Unifikation (1)

Allgemeines:

- Mechanismus zur Parameter-Übergabe.
- Zuweisung an Variablen (einmalig, symmetrisch).
- „Pattern-Matching“ (Muster-Vergleich).

Substitution:

- Abbildung von Variablen auf Terme.

  D.h. Variablen $\longrightarrow$ Konstanten, strukturierte Terme oder Variablen.
- subst: **array** [Variablen] **of** Term.

	subst
X:	a
Y:	Z
Z:	Z

- Übliche Notation: $\{X/a, Y/Z\}$.

  Uninstanziierte (freie) Variablen werden nicht dargestellt.

Anwendung einer Substitution auf Literal:

- Jede Variable durch ihren Wert ersetzen.
- $p(X, b, f(Y, Z))\ \{X/a, Y/Z\} = \ p(a, b, f(Z, Z))$.

  Für Substitutionen ist Postfixnotation üblich.

Unifikation (2)

Anwendung einer Substitution (Forts.):

- Rekursiv entsprechend der Termstruktur:

```
function apply(subst, term): term
    if term ist Konstante c then
        return c;
    else if term ist Variable v then
        return subst[v]
    else if term ist zusammengesetzer Term f(t1,...,tn) then
        return f(apply(subst, t1),..., apply(subst, tn))
```

- Prolog konstruiert nicht wirklich den neuen Term, sondern nimmt bei jedem Zugriff auf eine Variable automatisch ihren Wert aus der Tabelle.

Unifikator zweier Literale λ_1 und λ_2:

- Substitution θ mit $\lambda_1\theta = \lambda_2\theta$.
- λ_1 und λ_2 heißen unifizierbar gdw. es so ein θ gibt.
- Interessant sind nur allgemeinsten Unifikatoren, die keine überflüssigen Ersetzungen vornehmen.
- Formal: θ ist ein allgemeinster Unifikator von λ_1 und λ_2 gdw. es für jeden anderen Unifikator θ' eine Substitution σ gibt mit $\theta' = \theta \circ \sigma$. ($\theta \circ \sigma$ sei die Komposition/Hintereinanderausführung von θ und σ).
- Lemma: Wenn es einen Unifikator gibt, dann gibt es auch einen allgemeinsten.
- Lemma: Der allgemeinste Unifikator ist bis auf Variablenumbenennungen eindeutig.

Unifikation (3)

Beispiele:

- $p(X, b)$, $p(a, Y)$: unifizierbar durch $\{X/a, Y/b\}$.
- $q(a)$, $q(b)$: nicht unifizierbar.
- $q(X)$, $q(Y)$: unifizierbar durch $\{X/Y\}$.
  $\{X/a,\ Y/a\}$ ist Unifikator, aber nicht allgemeinster Unifikator.

Aufgaben:

Bestimmen Sie Unifikatoren, falls möglich:

- $\mathit{flaeche}(\mathit{quadr}(3), E)$ und $\mathit{flaeche}(\mathit{quadr}(X), F)$.
- $\mathit{length}([1, 2, 3], X)$ und $\mathit{length}([], 0)$.
- $\mathit{length}([E], X)$ und $\mathit{length}([], 0)$.
- $\mathit{length}([1, 2, 3], X)$ und $\mathit{length}([E|R], N1)$.
- $\mathit{append}(X, [2, 3], [1, 2, 3])$ und
  $\mathit{append}([E|R], L, [E|A])$.
- $p(f(X), Z)$ und $p(Y, a)$.
- $p(f(a), g(X))$ und $p(Y, Y)$.
- $q(X, Y, h(g(X)))$ und $q(Z, h(Z), h(Z))$.
- Lassen Sie die Lösungen von Prolog überprüfen.
  Berücksichtigen Sie aber ggf. den fehlenden “Occur check” (s.u.).

