IV.2 Syntax von Prolog

Freitag, 27. Januar 2017 17:00

Prolog-Progr = Folge von logischen Formeln spezieller Gestalt (sogenannte Klauseln).

Nach jeder Klausel muss ein Zeilenumbruch oder ein Leerzeichen kommen.

Literal: Prädikat($Term_1$, ..., $Term_n$)

↑
beschreibt eine
n-stellige Relation.

Sind Strings, die mit Kleinbuchstaben anfangen.

(Außerdem gibt es vordef. Prädikate wie
=, >, ...)

```
?- 3>2.
true
```

Man schreibt oft die Stelligkeit eines Prädikats mit "/" dazu: verheiratet/2

(Man kann Präd-Symbole überladen, d.h.
verheiratet/3 hat nichts mit
verheiratet/2 zu tun)

Literal ≙ Aussage, die wahr oder falsch sein kann.

Term ≙ Objekt

Terme: Variablen oder sie entstehen durch Anwendung von Funktionssymbolen auf andere Terme.

Variablen: Strings, die mit Großbuchst. oder mit _ anfangen.

Anonyme Var. _ : steht für bel. Objekt, aber ihre Belegung wird in der Antwortsubst. nicht mit ausgegeben. Mehrfache Vorkommen von _ können unterschiedlich instantiiert werden.

ist Ehemann(Person) :- verheiratet(Person, _).

- Funktionen: Strings, die mit Kleinbuchst. anfangen.
  Auch hier gibt man oft die Stelligkeit mit /... an:
  monika/0, werner/0

Auch mehrstellige Funktionssymbole sind nützlich:

Bsp: Präd. geboren um auszudrücken, wann Personen Geburtstag haben.

Mögliche Lösung: geboren/4 mit
geboren(monika, 25, 7, 1972).

Bessere Lösung: geboren/2, das nur Person und das Datum in Beziehung setzt.

Verwende ein 3-stelliges Funktionssymbol datum/3. Damit kann man Terme bilden wie:

datum(25, 7, 1972).

geboren(monika, datum(25, 7, 1972)).

Datenstrukturen in Prolog

- Eigene Datenstrukturen kann man in Prolog definieren, indem man Datenobjekte durch geeignete Terme repräsentiert.

D.h: verwende geeignete Funktionssymbole, mit denen sich alle Objekte der gewünschten Datenstruktur darstellen lassen. ≙ Datenkonstruktoren in Haskell

Bsp: Datenstruktur $\mathbb{N}$ lässt sich mit

zero / 0

succ / 1

repräsentieren: zero ≙ 0

succ(zero) ≙ 1

Prolog: Nur Prädikatssymbole werden ausgewertet, Funktionssymbole werden nicht ausgewertet (sie entsprechen Datenkonstruktoren in Haskell).

Daher kann man in Prolog Algorithmen nur als Prädikate, nicht als Funktionen implementieren.

Bsp: Addition

add(X, Y, Z) ≙

Die Addition von X und Y ergibt Z.

D.h.: Jede n-stellige Funktion lässt sich durch ein (n+1)-stelliges Prädikat realisieren.

add-Prog. lässt sich zum Addieren und Subtrahieren verwenden.

?- add(s(zero), s(zero), U)

| X = s(zero)
| Y = zero
| U = s(Z)

?- add(s(zero), zero, Z)

| X' = s(zero)
| Z = s(zero)

□

Bei jeder Anwendg. einer Prog-Klausel werden ihre Variablen in frische Var. umbenannt. D.h.: Hier gehen wir von folgendem Fakt aus:

add(X', zero, X').

Antwortsubstitution:

U = s(s(zero))

Durch versch. Anfragen kann man

dasselbe Prog. sowohl für Addition als auch für Subtraktion verwenden.
Weitere mögl. Anfrage:

?- add(succ(_), _, zero).

Dh: Gibt es hat Zahlen x, y mit $x+y=0$ und $x>0$?

Bsp: Multiplikation

$X * (Y+1) = Z$, falls

$X * Y = U$ und $X+U=Z$

?- mult(s(zero), s(zero), U)

| X = s(zero)
| Y = zero
| Z = U

Var. U in der Prog-Klausel wird zu U' umbenannt. Prog.-Klausel sollte bei jeder Anwendung frische Var. haben.

