Programmiersprachen

Alexandru Berlea

Institut für Informatik
TU München

Wintersemester 2006/2007

mgu-Berechnung: Beispiel

- Zu unifizieren: `append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

mgu-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

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

mgu-Berechnung: Beispiel

- Zu unifizieren: `append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

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]`$]$; $\theta = \{\}$

mgu-Berechnung: Beispiel

- Zu unifizieren: `append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

0. Initialisierung:
   $s = [\texttt{append([a|[b]],[c|[d]],Ls)} = \texttt{append([X|Xs],Ys,[X|Zs])}]$;
   $\theta = \{\}$

1. $s = [\texttt{[a|[b]]} = \texttt{[X|Xs]}, \texttt{[c|[d]]} = Ys, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

2. $s = [\texttt{a} = \texttt{X}, \texttt{[b]} = \texttt{Xs}, \texttt{[c|[d]]} = Ys, \texttt{Ls} = \texttt{[X|Zs]}]$; $\theta = \{\}$

3. $s = [\texttt{[b]} = \texttt{Xs}, \texttt{[c|[d]]} = Ys, \texttt{Ls} = \texttt{[a|Zs]}]$; $\theta = \{\texttt{X=a}\}$

mgu-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

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]`$\}$

mgu-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

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]]`$\}$

mgu-Berechnung: Beispiel

- Zu unifizieren:`append([a|[b]],[c|[d]],Ls)` und `append([X|Xs],Ys,[X|Zs])`.

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

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]`$\}$

Logik als Berechnungsmodell

- 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):

Atomische Ziele:	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.
- Wir schreiben $\wedge$ für logische Konjunktion.
- Das Ziel $\top$ heißt top/true/leeres Ziel (☞ Erfolg der Berechnung). Es gilt das **Identitätsgesetz**: $G \wedge \top \equiv G$.
- Das Ziel $\bot$ heißt bottom/false (☞ Scheitern der Berechnung). Es gilt das **Absorptiongesetz**: $G \wedge \bot \equiv \bot$

Zustände

- 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.

Reduktionen

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, Inferenzregel) von der Form dar:

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

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**

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

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; *no*, sonst

```
Resolvente := G; θ := ε;

while Resolvente ≠ ⊤
  sei Resolvente = C_1 ∧ ··· ∧ C_i ∧ A ∧ C_{i+1} ∧ ··· ∧ C_m
  if ∃ eine (umbennante) Klausel (A' ← B_1 ∧ ··· ∧ B_n) ∈ P ,
     so dass β der mgu von A und A' ist
  then Resolvente := (C_1 ∧ ... C_i ∧ B_1 ∧ ··· ∧ B_n ∧ C_{i+1} ∧ ··· ∧ C_m) β
          θ := θ β
  else break

if Resolvente = ⊤
  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 atomischen Ziels A aus der Resolventen **nicht deterministisch**.

- Eine LP-Sprache (z.B. Prolog) muss den Nichtdeterminismus nach einem Schema (*scheduling policy*) auflösen.
  - Die Selektion des atomischen Ziels 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: $\texttt{append([],Ys,Ys)} \leftarrow \top.$ (1)

$\texttt{append([X|Xs],Ys,[X|Zs])} \leftarrow \texttt{append(Xs,Ys,Zs)}.$ (2)

Ziel: $\texttt{append([a,b],[c,d],Ls)}$

Berechnung:

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

LP-Kalkül: Beispiel

Programm:

$$\texttt{append([],Ys,Ys)} \leftarrow \top. \qquad (1)$$

$$\texttt{append([X|Xs],Ys,[X|Zs])} \leftarrow \texttt{append(Xs,Ys,Zs).} \qquad (2)$$

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

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

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

LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

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

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

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

LP-Kalkül: Beispiel

Programm:
```
append([],Ys,Ys) ← ⊤.                                    (1)
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).             (2)
```

Ziel:
```
append([a,b],[c,d],Ls)
```

Berechnung:

