Bsp.: append

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

- `append([a,b,c],[d,e],Xs)?` gibt aus: `Xs=[a,b,c,d,e]`.
- `append(Xs,[d,e],[a,b,c,d,e])?` gibt aus: `Xs=[a,b]`.
- `append(As,Bs,[a,b,c,d])?` gibt aus die verschiedenen möglichen Aufspaltungen der Liste `[a,b,c,d]`.

  $\longrightarrow$ `append` kann benutzt werden, um Listen aufzuspalten...

Bsp.: append zum Listenaufspalten

```
prefix(Xs,Ys) ← append(Xs,As,Ys).

suffix(Xs,Ys) ← append(As,Xs,Ys).

member(X,Ys) ← append(As,[X|Xs],Ys).

last(X,Xs) ← append(As,[X],Xs).
```

Bsp.: Listen Umdrehen

- Erster Versuch:

```
reverse([],[]).
reverse([X|Xs],Zs) ← reverse(Xs,Ys), append(Ys,[X],Zs).
```

Ineffizient: quadratische Anzahl von Deduktionsschritten ($\longrightarrow$ Berechnungsmodell der logischen Programme)

- Besser:

```
reverse(Xs,Ys) ← reverse(Xs,[],Ys).
reverse([X|Xs],Acc,Ys) ← reverse(Xs,[X|Acc],Ys).
reverse([],Ys,Ys).
```

5.4 Das Berechnungsmodel der Logikprogrammierung

5.4.1 Unifikation

- Eine Substitution θ heißt Unifikator für zwei Termen T_1 und T_2 falls $T_1\ \theta = T_2\ \theta$.
  - ▷ Z.B. ist θ = {`X=1, Xs=[2,3], Ys=[3,4], List=[1|Zs]`} ein Unifikator für `append([1,2,3],[3,4],List)` und `append([X|Xs],Ys,[X|Zs])`.
- Gibt es einen Unifikator für T_1 und T_2, dann heißen diese unifizierbar.

Allgemeinster Unifikator

- Ein Unifikator heißt allgemeinster Unifikator (*most general unifier* = *mgu*) für T_1 und T_2, falls es für jeden Unifikator σ eine Substitution β gibt, so dass $\sigma = \theta\ \beta$ (Die Komposition der Substitutionen θ und β).
- Es kann gezeigt werden, dass unter den Unifikatoren von Termen einen allgemeinsten gibt (wenn es überhaupt einen gibt), und dass dieser bis auf die Umbennenung von Variablen (Variantenbildung) eindeutig ist.

Berechnung des allgemeinster Unfikators

- Basiert auf Lösung von Gleichungen zwischen Termen;
- Nutzt einen Keller, um noch nicht gelöste Gleichungen zu speichern und eine Liste von Substitutionen θ, die die Substitutionen für die Ausgabe sammelt.

Eingabe: Zwei Terme T_1 und T_2

Ausgabe: *failure*, wenn T_1 und T_2 nicht unifizierbar sind oder ihr *mgu* sonst.

Algorithm zur Berechnung des allgemeinster Unfikators

```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
  hole X = Y vom stack runter
  case
    X ist eine Variable, die in Y nicht auftritt:
      ersetze X durch Y im stack und in θ
      füge X = Y zu θ hinzu
    Y ist eine Variable, die in X nicht auftritt:
      ersetze Y durch X im stack und in θ
      füge Y = X zu θ hinzu
    X und Y sind identische Konstante oder Variable: continue
    X ist f(X1, ..., Xn) und Y ist f(Y1, ..., Yn) mit f=Funktor:
      push(stack, X1 = Y1, X2 = Y2, ..., Xn = Yn)
    sonst:
      failure := true
if failure then output failure else output θ
```

mgu-Berechnung: Beispiel

- Zu unifizieren: `append([a|[b]],[c|[d]],Ls)` und
  `append([X|Xs],Ys,[X|Zs])`.
- Initialisierung:
  Stack $s = [$`append([a|[b]],[c|[d]],Ls)` $=$ `append([X|Xs],Ys,[X|Zs])`$]$;
  Substitution $\theta = \{\}$

1. $s = [$`[a|[b]]` $=$ `[X|Xs]`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$
2. $s = [$`a` $=$ `X`, `[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[X|Zs]`$]$; $\theta = \{\}$
3. $s = [$`[b]` $=$ `Xs`, `[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`$\}$
4. $s = [$`[c|[d]]` $=$ `Ys`, `Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`, `Xs=[b]`$\}$
5. $s = [$`Ls` $=$ `[a|Zs]`$]$; $\theta = \{$`X=a`, `Xs=[b]`, `Ys=[c|[d]]`$\}$
6. $s = []$; $\theta = \{$`X=a`, `Xs=[b]`, `Ys=[c|[d]]`, `Ls=[a|Zs]`$\}$

