H}$

$\Rightarrow$  gleiche Variablen, daher  $U_{2,1} = \emptyset$

- Finde einen allgemeinsten Unifikator  $U_{2,2}$  für  $U_{2,1}(\mathbf{J})$  und  $U_{2,1}(\mathbf{F})$

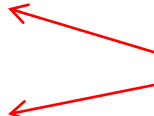
$\Rightarrow$  verschiedene Variablen, daher  $U_{2,2} = \{\mathbf{F}/\mathbf{J}\}$

$$U_2 = U_{2,2} \circ U_{2,1} = \{\mathbf{F}/\mathbf{J}\}$$

Ein allgemeinsten Unifikator insgesamt ist:

$$U = U_2 \circ U_1 = \{\mathbf{E}/\mathbf{J}, \mathbf{F}/\mathbf{J}\}$$

Zur Erinnerung:

2. Wenn  $T_1$  eine Variable ist, die nicht in  $T_2$  vorkommt,  
dann ist  $U = \{T_1 / T_2\}$
  3. Wenn  $T_2$  eine Variable ist, die nicht in  $T_1$  vorkommt,  
dann ist  $U = \{T_2 / T_1\}$
- 
- „occurs check“

Also zum Beispiel:

$$\mathbf{x} = \mathbf{q}(\mathbf{x}) \quad ?$$

$\Rightarrow$  Es existiert kein Unifikator.

In Prolog wird diese Überprüfung jedoch standardmäßig nicht durchgeführt!



Ohne „occurs check“:

$$p(x) = p(q(x)) ?$$

Strukturen mit gleichem Funktor, gleicher Anzahl von Komponenten, also:

1. Finde einen allgemeinsten Unifikator  $U_I$  für  $x$  und  $q(x)$   
 $\Rightarrow$  Variable vs. Term, daher  $U_I = \{x/q(x)\}$

$$U = U_I = \{x/q(x)\} !$$

Obwohl es ja eigentlich nicht stimmt, dass  $U(p(x))$  und  $U(p(q(x)))$  gleich sind!

# Deskriptive Programmierung

**Resolution**

### Resolutions-/(Beweis)prinzip in Prolog

Man kann den Beweis der Anfrage

$?- P, L, Q.$

( $P, L$  und  $Q$  seien **variablenfreie** Literale)

auf den Beweis der Anfrage

$?- P, L_1, L_2, \dots, L_n, Q.$

zurückführen, wenn  $L :- L_1, L_2, \dots, L_n.$  eine Regel ist (wobei  $n \geq 0$ ).

- Die Wahl des Literals  $L$  und der Regel dafür sind prinzipiell beliebig.
- Ist  $n = 0$ , so wird die Anfrage durch den Resolutionsschritt kürzer.

### Resolutionsprinzip – mit Variablen

Man kann den Beweis der Anfrage

$?- P, L, Q.$

( $P$ ,  $L$  und  $Q$  sind Literale)

auf den Beweis der Anfrage

$?- U(P), U(L_1), U(L_2), \dots, U(L_n), U(Q).$

zurückführen, wenn

- es eine Regel  $L_0 :- L_1, L_2, \dots, L_n.$  gibt ( $n \geq 0$ ), mit „zur Sicherheit“ umbenannten Variablen (so dass keine Überschneidung mit denen in  $P, L, Q$ ), und
- $U$  ein **allgemeinster Unifikator** von  $L$  und  $L_0$  ist.