Unifikation (4)

unify(Term/Literal t, u): Substitution θ:

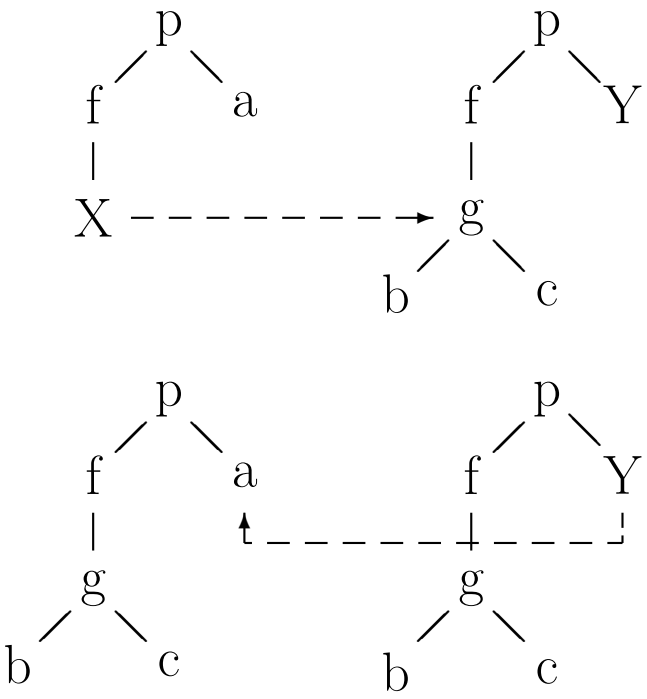
if $t = u$ **then**
  $\theta := \{\}$
elif t ist Variable v, die in u nicht vorkommt **then**
  $\theta := \{v/u\}$
elif u ist Variable v, die in t nicht vorkommt **then**
  $\theta := \{v/t\}$
elif t ist $f(t_1, \ldots, t_n)$ und u ist $f(u_1, \ldots, u_n)$ **then**
  $\theta := \{\}$;
  for $i = 1$ **to** n **do**
    $\theta := \theta \circ \text{unify}(t_i\theta, u_i\theta)$
else /* verschiedene Funktoren/Konstanten */
  $\theta :=$ „nicht unifizierbar“

Beispiele, Forts.:

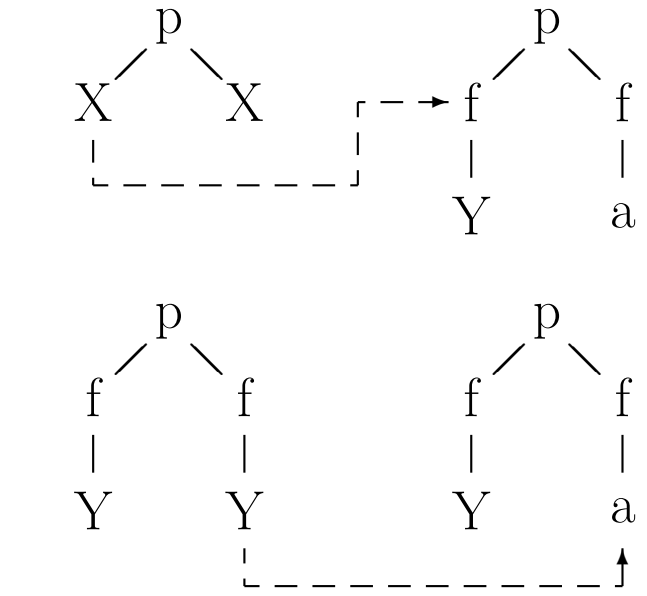
- $p(X, X)$, $p(a, b)$: nicht unifizierbar.
  Zuerst berechnet man die Bindung $\{X/a\}$. Danach sind $p(a, a)$ und $p(a, b)$ zu unifizieren. Das ist aber nicht möglich.
- $p(X, X)$, $p(Y, f(Y))$: nicht unifizierbar.
  Wegen Vorkommenstest (“occur check”).

Unifikation (5)

Beispiel:



Beispiel:



Unifikation (6)

Aufwand:

- $p(X_1, \ldots, X_n)$, $p(f(X_0, X_0), \ldots, f(X_{n-1}, X_{n-1}))$:
  $\theta = \{\ X_1/f(X_0, X_0),$
  $X_2/f(f(X_0, X_0), f(X_0, X_0)),$
  $\ldots\ \}.$
  Der Unifikator θ ersetzt X_k durch einen Term t_k mit $2^{k+1} - 1$ Vorkommen von f.
- Der Test, ob X_k in t_k vorkommt, kostet exponentiellen Aufwand.
- Außerdem kostet auch eine explizite Repräsentation von θ exponentiellen Aufwand. Das ist aber kein Problem, da man normalerweise mit Zeigern arbeitet und Teilterme doppelt nutzen kann.