Der *occurs check* Test

```
θ := ∅; push(stack, T1 = T2); failure = false;
while not empty(stack) and not failure do
   hole X = Y vom stack runter
   case
      ....
      X ist eine Variable, die in Y nicht auftritt:
         ersetze Y durch X im stack und in θ
         füge X = Y zu θ hinzu
      ....

if failure then output failure else output θ
```

- ...stellt sicher, dass die Unifikation terminiert; Z.B. gibt es keine endliche gemeinsame Instanz von `X` und `s(X)`;
- wird aus Effizienzgründen in Implementierungen wie Prolog weggelassen.

5.5 Logik als Berechnungsmodel

- Ein logischer Kalkül ist die Erweiterung logischer Formeln um den Ableitungsbegriff. Mittels eines Kalküls kann man auch die operationale Semantik einer Programmiersprache beschreiben, d.h. formal angeben, wie Berechnungen in der Programmiersprache erfolgen.

- Syntax unserer Logikprogrammiersprache (LP-Sprache):

Atome:	A, B	$::=$	$p(t_1, \ldots, t_n)$
Ziele:	G, H	$::=$	$\top \mid \bot \mid A \mid G \wedge H$
Klauseln:	K	$::=$	$A \leftarrow G$
Programme:	P	$::=$	$\{K_1, \ldots, K_m\}$

Bemerkungen

- Die obigen Klauseln (genannt Horn- o. definite Klauseln) sind Speziallfälle von Formeln der Prädikatenlogik erster Stufe.
- Das Ziel $\top$ heißt top/true/leeres Ziel ($\longrightarrow$ Erfolg der Berechnung). Es gilt das Identitätsgesetz: $G \wedge \top \equiv G$.
- Das Ziel $\bot$ heißt bottom/false ($\longrightarrow$ Scheitern der Berechnung). Es gilt das Absorptiongesetz: $G \wedge \bot \equiv \bot$
- Wir schreiben $\wedge$ für logische Konjunktion.

5.5.1 LP-Kalkül

- Ein Zustand ist ein Paar $< G, \theta >$, wobei G ein Ziel und θ eine Substitution ist. G wird Resolvente genannt.
- Ein Anfangszustand ist ein Zustand der Form $< G, \epsilon >$, wobei ϵ die leere Substitution ist.
- Ein Zustand heißt erfolgreicher Endzustand, falls er von der Form $< \top, \theta >$ ist.
- Ein Zustand heißt erfolgloser Endzustand, falls er von der Form $< \bot, \epsilon >$ ist.

LP-Kalkül

- Eine Reduktion (ein Zustandsübergang, Ableitungsschritt) von einem Ausgangszustand S zu einem Folgezustand S' kann erfolgen, wenn bestimmte Reduktionsbedingungen erfüllt sind. Man stellt das als Reduktionsregel (Ableitungsregel, Inferrenzregel) von der Form dar:

$$\frac{\begin{array}{l}\text{Bedingung 1}\\ \ldots\\ \text{Bedingung n}\end{array}}{S \mapsto S'}$$

LP-Reduktionsregel

- Entfalten

$$\frac{(B \leftarrow H) \in P \qquad \beta \text{ ist allgemeinster Unifikator von } B \text{ und } A\ \theta}{< A \wedge G, \theta > \mapsto_{Entfalten} < H \wedge G, \theta\ \beta >}$$

- Scheitern

$$\frac{\text{Es gibt keine Klausel } (B \leftarrow H) \in P, \qquad \text{so dass ein Unifikator von } B \text{ und } A\ \theta \text{ existiert}}{< A \wedge G, \theta > \mapsto_{Scheitern} < \bot, \epsilon >}$$

LP-Kalkül

- Eine Berechnung (engl. *computation*) ist eine Sequenz $S_0 \mapsto S_1 \mapsto \ldots \mapsto S_n$ von Reduktionen.
- Eine Ableitung (engl. *derivation*) ist eine Berechnung, die entweder in einem Endzustand endet oder unendlich ist.
- Eine Ableitung ist
  - ▷ erfolgreich, wenn ihr Endzustand erfolgreich ist;
  - ▷ erfolglos, wenn ihr Endzustand erfolglos ist;
  - ▷ unendlich, wenn sie keinen Endzustand hat.