```
                 < append([a,b],[c,d],Ls), ε >

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

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

↦Entfalten(1)    < ⊤,
 {X=a,Xs=[b],Ys=[c,d],Ls=[a,b,c,d],X1=b,Xs1=[],Ys1=[c,d],Zs=[b,c,d],
                                              Ys2=[c,d],Zs1=[c,d]} >
```

LP-Kalkül: Beispiel

Programm: `append([],Ys,Ys)` $\leftarrow \top$. (1)

`append([X|Xs],Ys,[X|Zs])` $\leftarrow$ `append(Xs,Ys,Zs)`. (2)

Ziel: `append([a,b],[c,d],Ls)`

Berechnung:

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

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

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

$\mapsto_{\textit{Entfalten}(1)}$ $< \top$,
`{X=a,Xs=[b],Ys=[c,d],Ls=[a,b,c,d],X1=b,Xs1=[],Ys1=[c,d],Zs=[b,c,d],`
`Ys2=[c,d],Zs1=[c,d]}` $>$

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

Negation

- Manchmal ist es natürlich negative Bedingungen zu spezifizieren. Z.B.:

```
bachelor(X) ← male(X), not married(X).
```

⟹ Syntax und Semantik der LP muss erweitert werden

- Syntax:

Atomische Ziele:	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 Ziele

Reduktionsregeln für negierte Ziele

- **Scheitern NaF:** *es gibt eine erfolgreiche Ableitung von A*

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

- **Erfolg NaF:** *es gibt keine erfolgreiche UND keine unendliche Ableitung von A*

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

Negation durch Scheitern: Bemerkungen

- NaF ist eine eingeschränkte Form der Negation aus der Prädikatenlogik erster Stufe.
- NaF führt unter Umständen zu semantischen Problemen in Zusammenhang mit Vollständigkeit und Terminierung.
- NaF zerstört die Eigenschaft des LP-Kalküls ohne Negation, dass erfolgreiche Ableitungen erhalten bleiben, wenn man dem Programm Klauseln hinzufügt. Z.B.:

  $P = \{\texttt{p(a).}\} \implies \texttt{p(b)}$ scheitert. $\implies \neg\texttt{p(b)}$ ist erfolgreich.

  $P = \{\texttt{p(a).}$
  $\quad\texttt{p(b).}\} \implies \texttt{p(b)}$ ist erfolgreich. $\implies \neg\texttt{p(b)}$ scheitert.

