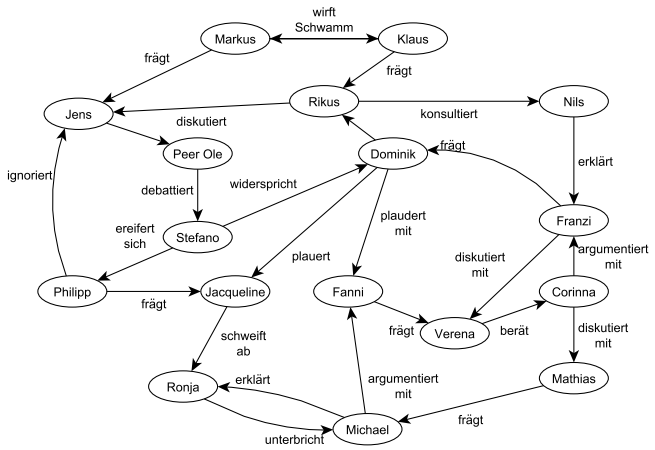
Inhaltsverzeichnis

Reguläre Sprachen und endliche Automaten



1 Aus der Kursankündigung

Reguläre Sprachen bilden eine tragende Säule der Informatik. Sie haben gute Abschlusseigenschaften und lassen sich sowohl durch Automaten darstellen, als auch über Logiken. Dies führt zu wichtigen Entscheidungsverfahren. Im Kurs wird die Theorie so weit erarbeitet, dass diese Verfahren implementiert werden können.

2 Einführung

Während der Akademie befasste sich der Kurs mit Eigenschaften von endlichen Automaten. Um eine Vorstellung von dem Ablauf eines Automaten zu bekommen, wurde die Funktionsweise eines Münztelefons näher betrachtet. Ein solches Telefon ist ein Automat, der von einem Menschen bedient wird. Dabei sieht der Nutzer, was passiert, wenn er eine bestimmte Aktion ausführt. Er kann aus der Ausgabe, also dem "Verhalten" des Münztelefons, auf dessen Zustand schließen. Wie in Abbildung 2.1 in runden Kreisen dargestellt, wird von fünf verschiedenen Zuständen bei dem Münztelefon ausgegangen. Es gibt den Anfangszustand, den Zustand, dass der Hörer abgenommen ist ("HAb"), dass der Hörer aufgelegt ist ("HAuf"), dass das Geld eingeworfen ist ("GEin") und dass beides der Fall ist. Es handelt sich dabei um eine Vereinfachung. Der Nutzer telefoniert folglich, sobald er Geld eingeworfen und den Hörer abgenommen hat. Die Zeit spielt dabei keine Rolle. Um von einem Zustand in den anderen Zustand zu gelangen benötigt der Automat gewisse Aktionen. Diese sind in Abbildung 2.1 als Pfeile dargestellt. Um zum Beispiel in den Zustand "HAb" zu gelangen, benötigt der Automat die Eingabe "Hörer abnehmen". Theoretisch ist auf jeden Zustand jede Aktion anwendbar. Bei dem praktischen Beispiel des Münztelefons wird jedoch schnell deutlich, dass bestimmte Aktionen aus physikalischen Gründen zu manchen Zeitpunkten nicht möglich sind. Zum Beispiel kann der Nutzer den Hörer nicht auflegen, wenn dieser zuvor nicht abgenommen wurde. Diese unmöglichen Fälle sind in der Grafik nicht aufgezeichnet und führen zu einem Fehler. Beendet ist die Benutzung des Telefons, wenn der Nutzer den Hörer wieder aufgelegt hat. Das schließt nicht aus, dass der Nutzer danach wieder von vorne beginnen kann. Dies wird durch den Pfeil mit der Beschriftung "neue Nutzung" beschrieben.

Nachdem sich der Kurs mit verschiedenen solcher praktischen Beispiele beschäftigt hatte, wurde das Konzept eines endlicher Automat definiert, also das eines Automaten mit nur endlich vielen Zuständen. Im weiteren Verlauf des Kurses wurde das Automatenmodell immer abstrakter, indem seine Zustände nicht mehr als für den Nutzer sichtbar dargestellt wurden. Auch die Aktionen waren keine "Taten" von Nutzern mehr, sondern eine Eingabe wurde zu einer Zahl oder einem Buchstaben. Ziel dieser Verallgemeinerung war, zu erkennen, dass man durch die Kombination von Aktionen, eine Zahlen- oder eine Buchstabenfolge erhält. Legt man nun noch Endzustände fest, kann man den Begriff von Automaten so definieren, dass entsprechende Automaten bestimmte Zeichenfolgen erkennen. Nur bei bestimmten Kombinationen von Aktionen gelangt der Automat dann in einen dieser festgelegten Endzustände. Alle möglichen endlichen Zeichenfolgen, die dabei entstehen können,

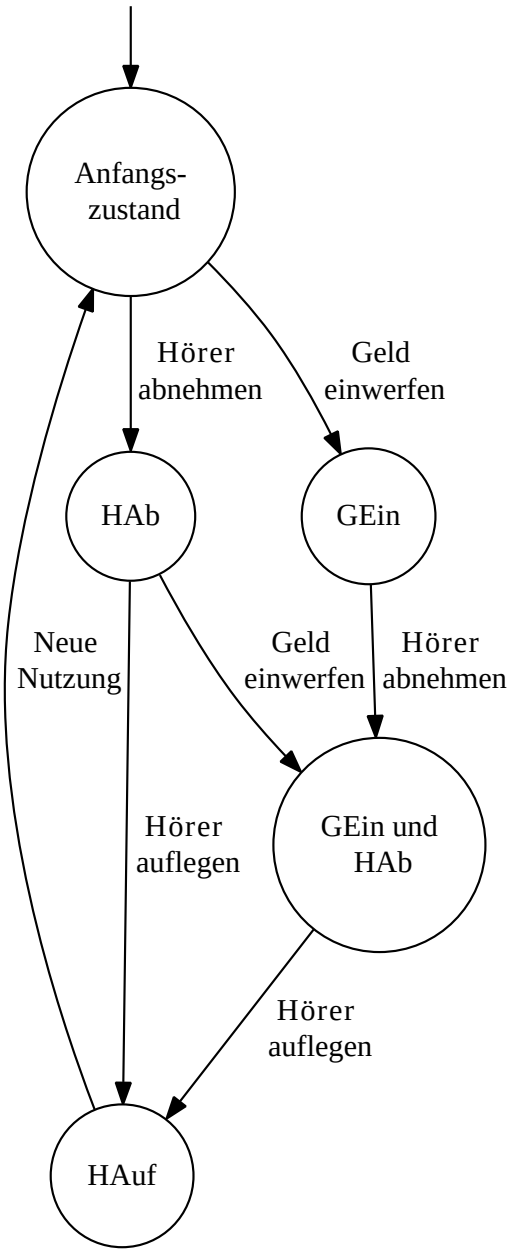


Abb. 2.1: Zustandsdiagramm eines Telefons

bilden zusammen die Sprache des Automaten. Eine Sprache nennen wir genau dann *regulär*, wenn sie die Sprache eines endlichen Automaten ist.

Besonders wurden die Abschlusseigenschaften endlicher Automaten betrachtet. So kann man zum Beispiel aus zwei gegebenen Automaten einen neuen konstruieren, dessen Sprache die Schnittmenge der der beiden ursprünglichen Automaten ist. Das gleiche gilt für Vereinigung, Projektion und Komplement. Weiter wurde die Presburger Arithmetik als mathematisches Modell dargestellt, deren Semantik als reguläre Sprache verstanden werden kann. Aus einer Formel kann man effektiv einen Automaten für diese Semantik konstruieren. Die Konjunktion der Presburger Arithmetik entspricht zum Beispiel dem Schnitt der den Konjunkten entsprechenden Automaten. Der Leerheitstest, der ebenfalls im Zusammenhang mit den Automaten eingeführt worden war, konnte genutzt werden, um festzustellen, ob eine geschlossene Formel richtig oder falsch ist.

Als eine weitere wichtige Eigenschaft wurde der Determinismus eines Automaten besprochen und eine Möglichkeit gesucht, aus einem nichtdeterministischen Automaten einen deterministischen zu machen. Nicht-deterministisch bedeutet, dass es bei einem Automaten die Möglichkeit gibt von einem Zustand aus über die gleiche Eingabe zu verschiedenen neuen Zuständen zu gelangen. Von einem deterministischen Automaten spricht man folglich, wenn es von jedem Zustand aus für eine bestimmte Eingabe immer genau eine Möglichkeit gibt, zu einem folgenden Zustand zu gelangen. Mit all diesen erarbeiteten Eigenschaften entwarf und realisierte der Kurs ein Konzept zur Implementierung von Automaten für die Semantik der Presburger Arithmetik.

2.1 **Beispiel Aufzug**

In der erste Gruppenarbeit sollten erste Definitionen von Automaten gefunden werden. Hierzu entwickelte eine Gruppe einen Automaten, der die Funktionsweise eines Aufzugs beschreibt. Hierbei wurde nicht auf Effizienz geachtet, sondern nur darauf, dass alle Personen irgendwann ihr Fahrtziel erreichen. In diesem Konzept wurde zwischen zwei verschiedenen Arten von Tasten unterschieden. Einer wird bei Fahrtwunsch nach oben gedrückt, der andere bei Fahrtwunsch nach unten. Außerdem sind im Aufzug Knöpfe für die einzelnen Stockwerke angebracht, mit denen der Fahrgast seine Zieletage angibt. Das Funktionsprinzip basiert darauf, dass sich der Aufzug grundsätzlich einmal vollständig von oben nach unten und anschließend umgekehrt bewegt, bis alle Fahrtwünsche erfüllt sind. Dabei öffnen sich die Aufzugtüren nur in einer Etage, in der die "Aufzugrufentaste" in Fahrtrichtung gedrückt wurde oder in der ein Fahrgast den Aufzug verlassen möchte. Von diesem Beispiel ausgehend entwickelte sich eine erste Definition von endlichen Automaten. Demnach ist ein endlicher Automat durch "Q" (der Menge der Zustände), "I" (der Menge der Anfangszustände), "Σ" (dem Eingabealphabet), "O" (dem Ausgabealphabet) und "Δ" (der Übergangsfunktion) mit $\Delta\colon Q \times \Sigma \to Q \times O$ definiert.

Ein Automat ist eine reale oder virtuelle Ma-

schine, die von einem Anfangszustand I mittels einer Eingabe aus dem Alphabet Σ und einer Übergangsfunktion Δ in einen neuen Zustand q' versetzt wird. Dieser Zustand kann durch weitere Eingaben geändert werden. Hierbei gilt, dass unterschiedliche Zustände zu verschiedenen Ausgaben o führen können. Endliche Automaten sind dadurch definiert, dass sie eine endliche Anzahl von Zuständen und Übergangsfunktion haben.

2.2 Beispiel Stammkunde

Eine weitere Gruppe hat das Beispiel eines Kellners behandelt, der einen einzelnen Stammkunden betreut. Dieser Kunde bestellt immer nur, wenn keine seiner Bestellungen mehr aussteht, das heißt, wenn das Bestellte bereits serviert wurde. Außerdem muss die Ausführung der Bestellung, sprich das Servieren innerhalb eines bestimmten Zeitrahmens erfolgen. Man betrachte die Zustände des Kellners als die Zustände des zu entwickelnden Automaten. Diese werden als "Warten auf den Kunden", "Bestellung", "Warten auf die Küche", "Beschweren", "Servieren" und "Kassieren" festgelegt. Der Anfangszustand ist "Warten auf den Kunden", der Endzustand "Kassieren". Der Übergang zwischen den einzelnen Zuständen wird dadurch ausgelöst, dass der Kunde eine Aktion initialisiert und der Kellner diese ausführt. Der Zustand "Beschweren" wurde eingeführt, damit der Kunde nicht ewig auf sein Essen warten muss. Der Kellner kann sich bei der Küche beschweren und das Essen kann danach sofort serviert werden.

Anhand dieses Beispiels wurde eine weitere Definition für einen Automat entwickelt. Diese besagt, dass ein endlicher Automat aus einer festen Menge an Zuständen besteht, wobei die Übergänge zwischen den Zuständen durch Aktionen ausgelöst werden. Die Abfolge der Zustände ist durch feste Regeln eingeschränkt.

Man kann ihn mathematisch darstellen als eine Funktion, deren Parameter der gegenwärtige Zustand des Automaten und die erfolgende Aktion sind. Ergebnis der Funktion ist ein neuer Zustand, enthalten in der Menge der definierten Zustände. Wenn eine Aktion nicht ausführbar ist, wird der letzte valide Zustand wiederhergestellt.

Zwei Automaten verhalten sich nicht gleich, wenn bei gleicher Abfolge von Aktionen die Automaten unterschiedliche Reaktionen zeigen. Das heißt, ein Automat muss anzeigen, ob eine Aktion zu einem definierten Zustand führt oder nicht.

3 Notationen

Um Missverständnisse zu vermeiden, einigte sich unser Kurs auf einige mathematische Notationen, die im Folgenden kurz erklärt werden.

