Theoretische Grundlagen des Software Engineering

13: Konfluenz und Termination

Stephan Schulz
schulz@eprover.org

Matching – Anmerkungen

Matching wird in vielen modenen Programmiersprachen eingesetzt:

- Haskell
- F#
- Python
- Aber auch: Finden der richtigen Methode bei Polymorphismus in C++

Matching wird in fast allen symbolischen Systemen eingesetzt:

- Expertensystem
- Planungssysteme
- Theorembeweiser
- Computeralgebrasystemen
- IDE's ("Intellisense")

Matching

Frage: Wie findet man einen Match?

Idee: Traversiere s und t in Vorordnung, baue (Repräsentation von) Substitution σ

Situationen:

- s ist eine Variable
  - Fall 1: s ist in σ nicht gebunden: Erweitere σ um $s \leftarrow t$
  - Fall 2: s ist gebunden, $\sigma(s) = t$: Ok, matcht schon
  - Fall 3: s ist gebunden, $\sigma(s) \neq t$: Es gibt keinen Match!
- s ist von der Form $f(s_1, \ldots, s_n)$
  - Fall 1: t ist von der Form $f(t_1, \ldots, t_n)$: Mache mit den s_i, t_i weiter
  - Fall 2 : t ist von anderer Form: Es gibt keinen Match!

Matching

```
def matchTerms(s,t, sigma={}):
    if isVar(s):
        if not s in sigma:
            sigma[s] = t
            return sigma
        elif sigma[s] == t:
            return sigma
        else:
            return None
    elif isVar(t):
        return None
    elif getFun(s)!=getFun(t):
        return None
    else:
        for s,t in zip(getArgs(s), getArgs(t)):
            if matchTerms(s,t, sigma) == None:
                return None
        return sigma
```

Programme als Terme (Beispiel)

Beispielprogramm

```
def fak1(a):
    if a==0:
        return 1
    else:
        return a*fak1(a-1)
```

Beispiel: Programme als Terme (Signatur Σ)

$arg\|_2$	Konstruktor für Argumentlisten
$stmt\|_2$	Konstruktor für Befehlslisten
$end\|_0$	Listenende
$fcall\|_2$	Funktionsaufruf
$def\|_3$	Kodiert **def**
$return\|_1$	Kodiert **return**
$equal\|_2$	Kodiert **==**
$mult\|_2$	Kodiert *
$minus\|_2$	Kodiert **-**
$fak1\|_0$	Name aus dem Programm
$a\|_0$	Name aus dem Programm
$0\|_0$	Konstante aus dem Programm
$1\|_0$	Konstante aus dem Programm

Programme als Terme (Kodiertes Program)

```
def(fak1, arg(a, end),
    stmt(if(equal(a, 0),
            stmt(return(1),
                 end),
            stmt(return(mult(a,
                             fcall(fak1,
                                   arg(minus(a,1)),
                                       end))),
                 end),
         end)
```

Programme als Terme (Rewriting)

Mögliche Anwendung: Optimierung

- $mult(x, 0) \rightarrow 0$
- $mult(0, x) \rightarrow 0$
- $mult(x, 1) \rightarrow x$
- $mult(1, x) \rightarrow x$
- $minus(x, 0) \rightarrow x$

. . . aber auch denkbar:

- Regeln, die nutzlose **else**- Statements entfernen
- Regeln, die konstante Ausdrücke auswerten
- Regeln, die tote **if**/**else** Zweige entfernen

Rewrite- Relation

Definition

Seien Σ, V gegeben und sei R eine Menge von Regeln über $Term(\Sigma, V)$. R definiert the Rewrite- Relation $\rightarrow_R$ auf $Term(\Sigma, V)$ wie folgt:

$$s \rightarrow_R t$$

gdw

$$\exists p \in Pos(s), l \rightarrow r \in R, \sigma : s|_p = \sigma(l) \text{ und } t = s[\sigma(r)]_p$$

Wenn die linke Seite einer Regel auf einen Teilterm matched, kann dieser durch die instanziierte rechte Seite ersetzt werden

Konfluenz, Termination, Konvergenz

Definition

Das Regelsystem R heißt...

