Termersetzungssysteme und Reduktion

Definition 2.24. Ein Term t heißt

- schwach normalisierend, wenn t eine Normalform hat, d.h. wenn s existiert mit $t \to^* s$ und s normal;
- stark normalisierend, wenn keine unendliche Folge $(t_i)_{i\in\mathbb{N}}$ mit $t = t_0 \to t_1 \to t_2 \to \dots$ existiert.

Ein Termersetzungssystem $(\Sigma, \to_0)$ heißt schwach/stark normalisierend, wenn jeder Term schwach/stark normalisierend in $\to$ ist.

Satz 2.28. Sei $>$ Reduktionsordnung und für alle Terme t, s gelte: Aus $t \to_0 s$ folgt: $t > s$.
Dann ist $\to$ stark normalisierend.

Satz 2.40. Polynomordnungen sind Reduktionsordnungen.

Korollar 2.41. Sei $\to_0$ ein Termersetzungssystem und $>_{\mathcal{A}}$ eine Polynomordnung. Falls

$$t \to_0 s \Rightarrow t \succ_{\mathcal{A}} s \text{ für alle Terme } t, s,$$

dann ist $\to$ stark nomalisierend.

Satz 2.46. Sei T ein konfluentes Termersetzungssystem.

1. Für Terme t, s gilt

$$s \leftrightarrow^* t \Leftrightarrow s \text{ und } t \text{ sind zusammenführbar}$$

2. Normalformen sind eindeutig, d.h. wenn Terme s, s' Normalformen einer Termes t sind, dann gilt $s = s'$ (d.h. s und s' sind syntaktisch gleich).

Satz 2.48 (Newman's Lemma)**.** Ein stark normalisierendes und lokal konfluentes Termersetzungssystem ist konfluent.

Satz 2.57 (Critical Pair Lemma)**.** Ein Termersetzungssystem T ist genau dann lokal konfluent, wenn in T alle kritischen Paare zusammenführbar sind.

Definition 3.6. Zwei Terme t_1, t_2 heißen α-äquivalent, wenn sie durch Umbenennung gebundener Variablen auseinander hervorgehen:

$$\lambda x.t =_\alpha \lambda x.t[y/x] \qquad \text{wenn } y \notin FV(t).$$

Ausführungsvorschrift β-Reduktion, die das Ausrechnen einer Funktionsanwendung modelliert:

$$(\lambda x.t)x \to_\beta t$$

erlaubt insbesondere:

$$(\lambda x.s)t \to_\beta s[t/x]$$

wobei $(\lambda x.s)t$ als β-Redex bezeichnet wird.

Zusätzliche Grundreduktionen wie η-Reduktion: $\lambda x.yx \to_\eta y$.

δ-Reduktion: Einsetzen einer Definition eines benannten λ-Ausdrucks ($fun = \lambda x.t$)

$$\lambda y.fun \to_\delta \delta y.(\delta x.t)$$

Definition 3.11. Die applikative (auch leftmost-innermost) Reduktion $\to_a$ ist induktiv definiert durch:

- $(\lambda x.t)s \to_a t[s/x]$, wenn t und s normal sind.
- $\lambda x.t \to_a \lambda x.t'$, wenn $t \to_a t'$.

- $ts \to_a t's$, wenn $t \to_a t'$.
- $ts \to_a ts'$, wenn $s \to_a s'$ und t normal ist.

Definition 3.12. Die normale (auch leftmost-outermost) Reduktion $\to_n$ ist definiert durch:

- $(\lambda x.t)s \to_n t[s/x]$.
- $\lambda x.t \to_n \lambda x.t'$, wenn $t \to_n t'$.
- $ts \to_n t's$, wenn $t \to_n t'$ und t keine λ-Abstraktion ist.
- $ts \to_n ts'$, wenn $s \to_n s'$ und t normal ist und keine λ-Abstraktion ist.

Hinweis: Die Reduktionsstrategie bezieht sich ausdrücklich auch auf δ-Reduktion. D.h. auch δ-Reduktionen sind bei normaler Reduktion zuerst auf linke Seiten von Funktionsanwendungen (also die Funktionen) anzuwenden, von außen nach innen, und erst dann auf Argumente von Funktionsanwendungen, bei applikativer Reduktion dagegen zuerst auf Argumente von Funktionsanwendungen.