3.1 Logische Notationen

Die Zeichen $\wedge$, $\vee$ und $\neg$ sind logisches »und«, »oder«, »nicht«. Ferner ist $\Rightarrow$ die logische Folgerung, also »Wenn ..., dann ...«. Es ist $A \Rightarrow B$ gleichbedeutend mit $(\neg A) \vee B$. Das Symbol $\Longleftrightarrow$ steht für »... genau dann, wenn ...«, das heißt, $A \Longleftrightarrow B$ ist also gleichbedeutend mit $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Schließlich stehen $\forall$, $\exists$ und $\exists!$ für »für alle«, »es gibt ein« und »es gibt genau ein«.

3.2 Mengentheoretische Notationen

Unter einer Menge verstehen wir eine Zusammenfassung von bestimmten Objekten unseres Denkens – den Elementen der Menge – zu einem Ganzen. Die leere Menge bezeichnen wir mit $\emptyset$. Das Symbol $\mathbb{N}$ bezeichnet die Menge $\{0, 1, 2, \ldots\}$ der natürlichen Zahlen. 1 bezeichnet auch die Menge $1 = \{0\}$.

Für zwei Mengen A und B sagen wir, dass A eine *Teilmenge* von B ist, falls jedes Element von A auch Element von B ist und schreiben abkürzend $A \subseteq B$. Ferner heißt $A \subsetneq B$, dass A eine echte Teilmenge von B ist, also dass $A \subseteq B \wedge A \neq B$. Die Schreibweise $A \subset B$ kann $A \subseteq B$ oder $A \subsetneq B$ bedeuten; es sollte jeweils aus dem Kontext hervorgehen, welches der beiden Zeichen gemeint ist.

$A \cup B = \{x \mid x \in A \vee x \in B\}$ ist die Vereinigung von zwei Mengen A und B. Es gilt also insbesondere $A \subset A \cup B$ und $B \subset A \cup B$. Ist M eine Menge von Mengen, so bezeichnet $\bigcup M$ die Vereinigung ihrer Elemente, das heißt $\bigcup M = \bigcup_{X \in M} X$.

$A \cap B = \{x \mid x \in A \wedge x \in B\}$ ist der Schnitt von zwei Mengen A und B. Es gilt also insbesondere $A \cap B \subset A$ und $A \cap B \subset B$. Ist M eine Menge von Mengen, so bezeichnet $\bigcap M$ den Schnitt ihrer Elemente, das heißt $\bigcap M = \bigcap_{X \in M} X$.

Für eine Menge X bezeichnet $\mathfrak{P}(X)$ ihre Potenzmenge, also die Menge aller ihrer Teilmengen. In anderen Worten $\mathfrak{P}(X) = \{Y \mid Y \subseteq X\}$.

Das geordnete Paar von a und b bezeichnen wir mit $\langle a, b\rangle$. Für zwei Mengen A und B ist das kartesische Produkt $A \times B$ definiert durch $A \times B = \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$. Ferner steht $A \times B \times C$ für $A \times (B \times C)$. Ist also $a \in A \wedge b \in B \wedge c \in C$, so ist $\langle a, \langle b, c\rangle\rangle \in A \times B \times C$. Für $\langle a, \langle b, c\rangle\rangle$ schreiben wir auch $\langle a, b, c\rangle$.

$A + B$ ist die disjunkte Vereinigung von A und B. Genauer definiert man $A + B = \{0\} \times A \cup$

$\{1\}\times B = \{\langle 0, a\rangle \mid a \in A\} \cup \{\langle 1, b\rangle \mid b \in B\}$. Ferner steht $A + B + C$ für $A + (B + C)$. Ist also $c \in C$, so ist $\langle 1, 1, c\rangle = \langle 1, \langle 1, c\rangle\rangle \in A + B + C$.

$f\colon A \to B$ meint »f ist eine Funktion von A nach B«. Das heißt, f weist jedem Element $a \in A$ ein Element $b = f(a) \in B$ zu. Man schreibt auch $a \mapsto b$, falls f aus dem Kontext klar ist. Formal ist eine Funktion $f\colon A \to B$ eine Menge $f \subseteq A \times B$ mit $\forall a \in A\, \exists!\, b \in B(\langle a, b\rangle \in f)$.

Definition 1 *(Relation)* Für zwei Mengen X und Y ist eine *(Binär-)Relation zwischen* X und Y eine Teilmenge R von $X \times Y$. Statt $\langle x, y\rangle \in R$ schreiben wir auch xRy. Im Fall $X = Y$ heißt R auch *(Binär-)Relation auf* (oder über) X.

Definition 2 *(Äquivalenzrelation)* Eine Relation R auf einer Menge X ist eine *Äquivalenzrelation*, falls sie folgende drei Eigenschaften erfüllt.

1. R ist reflexiv, das heißt $\forall x(xRx)$.
2. R ist symmetrisch, das heißt $\forall x \forall y(xRy \Rightarrow yRx)$.
3. R ist transtitiv, das heißt $\forall x \forall y \forall z(xRy \wedge yRz \Rightarrow xRz)$.

Definition 3 *(Äquivalenzklasse)* Sei R eine Äquivalenzrelation über X. Die Äquivalenzklasse von $x \in X$ ist $[x]_R := \{y \mid yRx\}$. Falls R aus dem Kontext klar ist, schreiben wird auch einfach nur $[x]$. Die Menge aller Äquivalenzklasse bezeichen wir mit $X/R := \{[x]_R \mid x \in X\}$.

Definition 4 *(Partitionierung)* Für eine Menge X ist $M \in \mathfrak{P}(X)$ eine *Partitionierung* von X, wenn folgendes gilt.

1. $\bigcup M = X$
2. $\emptyset \notin M$
3. Für alle $A, B \in M$ gilt $A = B$ oder $A \cap B = \emptyset$.

Ist M Partionierung von X so wird durch

$$x \sim_M y \iff \exists A \in M(x \in A \wedge y \in A)$$

eine Relation definiert, die von der Partitionierung *induzierte Relation*.

Proposition 5 Für eine Äquivalenzrelation R über X, ist X/R eine Partitionierung von X und jede Partitionierung entsteht in dieser Weise. Für eine Partitionierung M von X ist $\sim_M$ eine Äquivalenzrelation, und jede Äquivalenzrelation entsteht in dieser Weise.

4 Automaten für endliche Wörter

4.1 Definitionen

Definition 6 Ist Σ eine Menge, so bezeichnet Σ^* die Menge der endlichen Folgen von Elementen in Σ und $\Sigma^\omega = \Sigma^{\mathbb{N}}$ die Menge der unendlichen Folgen von Elementen in Σ. Ein Element $x \in \Sigma^*$ nennen wir *(endliches) Wort* über dem Alphabet Σ und ein Element $x \in \Sigma^\omega$ nennen wir *(unendliches) Wort* über dem Alphabet Σ. Wir schreiben ε für das leere Wort. Für zwei Wörter $x \in \Sigma^*$ und $y \in \Sigma^* \cup \Sigma^\omega$ sei xy das Wort, das durch Aneinanderhängen von x und y entsteht. Weiter sei x^n für jedes Wort $x \in \Sigma^*$ und alle $n \in \mathbb{N}$ rekursiv durch $x^0 = \varepsilon$ und $x^{n+1} = x^n x$ definiert.

Definition 7 Ein *Automat* $\mathfrak{A}$ ist gegeben durch

- eine endliche Menge Q, genannt *Menge von Zuständen*,
- eine endliche Menge Σ, genannt *Alphabet*,
- eine Relation $\Delta \subseteq Q \times \Sigma \times Q$, genannt *Transitions-* oder *Übergangsrelation*,
- eine Menge $I \subseteq Q$, genannt *Menge von Anfangszuständen* und
- eine Menge $F \subseteq Q$, genannt *gute Zustände* oder *Endzustände*.

Definition 8 Ein *Automat für endliche Wörter* ist ein Automat $\mathfrak{A}$. Seine *Sprache* $\mathcal{L}(\mathfrak{A}) \subset \Sigma^*$ ist definiert als

$$\begin{aligned}\mathcal{L}(\mathfrak{A}) = \{&a_1 \dots a_k \mid \exists q_0, \dots, q_k \in Q \\ &(q_0 \in I \wedge (\forall 0 \le i < k : \langle q_i, a_{i+1}, q_{i+1}\rangle \in \Delta) \\ &\wedge q_k \in F)\ \}\end{aligned}$$

wobei Q, I, F und Δ respektiv die Zustandsmenge, Anfangszustände, Endzustände und Transitionsrelation von $\mathfrak{A}$ sind.

Eine Sprache $\mathcal{L} \subset \Sigma^*$ heißt *regulär*, falls es einen Automaten $\mathfrak{A}$ (für endliche Wörter) gibt, für den $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$ gilt.

Im Zusammenhang mit endlichen Automaten ist vor allem interessant, Abschlusseigenschaften regulärer Sprachen zu untersuchen. Wir zeigen in den nächsten Abschnitten, dass reguläre Sprachen unter *Vereinigung-, Schnitt-* und *Komplementbildung* abgeschlossen sind.

4.2 Vereinigung

Satz 9 Seien $\mathcal{L}_1 \subset \Sigma_1^*$ und $\mathcal{L}_2 \subset \Sigma_2^*$ reguläre Sprachen. Dann ist auch $\mathcal{L}_1 \cup \mathcal{L}_2$ regulär.

Beweis Seien $\mathfrak{A}_1 = (Q_1, \Sigma_1, \Delta_1, I_1, F_1)$ und $\mathfrak{A}_2 = (Q_2, \Sigma_2, \Delta_2, I_2, F_2)$ zwei endliche Automaten mit $\mathcal{L}(\mathfrak{A}_1) = \mathcal{L}_1$ und $\mathcal{L}(\mathfrak{A}_2) = \mathcal{L}_2$. Sei $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ definiert durch

- die Menge der Zustände $Q = Q_1 + Q_2$,

– das Alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$,
– die Transitionsrelation

$$\begin{aligned}\Delta =&\{\langle\langle 0, q\rangle, a, \langle 0, q'\rangle\rangle | \langle q, a, q'\rangle \in \Delta_1\}\\ &\cup \{\langle\langle 1, q\rangle, a, \langle 1, q'\rangle\rangle | \langle q, a, q'\rangle \in \Delta_2\}\\ &\subset Q \times \Sigma \times Q\ ,\end{aligned}$$

– die Anfangszustände $I = I_1 + I_2$ und
– die Endzustände $F = F_1 + F_2$.

Es ist zu zeigen, dass $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathfrak{A}_1) \cup \mathcal{L}(\mathfrak{A}_2)$ gilt. Sei $w \in \mathcal{L}(\mathfrak{A})$ beliebig. Dann gibt es einen Zustand $\langle a, q\rangle$ in I mit $a \in \{0, 1\}$ und $q \in Q_1 \cup Q_2$, so dass der Zustand $\langle a, q'\rangle$, der nach der Ausführung gemäß Δ mit w aus $\langle a, q\rangle$ entsteht, in F liegt. Nach der Definition von Δ und I folgt hieraus, dass q' in F_1 oder in F_2 liegt. Das bedeutet, dass $w \in \mathcal{L}(\mathfrak{A}_1)$ oder $w \in \mathcal{L}(\mathfrak{A}_2)$ ist, woraus $w \in \mathcal{L}(\mathfrak{A}_1) \cup \mathcal{L}(\mathfrak{A}_2)$ folgt. Analoges gilt für $w \in \mathcal{L}(\mathfrak{A}_1) \cup \mathcal{L}(\mathfrak{A}_2)$. Somit gilt $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathfrak{A}_1) \cup \mathcal{L}(\mathfrak{A}_2)$. □

Die auszuführenden Aktionen spielen sich in einem der beiden gegebenen Automaten ab. Die Endzustände des neuen Automaten sind also jene Zustände, in denen einer der gegebenen Automaten einen Endzustand erreicht hat. Somit sind genau nur die Wörter erlaubt, die mindestens einer der gegebenen Automaten akzeptiert hätte.

4.3 Schnitt

Satz 10 Seien $\mathcal{L}_1 \subset \Sigma_1^*$ und $\mathcal{L}_2^* \subset \Sigma_2^*$ reguläre Sprachen. Dann ist auch $\mathcal{L}_1 \cap \mathcal{L}_2$ regulär.

Beweis Seien $\mathfrak{A}_1 = (Q_1, \Sigma_1, \Delta_1, I_1, F_1)$ und $\mathfrak{A}_2 = (Q_2, \Sigma_2, \Delta_2, I_2, F_2)$ zwei endliche Automaten mit $\mathcal{L}(\mathfrak{A}_1) = \mathcal{L}_1$ und $\mathcal{L}(\mathfrak{A}_2) = \mathcal{L}_2$. Sei $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ definiert durch
– die Menge der Zustände $Q = Q_1 \times Q_2$,
– das Alphabet $\Sigma = \Sigma_1 \cap \Sigma_2$,
– die Transitionsrelation

$$\begin{aligned}\Delta =&\{\langle\langle q_1, q_2\rangle, a, \langle q_1', q_2'\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in \Delta_1 \wedge\\ &\qquad\qquad \langle q_2, a, q_2'\rangle \in \Delta_2\}\\ \subset& Q \times \Sigma \times Q\ ,\end{aligned}$$

– die Anfangszustände $I = I_1 \times I_2$ und
– die Endzustände $F = F_1 \times F_2$.