?- mult(s(zero), zero, U'), add(s(zero), U', U)

| X' = s(zero)
| U' = zero

?- add(s(zero), zero, U)

| X'' = s(zero)
| U = s(zero)

□

Antwortsubst: U = succ(zero)

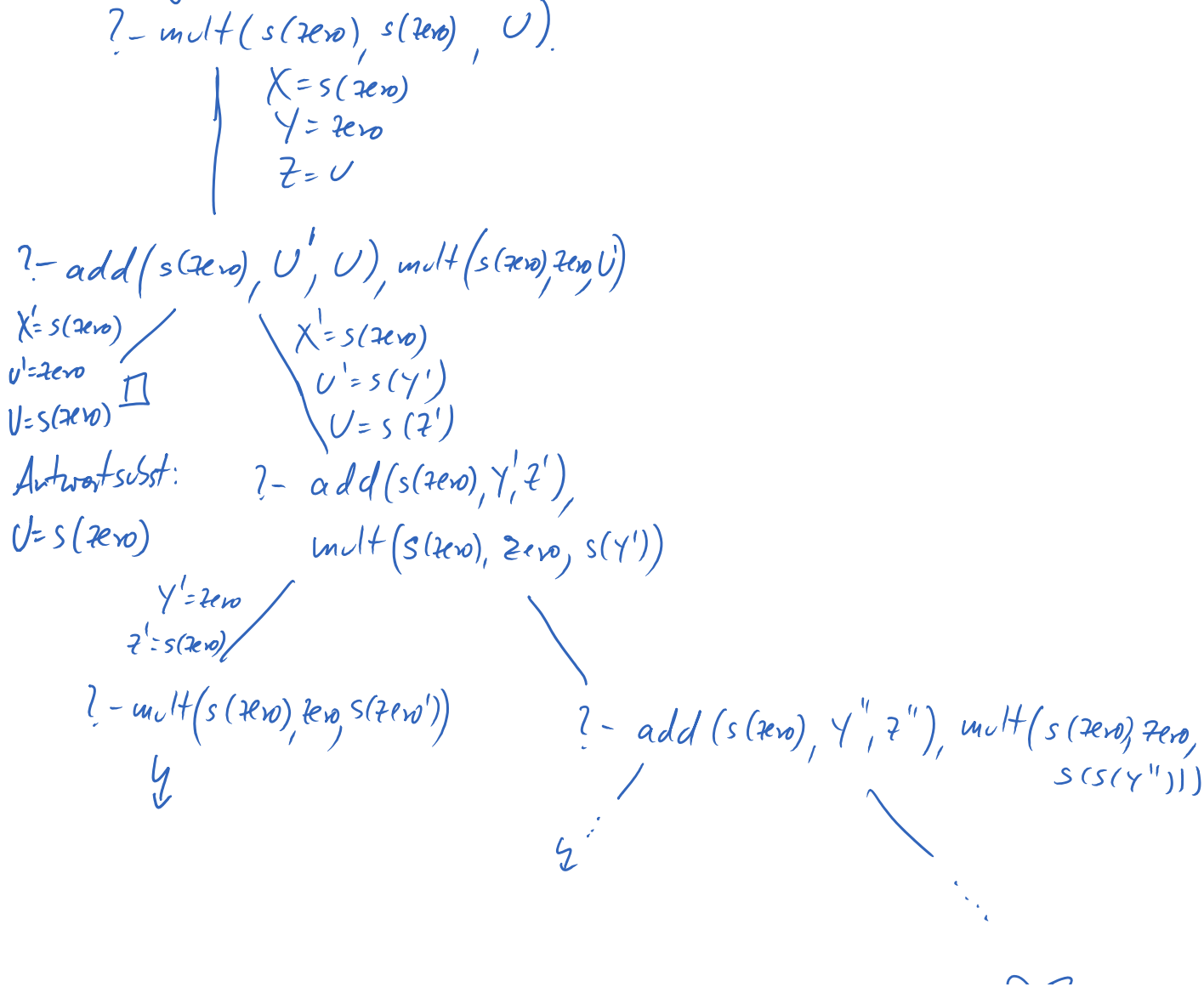
Prolog verwendet folgende Auswertungsstrategie:

- Um ein Beweisziel zu lösen,

werden die Prog.-Klauseln
von oben nach unten abgearbeitet.