Lösungen:

- Den „occur check“ weglassen (übliche Lösung).
  Dies kann zu falschen Ergebnissen führen:
  $p \leftarrow q(X, X)$.
  $q(Y, f(Y))$.
  Prolog-Systeme ohne „occur check“ beantworten „p“ mit „yes“.
  Die Unifikation kann dadurch auch in eine Endlosschleife geraten.
- Es gibt Algorithmen mit komplizierteren Datenstrukturen, die eine lineare Laufzeit haben.

Resolution (1)

Allgemeines:

- Vereinfache Anfrage schrittweise zu "true".
- Ersetze dazu Regel-Kopf durch Regel-Rumpf.
- Das Anfrage-Literal und der Regel-Kopf müssen mit einem Unifikator gleich gemacht werden.
- Fakten zählen als Regeln mit leerem Rumpf, sie führen also zu einer Verkürzung der Anfrage.

Beispiel:

- vorfahr(anke,gerd).
  vorfahr(X,Z) :- elternteil(X, Y), vorfahr(Y, Z). $\theta := \{X/anke, Z/gerd\}$.
- elternteil(anke, Y), vorfahr(Y, gerd).
  vorfahr(X, Y) :- mutter(X, Y).
- mutter(anke, Y), vorfahr(Y, gerd).
  mutter(anke, dora). $\theta := \{Y/dora\}$.
- vorfahr(dora, gerd).
  vorfahr(X, Y) :- vater(X, Y).
- vater(dora, gerd).
  vater(dora, gerd).
- $\square$ ("true").

Resolution (2)

Resolutions-Schritt:

- Anfrage: ?- $A_1, A_2, \ldots, A_n$.
- Regel: B_0 :- $B_1, \ldots, B_m$.
- Nenne Variablen der Regel so um, daß von allen bisher verwendeten verschieden: B'_0 :- $B'_1, \ldots, B'_m$.
  Jeder Aufruf der Regel hat ja seinen eigenen lokalen Variablensatz.
- Bestimme Unifikator θ von A_1 und B'_0.
- Neue Anfrage: ?- $B'_1\theta, \ldots, B'_m\theta, A_2\theta, \ldots, A_n\theta$.

Aufgabe:

- ?- $append(L,\ [1,2,3],\ [1,2,3])$.
  $append([\,],\ L,\ L)$.
- ?- $append([1],\ [2,3],\ X)$.
  $append([E|R],\ L,\ [E|A])$:- $append(R,\ L,\ A)$.
  anschließend: $append([\,],\ L,\ L)$.

Berechnete Antwort:

- Werte der Anfragevariablen in der Komposition aller benutzen Unifikatoren.

Resolution (3)

Algorithmus 1:

procedure prove($[A_1, \ldots, A_n]$)
  if $n = 0$ **then print** "yes";
  else wähle Regel B_0 :- $B_1, \ldots, B_m$;
    nenne Variablen um: B'_0 :- $B'_1, \ldots, B'_m$;
    θ := unify(A_1, B'_0);
    if nicht unifizierbar **then** Abbruch;
    prove($[B'_1\theta, \ldots, B'_m\theta, A_2\theta, \ldots, A_n\theta]$)

Algorithmus 2:

var θ: Substitution; /* Initialisiert mit {} */
procedure execute(A: Literal)
  print "Call $A\theta$";
  wähle Regel B_0 :- $B_1, \ldots, B_m$;
  nenne Variablen um: B'_0 :- $B'_1, \ldots, B'_m$;
  $\theta := \theta \circ$ unify($A\theta, B'_0$);
  if nicht unifizierbar **then** Abbruch;
  for $i := 1$ **to** m **do**
    execute(B'_i);
  print "Exit $A\theta$";

Backtracking (1)

Allgemeines:

- Obige Algorithmen sind nichtdeterministisch.

  D.h. es wird nur garantiert, daß es Auswahlmöglichkeiten gibt, die zum Ziel (**print** "yes") führen.

- Man muß die richtige Regel „raten".

  Eine falsche Auswahl führt zum Abbruch und nicht zu einem falschen Ergebnis (sonst wäre der Algorithmus ja sinnlos).