Es ist zu zeigen, dass $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathfrak{A}_1) \cap \mathcal{L}(\mathfrak{A}_2)$ gilt. Sei $w \in \mathcal{L}(\mathfrak{A})$ beliebig. Dann gibt es ein $\langle q_1, q_2\rangle \in I$, sodass der Zustand $\langle q_1', q_2'\rangle$, der nach der Ausführung gemäß Δ mit w aus $\langle q_1, q_2\rangle$ entsteht, in F liegt. Nach der Definition von Δ und I folgt hieraus jedoch induktiv, dass q_1' in F_1 und q_2' in F_2 liegt. Dies ist äquivalent zu $w \in \mathcal{L}(\mathfrak{A}_1)$ und $w \in \mathcal{L}(\mathfrak{A}_2)$, woraus $w \in \mathcal{L}(\mathfrak{A}_1) \cap \mathcal{L}(\mathfrak{A}_2)$ folgt. Analog lässt sich zeigen, dass ein beliebiges w in $\mathcal{L}(\mathfrak{A}_1)$ und $w \in \mathcal{L}(\mathfrak{A}_2)$ auch in $\mathcal{L}(\mathfrak{A})$ liegt. Somit gilt $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathfrak{A}_1) \cap \mathcal{L}(\mathfrak{A}_2)$. □

Die auszuführenden Aktionen müssen sich immer in beiden Urautomaten abspielen lassen. Ein Zustand besteht aus einem Pärchen von zwei Zuständen, jeweils aus den beiden Urautomaten, und eine Aktion bewirkt eine Änderung in beiden Komponenten. Die Endzustände sind also die Zustände, in denen beide Urautomaten einen Endzustand erreicht haben. Somit werden nur genau die Wörter erkannt, die von beide Urautomaten erkannt werden.

4.4 Shuffle

Definition 11 Für endliche Alphabete Σ_1, Σ_2 und ein Wort $w \in (\Sigma_1 + \Sigma_2)^*$, $w = (\langle d_0, x_0\rangle \ldots \langle d_k, x_k\rangle)$, sei $n_0 \ldots n_\ell$ die aufsteigende Folge derjenigen Indices n, für die $d_n = 0$ gilt. Das Wort $\hat{w} = x_{n_0} \ldots x_{n_\ell} \in \Sigma_1^*$ nennen wir dann *Projektion von w auf Σ_1* und schreiben $\pi_{\Sigma_1}(w) = \hat{w}$. Analog definieren wir die Projektion auf Σ_2.

Definition 12 Für endliche Alphabete Σ_1, Σ_2 und Sprachen $\mathcal{L}_1 \subset \Sigma_1^*$, $\mathcal{L}_2 \subset \Sigma_2^*$ bezeichnen wir die Sprache derjenigen Wörter aus $(\Sigma_1 + \Sigma_2)^*$, für die die Projektionen auf Σ_1 und Σ_2 in $\mathcal{L}_1$ beziehungsweise $\mathcal{L}_2$ liegen, als *Shuffle* von $\mathcal{L}_1$ und $\mathcal{L}_2$; als abkürzende Schreibweise wählen wir $\mathcal{L}_1 || \mathcal{L}_2$.

Satz 13 Seien $\mathfrak{A}_1$ und $\mathfrak{A}_2$ zwei endliche Automaten. Es gibt einen Automaten $\mathfrak{A}$, der die Sprache $\mathcal{L}(\mathfrak{A}) = \mathcal{L}(\mathfrak{A}_1) || \mathcal{L}(\mathfrak{A}_2)$ erkennt. Der neue Automat $\mathfrak{A}$ ist definiert durch
– die Menge der Zustände $Q = Q_1 \times Q_2$,
– das Alphabet $\Sigma = \Sigma_1 + \Sigma_2$,
– die Transitionsrelation

$$\begin{aligned}&\{\langle\langle q_1, q_2\rangle, \langle 0, a\rangle, \langle q_1', q_2\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in \Delta_1\} \cup\\ &\{\langle\langle q_1, q_2\rangle, \langle 1, a\rangle, \langle q_1, q_2'\rangle\rangle \mid \langle q_2, a, q_2'\rangle \in \Delta_2\}\\ &\subset (Q_1 \times Q_2) \times (\Sigma_1 + \Sigma_2) \times (Q_1 \times Q_2)\ ,\end{aligned}$$

– die Anfangszustände $I = I_1 \times I_2$ und
– die Endzustände $F = F_1 \times F_2$.

Die auszuführenden Aktionen werden jeweils in genau einem der beiden Automaten ausgeführt. Dabei ändert sich die jeweilige Komponente des Zustandes. Ein Zustand besteht wieder aus einem Paar von zwei Zuständen, jeweils aus den beiden Urautomaten. Eine Aktion darf genau dann ausgeführt werden, wenn sie in dem jeweiligen Urautomaten ausgeführt werden könnten. Die Projektion des neuen Automaten $\mathfrak{A}$ auf Urautomaten ist also korrekt. Die Endzustände des neuen Automaten sind genau die Zustände, in denen beide gegebenen Automaten einen Endzustand erreichen. Der Automat erkennt also genau die Wörter, die in der jeweiligen Projektion zurück auf ihre Urautomaten zu einem Endzustand führen.

4.5 Projektion

SATZ 14 Seien Σ_1 und Σ_2 endliche Alphabete und $\mathcal{L}$ eine reguläre Sprache über $\Sigma_1 \times \Sigma_2$. Es sei $\pi_1 \colon \Sigma_1 \times \Sigma_2 \to \Sigma_1$ die Projektion von $\mathcal{L}$ auf die erste Komponente, das heißt $\pi_1(\langle x, y\rangle) = y$. Dann ist auch die Sprache $\pi_1(\mathcal{L}) := \{\pi_1(a_1)\dots\pi_1(a_n) \mid a_1 \dots a_n \in \mathcal{L}\}$ regulär.

BEWEIS Sei $\mathfrak{A} = (Q, \Sigma_1 \times \Sigma_2, \Delta, I, F)$ ein Automat mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$. Wir betrachten nun den Automaten $\mathfrak{A}' = (Q, \Sigma_1, \{\langle q, x, q'\rangle \mid \exists y \in \Sigma_2(\langle q, \langle x, y\rangle, q'\rangle \in \Delta)\}, I, F)$. Offensichtlich akzeptiert $\mathfrak{A}'$ nun genau die Wörter $x = x_1 \dots x_n \in \Sigma_1^*$, für die es ein Wort $y = y_1 \dots y_n \in \Sigma_2^*$ gibt, für das der Automat $\mathfrak{A}$ das Wort $\langle x_1, y_1\rangle \dots \langle x_n, y_n\rangle$ akzeptiert. Damit gilt wie gewünscht $\mathcal{L}(\mathfrak{A}') = \pi_1(\mathcal{L})$, das heißt $\pi_1(\mathcal{L})$ ist regulär. □

4.6 Leerheitstest

Im Folgenden wird ein Algorithmus vorgestellt, der prüft, ob die Sprache eines Automaten leer ist. Der Algorithmus beginnt in jedem Initialzustand, prüft welche Pfade zu einem anderen Zustand ausführbar sind und ruft sich in den dementsprechenden Zuständen neu auf. Er speichert dabei, welche Zustände er schon bearbeitet hat. Erreicht er einen Endzustand, ist erwiesen, dass die Sprache des Automaten nicht leer ist. Wird der Algorithmus ausschließlich dadurch beendet, dass er auf einen Zustand trifft, den er schon bearbeitet hat oder von dem keine Pfade zu anderen Zuständen führen, ist die Sprache leer.

4.7 Determinismus

DEFINITION 15 Ein Automat $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ wird *deterministisch* genannt, wenn zu jedem Zustand $q \in Q$ und jeder Eingabe $x \in \Sigma$ genau ein Zustand $q' \in Q$ mit $\langle q, x, q'\rangle \in \Delta$ existiert und es genau einen Anfangszustand gibt, das heißt I genau ein Element enthält.

Bevor wir beweisen können, dass jeder Automat determinisiert werden kann, definieren wir.

DEFINITION 16 Für einen endlichen Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$, eine Menge $P \subset Q$ von Zuständen und ein Wort $x \in \Sigma^*$ sei $\widehat{\delta}_{\mathfrak{A}}(P, x)$ die Menge derjenigen Zustände, in denen sich $\mathfrak{A}$ nach Eingabe des Wortes x befinden kann, wenn er in einem beliebigen Zustand $p \in P$ startet und sich gemäß Δ bewegt.

DEFINITION 17 Für einen endlichen, deterministischen Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$, einen Zustand $q \in Q$ und ein Wort $x \in \Sigma^*$ sei $\delta_{\mathfrak{A}}(q, x)$ derjenige Zustand p, in dem sich $\mathfrak{A}$ nach Eingabe des Wortes x befindet, wenn er in q startet und sich gemäß Δ bewegt. Da $\mathfrak{A}$ deterministisch ist, existiert $\delta_{\mathfrak{A}}(q, x)$ für alle $q \in Q$, $x \in \Sigma^*$ und ist eindeutig. Es gilt also $\{\delta_{\mathfrak{A}}(q, x)\} = \widehat{\delta}_{\mathfrak{A}}(\{q\}, x)$.

SATZ 18 Sei $\mathcal{L} \subset \Sigma^*$ regulär. Dann gibt es einen deterministischen Automaten $\mathfrak{A}'$ mit $\mathcal{L}(\mathfrak{A}') = \mathcal{L}$.

BEWEIS Wir betrachten einen Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$. Sei $\mathfrak{A}' = (Q', \Sigma', \Delta', I', F')$ definiert durch $Q' = \mathfrak{P}(Q)$, $\Sigma' = \Sigma$, $\Delta' = \{\langle p', x, q'\rangle \in Q' \times \Sigma \times Q' \mid q' = \widehat{\delta}_{\mathfrak{A}}(p', x)\}$, $I' = \{I\}$ und $F' = \{q' \subseteq Q \mid q' \cap F \neq \emptyset\}$. Offensichtlich ist der konstruierte Automat $\mathfrak{A}$ deterministisch, denn für vorgegebene $p' \in Q'$ und $x \in \Sigma$ gibt es genau ein $q' \in Q'$ mit $\langle p', x, q'\rangle \in \Delta$, nämlich $q' = \widehat{\delta}_{\mathfrak{A}}(p', x)$. Wir beweisen nun, die folgende Behauptung. Für jedes Wort $x \in \Sigma^*$ gilt

$$\widehat{\delta}_{\mathfrak{A}}(I, x) = \delta_{\mathfrak{A}'}(I, x) \; .$$

Beweis mit Induktion nach der Länge n des Wortes x. Induktionsbeginn $n = 0$. Klar, denn $\widehat{\delta}_{\mathfrak{A}}(I, \varepsilon) = I = \delta_{\mathfrak{A}'}(I, \varepsilon)$. Induktionsschritt $n \to (n+1)$. Wir betrachten ein Wort $x = x_1 \dots x_{n+1}$ aus $n+1$ Zeichen. Sei $x' = x_1 \dots x_n$. Dann ist

$$\begin{aligned}
\widehat{\delta}_{\mathfrak{A}}(I, x) &= \widehat{\delta}_{\mathfrak{A}}(\widehat{\delta}_{\mathfrak{A}}(I, x'), x_{n+1}) \\
&\overset{I.V.}{=} \widehat{\delta}_{\mathfrak{A}}(\delta_{\mathfrak{A}'}(I, x'), x_{n+1}) \\
&\overset{\text{Def.}\mathfrak{A}'}{=} \delta_{\mathfrak{A}'}(\delta_{\mathfrak{A}'}(I, x'), x_{n+1}) \\
&= \delta_{\mathfrak{A}'}(I, x) \; .
\end{aligned}$$

Dies beendet den Induktionsschritt und damit den Beweis der Behauptung.

Zeigen wir nun, dass $\mathcal{L}(\mathfrak{A})' = \mathcal{L}(\mathfrak{A}) = \mathcal{L}$ gilt. Wir haben

$$\begin{aligned}
x \in \mathcal{L}(\mathfrak{A}') &\iff \delta_{\mathfrak{A}'}(I, x) \in F' \\
&\iff \delta_{\mathfrak{A}'}(I, x) \cap F \neq \emptyset \\
&\iff \widehat{\delta}_{\mathfrak{A}}(I, x) \cap F \neq \emptyset \iff x \in \mathcal{L}(\mathfrak{A}) \; .
\end{aligned}$$

Also sind die Sprachen $\mathcal{L}(\mathfrak{A}')$ und $\mathcal{L}(\mathfrak{A})$ tatsächlich identisch. □

4.8 Komplementierung

Nun sind wir in der Lage, eine weitere Abschlusseigenschaft regulärer Sprachen – die Komplementierung – nachzuweisen.