- Beweisziele mit mehreren Literalen werden von links nach rechts gelöst.

Im Bsp. ist der Beweisbaum
zu ?- mult(succ(zero), succ(zero), U)
schon vollständig aufgebaut.
Was würde passieren, wenn man
die Literale im Rumpf der
mult-Regel vertauscht?



unendlicher Pfad

Wenn man nach der ersten Lösung ";" eingibt, dann baut Prolog den restlichen (unendlichen) Beweisbaum auf
⇒ terminiert nicht.

Schreibe Programme so, dass im ersten Literal des Klauselrumpfs möglichst viele Variablen schon instantiiert sind, wenn der Rumpf ausgewertet wird.

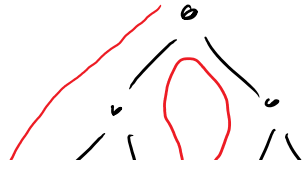
Durch entspr. Anfrage lässt sich das Multipl.-Prog. auch zum dividieren verwenden.

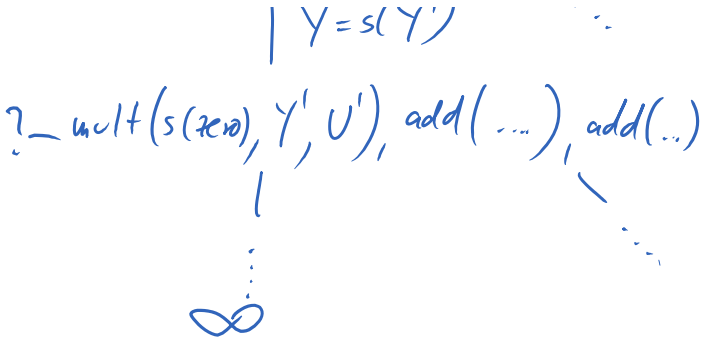
Bsp, um zu demonstrieren, dass die Reihenfolge der Prog.-Klauseln wichtig ist:

Vertausche rekursive und nicht-rekursive mult-Klausel.

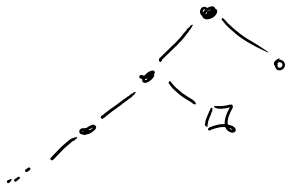
```
?- mult(s(zero), V, s(s(zero)))
   | X=s(zero)
   | V=s(Y)
   | Z=s(s(zero))
?- mult(s(zero), Y, U), add(s(zero), U, s(s(zero)))
   | Y=s(Y')
```

Prolog baut den Beweisbaum in Tiefensuche von links nach rechts auf.





Wenn der linkeste Pfad ∞ ist, wird eine Lösung im Baum nicht gefunden.



Programmierer muss über die Reihenfolge der Prog.-Klauseln nachdenken. Nicht-rekursive Klauseln für Präd. p sollten vor rekursiven Klauseln für p kommen.

Bsp Definiere die Datenstr. der Listen selbst.

nil/0 steht für leere Liste

cons/2 steht für nicht-leere Listen

cons(zero, cons(succ(zero), nil))
steht für 2-elementige Liste mit Elementen 0 und 1.

Prädikat len(Z, N) ≙
"Die Liste Z hat die Länge N."

Da Zahlen und Listen oft benutzt werden, sind diese in Prolog vordefiniert.
(D.h. anstelle der selbstdef. Zahlen u. Listen könnte man auch vordef. Versionen benutzen)

Vordef. Listen

2 Funktionssymbole:

[] .

statt nil und cons

Dh:

.(1, .(2, .(3, [])))

steht für die Liste mit den Elementen 1, 2, 3.

Für die vordef. Listen existieren Kurzschreibweisen:

[1, 2, 3] oder

[1 | [2, 3]] oder

[1, 2 | [3]] oder

[1, 2, 3 | []]

$[t_1, \ldots, t_n \mid l]$ steht für die Liste,

die aus der Liste l entsteht, in-
dem $t_1, \ldots, t_n$ vorne eingefügt werden.