Prolog

- ... ist die bekannteste Implementierung einer LP-Sprache;
- wurde Anfang der 1970er von Alain Colmerauer (Marseille) und Robert Kowalski (Edinburgh) entwickelt.
- konkretisiert den vorgestellten LP-Kalkül zur Bearbeitung von Zielen durch:
  - Auflösung des Nichtdeterminismus der Auswahl von Klauseln und Atomen nach einem festen Schema;
  - Erlauben der Negation durch Scheitern nur für zur Laufzeit unter den aktuellen Substitutionen variablenfreie Ziele.

$\Longrightarrow$ SLDNF-Resolution (Linear resolution with Selection function for Definite clauses with Negation as Failure)

Auswahl von Klauseln und Atomen in Prolog

- In Prolog wird immer das am weitesten links stehende **Literal** (atomisches Ziel oder negiertes atomisches Ziel) eines Ziels selektiert und ganz entfaltet.

- Klauseln werden in textueller Reihenfolge ausgewählt.
  - Scheitert eine Ableitung mit einer bestimmten Klausel, versucht man eine neue Ableitung für das Literal mit Hilfe der nächsten Klausel. (Rücksetzen, **backtracking**)
  - Wird eine erfolgreiche Ableitung gefunden, werden weitere erfolgreiche Ableitungen gesucht, indem man immer das zuletzt gewählte Literal, bei dem noch Klauseln zur Auswahl stehen, erneut zu entfalten versucht.

- Die Auswahl der Klauseln bei der Suche durch Rücksetzen wird als **don't know Nichtdeterminismus** bezeichnet.

Suchbäume

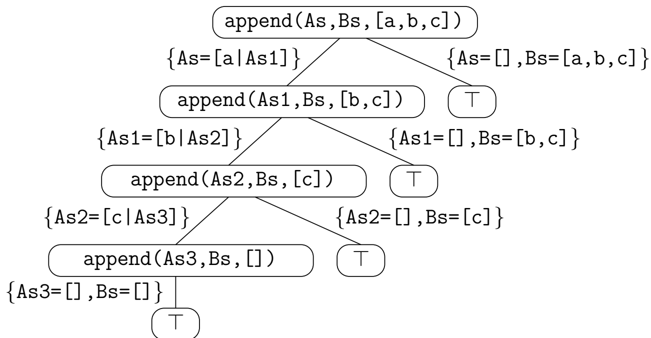
- Ein **Suchbaum** eines Zieles G_{start} bezüglich eines Programms P ist wie folgt definiert:
  - Knoten sind Ziele.
  - Die Wurzel des Baumes ist G_{start}.
  - Für jede anwendbare Entfalten-Reduktionsregel:

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

    existiert eine mit β beschrifteter Kante vom Knoten $A \wedge G$ zum Knoten $H \wedge G$.
- Durch die Auswahlstrategie von Prolog entspricht jedem Ziel genau ein Suchbaum.

Suchbäume

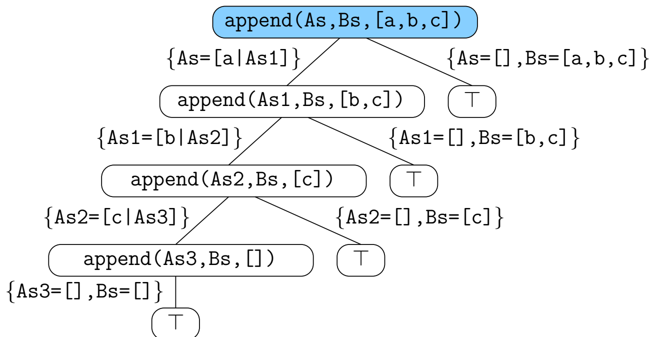
```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
append([],Ys,Ys).
```



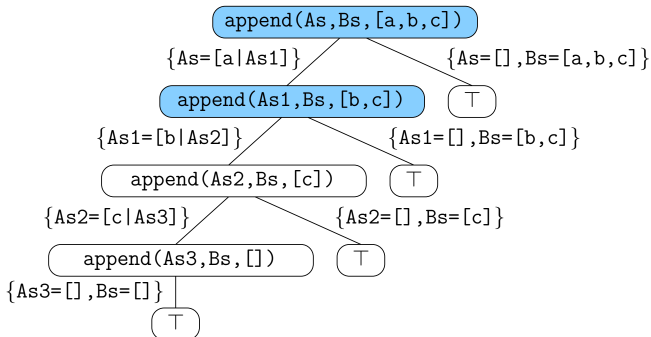
Suchbäume