SATZ 19 Sei Σ ein endliches Alphabet und $\mathcal{L} \subset \Sigma^*$ eine reguläre Sprache. Dann ist auch die Sprache $\Sigma^* \backslash \mathcal{L}$ regulär.

BEWEIS Nach dem bisher Gezeigten existiert ein deterministischer Automat $\mathfrak{A} = (Q, \Sigma, \Delta, \{q_0\}, F)$

mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$. Wir betrachten nun den Automaten $\mathfrak{A}' = (Q, \Sigma, \Delta, \{q_0\}, Q\backslash F)$. Für diesen Automaten gilt für alle $x \in \Sigma^*$

$$\begin{aligned} x \in \mathcal{L}(\mathfrak{A}') &\Leftrightarrow \delta_{\mathfrak{A}'}(q_0, x) \in Q\backslash F \\ &\Leftrightarrow \neg(\delta_{\mathfrak{A}}(q_0, x) \in F) \\ &\Leftrightarrow \neg(x \in \mathcal{L}) \ . \end{aligned}$$

Also haben wir tatsächlich einen Automaten $\mathfrak{A}'$ mit $\mathcal{L}(\mathfrak{A}') = \Sigma^*\backslash\mathcal{L}$ konstruiert. □

4.9 Minimalautomaten

Ziel dieses Abschnittes ist es zu beweisen, dass für jeden Automaten $\mathfrak{A}$ ein deterministischer Automat $\mathfrak{A}'$ mit $\mathcal{L}(\mathfrak{A}') = \mathcal{L}(\mathfrak{A})$ *konstruiert* werden kann, der eine minimale Anzahl von Zuständen besitzt. Da dieses Resultat nicht völlig auf der Hand liegt, bedarf es einiger Vorbereitung. Wir erinnern daran, dass für einen endlichen, deterministischen Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$, einen Zustand $p \in Q$ und ein Wort $x \in \Sigma^*$ mit $\delta_{\mathfrak{A}}(p, x)$ derjenige eindeutige Zustand q bezeichnet wird, in dem sich $\mathfrak{A}$ nach Eingabe des Wortes x befindet, wenn er in p startet und sich gemäß Δ bewegt.

Definition 20 Sei $\mathcal{L} \subset \Sigma^*$ eine Sprache über dem Alphabet Σ und $\sim$ eine Äquivalenzrelation auf $\mathcal{L}$. Die Relation $\sim$ heißt *rechtsinvariant*, falls $\forall x, y, z \in \Sigma^*(x \sim y \Rightarrow xz \sim yz)$ gilt.

Wir beweisen zunächst den folgenden Satz.

Satz 21 Sei Σ ein endliches Alphabet und $\mathcal{L} \subset \Sigma^*$ eine Sprache. Dann sind folgenden Aussagen äquivalent.

(1) $\mathcal{L}$ ist regulär.
(2) $\mathcal{L}$ ist die Vereinigung von einigen der Äquivalenzklassen einer rechtsinvarianten Äquivalenzrelation $\sim$ auf Σ^*; dabei gibt es insgesamt nur endlich viele Äquivalenzklassen.
(3) Die Äquivalenzrelation $\sim_{\mathcal{L}}$ auf Σ^* sei definiert durch $x \sim_{\mathcal{L}} y :\Leftrightarrow (\forall z \in \Sigma^*(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}))$. Dann gibt es nur endlich viele verschiedene Äquivalenzklassen.

Beweis (1) ⇒ (2): Sei $\mathfrak{A} = (Q, \Sigma, \Delta, \{q_0\}, F)$ ein deterministischer Automat mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$. Definiere eine Äquivalenzrelation $\sim$ auf Σ^* durch $x \sim y :\Leftrightarrow \delta_{\mathfrak{A}}(q_0, x) = \delta_{\mathfrak{A}}(q_0, y)$. Offensichtlich ist $\sim$ wirklich eine Äquivalenzrelation. Ferner kann es nicht mehr Äquivalenzklassen als Zustände in Q geben, insbesondere gibt es also nur endlich viele Äquivalenzklassen. Bezeichnen wir die zu einem Wort $x \in \Sigma^*$ gehörende Äquivalenzklasse mit $[x]_\sim$, so gilt $\mathcal{L} = \bigcup_{x\in\mathcal{L}}[x]_\sim$, das heißt, $\mathcal{L}$ lässt sich darstellen als Vereinigung von einigen der Äquivalenzklassen von $\sim$. Schließlich ist $\sim$ rechtsinvariant, denn für alle $x, y, z \in \Sigma^*$ folgt aus $x \sim y$ zunächst $\delta_{\mathfrak{A}}(q_0, x) = \delta_{\mathfrak{A}}(q_0, y)$ und damit $\delta_{\mathfrak{A}}(q_0, xz) = \delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(q_0, x), z) = \delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(q_0, y), z) = \delta_{\mathfrak{A}}(q_0, yz)$, das heißt, $xz \sim yz$.

(2) ⇒ (3): Es genügt zu zeigen, dass jede Äquivalenzklasse von $\sim$ ganz in einer Äquivalenzklasse von $\sim_{\mathcal{L}}$ enthalten ist, denn nach Voraussetzung zerlegt $\sim$ die Menge Σ^* in nur endlich viele verschiedene Äquivalenzklassen. Seien also $x, y \in \Sigma^*$ mit $x \sim y$ vorgelegt. Dann gilt, da $\sim$ rechtsinvariant ist, $xz \sim yz$ für alle $z \in \Sigma^*$. Insbesondere gilt $(\forall z \in \Sigma^*(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}))$, das heißt, $x \sim_{\mathcal{L}} y$.

(3) ⇒ (1): Wir zeigen zunächst, dass $\sim_{\mathcal{L}}$ rechtsinvariant ist. Seien dazu $x, y \in \Sigma^*$ mit $x \sim_{\mathcal{L}} y$ vorgelegt. Wir wollen zeigen, dass für alle $z \in \Sigma^*$ $xz \sim_{\mathcal{L}} yz$ gilt, das heißt, dass $(\forall w \in \Sigma^*(xzw \in \mathcal{L} \Leftrightarrow yzw \in \mathcal{L}))$. Dies ist jedoch klar nach der Voraussetzung $(\forall v \in \Sigma^*(xv \in \mathcal{L} \Leftrightarrow yv \in \mathcal{L}))$ mit $v = zw$. Nun sei Q' die (endliche) Menge der Äquivalenzklassen von $\sim_{\mathcal{L}}$ und $[z]$ die zu einem $z \in \Sigma^*$ gehörende Äquivalenzklasse. Durch $\Delta' = \{\langle [z], x, [zx]\rangle \mid z \in \Sigma^*, x \in \Sigma\}$, $I' = \{[\varepsilon]\}, F' = \{[z] \mid z \in \mathcal{L}\}$ wird ein endlicher Automat $\mathfrak{A}' = (Q', \Sigma, \Delta', I', F')$ definiert, der sogar deterministisch ist. Diese Definition ist konsistent, da $\sim_{\mathcal{L}}$ rechtsinvariant ist, denn für $z_1, z_2 \in \Sigma^*$ mit $[z_1] = [z_2]$, das heißt, $z_1 \sim_{\mathcal{L}} z_2$ gilt $z_1x \sim_{\mathcal{L}} z_2x$ und damit $[z_1x] = [z_2x]$ für alle Eingabezeichen $x \in \Sigma$. Die Definition von Δ ist also vom jeweiligen Vertreter z des Zustandes $[z] \in Q'$ unabhängig. Schließlich gilt $\mathcal{L}(\mathfrak{A}') = \mathcal{L}$, denn $x \in \mathcal{L}(\mathfrak{A}') \Leftrightarrow \delta_{\mathfrak{A}'}(q'_o, x) = [x] \in F' \Leftrightarrow x \in \mathcal{L}$. Also ist $\mathcal{L}$ regulär. □

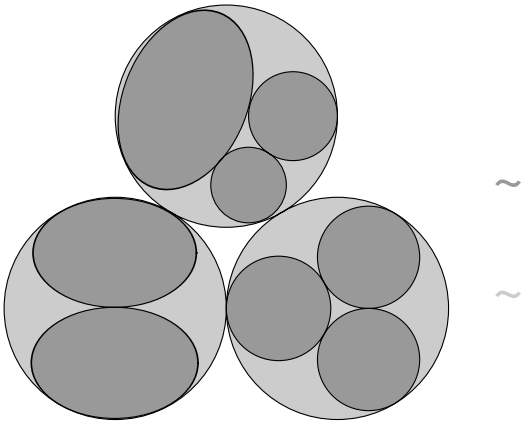


Abb. 2.2: Verfeinerung von Äquivalenzrelationen. Die durch einen Automaten induzierte Äquivalenzrelation ist dunkelgrau dargestellt, die durch dessen Sprache induzierte Äquivalenzrelation ist hellgrau. Dabei werden alle hellgrauen Fläche durch einige dunkelgraue vollständig *überdeckt.*

Wir sehen also aus dem Beweis, dass für jede reguläre Sprache $\mathcal{L}$ jeder deterministische Auto-

mat $\mathfrak{A}$ mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$ auf Σ^* eine Äquivalenzrelation $\sim$ induziert, die eine Verfeinerung der durch $x \sim_{\mathcal{L}} y :\Leftrightarrow (\forall z \in \Sigma^*(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}))$ auf Σ^* definierten Äquivalenzrelation $\sim_{\mathcal{L}}$ darstellt, siehe Abbildung 2.2. Insbesondere ist also die Zahl der Zustände eines deterministischen, die Sprache $\mathcal{L}$ akzeptierenden Automaten $\mathfrak{A}$ mindestens ebenso groß wie die Anzahl der Äquivalenzklassen, in die Σ^* durch $\sim_{\mathcal{L}}$ unterteilt wird. Ein gegebener deterministischer Automat $\mathfrak{A}$ mit $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$ besitzt also genau dann eine minimale Anzahl von Zuständen, wenn die durch ihn induzierte Äquivalenzrelation $\sim$ mit $\sim_{\mathcal{L}}$ übereinstimmt.

Wir betrachten nun einen deterministischen Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, \{q_0\}, F)$. Auf der Menge Q seiner Zustände definieren wir durch $p \equiv q :\Leftrightarrow (\forall x \in \Sigma^*(\delta(p,x) \in F \Leftrightarrow \delta(q,x) \in F))$ eine Äquivalenzrelation $\equiv$. Die Äquivalenzklasse eines Zustandes q bezeichnen wir mit $[q]$. Betrachten wir nun den Automaten $\mathfrak{A}' = (Q', \Sigma, \Delta', I', F')$, wobei $Q' = \{[q] \mid q \in Q\}$, $\Delta' = \{\langle [p], z, [\delta_{\mathfrak{A}}(p,z)]\rangle \mid p \in Q, z \in \Sigma\}$, $I' = \{[q_0]\}$ und $F' = \{[x] \mid x \in F\}$ sei. Zeigen wir zunächst, dass Δ' wohldefiniert ist. Seien dazu zwei Zustände $p, q \in Q$ mit $[p] = [q]$, also $p \equiv q$ vorgelegt. Ferner sei $z \in \Sigma^*$ beliebig. Wir betrachten die Zustände $p' := \delta_{\mathfrak{A}}(p,z)$ und $q' := \delta_{\mathfrak{A}}(q,z)$. Dann gilt für alle $w \in \Sigma^*$, dass

$$\delta_{\mathfrak{A}}(p', w) = \delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(p,z), w) = \delta_{\mathfrak{A}}(p, zw) \in F$$
$$\overset{p \equiv q}{\Leftrightarrow} F \ni \delta_{\mathfrak{A}}(q, zw) = \delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(q,z), w) = \delta_{\mathfrak{A}}(q', w).$$

Somit haben wir auch $p' \equiv q'$, das heißt

$$\begin{aligned}\delta_{\mathfrak{A}'}([p], z) &= [\delta_{\mathfrak{A}}(p,z)] = [p'] \\ &= [q'] = [\delta_{\mathfrak{A}}(q,z)] = \delta_{\mathfrak{A}'}([q], z)\ .\end{aligned}$$

Es ist relativ einsichtig, dass der Automat $\mathfrak{A}'$ die gleiche Sprache $\mathcal{L}$ wie der gegebene Automat $\mathfrak{A}$ akzeptiert, denn für alle $z \in \Sigma^*$ gilt

$$\begin{aligned}z \in \mathcal{L}(\mathfrak{A}') &\Leftrightarrow \delta_{\mathfrak{A}'}(q_0', z) \in F' \\ &\Leftrightarrow [\delta_{\mathfrak{A}}(q_0, z)] \in F' \\ &\Leftrightarrow \exists q \in F(\delta_{\mathfrak{A}}(q_0, z) \equiv q) \\ &\Leftrightarrow \delta_{\mathfrak{A}}(q_0, z) \in F \Leftrightarrow z \in \mathcal{L}(\mathfrak{A})\ .\end{aligned}$$

Zeigen wir nun, dass $\mathfrak{A}'$ unter allen deterministischen Automaten mit der Sprache $\mathcal{L} = \mathcal{L}(\mathfrak{A})$ eine minimale Anzahl von Zuständen besitzt. Dazu müssen wir lediglich nachweisen, dass je zwei bezüglich der durch die Sprache $\mathcal{L}$ induzierten Äquivalenzrelation $\sim_{\mathcal{L}}$ äquivalenten Wörter $x, y \in \Sigma^*$ auch bezüglich der durch den Automaten $\mathfrak{A}'$ (wie oben) induzierten Äquivalenzrelation $\sim$ äquivalent sind. Seien also $x, y \in \Sigma^*$ vorgelegt mit $x \sim_{\mathcal{L}} y$. Dann gilt nach Definition von $\sim_{\mathcal{L}}$ für alle $z \in \Sigma^*$: $(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L})$. Es folgt

$$\begin{aligned}\forall z \in \Sigma^*(\delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(q_0, x), z) = \delta_{\mathfrak{A}}(q_0, xz) \in F \\ \Leftrightarrow F \ni \delta_{\mathfrak{A}}(q_0, yz) = \delta_{\mathfrak{A}}(\delta_{\mathfrak{A}}(q_0, y), z)),\end{aligned}$$

also $\delta_{\mathfrak{A}}(q_0, x) \equiv \delta_{\mathfrak{A}}(q_0, y)$, das heißt

$$\delta_{\mathfrak{A}'}(q_0', x) = [\delta_{\mathfrak{A}}(q_0, x)] = [\delta_{\mathfrak{A}}(q_0, y)] = \delta_{\mathfrak{A}'}(q_0', y),$$

was zu $x \sim y$ äquivalent ist.