- lokal konfluent
- konfluent
- terminierend
- konvergent

...wenn die zugehörige Rewrite- Relation $\rightarrow_R$ lokal konfluent, konfluent, terminierend, konvergent ist.

Unentscheidbarkeit

Satz

Im allgemeinen ist unentscheidbar, ob ein Regelsystem R

- konfluent
- terminierend
- konvergent

ist

Beweisidee: Regelsysteme können beliebige Berechnungen kodieren, u.a. das Halteproblem und das Wortproblem

Entscheidbarkeit

Satz

Für terminierende Regelsysteme ist Konfluenz entscheidbar!

Idee:

- Zeige lokale Konvergenz
- Reduzierbar auf endlich viele Divergenzen (Kritische Paare)
- Zeige Zusammenführbarkeit dieser kritischen Paare

Termination?

- Hinreichende Kriterien
- Insbesondere Reduktionsordnungen

Reduktionsordnungen

Definition

Eine Reduktionsordnung $>$ ist eine Partialordnung auf $Term(\Sigma, V)$ mit folgenden Eigenschaften:

- $>$ ist terminierend
- $>$ ist verträglich mit Substitutionen: $s_1 > s_2$ impliziert $\sigma(s_1) > \sigma(s_2)$
- $>$ ist verträglich mit der Termstruktur: $s_1 > s_2$ impliziert $t[s_1]|_p > t[s_2]|_p$

Termination von Rewrite- Systemen

Satz

Sei R ein Regelsystem. $\rightarrow_R$ terminiert genau dann, wenn es eine Reduktionsordnung $>$ gibt, so dass $l > r$ für alle $l \rightarrow r \in R$ gilt.

Unifikation

Definition

Seien Σ, V gegeben und seien $s, t \in Term(\Sigma, V)$. Seien σ, τ Substitutionen.

- Eine Substitution Σ heißt Unifikator von s und t, falls $\sigma(s) = \sigma(t)$
- Die Menge der Unifikatoren von s und t heißt $\mathcal{U}(s, t)$.
- σ heißt allgemeiner als τ, falls eine Substitution ρ existiert und $\sigma = \tau \circ \rho$.
- σ heißt allgemeinster Unifikator, falls $\sigma \in \mathcal{U}(s, t)$ und für alle $\tau \in \mathcal{U}(s, t)$ gilt: σ ist allgemeiner als τ.

Fakt: Der allgemeinste Unifikator von s und t ist bis auf Variablenpermutationen eindeutig. Wir bezeichnen ihn mit $mgu(s, t)$.

Kritische Paare

Definition

Ein kritisches Paar ist das Ergebnis einer Überlappung von zwei Regen an einer nicht-Variablen-Position.

Formal: Seien $l_1 \rightarrow r_1$ und $l_2 \rightarrow r_2$ zwei Regeln über $Term(\Sigma, V)$. Sei $p \in Pos(l_1)$ und $l_1|_p \notin V$. Sei $\sigma = mgu(l_1|_p, l_2)$.

- $\langle \sigma(l_1[r_2]_p), \sigma(r_2) \rangle$ ist ein kritisches Paar zwischen $l_1 \rightarrow r_1$ und $l_2 \rightarrow r_2$.
- Sei R ein Regelsystem und sei $\langle s, t \rangle$ ein kritisches Paar. $\langle s, t \rangle$ heißt zusammenführbar, falls $s \downarrow_R t$.

Lokale Konfluenz von Rewrite- Systemen

Satz

Ein Regelsystem R ist lokal konfluent, wenn alle kritischen Paare zusammenführbar sind.

Zusammenfassung

Matching-Algorithmus

Anwendungsbeispiel: Programme als Terme

Termination und Konfluenz

- Reduktionsordnung
- Kritische Paare