- Lösung: Alle Möglichkeiten durchprobieren.

Backtracking:

- Falls mehrere anwendbare Regeln: Berechnungszustand sichern, erste Regel probieren.
- Statt „Abbruch" zurück zum letzten gesicherten Zustand. Dort nächste Möglichkeit probieren.

  „Letzten" im Sinne des LIFO (Stack-) Prinzips.

- "Backtracking": Spur zurück verfolgen, bis zur letzten noch nicht probierten Abzweigung.

Backtracking (2)

Darstellung der Alternativen als Baum:

- Jeder Knoten entspricht einem Berechnungszustand (aktuelle Anfrage).
- Nachfolge-Knoten für mögliche Fortsetzungen.
- Backtracking entspricht "depth-first" Suche.

Beispiel:

- ?- p(X), write(X), s(X).
- p(X) :- write('erste Regel über p'), q(X).
  p(X) :- write('zweite Regel über p'), r(X).
- q(a).
  q(b).
- r(c).
  r(d).
- s(d).

Bemerkung zu Systemprädikaten:

- Systemprädikate haben meist nur eine Lösung.
- "write" gelingt beim ersten Aufruf sofort (d.h. CALL⟶EXIT). Es gelingt aber nur ein Mal (d.h. REDO⟶FAIL).

Backtracking (3)

Ein einfacher Prolog-Interpreter:

- **procedure** prove($[A_1, \ldots, A_n]$)
  if $n = 0$ **then print** "yes";
  else for each Regel B_0 :- $B_1, \ldots, B_m$ **do**
  nenne Variablen um: B'_0 :- $B'_1, \ldots, B'_m$
  θ := unify(A_1, B'_0);
  if unifizierbar **then**
  prove([$B'_1\theta, \ldots, B'_m\theta,$
  $A_2\theta, \ldots, A_n\theta$])
- Sichern des aktuellen Zustandes: Anwendung der Substitution erzeugt immer neue Kopie.
- Berechnung der Antwort: Antwortvariablen als zweites Argument, θ jeweils darauf anwenden.

Aufgabe*:

Angenommen, Ihr Interpreter hat keinen Trace-Modus. Definieren Sie ein Prädikat p', das sich wie ein Prädikat p verhält, aber zusätzlich Ausgaben entsprechend dem Vierport-Modell macht.

Das Systemprädikat `fail` (logisch falsch) führt zum Backtracking.

Der "Cut" (1)

Vorbemerkungen:

- Das Sichern des Systemzustandes ("Choicepoint" erzeugen) ist recht aufwendig, besser vermeiden.
- Man kann dem Interpreter mitteilen, daß er bestimmte Alternativen nicht betrachten muß.

Wirkung:

- Der „cut", geschrieben `!`, entfernt alle bisherigen Alternativen für den aktuellen Aufruf.

  Das bedeutet: Die aktuelle Klausel wird festgeschrieben und alle Werte für die in dieser Klausel bisher gebundenen Variablen.

Beispiel:

- ```
  p(X) :- q(X), !, r(X).
  p(X) :- s(X).
  q(a).
  q(b).
  r(X).
  s(c).
  ```
- Mit Cut liefert `p(X)` nur `X=a`, ohne Cut kommen auch `b` und `c` heraus.
  ```

Der "Cut" (2)

Aufgabe:

Zeichnen Sie ein Vierport-Diagramm zum Beispiel-Programm (mit und ohne Cut).

Der Cut ist nicht ungefährlich:

- Ursprüngliche Idee: Teile des Beweisbaums abschneiden, die ohnehin scheitern würden.
  Man spricht dann von einem „grünen" Cut (bzw. „blauen" Cut, falls das Prolog-System dies eigentlich selbst erkennen sollte).
- Der Interpreter hält sich an den „Hinweis" aber auch dann, wenn er falsch ist, die abgeschnittenen Alternativen also weitere Lösungen enthalten.
  Wie in dem Beispiel. Solche Cuts werden „rote" Cuts genannt.
- Beide Arten von Cuts sind manchmal nützlich, aber sollten sparsam verwendet werden.

Bemerkung:

- Bessere Interpreter haben einen Index über dem äußersten Funktor des ersten Arguments. Dies schränkt die anwendbaren Regeln auch schon ein.