Schließlich muss noch nachgewiesen werden, dass der beschriebene Automat $\mathfrak{A}$ *konstruiert* werden kann. Wir müssen also einen Algorithmus angeben, der für gegebene Zustände $p, q \in Q$ entscheidet, ob $p \equiv q$ oder nicht. Wir können ähnlich vorgehen wie bei der Aufgabe, für einen gegebenen Automaten herauszufinden, ob dessen Sprache leer ist. Wir müssen lediglich für alle Wörter $z \in \Sigma^*$, die aus maximal $|Q|^2$ Zeichen bestehen, die Äquivalenz $\delta_{\mathfrak{A}}(p,z) \in F \Leftrightarrow \delta_{\mathfrak{A}}(q,z) \in F$ nachprüfen, da nur für solche Wörter $z = z_1 \dots z_n$ die Äquivalenz überprüft werden muss, für die die Paare $\langle \delta_{\mathfrak{A}}(p, z_1 \dots z_i), \delta_{\mathfrak{A}}(q, z_1 \dots z_i)\rangle$ für $i = 1, \dots, n$ paarweise verschieden sind, was für $n > |Q|^2$ nicht mehr der Fall sein kann, da es nur $|Q|^2$ viele Paare von Zuständen gibt. Insgesamt muss die Äquivalenz $\delta_{\mathfrak{A}}(p,z) \in F \Leftrightarrow \delta_{\mathfrak{A}}(q,z) \in F$ also nur für endlich viele Wörter z überprüft werden; der oben beschriebene Minimalautomat $\mathfrak{A}'$ ist damit konstruierbar.

4.10 **Nichtregularität**

Der zu Beginn des vorigen Abschnitttes vorgestellte Satz ist noch aus einem weiteren Grund von besonderem Interesse für die Theorie regulärer Sprachen: Wir können mithilfe der Implikation (1) ⇒ (3) nachweisen, dass gewisse Sprachen nicht regulär sind. Diese Methode wollen wir im Folgenden anhand zweier Beispiele demonstrieren.

Beispiel 22 Die Sprache $\mathcal{L} = \{a^n b^n \mid n \in \mathbb{N}\}$ ist nicht regulär.

Beweis Wäre $\mathcal{L}$ regulär, so dürfte die auf Σ^* durch $x \sim_{\mathcal{L}} y :\Leftrightarrow (\forall z \in \Sigma^*(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}))$ definierte Äquivalenzrelation die Menge Σ^* in nur endlich viele Äquivalenzklassen zerlegen. Wir behaupten aber, dass die Äquivalenzklassen $[a^n b]$ für $n \in \mathbb{N}$ paarweise verschieden sind. Zum Beweis beachten man, dass für alle $i < j$ zwar $a^i b b^{j-1} = a^i b^j \notin \mathcal{L}$, aber $a^j b b^{j-1} = a^j b^j \in \mathcal{L}$ gilt, das heißt, nach der Definition von $\sim_{\mathcal{L}}$ die Wörter $a^i b$ und $a^j b$ nicht äquivalent sind, also $[a^i b] \neq [a^j b]$ gilt. Somit ist $\mathcal{L}$ tatsächlich keine reguläre Sprache. □

Beispiel 23 Die Sprache $\mathcal{L} = \{a^{n^2} \mid n \in \mathbb{N}\}$ ist nicht regulär.

Beweis Wäre $\mathcal{L}$ regulär, so dürfte die auf Σ^* durch $x \sim_{\mathcal{L}} y :\Leftrightarrow (\forall z \in \Sigma^*(xz \in \mathcal{L} \Leftrightarrow yz \in \mathcal{L}))$ definierte Äquivalenzrelation die Menge Σ^* in nur endlich viele Äquivalenzklassen unterteilen. Wir behaupten aber, dass die Äquivalenzklassen

$[a^{n(n-1)}]$ für $n \in \mathbb{N}$ paarweise verschieden sind. Zum Beweis beachte man, dass für alle $i < j$ zwar $a^{i(i-1)}a^i = a^{i^2} \in \mathcal{L}$, aber $a^{j(j-1)}a^i = a^{j^2-j+i} \notin \mathcal{L}$, denn wegen $(j-1)^2 = j^2-2j+1 < j^2-j+i < j^2$ ist j^2-j+i keine Quadratzahl. Folglich sind für alle $i < j$ nach Definition von $\sim_{\mathcal{L}}$ die Wörter $a^{i(i-1)}$ und $a^{j(j-1)}$ nicht äquivalent, das heißt, es gilt tatsächlich $[a^{i(i-1)}] \neq [a^{j(j-1)}]$. Somit ist $\mathcal{L}$ in der Tat keine reguläre Sprache. □

4.11 Addierautomat

Ein Addierautomat soll überprüfen, ob zwei eingegebene Zahlen in Binärdarstellung zusammen addiert die dritte eingegebene Zahl ergeben. Dafür wird in den Automaten immer ein Tripel $\langle x, y, z\rangle \in \{0,1\}^3$ als Eingabezeichen gegeben. Wenn die Tripel hintereinander gelesen werden, ergibt die erste Spalte den ersten und die zweite Spalte den zweiten Summanden in little-endian Binärdarstellung. Die dritte Spalte stellt die Lösung dar, die überprüft werden soll.

Der Automat $\mathfrak{A}$ ist definiert durch die Zustandsmenge $Q = \{q_0, q_1\}$, die Startzustandsmenge $I = \{q_0\}$ und die Endzustandsmenge $F = \{q_0\}$. Die Zustände geben an, ob sich der Automat einen Übertrag merken muss oder nicht. Bei q_0 merkt er sich nichts, bei q_1 den Übertrag 1. Der Automat $\mathfrak{A}$ hat die Transitionsrelation

$$\begin{aligned}\Delta = \{&\langle q_0, \langle 0,0,0\rangle, q_0\rangle, \langle q_0, \langle 0,1,1\rangle, q_0\rangle,\\ &\langle q_0, \langle 1,0,1\rangle, q_0\rangle, \langle q_0, \langle 1,1,0\rangle, q_1\rangle,\\ &\langle q_1, \langle 0,0,1\rangle, q_0\rangle, \langle q_1, \langle 0,1,0\rangle, q_1\rangle,\\ &\langle q_1, \langle 1,0,0\rangle, q_1\rangle, \langle q_1, \langle 1,1,1\rangle, q_1\rangle\}\end{aligned}$$

und das Eingabealphabet $\Sigma = \{0,1\}^3$. Daraus ergibt sich als Sprache des Automaten $\mathcal{L}(\mathfrak{A}) = \{\langle x, y, z\rangle \mid x + y = z\}$. Dieser Automat ist auch in Abbildung 2.3 dargestellt.

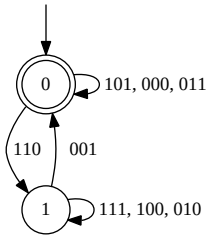


Abb. 2.3: Automat für die Addition

5 Transducer

5.1 Definitionen

Als Transducer bezeichnet man die Automaten, die in irgendeiner Form auf eine Abfolge von Aktionen mit einer Ausgabe reagieren. Sie besitzten also neben dem Eingabealphabet Σ auch ein Ausgabealphabet O. Die Ausgabe kann sich zum Beispiel um ein Zeichen handeln, das ausgegeben wird, wenn eine bestimmte Abfolge von Aktionen vorhergegangen ist. Die Übergangsrelation wird dementsprechend angepasst zu $\Delta \subseteq Q \times \Sigma \times O \times Q$. Über die Zustände im Inneren des Automaten erhält der Benutzer keine Informationen. Diese Eigenschaft der Ausgabe wird nun anhand von zwei Beispielen verdeutlicht. Wichtig ist hier zu erkennen, dass auch Transducer den grundlegenden Aufbau eines endlichen Automatens besitzen.

5.2 Rechtsshift

Die Aufgabenstellung bestand aus der Erstellung eines Automaten, dessen Ausgabe die erfolgte Eingabe ist, der eine Null vorangestellt wird. Es wird also nicht das aktuelle Eingabesymbol ausgegeben, sondern das vorhergegangene. Die Reihe der Ausgaben wird also sozusagen um eine Stelle nach rechts verschoben. Wird also beispielsweise die Folge 0001101001 … eingegeben, soll die Folge 00001101001 … ausgegeben werden.

Eine mögliche Lösung beschreibt der Automat $\mathfrak{A}$ mit

$$\begin{aligned}\mathfrak{A} = (&\{0,1\}, \{0,1\}, \{0,1\}\\ &\{\langle 0,0,0,0\rangle\langle 0,1,0,1\rangle\langle 1,0,1,0\rangle\langle 1,1,1,1\rangle\},\\ &\{0\})\,.\end{aligned}$$

Der Automat hat also zwei Zustände: den Zustand 0, den er erreicht, nachdem die Eingabe 0 erfolgt ist, und den Zustand 1, der durch die Eingabe 1 erreicht wird. Bei einer Eingabe liefert der Automat immer die dem aktuellen Zustand entsprechende Ausgabe. Befindet er sich also im Zustand 1, wird die 1 ausgegeben, egal welche Eingabe erfolgt. Die erste Ausgabe wird durch den Anfangszustand festgelegt. Sie wurde durch die Aufgabenstellung auf die Ausgabe 0 gesetzt. Der Graph des Automaten $\mathfrak{A}$ sieht also wie in Abbildung 2.4 aus.

5.3 Linksshift

Die Aufgabenstellung war es zu zeigen, dass man keinen deterministischen Automaten konstruieren kann, der als Ausgabe das darauf folgende Eingabesymbol liefert, wenn $|\Sigma| > 1$ ist. Die Reihe der Ausgabe soll also analog zum Rechtsshift um eine Stelle nach links verschoben werden.

Angenommen es gäbe einen deterministischen

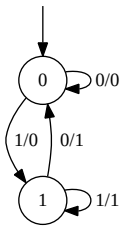


Abb. 2.4: Ein Transducer für den Shift der Eingabe um ein Zeichen nach rechts.

Automaten

$$\mathfrak{A} = (Q, \{0,1\}, \{0,1\}, \Delta, I),$$

der die Aufgabenstellung erfüllt. Seien w_1 und w_2 zwei Wörter aus Σ^ω mit $w_1 = 00\ldots$ und $w_2 = 01\ldots$. Da $\mathfrak{A}$ deterministisch ist, reagiert er auf das erste Symbol der Eingabe von w_1 und w_2 mit der gleichen Ausgabe. Diese Ausgabe soll aber die nun folgende Eingabe beschreiben. Hierin unterscheiden sich jedoch die beiden Wörter. Der Automat reagiert also auf mindestens ein Wort falsch.

Aus diesem Grund betrachtet man oft Transducer, die statt einem Zeichen auch ein Wort ausgeben können. Solche Automaten mit $\Delta \subset Q \times \Sigma \times O^* \times Q$ können auch den Linksshift berechnen, wie man sich leicht überlegt.

6 Büchi-Automat

6.1 Definitionen

Definition 24 Ein *Büchi-Automat* ist ein Automat $\mathfrak{A}$. Seine *Sprache* $\mathcal{L}^\omega(\mathfrak{A}) \subset \Sigma^\omega$ ist definiert als $\mathcal{L}^\omega(\mathfrak{A}) = \{(a_n)_{n\in\mathbb{N}} \mid \exists (q_n)_{n\in\mathbb{N}} (q_0 \in I \wedge (\forall i. \langle q_i, a_i, q_{i+1}\rangle \in \Delta) \wedge \forall k \exists \ell (k < \ell \wedge q_\ell \in F))\}$, wobei Q, I, F und Δ die Zustandsmenge, die Menge der Anfangszustände, die Menge der guten Zustände, die in diesem Zusammenhang oft auch *Fortschrittszustände* genannt werden, beziehungsweise die Transitionsrelation von $\mathfrak{A}$ sind. Eine Sprache $\mathcal{L} \subset \Sigma^\omega$ heißt *ω-regulär*, oder kurz *regulär*, falls es einen Büchi-Automaten $\mathfrak{A}$ gibt, für den $\mathcal{L}(\mathfrak{A}) = \mathcal{L}$ gilt.