LP-Kalkül

- Ein Ziel G ist
  - ▷ erfolgreich, wenn es eine erfolgreiche Ableitung beginnend mit $< G, \epsilon >$ gibt;
  - ▷ erfolglos (endlich gescheitert), wenn es nur erfolglose Ableitungen beginnend mit $< G, \epsilon >$ hat.
- Eine Substitution θ wird Antwort eines Zieles G genannt, falls es eine erfolgreiche Ableitung $< G, \epsilon > \mapsto \ldots \mapsto < \top, \beta >$ gibt, so dass θ die eingeschränkte Substitution von β auf die Variablen von G ist.

Eine Implementierung des LP-Kalküls

Input: Ein Ziel G und ein Programm P
Output: Eine berechnete Antwort des Zieles G , wenn es eine gibt, oder *no*, sonst

Resolvente := G; $\theta := \epsilon$;

while Resolvente $\neq \top$
  sei Resolvente $= C_1 \wedge \ldots C_i \wedge A \wedge C_{i+1} \ldots C_m$
  if existiert eine (umbennante) Klausel $(A' \mapsto B_1, \ldots, B_n) \in P$,
    so dass β der mgu von A und A' ist
  then Resolvente := $(C_1 \wedge \ldots C_i \wedge B_1 \ldots B_n \wedge C_{i+1} \ldots C_m)\ \beta$
    $\theta := \theta\ \beta$
  else break

if Resolvente $= \top$
  then output θ
  else output *no*

Nichtdeterminismus im LP-Kalkül

- In unserem LP-Kalkül ist die Auswahl der Klausel innerhalb eines Programms und die Auswahl des Atoms A aus der Resolventen nicht deterministisch.
- Eine LP-Sprache (z.B. Prolog) muss den Nichtdeterminismus nach einem Schema (*scheduling policy*) auflösen.
  - ▷ Die Selektion des Atoms A kann (nur) die Länge der Ableitung beeinflußen (im schlimmsten Fall unendlich).
  - ▷ Die Selektion der Klausel kann über Erfolg oder Scheitern und über die berechnete Antwort entscheiden.

LP-Kalkül: Beispiel

Programm		Ziel
`append([],Ys,Ys)` $\leftarrow \top$.	(1)	`append([a,b],[c,d],Ls)`
`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`.	(2)	

$< \texttt{append([a,b],[c,d],Ls)}, \epsilon >$

$\mapsto_{Entfalten(2)}$ $< \texttt{append(Xs,Ys,Zs)},$
$\{\texttt{X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs]}\} >$

$\mapsto_{Entfalten(2)}$ $< \texttt{append(Xs1,Ys1,Zs1)},$
$\{\texttt{X=a,Xs=[b],Ys=[c,d],Ls=[a|Zs],X1=b,Xs1=[],Ys1=[c,d],Zs=[b|Zs1]}\} >$

$\mapsto_{Entfalten(1)}$ $< \top,$
$\{\texttt{X=a,Xs=[b],Ys=[c,d],Ls=[a,b,c,d],X1=b,Xs1=[],Ys1=[c,d],Zs=[b,c,d],}$
$\texttt{Ys2=[c,d],Xs1=[c,d]}\} >$

$\Longrightarrow$ Antwort: $\{\texttt{Ls=[a,b,c,d]}\}$.

5.5.2 Negation durch Scheitern

- Manchmal ist es natürlich negative Bedingungen zu spezifizieren.
  Z.B.:
  `bachelor(X)` ← `male(X), not married(X).`

$\Longrightarrow$ Syntax und Semantik der LP muss erweitert werden

- Syntax:

Atome:	A, B	$::=$	$p(t_1, \ldots, t_n)$
Ziele:	G, H	$::=$	$\top \mid \bot \mid A \mid \neg A \mid G \wedge H$
Klauseln:	K	$::=$	$A \leftarrow G$
Programme:	P	$::=$	$\{K_1, \ldots, K_m\}$

Negation durch Scheitern: Semantik

- Ein Ziel $\neg A$ ist erfolgreich genau dann, wenn das Ziel A endlich scheitert.
- Diese Art der Negation wird Negation durch Scheitern genannt (*Negation as Failure, NaF*).

$\Longrightarrow$ Reduktionsregeln für negierte Atome

Reduktionsregeln für negierte Atome

- Scheitern NaF

$$\frac{< A, \theta > \mapsto^* < \top, \beta >}{< \neg A \wedge G, \theta > \mapsto < \bot, \epsilon >}$$

- Erfolg NaF

$$\frac{\text{Jede Ableitung von } A \text{ scheitert endlich: } < A, \theta > \mapsto^* < \bot, \epsilon >}{< \neg A \wedge G, \theta > \mapsto < G, \theta >}$$