- Die Blätter eines Suchbaumes, wo das leere Ziel erreicht wird, heißen **Erfolgsknoten**.
- Der Pfad zu einem Erfolgsknoten entspricht der Berechnung einer Antwort.
- Erfolgsknoten entsprechen einer berechneten Antwort.
- Die übrigen Blätter heißen **Scheiternknoten**.
- Die Auswertung eines Zieles in Prolog entspricht einem Tiefendurchlauf über den entsprechenden Suchbaum. Wird ein Erfolgsknoten erreicht, so werden die entsprechenden Substitutionen für die Variablen in der Anfrage ausgegeben. Der Benutzer hat die Möglichkeit nach weiteren Lösungen suchen zu lassen.

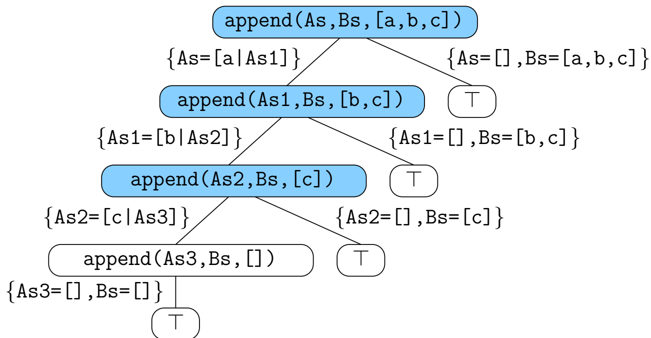
Prolog's Suchstrategie



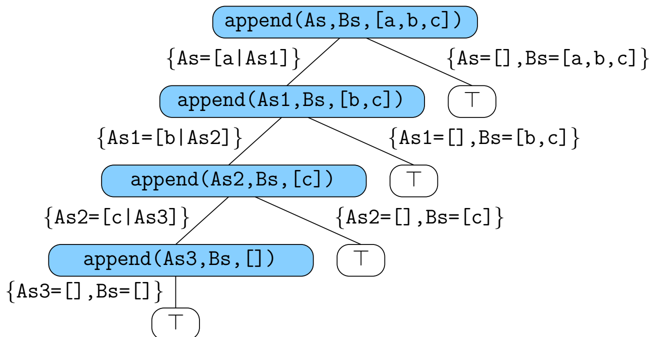
Prolog-Suchbaumdurchläufe



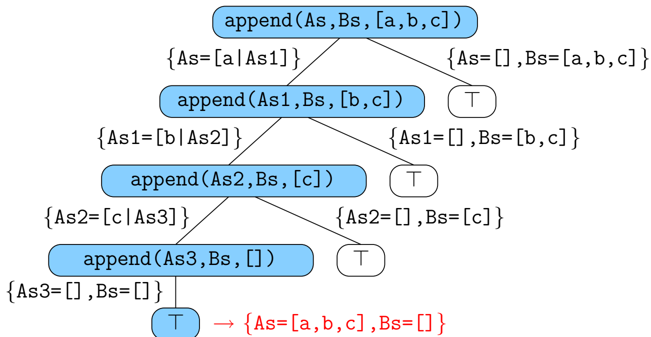
Prolog-Suchbaumdurchläufe



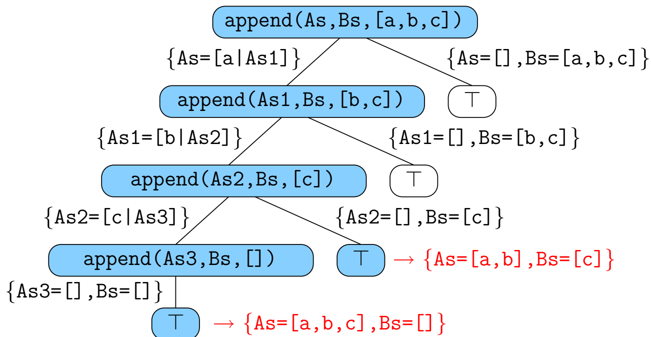
Prolog-Suchbaumdurchläufe



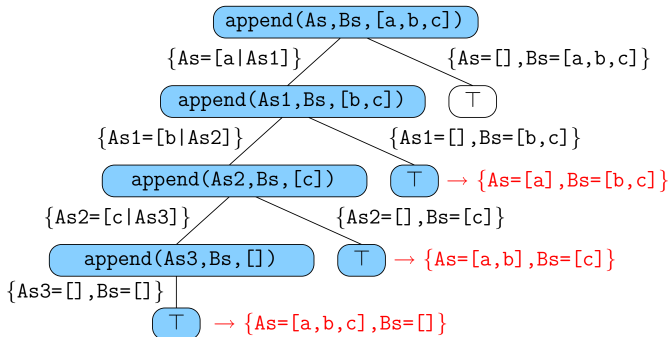
Prolog-Suchbaumdurchläufe



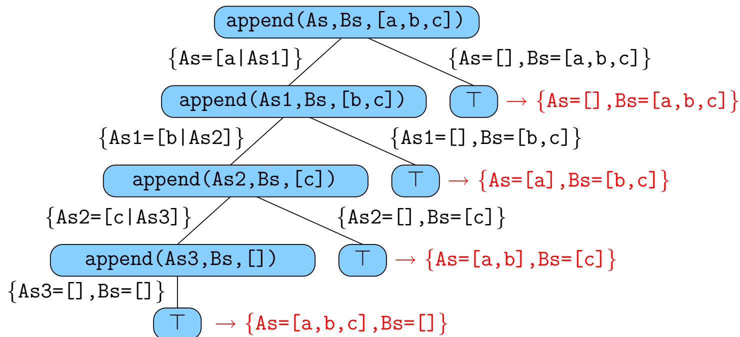
Prolog-Suchbaumdurchläufe



Prolog-Suchbaumdurchläufe



Prolog-Suchbaumdurchläufe



Folgen der Tiefensuche

- Die Tiefensuche ist effizient und einfach zu implementieren, aber...
- Unter Umständen ist der zuerst gefolgte Pfad unendlich, so dass andere eventuell existierende Erfolgsknoten nicht mehr erreicht werden können.
- Durch die Änderung der Reihenfolge der Klauseln und Literale kann erreicht werden, dass unendliche Berechnungen vermieden werden oder später auftretten.

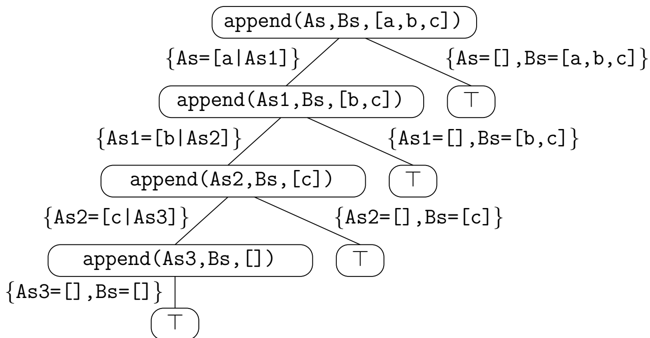
$\Longrightarrow$ Deklarativität der Logikprogrammierung wird (zugunsten der Effizienz) verletzt.

Fazit

- Aus Sicht der LP-Programmierung ist die Reihenfolge der Klauseln und der Ziele irrelevant.
- Die Effizienz der Prolog-Programme allerdings hängt oft maßgeblich von dieser Reihenfolge ab.
- Im Extremfall (oft) terminieren korrekte LP-Programme nicht für eine bestimmte Anordnung der Klauseln und der Ziele.

Reihenfolge in der Lösungen gefunden werden

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



Reihenfolge in der Lösungen gefunden werden

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