Beispiel

$$\begin{array}{lll}
\underline{pow2\ three} & \to_n & \underline{three}\ (\ mult\ two\ )\ one \\
& \to_n & \underline{\lambda f\ a.\ f\ (\ f\ (\ f\ a\ )\ )\ (\ mult\ two\ )}\ one \\
& \to_n & \underline{\lambda a.\ (\ mult\ two\ )\ (\ (\ mult\ two\ )\ (\ (\ mult\ two\ )\ a\ )\ )}\ one \\
& \to_n & (\ mult\ two\ )\ (\ (\ mult\ two\ )\ (\ (\ mult\ two\ )\ one\ )\ ) \\
 & & \\
pow2\ \underline{three} & \to_a & \underline{pow2\ (\lambda f\ a.\ f\ (\ f\ (\ f\ a\ )\ )\ )} \\
& \to_a & (\lambda f\ a.\ f\ (\ f\ (\ f\ a\ )\ )\ )\ (\ mult\ \underline{two}\ )\ one \\
& \to_a & (\lambda f\ a.\ f\ (\ f\ (\ f\ a\ )\ )\ )\ (\ mult\ \lambda f\ a.\ f\ (\ f\ \ a\ )\ )\ \underline{one} \\
& \to_a & (\lambda f\ a.\ f\ (\ f\ (\ f\ a\ )\ )\ )\ (\ mult\ \lambda f\ a.\ f\ (\ f\ \ a\ )\ )\ \lambda f\ a.\ f\ (\ a\ )
\end{array}$$

Funktionen

$$\begin{array}{rclrclrcl}
flip & = & \lambda f\ x\ y.\ f\ y\ x & pair\ a\ b & = & \lambda s.\ s\ a\ b & swap\ p & = & p\ (\ \lambda\ x\ y.\ pair\ y\ x\ ) \\
const & = & \lambda x\ y.\ x & fst\ p & = & p\ \lambda\ x\ y.\ x & true & = & \lambda\ x\ y.\ x \\
twice & = & \lambda f\ x.\ f\ (\ f\ x\ ) & snd\ p & = & p\ \lambda\ x\ y.\ y & false & = & \lambda\ x\ y.\ y \\
zero & = & \lambda\ f\ a.\ a & add & = & n\ succ\ m & if_then_else & = & \lambda\ b\ x\ y.\ b\ x\ y \\
succ\ n & = & \lambda\ f\ a.\ f\ (\ n\ f\ a\ ) & mult & = & n\ (\ add\ m\ )\ zero & isZero\ n & = & n\ \lambda\ x.\ false\ true \\
pred\ n & = & fst\ (\ n\ (\ \lambda p.\ pair\ (\ snd\ p\ )\ (\ succ\ (\ snd\ p\ )))\ (\ pair\ zero\ zero\ )) & & & & sub\ n\ m & = & m\ pred\ n
\end{array}$$

System F

Bemerkung 3.23. Funktionen mit mehreren Argumenten stellen wir wieder mittels Currying dar.
Der Typ

$$\alpha_1 \to (\alpha_2 \to (\cdots \to (\alpha_n \to \beta)\ldots)),$$

den wir per Rechtsassoziativität kurz als $\alpha_1 \to \alpha_2 \to \ldots \alpha_n \to \beta$ schreiben, kann als ein Typ von n-stelligen Funktionen mit Argumenten der Typen $\alpha_1, \ldots, \alpha_n$ und Resultattyp β angesehen werden.