Die Sprache $\mathcal{L}^\omega(\mathfrak{A})$ eines Büchi-Automaten $\mathfrak{A}$ besteht also aus allen Wörtern $x = (x_n)_{n\in\mathbb{N}}$, die es dem Automaten ermöglichen, sich, ausgehend von einem Zustand $q_0 \in I$, gemäß Δ so durch seine Zustände zu bewegen, dass er sich unendlich oft in einem Zustand aus F befindet.

Auch für ω-reguläre Sprachen werden wir im folgenden zunächst einige Abschlusseigenschaften untersuchen. Schließlich werden wir zeigen, dass im Gegensatz zu Automaten für endliche Wörter nicht jeder Büchi-Automat determinisiert werden kann.

6.2 Die Schnittmenge zweier ω-regulärer Sprachen

Satz 25 Sind $\mathcal{L}_1 \subset \Sigma_1^\omega$ und $\mathcal{L}_2 \subset \Sigma_2^\omega$ reguläre Sprachen, so ist auch die Schnittmenge $\mathcal{L}_1 \cap \mathcal{L}_2$ regulär.

Beweis Seien $\mathfrak{A}_1 = (Q_1, \Sigma_1, \Delta_1, I_1, F_1)$ und $\mathfrak{A}_2 = (Q_2, \Sigma_2, \Delta_2, I_2, F_2)$ zwei endliche Automaten mit $\mathcal{L}_1 = \mathcal{L}^\omega(\mathfrak{A}_1)$ und $\mathcal{L}_2 = \mathcal{L}^\omega(\mathfrak{A}_2)$, so hat der Automat mit der Schnittmenge der beiden Sprachen $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ folgende Eigenschaften.

- $Q = Q_1 \times Q_2 \times C$ mit $C = \{0,1\}$
- $\Sigma = \Sigma_1 \cap \Sigma_2$
- die Transitionsrelation

$$\begin{aligned}\Delta = &\{\langle\langle q_1, q_2, c\rangle, a, \langle q_1', q_2', c\rangle\rangle \mid \\ &\qquad \langle q_1, a, q_1'\rangle \in \Delta_1 \wedge \langle q_2, a, q_2'\rangle \in \Delta_2 \\ &\qquad \wedge ((q_1 \notin F_1) \wedge (q_2 \notin F_2))\} \qquad \cup \\ &\{\langle\langle q_1, q_2, c\rangle, a, \langle q_1', q_2', 1\rangle\rangle \mid \\ &\qquad \langle q_1, a, q_1'\rangle \in \Delta_1 \wedge \langle q_2, a, q_2'\rangle \in \Delta_2 \\ &\qquad \wedge (q_1 \in F_1)\} \qquad \cup \\ &\{\langle\langle q_1, q_2, c\rangle, a, \langle q_1, q_2', 0\rangle\rangle \mid \\ &\qquad \langle q_1, a, q_1'\rangle \in \Delta_1 \wedge \langle q_2, a, q_2'\rangle \in \Delta_2 \\ &\qquad \wedge (q_2 \in F_2)\} \\ &\subseteq Q \times \Sigma \times Q\end{aligned}$$

- $I = I_1 \times I_2 \times \{0\}$
- $F = F_1 \times Q_2 \times \{0\}$

Die Tatsache, die den Schnitt zweier Sprachen von Büchi-Automaten gegenüber dem zweier Automaten für endliche Wörter unterscheidet, ist, dass die Fortschrittszustände in *beiden* Automaten unendlich oft erreicht werden müssen. Dies muss aber nicht unbedingt zur gleichen Zeit geschehen. Hierzu wird dem kartesischen Produkt der Zustandsmengen die Menge C hinzugefügt. Zu Beginn befindet sich der Automat im Zustand $\langle q_1, q_2, 0\rangle$. Wird im gegebenen ersten Automat ein Fortschrittszustand erreicht, wechselt der Automat mit der darauf folgenden Aktion in den Zustand $\langle q_1, q_2, 1\rangle$. Diese Zustände, die dem Zustand $\langle q_1, q_2, 1\rangle$ vorausgehen, werden als die Fortschrittszustände des neuen Automaten definiert. Erreicht nun der zweite Automat einen Fortschrittszustand, so wechselt der Automat wiederherum mit der darauf folgenden Aktion in den Zustand $\langle q_1, q_2, 0\rangle$. Diese beiden Schritte können auch gleichzeitig auftreten. Soll ein Fortschrittszustand unendlich oft erreicht werden, so muss der

Automat diesen Zyklus unendlich oft durchlaufen, die gegebenen Automaten also unendlich oft einen Fortschrittszustand erreichen. Dies war das vorgegebene Ziel. □

6.3 Die Vereinigung zweier ω-regulärer Sprachen

Satz 26 Seien $\mathcal{L}_1 \subset \Sigma_1^\omega$ und $\mathcal{L}_2 \subset \Sigma^\omega$ reguläre Sprachen. Dann ist auch $\mathcal{L}_1 \cup \mathcal{L}_2$ regulär.

Beweis Seien $\mathfrak{A}_1 = (Q_1, \Sigma_1, \Delta_1, I_1, F_1)$ und $\mathfrak{A}_2 = (Q_2, \Sigma_2, \Delta_2, I_2, F_2)$ zwei endliche Automaten mit $\mathcal{L}^\omega(\mathfrak{A}_1) = \mathcal{L}_1$ und $\mathcal{L}^\omega(\mathfrak{A}_2) = \mathcal{L}_2$. Sei $\mathfrak{A} = (Q, \Sigma_1 \cup \Sigma_2, \Delta, I, F)$ definiert durch
- die Menge der Zustände $Q = Q_1 + Q_2$,
- das Alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$,
- die Transitionsrelation

$$\begin{aligned}\Delta = \quad & \{\langle\langle 0, q\rangle, x, \langle 0, q'\rangle\rangle \mid \langle q, x, q'\rangle \in \Delta_1\} \\ & \cup \{\langle\langle 1, q\rangle, x, \langle 1, q'\rangle\rangle \mid \langle q, x, q'\rangle \in \Delta_2\} \\ & \subset Q \times \Sigma \times Q\ ,\end{aligned}$$

- die Anfangszustände $I = I_1 + I_2$ und
- die Endzustände $F = F_1 + F_2$.

Es ist zu zeigen, dass $\mathcal{L}^\omega(\mathfrak{A}) = \mathcal{L}^\omega(\mathfrak{A}_1) \cup \mathcal{L}^\omega(\mathfrak{A}_2)$ gilt. Sei $x = (x_n)_{n\in\mathbb{N}} \in \mathcal{L}^\omega(\mathfrak{A})$ beliebig. Dann gibt es eine Folge $(\langle d_n, q_n\rangle)_{n\in\mathbb{N}}$ mit $\langle d_0, q_0\rangle \in I$ und $\langle\langle d_n, q_n\rangle, x_n, \langle d_{n+1}, q_{n+1}\rangle\rangle \in \Delta$ für alle $n \in \mathbb{N}$. Mit der Definition von Δ folgt $d_n = 0$ für alle $n \in \mathbb{N}$ oder $d_n = 1$ für alle $n \in \mathbb{N}$ und damit $q := (q_n)_{n\in\mathbb{N}} \in Q_1^\omega$ oder $q \in Q_2^\omega$, womit q aufgrund der Definition von Δ eine zulässige Folge von Zuständen in Q_1 oder in Q_2 für die Eingabe x darstellt, woraus $x \in \mathcal{L}^\omega(\mathfrak{A}_1)$ oder $x \in \mathcal{L}^\omega(\mathfrak{A}_2)$, das heißt $x \in \mathcal{L}^\omega(\mathfrak{A}_1) \cup \mathcal{L}^\omega(\mathfrak{A}_2)$, folgt. Analog kann man umgekehrt für $x \in \mathcal{L}^\omega(\mathfrak{A}_1) \cup \mathcal{L}^\omega(\mathfrak{A}_2)$ argumentieren. Somit gilt $\mathcal{L}^\omega(\mathfrak{A}) = \mathcal{L}^\omega(\mathfrak{A}_1) \cup \mathcal{L}^\omega(\mathfrak{A}_2)$. □

6.4 Das Shuffle-Produkt zweier ω-regulärer Sprachen

Das (faire) Shufflen (»Mischen«) zweier Sprachen $L_1 \subset \Sigma_1^\omega$ und $L_2 \subset \Sigma_2^\omega$ bedeutet, dass neue Wörter mit $w \in (\Sigma_1 + \Sigma_2)^\omega$ entstehen. Diese sind demnach zusammengesetzt aus einzelnen Elementen aus Σ_1 und Σ_2. Die einzelnen Aktionen werden hierbei nur in den entsprechenden Automaten ausgeführt. Um das Shuffle zu verstehen, muss zuerst die Projektion betrachtet werden. Projektion bedeutet, dass die Komponenten der neuen Wörter so aufgeteilt werden, dass die Komponenten aus Σ_1 von denen aus Σ_2 getrennt werden. So entstehen wieder die Wörter $w_1 \in \Sigma_1^\omega$ und $w_2 \in \Sigma_2^\omega$.

Definition 27 Für endliche Alphabete Σ_1, Σ_2 und ein Wort $w \in (\Sigma_1 + \Sigma_2)^\omega$, $w = (\langle d_n, x_n\rangle)_{n\in\mathbb{N}}$, sei $(n_i)_i$ die aufsteigende Folge derjenigen Indices n, für die $d_n = 0$ gilt. Das Wort $\hat{w} = (x_{n_i})_i$ nennen wir dann *Projektion* von w auf Σ_1 und schreiben $\pi_{\Sigma_1}(w) = \hat{w}$. Analog definieren wie die Projektion auf Σ_2.

Mit Hilfe der Projektion kann nun auch das Shuffle definiert werden.

Definition 28 Für endliche Alphabete Σ_1, Σ_2 und Sprachen $\mathcal{L}_1 \subset \Sigma_1^\omega$, $\mathcal{L}_2 \subset \Sigma_2^\omega$ bezeichnen wir die Sprache derjenigen Wörter aus $(\Sigma_1 + \Sigma_2)^\omega$, für die die Projektionen auf Σ_1 und Σ_2 in $\mathcal{L}_1$ beziehungsweise $\mathcal{L}_2$ liegen, als *Shuffle* von $\mathcal{L}_1$ und $\mathcal{L}_2$; als abkürzende Schreibweise wählen wir $\mathcal{L}_1||\mathcal{L}_2$.

Satz 29 Sind $\mathcal{L}_1 \subset \Sigma_1^\omega$ und $\mathcal{L}_2 \subset \Sigma_2^\omega$ reguläre Sprachen, so ist auch das Shuffle $\mathcal{L}_1||\mathcal{L}_2$ regulär.