```
?- append(As,Bs,[a,b,c]).
As = [a, b, c]
Bs = [] ;

As = [a, b]
Bs = [c] ;

As = [a]
Bs = [b, c] ;

As = []
Bs = [a, b, c] ;
No
```

Reihenfolge in der Lösungen gefunden werden

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

```
?- append(As,Bs,[a,b,c]).
As = []
Bs = [a, b, c] ;

As = [a]
Bs = [b, c] ;

As = [a, b]
Bs = [c] ;

As = [a, b, c]
Bs = [] ;
No
```

Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 1. Versuch, eine kommutative Relation zu definieren.

```
married(X,Y) :- married(Y,X).
married(abraham,sarah).
```

```
?- married(abraham,sarah).
```
Nichtterminierende Berechnung

Grund:

```
married(abraham,sarah)
   married(sarah,abraham)
      married(abraham,sarah)
         married(sarah,abraham)
            ...
```

Terminierung

Rekursive Klauseln können zu Nichtterminierung führen.

Bsp.: 2. Versuch, eine kommutative Relation zu definieren.

```
married(abraham,sarah).
married(X,Y) :- married(Y,X).
```

```
?- married(abraham,sarah).
Yes
?- married(sarah,abraham).
Yes
?- married(lot,sarah).
```
Nichtterminierende Berechnung

Terminierung

Kommutative Relationen können mit einem neuen Prädikat definiert werden, das eine Klauseln für jede Permutation der Argumente der Relation hat:

```
are_married(X,Y) :- married(X,Y).
are_married(X,Y) :- married(Y,X).
married(abraham, sarah).
```

```
?- are_married(abraham, sarah).
Yes
?- are_married(sarah, abraham).
Yes
?- are_married(sarah, lot).
No
```

Anordnungen der Literale

Die Anordnungen der Literale bestimmen den Prolog-Suchbaum.
(im Unterschied zur Anordnung der Klausel, die nur die Reihenfolge ändert, in der Teilbäume besucht werden sollen.)

- kann die Effizienz maßgeblich beeinflußen.
- kann bestimmen, ob eine Berechnung terminiert oder nicht.

Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  $\longrightarrow$ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  $\longrightarrow$ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

Anordnung der Literale und Effizienz

Bsp.: Berechnung für `son(X,lot)`

- 1. Möglichkeit: `son(X,Y) :- male(X),parent(Y,X).`
  ⟶ man betrachtet die Männer nacheinander und prüft, ob sie Kinder von `lot` sind.

- 2. Möglichkeit: `son(X,Y) :- parent(Y,X),male(X).`
  ⟶ man betrachtet die Kinder von `lot` nacheinander und prüft, ob sie Männer sind.

⟹ Die zweite Anordnung ist günstiger für die Anfrage `son(X,lot)` aber die erste ist besser für `son(sarah,X)`

⟹ Die optimale Anordnung hängt von der beabsichtigten Benutzung ab.