Definition 5.1 (System-F nach Curry)**.** Für „im Kontext Γ hat der Term t den Typ α“ schreibt man

$$\Gamma \vdash t : \alpha$$

Diese Relation ist induktiv definiert durch die folgenden Regeln:

$$(Axiom)\ \frac{}{\Gamma \vdash x : \alpha}\ x : \alpha \in \Gamma$$

$$(\to e)\ \frac{\Gamma \vdash t : \alpha \to \beta \quad \Gamma \vdash s : \alpha}{\Gamma \vdash ts : \beta}$$

$$(\to_i)\ \frac{\Gamma, x:\alpha \vdash t:\beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}$$

Die Typisierung $\Gamma \vdash t : \alpha$ vom Termen wird induktiv definiert durch die Typregeln von $\lambda\to$ sowie

$$(\forall_i)\ \frac{\Gamma \vdash s:\alpha \quad a \notin FV(\Gamma)}{\Gamma \vdash s : \forall a.\alpha}$$

$$(\forall e)\ \frac{\Gamma \vdash s : \forall a.\alpha}{\Gamma \vdash s : (\alpha[a := \beta])}$$

Beispiel $\Gamma := \{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\}$

$$(\to_i)\frac{(\to e)\dfrac{(\forall e)\dfrac{(Ax)\overline{\Gamma_2 \vdash n : \forall a.(a \to a) \to a \to a}}{\Gamma_2 \vdash n : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}} \quad \dfrac{\overline{\Gamma_2 \vdash mult : \mathbb{N} \to\to \mathbb{N}} \quad \overline{\Gamma_2 \vdash two : \mathbb{N}}}{\Gamma_2 \vdash mult\ two : \mathbb{N} \to \mathbb{N}}}{\Gamma_2 \vdash n\ (\ mult\ two\ ) : \mathbb{N} \to \mathbb{N}} \quad \overline{\Gamma_2 \vdash one : \mathbb{N}}}{\Gamma_2 := \{\Gamma, n : \mathbb{N}\} \vdash n\ (\ mult\ two\ )\ one : \mathbb{N}}}{\{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n.\ n\ (\ mult\ two\ )\ one : \mathbb{N} \to \mathbb{N}}$$

Algorithmus 3.29 (Algorithmus W nach Hindley/Milner)**.** Wir definieren rekursiv eine Menge $PT(\Gamma; t; \alpha)$ von Typgleichungen (Notation $\alpha \doteq \beta$), so dass der *most general unifier* $\sigma = \mathrm{mgu}(PT(\Gamma; t; \alpha))$ die allgemeinste Lösung von $\Gamma \vdash t : \alpha$ ist, wenn denn Lösungen existieren.

1. $PT(\Gamma; x; \alpha) = \{\alpha \doteq \beta \mid x : \beta \in \Gamma\}$
2. $PT(\Gamma; \lambda x.t; \alpha) = PT((\Gamma, x : a); t; b) \cup \{a \to b \doteq \alpha\}$, mit a, b frisch.
3. $PT(\Gamma; ts; \alpha) = PT(\Gamma; t; a \to \alpha) \cup PT(\Gamma; s; a)$, mit a frisch.

Beispiel Wir berechnen den Prinzipaltyp von $(\emptyset, \lambda xy.\ xy)$

$$\begin{aligned} PT(\emptyset; \lambda xy.\ xy; a) &= PT(x : b; \lambda y.\ xy; c) \cup \{a \doteq b \to c\} \\ &= PT(x : b, y : d; xy; e) \cup \{a \doteq b \to c; c \doteq d \to e\} \\ &= PT(x : b, y : d; x; f \to e) \cup PT(x : b, y : d; y; f) \cup \{a \doteq b \to c; c \doteq d \to e\} \\ &= \{b \doteq f \to e; d \doteq f; a \doteq b \to c; c \doteq d \to e\} =: \mathcal{E} \end{aligned}$$

Wir erhalten

$$mgu(\mathcal{E}) = [d/f, d \to e/b, (d \to e)/c, (d \to e) \to d \to e/a].$$

Also hat $\lambda xy.\ xy$ den Prinzipaltyp $mgu(\mathcal{E})(a) = (d \to e) \to d \to e$.
Beispiel Wir berechnen den Prinzipaltyp von $(x : a, x\lambda z.\ z)$

$$\begin{aligned} PT(x : a; x\lambda z.\ z; b) &= PT(x : a; x; c \to b) \cup PT(x : a; \lambda z.\ z) \\ &= \{a \doteq c \to b\} \cup PT(x : a, z : d; z; e) \cup \{c \doteq d \to e\} \qquad = \{a \doteq c \to b, d \doteq e, c \doteq d \to e\} \end{aligned}$$