Beweis Seien $\mathfrak{A}_1 = (Q_1, \Sigma_1, \Delta_1, I_1, F_1)$ und $\mathfrak{A}_2 = (Q_2, \Sigma_2, \Delta_2, I_2, F_2)$ zwei endliche Automaten mit $\mathcal{L}_1 = \mathcal{L}^\omega(\mathfrak{A}_1)$ und $\mathcal{L}_2 = \mathcal{L}^\omega(\mathfrak{A}_2)$. Wir betrachten den Automaten $\mathfrak{A} = (Q, \Sigma, \Delta, I, F)$ mit $Q = Q_1 \times Q_2 \times \{0, 1\}$, $\Sigma = \Sigma_1 + \Sigma_2$, der Transitionsrelation $\Delta =$

$$\begin{aligned}&\{\langle\langle q_1, q_2, 0\rangle, \langle 0, a\rangle, \langle q_1', q_2, d\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in \Delta_1 \\ &\quad \wedge ((q_1 \notin F_1 \wedge d = 0) \vee (q_1 \in F_1 \wedge d = 1))\} \\ &\cup \{\langle\langle q_1, q_2, 0\rangle, \langle 1, a\rangle, \langle q_1, q_2', 0\rangle\rangle \mid \langle q_2, a, q_2'\rangle \in \Delta_2\} \\ &\cup \{\langle\langle q_1, q_2, 1\rangle, \langle 0, a\rangle, \langle q_1', q_2, 1\rangle\rangle \mid \langle q_1, a, q_1'\rangle \in \Delta_1\} \\ &\cup \{\langle\langle q_1, q_2, 1\rangle, \langle 1, a\rangle, \langle q_1, q_2', d\rangle\rangle \mid \langle q_2, a, q_2'\rangle \in \Delta_2 \\ &\quad \wedge ((q_2 \notin F_2 \wedge d = 1) \vee (q_2 \in F_2 \wedge d = 0))\} \\ &\subseteq \ Q \times \Sigma \times Q\ ,\end{aligned}$$

$I = I_1 \times I_2 \times \{0\}$ und $F = F_1 \times Q_2 \times \{0\}$. Der Automat $\mathfrak{A}$ agiert also, falls er eine Eingabe aus Σ_1 erhält, nur in der ersten Komponente seines Zustandes $\langle q_1, q_2, d\rangle \in Q_1 \times Q_2 \times \{0, 1\}$, falls er eine Eingabe aus Σ_2 erhält, analog nur in der zweiten Komponente und zwar jeweils gemäß Δ_1 beziehungsweise Δ_2. Für $\mathfrak{A}$ besteht ein Fortschritt darin, in der dritten Komponente des Zustandes von 0 zu 1 zu gelangen, was genau dann unendlich oft passiert, wenn die Projektionen des Eingabewortes w in Σ_1 beziehungsweise Σ_2 in den ursprünglichen Automaten $\mathfrak{A}_1$ und $\mathfrak{A}_2$ unendlich oft einen Fortschrittszustand errreichen würden, was zu $\pi_{\Sigma_1}(w) \in \mathcal{L}_1 \wedge \pi_{\Sigma_2}(w) \in \mathcal{L}_2$ und damit zu $w \in \mathcal{L}_1||\mathcal{L}_2$ äquivalent ist. □

6.5 Leerheitstest

Dieser Abschnitt befasst sich mit der Frage, ob ein Automat $\mathfrak{A}$ nach Büchi eine nicht-leere Sprache $\mathcal{L}(\mathfrak{A}) \subset \Sigma^\omega$ besitzt, das heißt ob es ein Wort $x \in \Sigma^\omega$ gibt, bei dessen Eingabe er unendlich oft durch einen Fortschrittszustand kommen kann.

Dazu ist nach einer erreichbaren Schleife innerhalb des Automaten zu suchen, in der mindestens ein Fortschrittszustand vorkommt. Eine Schleife bedeutet, dass irgendwann sich die Abfolge von Zuständen und Aktionen wiederholt. Danach kann gesucht werden, indem überprüft wird, ob ein erreichter Zustand vorher bereits erreicht wurde. Darüberhinaus muss gelten, dass in der Abfolge der Zustände, die zwischen den beiden gleichen Zuständen sind, ein Fortschrittszustand liegt. Wenn eine solche Schleife gefunden wird, akzeptiert der Automat $\mathfrak{A}$ mindestens ein Wort und damit gilt $\mathcal{L}(\mathfrak{A}) \neq \emptyset$. Gibt es keine solche Schleife, ist $\mathcal{L}(\mathfrak{A}) = \emptyset$, das heißt, die Sprache das Automaten ist leer.

6.6 Unmöglichkeit der Deterministierung

Im Gegensatz zu Automaten für endliche Wörter können Büchi-Automaten im Normalfall nicht deterministiert werden, das heißt, im Allgemeinen findet man zu einem gegebenen Büchi-Automaten $\mathfrak{A}$ keinen deterministischen Büchi-Automaten $\mathfrak{A}'$ mit $\mathcal{L}^\omega(\mathfrak{A}) = \mathcal{L}^\omega(\mathfrak{A}')$. Ein Beispiel eines solchen Büchi-Automaten wollen wir im Folgenden erklären. Sei

$$\begin{aligned}\mathfrak{A} = (&\{A, B\}, \{0, 1\},\\ &\{\langle A, 0, A\rangle, \langle A, 1, A\rangle, \langle A, 0, B\rangle, \langle B, 0, B\rangle\},\\ &\{A\}, \{B\})\ .\end{aligned}$$

Offensichtlich besteht die Sprache des Automaten $\mathfrak{A}$ aus denjenigen 0-1-Wörtern, die nur endlich viele 1en enthalten; in anderen Worten $\mathcal{L}^\omega(\mathfrak{A}) = \{(x_n)_{n\in\mathbb{N}} \mid \exists \ell \in \mathbb{N}(\forall n > \ell(x_n = 0))\}$. Wir behaupten, dass es keinen deterministische Büchi-Automaten mit derselben Sprache $\mathcal{L}$ gibt. Angenommen, wir könnten doch einen Büchi-Automaten $\mathfrak{A}' = (Q', \{0, 1\}, \Delta', \{q_0\}, F')$ mit $\mathcal{L}^\omega(\mathfrak{A}') = \mathcal{L}^\omega(\mathfrak{A})$ finden. Da der deterministische Automat $\mathfrak{A}'$ sich bei Eingabe des Wortes 0^ω irgendwann in einem Zustand aus F befindet, gibt es eine natürliche Zahl n_1, so dass $\delta_{\mathfrak{A}'}(q_0, 0^{n_1}) =: f_1 \in F$ gilt. Betrachten wir das Wort $0^{n_1}10^\omega$ und beachten, dass $\mathfrak{A}'$ deterministisch ist, finden wir also eine natürliche Zahl n_2 mit $\delta_{\mathfrak{A}'}(q_0, 0^{n_1}10^{n_2}) =: f_2 \in F$. Allgemein zeigt man auf diese Weise, dass eine Folge $(n_i)_{i\in\mathbb{N}}$ natürlicher Zahlen existiert, so dass $\delta_{\mathfrak{A}'}(q_0, 0^{n_1}10^{n_2}1\cdots10^{n_k}) =: f_k$ in F liegt für alle $k = 1, 2, 3, \ldots$. Angenommen, es gäbe Indices $i < j$ mit $f_i = f_j$. Dann würde das Wort $w = 0^{n_1}1\cdots10^{n_i}(1\cdots10^{n_j})^\omega$ in $\mathcal{L}^\omega(\mathfrak{A}')$ liegen, da der Automat $\mathfrak{A}'$ sich bei Eingabe dieses Wortes unendlich oft im Fortschrittszustand f_j befinden würde. Da das Wort w jedoch unendlich viele 1en enthält, widerspräche dies der Annahme $\mathcal{L}^\omega(\mathfrak{A}') = \mathcal{L}^\omega(\mathfrak{A})$. Also müssen die Zustände $f_1, f_2, f_3, \ldots$ paarweise verschieden sein. Dies ist jedoch nicht möglich, da F als Teilmenge von Q endlich ist. Also war unsere Annahme falsch und es existiert tatsächlich kein deterministischer Büchiautomat $\mathfrak{A}'$, der die Sprache derjenigen Wörter aus $\{0, 1\}^\omega$ erzeugt, die nur endlich viele 1en enthalten.

7 Presburger Arithmetik

In diesem Abschnitt wird es von Nutzen sein, jede natürliche Zahl c mit der Menge derjenigen Wörter $x = x_0 \ldots x_n \in (\{0, 1\})^*$ zu identifizieren, die eine little-endian Darstellung von c sind, das heißt solchen Wörtern, für die $\sum_{i=0}^{n} x_i \cdot 2^i = c$ gilt. Natürliche Zahlen werden also binär dargestellt, im Unterschied zur konventionellen Binärdarstellung hat die erste Ziffer jedoch den Wert 1, die zweite den Wert 2, die dritte den Wert 4, und so weiter. Weiter darf eine beliebige, endliche Zahl von Nullen angefügt werden, ohne dass sich die Bedeutung des Binärwortes ändert. Entsprechend fassen wir jede Menge M von k-Tupeln natürlicher Zahlen als Teilmenge von $\left(\{0, 1\}^k\right)^*$ auf.

Definition 30 *(Syntax der Presburger Arithmetik)* Mit Var bezeichen wir eine Menge von Variablen. Variablen denotieren wir mit $x, y, z, \ldots$. Die Menge der Formeln der Presburger Arithmetik ist (induktiv) wie folgt definiert.

- $(x = c)$ ist eine Formel für alle $x \in \mathsf{Var}$ und $c \in \mathbb{N}$.
- $(x + y = z)$ ist eine Formel für alle $x, y, z \in \mathsf{Var}$.
- $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$ und $(\neg\varphi)$ sind Formeln für alle Formeln φ und ψ.
- $(\exists x\varphi)$ und $(\forall x\varphi)$ sind Formeln für alle Formeln φ und alle Variablen x.

Klammerpaare können weggelassen werden, falls das entstandene Objekt im oberen Sinne eindeutig zu lesen ist.

Beispiel 31 $(\exists x(\forall y(\exists z((x + y = z) \wedge (y = 7)))))$ ist eine Formel der Presburger Arithmetik.

Beispiel 32 $(\forall x(\exists y(x+2 = y)))$ ist keine Formel der Presburger Arithmetik aber bedeutungsgleich mit der Formel $(\forall x(\exists y(\exists z((x+z = y)\wedge(z = 2)))))$.

Definition 33 *(Semantik der Presburger Arithmetik)* Unter einer *Belegung* verstehen wir eine Abbildung $\eta\colon \mathsf{Var} \to \mathbb{N}$. Die Relation $\eta \models \varphi$ ist induktiv wie folgt definiert.

- $\eta \models x = c$ genau dann, wenn $\eta(x) = c$.
- $\eta \models x+y = z$ genau dann, wenn $\eta(x)+\eta(y) = \eta(z)$.
- $\eta \models \varphi \wedge \psi$ genau dann, wenn $\eta \models \varphi$ und $\eta \models \psi$.

- $\eta \models \varphi \vee \psi$ genau dann, wenn $\eta \models \varphi$ oder $\eta \models \psi$.
- $\eta \models \neg\varphi$ genau dann, wenn $\eta \models \varphi$ nicht gilt.
- $\eta \models \exists x\varphi$ genau dann, wenn es ein $a \in \mathbb{N}$ gibt mit $\eta[a/x] \models \varphi$.
- $\eta \models \forall x\varphi$ genau dann, wenn $\eta[a/x] \models \varphi$ für alle $a \in \mathbb{N}$ gilt.

Dabei überschreibt $\eta[a/x]$ die Belegung η an der Stelle x mit a, das heißt

$$(\eta[a/x])(y) = \begin{cases} \eta(y) & \text{falls } y \neq x \\ a & \text{falls } y = x \end{cases}$$

für alle $y \in \mathsf{Var}$. Im Fall $\eta \models \varphi$ sagen wir, dass η ein Modell der Formel φ ist.

Beispiel 34 Für eine Belegung $\eta\colon \mathsf{Var} \to \mathbb{N}$ gilt $\eta \models (\exists x((x = 0) \wedge (y = 0)))$ genau dann, wenn $\eta(y) = 0$ gilt.

Beispiel 35 Für eine Belegung $\eta\colon \mathsf{Var} \to \mathbb{N}$ gilt $\eta \models (\exists x(x+y = z))$ genau dann, wenn $\eta(y) \leq \eta(z)$ gilt.

Definition 36 Für eine Formel φ der Presburger Arithmetik mit freien Variablen $x_1, \ldots, x_k$ definiert man

$$\begin{aligned}[\![\varphi]\!] =& \{\langle n_1, \ldots, n_k\rangle \mid \eta \models \varphi \text{ wobei} \\ & \eta(x_i) = n_i \text{ für alle } i\} \\ \subset& \left(\{0,1\}^k\right)^*\end{aligned}$$

Beispiel 37 Für $\varphi = \exists x \exists z(x + y = z \ \wedge \ z = 5)$ gilt $[\![\varphi]\!] = \{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle, \langle 3\rangle, \langle 4\rangle, \langle 5\rangle\}$.

Beispiel 38 Für $\varphi = \exists x(x + y = z \ \wedge \ x = 1)$ gilt $[\![\varphi]\!] = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \ldots\}$.

Satz 39 Für jede Formel φ der Presburger Arithmetik gibt es einen Automaten $\mathfrak{A}$ für endliche Wörter mit $\mathcal{L}(\mathfrak{A}) = [\![\varphi]\!]$.

Mehr noch, der Automat $\mathfrak{A}$ kann aus φ algorithmisch konstruiert werden.

