Grundzüge der Theoretischen Informatik

12. November 2021

Markus Bläser

Universität des Saarlandes

Kapitel 5: Das Myhill-Nerode-Theorem

Isomorphismen von Automaten

- Seien $M = (Q, \Sigma, \delta, q_0, Q_{\mathrm{acc}})$ und $M' = (Q', \Sigma, \delta', q'_0, Q'_{\mathrm{acc}})$ DEAs.
- δ und δ' seien total.

Definition

M und M' sind *isomorph* falls es eine Bijektion $b : Q \to Q'$ gibt mit:

1. Für alle $q \in Q$ und $\sigma \in \Sigma$ gilt $b(\delta(q, \sigma)) = \delta'(b(q), \sigma)$.
2. $b(q_0) = q'_0$.
3. $b(Q_{\mathrm{acc}}) = Q'_{\mathrm{acc}}$.

Solch ein b heißt *Isomorphismus*.

$$
\begin{tikzcd}
Q \arrow[d, "b"'] & \times & \Sigma \arrow[d, "\mathrm{id}"] \arrow[r, "\delta"] & Q \arrow[d, "b"] \\
Q' & \times & \Sigma \arrow[r, "\delta'"] & Q' \arrow[u, bend right, "b^{-1}"']
\end{tikzcd}
$$

Der Minimalautomat

Theorem (5.9)

Sei $L \subseteq \Sigma^*$ *regulär.*

1. *Jeder DEA* $M' = (Q', \Sigma, \delta', q_0', Q_{acc}')$ *mit* δ' *total und* $L(M') = L$ *hat mindestens* $\mathrm{index}(\sim_L)$ *Zustände.*
2. *Jeder DEA mit totaler Übergangsfunktion, der* L *erkennt und* $\mathrm{index}(\sim_L)$ *Zustände hat, ist isomorph zum Myhill-Nerode-Automaten* $M = (Q, \Sigma, \delta, q_0, Q_{acc})$ *(aus dem Beweis von "3. $\Longrightarrow$ 1." im Myhill-Nerode-Theorem).*

Es kann nur einen Minimalautomat geben.

Bis auf Isomorphie.

$$x \not\sim_L y \iff \exists z \in \Sigma^*: \; xz \notin L \;\wedge\; yz \in L$$

oder umgekehrt

Wenn es so ein z gilt, dann gilt es auch ein "kurzes" z.

$\equiv_M \longrightarrow \sim_L$

Hopcrofts Algorithmus (1)

Ziel: Gegeben $M = (Q, \Sigma, \delta, q_0, Q_{acc})$, finde den Minimalautomat!

- Algorithmus speichert eine Partition von Q, d.h. wir schreiben $Q = P_1 \cup \cdots \cup P_t$, so dass $P_1, \ldots, P_t$ sind paarweise disjunkt sind.
- $[P_i] := \bigcup_{q \in P_i} \overbrace{\{x \mid \delta^*(q_0, x) = q\}}^{\text{Äqklasse von } q \text{ wi } \equiv_M}$.
- Schreiben $[q]$ statt $[\{q\}]$ für ein einzelnes $q \in Q$.

Invariante

Für jede Äquivalenzklasse C von $\sim_{L(M)}$, gibt es ein i mit $C \subseteq [P_i]$.

- Nutzen: Für $p \in P_i$ und $q \in P_j$, $i \neq j$, sind $[p]$ und $[q]$ in verschiedenen Myhill-Nerode-Äquivalenzklassen.

Hopcrofts Algorithmus (2)

Lemma (5.12)

Sei $P_1, \ldots, P_t$ *eine Partition wie zuvor.*
Seien $1 \leq i, j \leq t$, $\sigma \in \Sigma$, *und* $q, p \in P_i$.

1. *Falls* $\delta(p, \sigma) \in P_j$, *aber* $\delta(q, \sigma) \notin P_j$, *dann sind* $[p]$ *und* $[q]$ *Teilmengen verschiedener Myhill-Nerode-Klasse.*
2. *Falls* $[p]$ *und* $[q]$ *Teilmengen von verschiedenen Myhill-Nerode-Klassen sind, dann gibt es Indizes* i' *und* j', $\sigma' \in \Sigma$ *und* $p', q' \in P_{i'}$, *so dass* $\delta(p', \sigma) \in P_{j'}$ *und* $\delta(q', \sigma) \notin P_{j'}$.

2 $\hat{=}$ der Beweis z Lemma vor die Lösung 1
haben, allerdings werden dann evtl.
andere Klassen getrennt

Beweis 5.12.

1) Sei $x \in [p]$ und $y \in [q]$. Dann gilt $x\sigma \in [P_j]$ aber $y\sigma \notin [P_j]$.

Da $P_1, \dots, P_t$ gröber ist als $\sim_L$ gilt

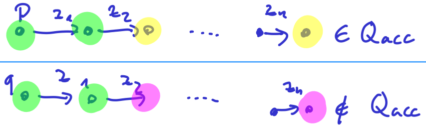
damit $x\sigma \not\sim_L y\sigma$

Daraus folgt, dass $x \not\sim_L y$ wegen Rechtsinvarianz.