Wir erhalten

$$mgu(\mathcal{E}) = [d/e, d \to d/c, (d \to d) \to b/a].$$

Also hat $(x : a, x\lambda z.\ z)$ den Prinzipaltyp $[(d \to d) \to b/a, b/b]$; Einsetzen in $x : a \vdash x\lambda z.\ z : b$ ergibt:

$$x : (d \to d) \to b \vdash x\lambda z.\ z : b.$$

Strukturelle Induktion

Wie der Beweis der Initialität der Termalgebra schon andeutet, ist Initialität einfach die abstrakte Verkapselung eines Rekursionsprinzips. Dieses Rekursionsprinzip bezeichnet man in der funktionalen Programmierung als *fold*.

Beispiel Allgemeine Listen

data List a **where**
$Nil : () \to$ **List** a
$Cons : a \to$ **List** $a \to$ **List** a

Dazugehörige *fold*-Funktion:

$$fold\ a\ c\ Nil = a$$
$$fold\ a\ c\ (\ Cons\ x\ xs\ ) = c\ x\ (\ fold\ a\ c\ xs\ )$$

Als Beweisprinzip für rekursive Funktionen bietet sich typischerweise Induktion an; dabei sollte das verwendete Induktionsprinzip dieselbe Struktur haben wie die rekursive Definition.

- Ein Induktionsanfang je Basisfall bzw. nicht-rekursivem Konstruktor.
- Ein Schritt je rekursivem Konstruktor → dafür setze in IV voraus, dass für die Argumente des Konstruktors die Aussage bereits bewiesen ist.

Funktionen

$$\begin{aligned} length\ Nil &= 0 \\ length\ (\ Cons\ x\ xs\ ) &= 1 + (\ length\ (\ xs\ ) \end{aligned}$$

$$\begin{aligned} snoc\ Nil\ x &= Cons\ x\ Nil \\ snoc\ (\ Cons\ y\ ys\ )\ x &= Cons\ y\ (\ snoc\ ys\ x\ ) \end{aligned}$$

$$\begin{aligned} reverse\ Nil &= Nil \\ reverse\ (\ Cons\ x\ xs\ ) &= snoc\ (\ reverse\ xs\ )\ x \end{aligned}$$

$$\begin{aligned} drop\ a\ Nil &= Nil \\ drop\ a\ (\ Cons\ x\ xs\ ) &= if\ a == x\ then\ drop\ a\ xs \\ &\quad else\ Cons\ x\ (\ drop\ a\ xs\ ) \end{aligned}$$

$$\begin{aligned} elem\ a\ Nil &= false \\ elem\ a\ (\ Cons\ x\ xs\ ) &= if\ a == x\ then\ true \\ &\quad else\ elem\ a\ xs \end{aligned}$$

$$\begin{aligned} min\ Nil &= 0 \\ min\ (\ Cons\ x\ Nil\ ) &= x \\ min\ Cons\ x\ xs &= if\ x < min\ xs\ then\ x \\ &\quad else\ min\ xs \end{aligned}$$

$$\begin{aligned} map\ f\ Nil &= Nil \\ map\ f\ (\ Cons\ x\ xs\ ) &= Cons\ (\ f\ x\ )\ (\ map\ f\ xs\ ) \end{aligned}$$

$$\begin{aligned} flatten\ Nil &= Nil \\ flatten\ (\ Cons\ x\ xs\ ) &= concat\ (\ flatten\ x\ )\ (\ flatten\ xs\ ) \end{aligned}$$

$$\begin{aligned} concat\ Nil\ x &= x \\ concat\ (\ Cons\ x\ xs\ )\ ys &= Cons\ x\ (\ concat\ xs\ ys\ ) \end{aligned}$$