Beweis Induktion über φ. Für die Formel $\varphi = (x = c)$ mit $c \in \mathbb{N}$ gilt $[\![\varphi]\!] = \{\langle c\rangle\}$. Man überlegt sich leicht, dass ein Automat konstruiert werden kann, dessen Sprache $\{\langle c\rangle\}$, aufgefasst als Sprache derjenigen Wörter aus $(\{0,1\}^1)^*$, die mit der little-endian Binärdarstellung von c beginnen und beliebig viele weitere Nullen enthalten. Auch für die Formel $\varphi = (x + y = z)$, für die $[\![\varphi]\!] = \{\langle a, b, c\rangle \mid a + b = c\}$ gilt, kann ein Automat $\mathfrak{A}$ mit $\mathcal{L}(\mathfrak{A}) = [\![\varphi]\!]$ konstruiert werden; siehe Addierautomat oben. Ist φ nun eine Formel der Presburger Arithmetik, für die schon bekannt ist, dass ein Automat $\mathfrak{A}$ konstruiert werden kann, dessen Sprache $\mathcal{L}(\mathfrak{A}) = [\![\varphi]\!]$ erfüllt, so betrachten wir den (konstruierbaren) Automaten $\mathfrak{A}'$, dessen Sprache $\mathcal{L}(\mathfrak{A}')$ das Komplement $(\{0,1\}^k)^* \backslash \mathcal{L}(\mathfrak{A})$ ist. Wegen $[\![\neg\varphi]\!] = (\{0,1\}^k)^* \backslash [\![\varphi]\!]$ haben wir damit einen Automaten mit Sprache $[\![\neg\varphi]\!]$ konstruiert. Sind φ und ψ Formeln der Presburger Arithmetik, für die schon bekannt ist, dass Automaten $\mathfrak{A}$ und $\mathfrak{B}$ mit $\mathcal{L}(\mathfrak{A}) = [\![\varphi]\!]$ und $\mathcal{L}(\mathfrak{B}) = [\![\psi]\!]$ existieren, so können wir wegen $[\![\varphi \wedge \psi]\!] = \mathcal{L}(\mathfrak{A}) \cap \mathcal{L}(\mathfrak{B})$ und $[\![\varphi \vee \psi]\!] = \mathcal{L}(\mathfrak{A}) \cup \mathcal{L}(\mathfrak{B})$ auch Automaten $\mathfrak{C}$ beziehungsweise $\mathfrak{D}$ mit $\mathcal{L}(\mathfrak{C}) = [\![\varphi \wedge \psi]\!]$ sowie $\mathcal{L}(\mathfrak{D}) = [\![\varphi \vee \psi]\!]$ konstruieren, wie wir weiter oben (Abgeschlossenheit regulärer Sprachen unter Schnitt und Vereinigung) gesehen haben. Ist schließlich φ eine Formel der Presburger Arithmetik, für die ein Automat $\mathfrak{A}$ mit $\mathcal{L}(\mathfrak{A}) = [\![\varphi]\!]$ existiert, und hat φ die freien Variablen $x_1, \ldots, x_k$, so ist $[\![\exists x_k(\varphi)]\!] \subset (\{0,1\}^{k-1})^*$ gerade die Projektion von $[\![\varphi]\!] \subset (\{0,1\}^k)^*$ auf die ersten $k-1$ Komponenten. Oben haben wir gesehen, dass beliebige Projektionen regulärer Sprachen regulär sind; mehr noch, dass die jeweiligen Automaten *konstruierbar* sind. Also können wir einen Automaten $\mathfrak{A}'$ mit $\mathcal{L}(\mathfrak{A}') = [\![\exists x_k(\varphi)]\!]$ konstruieren. Wegen $[\![\forall x\varphi]\!] = [\![\neg\exists x\neg\varphi]\!]$ für jede Formel φ der Presburger Arithmetik müssen wir uns nicht weiter um den Allquantor $\forall$ kümmern und haben die Behauptung bewiesen. □

Beispiel 40 Auch $x \leq y + c$ mit $c \in \mathbb{N}$ lässt sich in eine Formel der Presburger Arithemtik transformieren: $\exists u \exists w \exists z((x + z = u) \wedge ((w = c) \wedge (y + w = u)))$.

Dieses Beispiel soll im nun Folgenden noch etwas genauer betrachtet werden. Um einen Überblick über die durch die Presburger Arithmetik erzeugten Automaten zu bekommen, sind hier einige der Automaten abgebildet. Als Beispiel wählen wir konkret den Automaten für $x \leq y + 7$. Dies entspricht dem Automaten für $\exists u \exists w \exists z((x + z = u) \wedge ((w = 7) \wedge (y + w = u)))$. Die Automaten für $y + w = u$, $w = 7$ und $(x + z = u) \wedge ((w = 7) \wedge (y + w = u))$ sind in den Abbildungen 2.3, 2.6, beziehungsweise 2.5 zu finden.

Beispiel 41 Auch $y = c \cdot x$ lässt sich in eine Formel der Presburger Arithemtik transformieren, nämlich $\exists x_2 \exists x_3 \ldots \exists x_{c-1}((x_2 = x + x) \wedge (x_3 = x + x_2) \wedge \ldots \wedge (x_{c-1} = x + x_{c-2}) \wedge (y = x + x_{c-1}))$.

8 Implementierung

8.1 Gemeinsame Projektarbeit

Ein großer Teil der gemeinsamen Kurseinheiten des Kurses »Reguläre Sprachen und endliche Automaten« bestand darin, dass die Kursteilnehmer

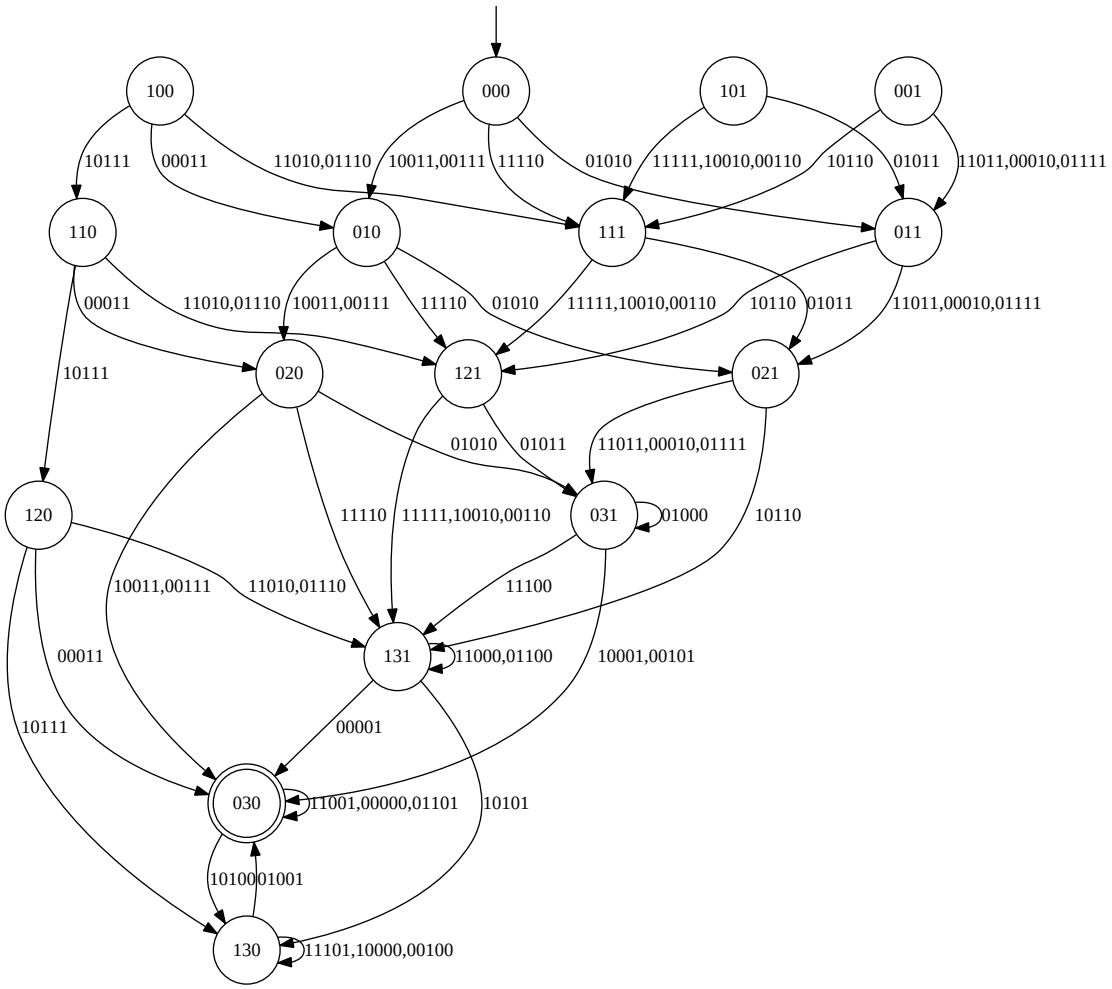


Abb. 2.5: Automat für die Formel $(x + z = u) \wedge ((w = 7) \wedge (y + w = u))$. Dieser Automat ist noch nicht minimiert. Beispielsweise erkennt man dies an den 5 unerreichbaren Zuständen.

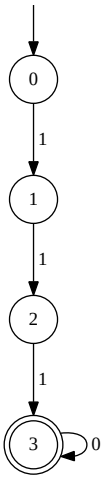


Abb. 2.6: Automat für $w = 7$

gemeinschaftlich an einem großen Projekt arbeiteten – genauer gesagt an zwei großen Projekten, zum einen diese Dokumentation und zum anderen die Implementierung der Presburger Arithmetik in der funktionalen Programmiersprache Haskell. Um überhaupt an einem Projekt arbeiten zu können, mussten die Kursteilnehmer erstmal essentielle Dinge lernen. Dazu gehörte die Einführung in CVS und LaTeX. Es war wichtig, dass die Dateien zentral auf einem Server mit einem Versionskontrollsystem gespeichert und bearbeitet wurden. Außerdem war auch die Arbeit mit vielen, kleinen Modulen wichtig, damit die gleichzeitige Bearbeitung einer Datei vermieden wurde. Das Wichtigste jedoch war, dass man miteinander über das Projekt sprach; sowohl im direkten Austausch als auch am Anfang jeder Kurseinheit mit allen Kursteilnehmern. Für die beiden Projekte gab es jeweils einen Kursteilnehmer, der das Projekt koodinier-

te, da auch dies zum gemeinschaftlichen Arbeiten gehört. Während der Kurseinheiten gab es immer wieder Gruppenarbeiten, die dann regelmäßig in unser Repository eingebunden wurden. So konnte ein Grundstock für die eigentliche Dokumentation angelegt werden. Ebenso gab es eine Einführung in die Programmiersprache Haskell, für die es viele Übungsaufgaben gab. Danach ging es an die wirkliche Implementierung. Für die Dokumentation war es besonders wichtig, sich erst einmal auf verbindliche Notationen zu einigen und eine Ordner- und Dateistruktur anzulegen. Von dieser hing im Prinzip alles ab. Bevor diese erstellt war, herrschte Chaos, da keiner wusste, wo er speichern sollte, was zu etlichen Problemen führte. Auch musste man sich auf Deadlines einigen, die zu sehr interessanten Uhrzeiten waren. Doch wie es Murphys Gesetz sagt, ging erstmal alles schief. Am letzten Tag an dem die Dokumentation geschrieben wurde, fielen alle Rechner und der Server wegen eines Stromausfalles aus, wodurch sich die Deadline von 24 auf 26 Uhr nach hinten verschob.

8.2 Einführung in CVS

CVS (Concurrent Versions System) [1] ist ein System zur Versionsverwaltung von Dateien. Es dient vor allem zum Datenaustausch bei Projekten, an denen mehrere Personen beteiligt sind. Im Allgemeinen ist CVS ein auf dem Server installiertes reines Kommandozeilenprogramm, auf das durch bestimmte Befehle zugegriffen werden kann. Jedoch gibt es ebenso für alle gängigen Systeme eine grafische Oberfläche.

Die Funktionsweise von CVS ist recht simpel. Alle Dateien des Projektes werden an zentraler Stelle, dem sogenannten *Repository*, abgespeichert. Der gesamte Inhalt oder auch nur Teile davon werden vom Benutzer bei der Anmeldung lokal kopiert und können dann auch lokal bearbeitet werden. Diesen Vorgang nennt man *check out*. In den zu den Dateien gehörenden Zusatzdaten wird die Versionshistorie gespeichert. Somit ist für jeden Nutzer der gesamte Verlauf der Änderungen in einer Datei ersichtlich. Es sind alle älteren Versionen der Dateien gespeichert und können bei Bedarf wiederhergestellt werden. Hat der Nutzer eine Datei bearbeitet, so lädt er sie zurück in das Repository. Dieser Vorgang wird als *check in* oder *commit* bezeichnet. Der Vorteil von CVS ist hier, dass die Software Möglichkeiten zur Versionszusammenführung und zum Versionsvergleich bietet. Haben zum Beispiel zwei Nutzer an der gleichen Datei gearbeitet, schlägt CVS eine mögliche Zusammenführung beziehungsweise Differenzierung beider neuer Versionen selbstständig vor. Auch sonst arbeitet das System weitgehend selbstständig. Der gesamte Vorgang, ob über Kommandozeile oder visuell, wird schnell und weitgehend reibungslos ausgeführt.

Da im Kurs sowohl an dem Projekt als auch an der Dokumentation sehr viele Personen gleichzeitig arbeiteten, bot sich CVS als System zur Versionskontrolle an. So musste das Projekt nicht in einzelne Teile partitioniert werden, sondern es konnte gemeinschaftlich gearbeitet werden. Außderdem war es auf diese Weise möglich, plattformunanbhängig auf Windows und Unix zu arbeiten.

8.3 Einführung in Haskell

Haskell [2] ist eine rein funktionale Programmiersprache, die vor allem auf mathematischer Logik aufgebaut ist. Eine funktionale Programmiersprache ist, im Gegensatz zu einer imperativen Sprache, eine Sprache, deren Programmfluss nur auf der Berechnung einzelner Funktionen basiert. