

First-Longest-Match Analyse

Niklas Rieken

June 21, 2017

Wir schauen uns hier einmal eine praktische Anwendung von regulären Ausdrücken und endlichen Automaten an. Bisher haben wir stets das *einfache Matching-Problem* für reguläre Ausdrücke betrachtet. Dabei ging es darum zu prüfen, ob ein gegebenes Wort $w \in \Sigma^*$ zur Sprache eines regulären Ausdrucks $r \in \text{RE}_\Sigma$ gehört. Dies ließ sich mit Hilfe der Thompson-Konstruktion einfach als das Wortproblem eines ε -NFA betrachten. In der Praxis ist das einfache Matching-Problem aber zu primitiv um echte Probleme lösen zu können. Vorallem im Compilerbau benötigen wir etwas mehr, wenn wir Programmcode in seine Bestandteile zerlegen wollen um später Syntax-Checks darauf durchführen zu können und schließlich eine Semantik für das Programm festzulegen. In der Fachsprache nennt man diesen Teil eines Compilers auch *Scanner* oder *Lexer* (siehe auch `lex`, `flex`).

Problemstellung: Extended Matching Problem

Das *erweiterte (extended) Matching Problem* kommt direkt aus der Anwendung des Compilerbaus. Gegeben ist ein Wort $w \in \Sigma^*$ und eine Reihe von regulären Ausdrücken $r_1, \dots, r_n \in \text{RE}_\Sigma$. (*Bemerkung:* In der Praxis ist die Annahme $\varepsilon \notin L(r_i) \neq \emptyset$ für alle i sinnvoll). Gesucht ist nun eine Zerlegung von dem Wortes $w = u_1 \dots u_k$, wobei für jedes u_j ein r_{i_j} existieren muss, sodass $u_j \in L(r_{i_j})$. Man nennt die Zerlegung in (u_1, \dots, u_k) auch *Dekomposition* und die zugehörigen Indizes der regulären Ausdrücke (i_1, \dots, i_k) *Analyse* von w bezüglich r_1, \dots, r_n .

Wir sehen schnell, dass weder Dekomposition, noch Analyse eindeutig sein müssen (insbesondere dann, wenn wir $\varepsilon \in L(r_i)$ zulassen).

Beispiel 1. (i) $r_1 = a^+, w = aa$. Ergibt Dekompositionen (aa) und (a, a) mit zugehöriger (eindeutiger) Analyse (1) bzw. $(1, 1)$.

(ii) $r_1 = a + b, r_2 = a + c, w = a$. Ergibt eindeutige Dekomposition (a) mit zwei verschiedenen Analysen (1) und (2) .

Im zweiten Fall stellen wir uns vor, dass r_1 mögliche Schlüsselwörter einer Programmiersprache (`while`, `true`, `if`) beschreibt und r_2 mögliche Variablennamen (Identifier), dann sieht man warum eine eindeutige Analyse wichtig ist. In diesem Anwendungsfall ist es natürlich zu sagen, dass der

Ausdruck r_1 "wichtiger" ist als r_2 (deshalb verbieten die meisten Programmiersprachen auch Schlüsselwörter als Identifier). Ähnliche Beispiele lassen sich auch für die Dekomposition finden.

Wir wollen nun sowohl, die Dekomposition, als auch die Analyse eindeutig machen. Ein erster Ansatz wäre zunächst das leere Wort zu verbieten und sicherzustellen, dass $L(r_i) \cap L(r_j) = \emptyset$ für alle $i \neq j$, indem wir den gemeinsamen Schnitt von zwei Ausdrücken aus dem "unwichtigeren" Ausdruck entfernen. Dadurch würde zumindest die Analyse eindeutig werden. Diese Idee ist jedoch nicht sinnvoll, u.a. deshalb, da das Entfernen des Schnittes – also das Erkennen von $L(r'_j) := L(r_j) \setminus L(r_i)$ – eine Produktkonstruktion erfordert und somit tendenziell teuer ist.

Eindeutigkeit durch First-Longest-Match

Der am meisten verbreitete Weg ist es folgende zwei Prinzipien zu implementieren:

Longest Match Mache jedes u_i der Zerlegung so lang wie möglich.

First Match Wähle aus den matchenden regulären Ausdrücken den mit dem kleinsten Index.

Definition 2. Eine Dekomposition (u_1, \dots, u_k) von $w \in \Sigma^*$ bezüglich $r_1, \dots, r_n \in \text{RE}_\Sigma$ heißt *Longest-Match-Dekomposition* (LMD), wenn für jedes $i \in \{1, \dots, k\}$, $x \in \Sigma^+$, $y \in \Sigma^*$ mit $w = u_1 \dots u_i x y$ gilt, dass kein $j \in \{1, \dots, n\}$ existiert, sodass $u_i x \in L(r_j)$.

Umgangssprachlich bedeutet das, dass wir an einen Teil der Komposition u_i kein nicht-leeres Wort mehr anhängen können, was noch nicht verarbeitet wurde. Es ist klar, dass eine LMD eindeutig ist, falls sie existiert. Sie muss jedoch nicht immer existieren:

Beispiel 3. $r_1 = a^+$, $r_2 = ab$, $w = aab$ hat Dekomposition (a, ab) aber keine LMD.

Definition 4. Sei u_1, \dots, u_k eine LMD von $w \in \Sigma^*$ bezüglich $r_1, \dots, r_n \in \text{RE}_\Sigma$. Die zugehörige *First-Longest-Match-Analyse* (FLM-Analyse) (i_1, \dots, i_k) ist gegeben durch

$$i_j := \min\{\ell \mid u_j \in L(r_\ell)\}$$

für jedes $j \in \{1, \dots, k\}$.

Auch hier ist klar, dass es höchstens eine FLM-Analyse gibt und sie existiert, wenn die LMD existiert.

Eine mögliche Implementierung

Es gibt viele Möglichkeiten eine FLM-Analyse durchzuführen. Die von mir hier vorgestellte Art ist nicht die von mir favorisierte, aber sie ist insofern intuitiv, als dass sie nur Konzepte aus der FoSAP-Vorlesung verwendet und ein wenig über Arrays.

Als Eingabe erhalten wir also $w \in \Sigma^*$ und die regulären Ausdrücke $r_1, \dots, r_n \in \text{RE}_\Sigma$. Gesucht ist eine FLM-Analyse inklusive zugehöriger LMD oder ein Error, falls diese nicht existieren.

Algorithmus 1 : Pseudocode

```

1  $\text{FLM}(w, (r_i)_{i=1}^n)$ 
   Output : FLM-Analyse  $(i_j)_{j=1}^k$  und LMD  $(u_j)_{j=1}^k$ 
   // Vorbereitung
2 Wandle alle  $r_i$  mit Thompson-Konstruktion in  $\varepsilon$ -NFAs
    $\mathcal{A}_i = (Q_i, \Sigma, \delta_i, q_0^i, F_i)$  um.
3 Baue neuen  $\varepsilon$ -NFA  $\mathcal{A} = (\bigcup_i Q_i \cup \{q_0, q_f\}, \Sigma \uplus \{1, \dots, n\}, \delta, q_0, \{q_f\})$ 
   mit  $\delta(p, a) = \delta_i(p, a)$  falls  $p \in Q_i$  und  $\delta(q_0, \varepsilon) = \{q_0^i \mid i = 1, \dots, n\}$  und
    $\delta(q, i) = \{q_f\}$ , falls  $q \in F_i$ .
4 Eliminiere  $\varepsilon$ -Transitionen.
5 Determinisiere mit Potenzmengenkonstruktion.
6 Minimiere mit Markierungsalgorithmus.
   /* Wir haben nun einen Automaten mit folgender
      Eigenschaft: Vom Zustand  $\hat{\delta}(q_0, u)$  ist genau dann eine
      Transition mit  $i = 1, \dots, n$  möglich, wenn  $u \in L(r_i)$ . */
   // Löse nun das erweiterte Matching-Problem
7  $\ell = 1, w_\ell = w$ 
8 while  $w_\ell \neq \varepsilon$  do
9    $A =$  leeres Array der Länge  $|w_\ell|$ 
10  Simuliere  $\mathcal{A}$  auf  $w_\ell$ , prüfe in jedem Schritt  $j$ , ob eine  $i$ -Transition
    in einen akzeptierenden Zustand möglich ist. Falls ja, setze  $A[j]$ 
    auf das kleinste mögliche  $i$ .
11  Nach der Simulation laufe  $A$  von hinten nach vorne durch bis wir
    den ersten nicht-leeren Eintrag an Stelle  $h$  finden. Sollte es keinen
    geben gebe einen Error aus, ansonsten ist  $u_\ell = w_{\ell_1} \dots w_{\ell_h}$  und
     $i_\ell = A[h]$ .
12  Setze  $w_{\ell+1} = w_{\ell_h} \dots w_{|w_\ell|}, \ell = \ell + 1$ .
   // Jetzt ist  $w_\ell = \varepsilon$ 
13 Gebe  $(u_1, \dots, u_{\ell-1})$  und  $(i_1, \dots, i_{\ell-1})$  aus.
```