2) Da $[p]$, $[q]$ zu unterschiedlichen Myhill-Nerode-Klassen gehören, gibt es ein $z \in \Sigma^*$:

$$\delta(p, z) \in Q_{acc} \quad \text{und} \quad \delta(q, z) \notin Q_{acc}$$

(oder umgekehrt)



Sei z' der längste Präfix von z, so dass

$$\underbrace{\delta^*(p, z')}_{=: p'} \text{ und } \underbrace{\delta^*(q, z')}_{q'} \text{ in gleichen } P_j$$

sind

z' ist ein echter Präfix von z ($z' \neq z$), denn $\delta^*(p, z)$ und $\delta^*(q, z)$ liegen nicht in gleichen P_j

Sei σ das nächste Zeichen nach z' in z

Dann gilt, dass $\delta^*(p', \sigma)$ und $\delta^*(q', \sigma)$ nicht in gleichen P_j liegen. $\square$

Hopcrofts Algorithmus (3)

$P_i \cap X = \{ q \in P_i : \delta(q,\sigma) \in P_j \}$

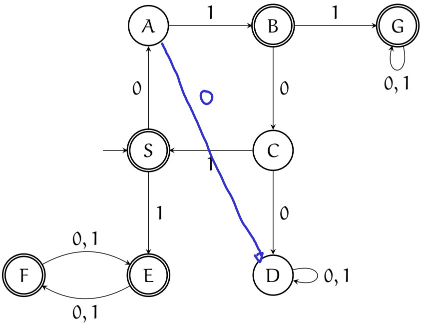
Minimierungsalgorithmus

$P_i \setminus X = \{ q \in P_i : \delta(q,\sigma) \notin P_j \}$

1. Starte mit $P_1 = Q_{\mathrm{acc}}$ und $P_2 = Q \setminus Q_{\mathrm{acc}}$.
   ($x \in [P_1]$ und $y \in [P_2]$ sind nicht Myhill-Nerode-äquivalent.)
2. In jeden Schritt, wähle ein P_j und ein $\sigma \in \Sigma$.
3. Sei $X = \{q \in Q \mid \delta(q, \sigma) \in P_j\}$.
4. Ersetze jedes P_i durch $P_i \cap X$ und $P_i \setminus X$.
5. Solange in Schritt 4 neue Klassen entstehen, gehe zu 2.

- Lemma 5.12(1): Die Invariante bleibt in Schritt 4 erhalten
- Lemma 5.12(2): Wenn kein j und σ in Schritt 2 gefunden wird, so dass sich etwas ändert, dann sind $P_1, \ldots, P_t$ die Myhill-Nerode-Klassen.

Beispiel



$P_1 = \{A, C, D\}$

$P_2 = \{B, G, S, F, E\}$

0

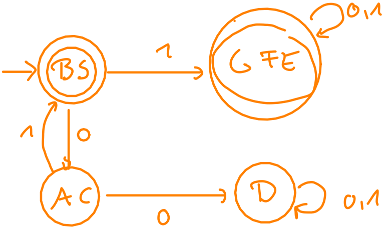
P_2

$\{B, S\}, \quad \{G, F, E\}$

1

P_1

$\{A, C\} \quad \{D\}$



0 : keine neuen Klassen

1 : keine neue Klassen

Einfaches mathematisches Berechnungsmodell

Wir können beweisen, dass

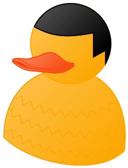
- eine Sprache regulär ist
  (von diesem Modell berechnet werden kann)
- eine Sprache nicht regulär ist
  (von diesem Modell nicht berechnet werden kann)

And now for something completely different. . .

Berechenbarkeitstheorie!

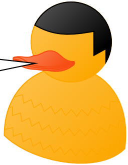
Kapitel 6: Grenzen der Berechenbarkeit

Grenzen der Berechenbarkeit



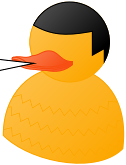
Grenzen der Berechenbarkeit

Warum überprüft ein Compiler nicht vorab, ob mein Programm terminiert?



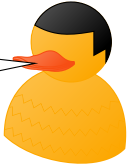
Grenzen der Berechenbarkeit

Warum überprüft ein Compiler nicht vorab, ob mein Programm terminiert? Warum testet er nicht, ob mein Programm korrekt ist?



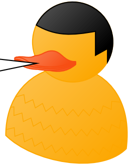
Grenzen der Berechenbarkeit

Warum überprüft ein Compiler nicht vorab, ob mein Programm terminiert? Warum testet er nicht, ob mein Programm korrekt ist? Warum habe ich so eine Frisur?



Grenzen der Berechenbarkeit

Warum überprüft ein Compiler nicht vorab, ob mein Programm terminiert? Warum testet er nicht, ob mein Programm korrekt ist? Warum habe ich so eine Frisur?



Gibt folgendes Programm immer 0 aus?

Input: $x_0, \dots, x_3$

if $x_0 \le 2$ **then**
    return 0
fi
if $x_1 = 0$ or $x_2 = 0$ or $x_3 = 0$ **then**
    return 0
fi
if $x_1^{x_0} + x_2^{x_0} = x_3^{x_0}$ **then**
    return 1
else
    return 0
fi

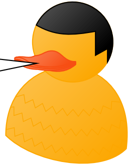
( )

if fi

Fermatsche Vermutung / Theorem

Grenzen der Berechenbarkeit

Warum überprüft ein Compiler nicht vorab, ob mein Programm terminiert? Warum testet er nicht, ob mein Programm korrekt ist? Warum habe ich so eine Frisur?



Gibt folgendes Programm immer 0 aus?

Input: $x_0, \ldots, x_3$

if $x_0 \leq 2$ **then**
    return 0
fi
if $x_1 = 0$ or $x_2 = 0$ or $x_3 = 0$ **then**
    return 0
fi
if $x_1^{x_0} + x_2^{x_0} = x_3^{x_0}$ **then**
    return 1 $\longleftarrow$ Endlosschleife $\Rightarrow$ Terminierung?
else
    return 0
fi

Kapitel 7: WHILE- und FOR-Programme

Die WHILE-Progammiersprache

Variablen: x_0, x_1, x_2, ... unendliches Alphabet

Konstanten: 0, 1, 2, ...

Schlüsselwörter: **while**, **do**, **od**

Sonstige Zeichen: :=, $\neq$, ; , +, −, [,]

Definition (7.1)

WHILE-Programme sind induktiv definiert wie folgt:

1. Einfache Anweisungen haben die Form

$$x_i := x_j + x_k \quad \textit{oder} \quad x_i := x_j - x_k \quad \textit{oder} \quad x_i := c,$$

   wobei $i, j, k \in \mathbb{N}$ und $c \in \mathbb{N}$.

2. Ein WHILE-Program P ist entweder eine einfache Anweisung oder lässt sich schreiben als:

   2.1 **while** $x_i \neq 0$ **do** P_1 **od** or
   2.2 $[P_1; P_2]$

   für ein $i \in \mathbb{N}$ und WHILE-Programme P_1 und P_2.

Die WHILE-Progammiersprache (2)

$\mathcal{W}_0 \subseteq \mathcal{W}_1 \subseteq \mathcal{W}_2 \subseteq \ldots$

$$\mathcal{W}_0 = \text{Menge aller einfachen Anweisungen}$$

$$\begin{aligned}\mathcal{W}_n = \mathcal{W}_{n-1} \cup \{P \mid\ & \exists P_1 \in \mathcal{W}_{n-1},\ x_i \in X, \\ & \text{so dass } P = \textbf{while } x_i \neq 0 \textbf{ do } P_1 \textbf{ od} \text{ oder} \\ & \exists P_1 \in W_j,\ P_2 \in W_k \text{ mi } j + k \leq n - 1, \\ & \text{so dass } P = [P_1; P_2]\}\end{aligned}$$

Bemerkung

Ein WHILE-Programm P ist in $\mathcal{W}_n$, genau dann wenn es aus einfachen Anwendungen durch höchstes n Anwendungen einer Regel aus 2. in Definition 7.1 gebaut werden kann.

Die FOR-Programmiersprache

Definition (7.1)

FOR-Programme sind induktiv definiert wie folgt:

1. Einfache Anweisungen haben die Form

$$x_i := x_j + x_k \quad \textit{oder} \quad x_i := x_j - x_k \quad \textit{oder} \quad x_i := c,$$

   wobei $i, j, k \in \mathbb{N}$ und $c \in \mathbb{N}$.

2. Ein FOR-Program P ist entweder eine einfache Anweisung oder lässt sich schreiben als:

   2.1 **for** x_i **do** P_1 **od**   or

   2.2 $[P_1;\ P_2]$

   für ein $i \in \mathbb{N}$ und FOR-Programme P_1 und P_2.

$x_0 := 1;$ while $x_i \neq 0$ do $x_0 := 1$ od

Semantik

- Eingabe eines Programms P: $\alpha_0, \ldots, \alpha_{s-1} \in \mathbb{N}$.
- Gespeichert in $x_0, \ldots, x_{s-1}$.
- Ausgabe: Inhalt von x_0 nach Ausführung von P.

- Die Menge $X = \{x_0, x_1, x_2, \ldots\}$ aller Variablen ist unendlich.
- Aber jedes Programm P kann nur endlich viele Variablen benutzen.
- $\ell = \ell(P)$ größte Index einer Variablen in P.
- Annahme: $\ell \geq s - 1$.

$P_1 \quad ; \quad P_2$

$\ell(P_1) = 7 \qquad \ell(P_2) = 42$

Zustand eines Programms P: Element aus $\mathbb{N}^{\ell(P)+1}$.
Besser: Unendliche Folge $\mathbb{N} \to \mathbb{N}$ mit endlichem Träger