Beispiel Klausur [Probe] SoSe 2017, Aufgabe 3

Beweisen Sie mittels struktureller Induktion, dass

$$\forall e\ xs\ ys.\ xs \oplus (\ Cons\ e\ ys\ ) = (snoc\ xs\ e\ ) \oplus ys$$

IA) Sei $xs = Nil$, dann gilt für e beliebig aber fest:

$$\begin{aligned} Nil \oplus (\ Cons\ e\ ys\ ) &= (\ snoc\ Nil\ e\ ) \oplus ys \\ (\ Cons\ e\ ys\ ) &= (\ Cons\ e\ Nil\ ) \oplus ys \\ (\ Cons\ e\ ys\ ) &= Cons\ e\ (\ Nil \oplus ys\ ) \\ Cons\ e\ ys &= Cons\ e\ ys \end{aligned}$$

IV) Sei Aussage für as bereits bewiesen.

IS) Sei $xs = Cons\ a\ as$

$$\begin{aligned}
(\ Cons\ a\ as\ ) \oplus (\ Cons\ e\ ys\ ) &= (\ snoc\ (\ Cons\ a\ as\ )\ e\ ) \oplus ys \\
Cons\ a\ (\ \underbrace{as \oplus (\ Cons\ e\ ys\ )}_{\text{(IV)}}\ ) &= (\ Cons\ a\ (\ snoc\ as\ e\ )\ ) \oplus ys \\
Cons\ a\ (\ (\ snoc\ as\ e\ ) \oplus ys\ ) &= Cons\ a\ (\ (\ snoc\ as\ e\ ) \oplus ys\ )
\end{aligned}$$

q.e.d

Korekursion und Koinduktion

Definition 4.2. Ein Stream über einem Alphabet A ist eine unendliche Sequenz $(a_0, a_1, a_2, \dots)$ mit $a_i \in A$ für alle $i \in \mathbb{N}$. Man kann Streams in der von Listen gewohnten Weise destruieren – man hat:

$$\begin{aligned}
hd : S &\to A \\
(a_0, a_1, \dots) &\mapsto a_0
\end{aligned}$$

und

$$\begin{aligned}
tl : S &\to S \\
(a_0, a_1, \dots) &\mapsto (a_1, a_2, \dots)
\end{aligned}$$

Dies macht gleichzeitig die Analogie zwischen Destruktoren und Beobachtungen klar: hd liest das aktuelle Element eines Streams, tl geht zum nächsten Element über.

Beispiel 4.37. Wir definieren:

$$blink : 2 \times a \times Stream\ a \to Stream\ a$$

$$blink\ c\ x\ (a_0, a_1, \dots) = \begin{cases} (x, a_0, x, a_1, \dots) & \text{falls } c = 0 \\ (a_0, x, a_1, x, \dots) & \text{falls } c = 1 \end{cases}$$

wobei 2 der Datentyp mit Konstruktoren 0, 1 ist. Wir schreiben kurz $blink\ 0 = b_0$, $blink\ 1 = b_1$ und definieren dann b_0, b_1 per unfold:

$$\begin{aligned}
hd\ (\ b_0\ x\ s\ ) &= x \\
tl\ (\ b_0\ x\ s\ ) &= b_1\ x\ s \\
hd\ (\ b_1\ x\ s\ ) &= hd\ s \\
tl\ (\ b_1\ x\ s\ ) &= b_0\ x\ (\ tl\ s\ )
\end{aligned}$$

Beispiel 4.38. Als Beispiel betrachten wir folgende korekursive Definition zweier Funktionen, die aus einem Stream die Teilstreams an den geraden bzw. ungeraden Positionen herausgreifen:

$$\begin{aligned}
hd\ (\ even\ s\ ) &= x \\
tl\ (\ b_0\ x\ s\ ) &= b_1\ x\ s \\
hd\ (\ b_1\ x\ s\ ) &= hd\ s \\
tl\ (\ b_1\ x\ s\ ) &= b_0\ x\ (\ tl\ s\ )
\end{aligned}$$

Definition 4.39. Eine Relation $R \subseteq A^\omega \times A^\omega$ heißt Bisimulation, wenn für alle $(s, t) \in R$ gilt:

- $hd\ s = hd\ t$
- $(\ tl\ s\ )\ R\ (\ tl\ t\ )$

Satz 4.40. x Wenn R eine Bisimulation ist, dann gilt $R \subseteq id$, d. h.

$$sRt \Rightarrow s = t \qquad \text{für alle } s, t \in A^\omega.$$

Bemerkung 4.41. Man kann obigen Satz als Korrektheit des Bisimulationsprinzips auffassen: Wenn ich zwei Streams durch eine Bisimulation in Beziehung setzen kann, sind sie tatsächlich gleich.

Beispiel Klausur [Probe] SoSe 2017, Aufgabe 3

1. Definieren Sie rekursiv eine Funktion $sampler : Signal \to Signal \to Signal$, sodass

   - $sampler\ (\ square\ 0\ 1\ )\ (\ square\ x\ 0\ ) = flat\ 0$
   - $sampler\ (\ square\ 1\ 0\ )\ (\ flat\ x\ ) = square\ x\ 0$

   Funktion in Codata-Funktionen einsetzen und Verhalten anwenden:

   $currentSample\ (\ sampler\ t\ s\ ) = if\ (\ currentSample\ t > 0\ )\ then\ (\ currentSample\ s\ )\ else\ 0$
   $discardSample\ (\ sampler\ t\ s\ ) = sampler\ (\ discardSample\ t\ )\ (\ discardSample\ s\ )$

2. Beweis der Aussagen per Koinduktion:

   (1): $currentSample\ (\ sampler\ (\ square\ 0\ 1\ )\ (\ square\ x\ 0\ )\ ) =$
   $= if\ currentSample\ (\ square\ 0\ 1\ ) > 0\ then\ currentSample\ (\ square\ x\ 0\ )\ else\ 0$
   $= if\ 0 > 0\ then\ x\ else\ 0$
   $= 0$
   $= currentSample\ (\ flat\ 0\ )$ ✓

   $discardSample\ (\ sampler\ (\ square\ 0\ 1\ )\ (\ square\ x\ 0\ )\ ) =$
   $= sampler\ (\ discardSample\ (\ square\ 0\ 1\ )\ )\ (\ discardSample\ (\ square\ x\ 0\ )\ )$
   $= sampler\ (\ square\ 1\ 0\ )\ (\ square\ 0\ x\ )$
   $\to currentSample\ (\ sampler\ (\ square\ 1\ 0\ )\ (\ square\ 0\ x\ )\ ) =$
   $= if\ currentSample\ (\ square\ 1\ 0\ ) > 0\ then\ currentSample\ (\ square\ 0\ x\ )\ else\ 0$
   $= if\ 1 > 0\ then\ 0\ else\ 0$
   $= 0$
   $= currentSample\ (\ discardSample\ (\ flat\ 0\ )\ )$ ✓

   (2): $currentSample\ (\ sampler\ (\ square\ 1\ 0\ )\ (\ flat\ x\ )\ ) =$
   $= if\ currentSample\ (\ square\ 1\ 0\ ) > 0\ then\ currentSample\ (\ flat\ x\ )\ else\ 0 =$
   $= if\ 1 > 0\ then\ x\ else\ 0$
   $= x$
   $= currentSample\ (\ square\ x\ 0\ )$ ✓

   $discardSample\ (\ sampler\ (\ square\ 1\ 0\ )\ (\ flat\ x\ )\ ) =$
   $= sampler\ (\ discardSample\ (\ square\ 1\ 0\ )\ )\ (\ discardSample\ (\ flat\ x\ )\ )$
   $= sampler\ (\ square\ 0\ 1\ )\ (\ flat\ x\ )$
   $\to currentSample\ (\ sampler\ (\ square\ 0\ 1\ )\ (\ flat\ x\ )\ ) =$
   $= if\ currentSample\ (\ square\ 0\ 1\ ) > 0\ then\ currentSample\ (\ flat\ x\ )\ else\ 0$
   $= if\ 0 > 0\ then\ x\ else\ 0$
   $= 0$
   $= currentSample\ (\ discardSample\ (\ square\ x\ 0\ ))$ ✓

Reguläre Sprachen

Satz 6.17. Eine Sprache L ist genau dann regulär, wenn $\langle L \rangle$, der minimale Automat für L, endlich ist.

Algorithmus 6.26 (Minimiere einen DFA $A = (Q, \Sigma, \delta, s, F)$)**.** Der Algorithmus verwendet eine globale Variable $R \subseteq Q \times Q$. Er läuft wie folgt:

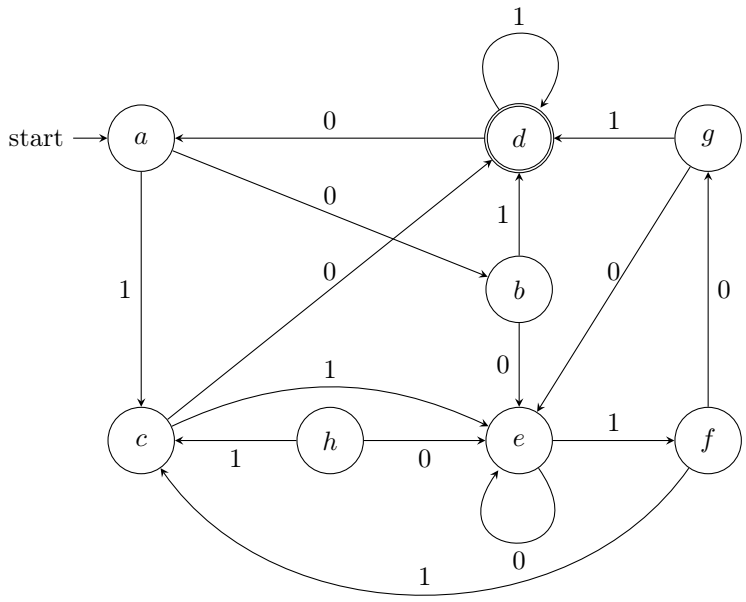
1. Entferne aus Q alle nicht erreichbaren Zustände.
2. Initialisiere R auf $\{(q_1, q_2) \mid q_1 \in F \Leftrightarrow q_2 \in F\}$
3. Suche ein Paar $(q_1, q_2) \in R$ und einen Buchstaben $a \in \Sigma$ mit

   $$(\delta(a, q_1), \delta(a, q_2)) \notin R$$

   Wenn kein solches Paar gefunden wird, gehe zu Schritt 4. Andernfalls entferne (q_1, q_2) aus R und fahre bei 3. fort.
4. Identifiziere alle Zustandspaare in R.

Wenn man den Algorithmus von Hand verwendet, sollte man ausnutzen, dass offenbar R stets symmetrisch ist und alle Paare (q, q) stets in R verbleiben. Es reicht also, eine dreieckige Tabelle zu führen, in der jedes Paar nur in einer der beiden möglichen Anordnungen vorkommt, und die Diagonalelemente (q, q) gar nicht. Man streicht dann zunächst die Zustandspaare, die hinsichtlich Finalität nicht übereinstimmen, und streicht dann wiederholt Paare heraus, die nach dem Kriterium in Schritt 3 aus R zu entfernen sind.

Beispiel Klausur [Probe] SoSe 2017, Aufgabe 3



1. Streiche Zustand h
2. Stelle R auf: teste Eingabe $\in \{0, 1\}$ und prüfe Ergebnis(-feld) – beispielsweise:
   - (d, f) mit Eingabe $\{\} \rightarrow$ anderes Ergebnis $\Rightarrow$ mit 0 füllen
   - (b, e) mit Eingabe $1 \rightarrow$ in (d, f) bereits eine $0 \Rightarrow$ mit 1 füllen
   - (a, e) mit Eingabe $0 \rightarrow$ in (b, e) bereits eine $1 \Rightarrow$ mit 2 füllen

	g	f	e	d	c	b
a	1	$\times$	2	0	1	1
b	$\times$	1	1	0	1	
c	1	1	1	0		
d	0	0	0			
e	1	2				
f	1					

Beschreibung von R:

- $\times$: gleiches Ergebnis bei beliebiger Eingabe $\Rightarrow$ können zusammengefasst werden
- n : bei beliebiger Eingabe der Länge n unterschiedliches Ergebnis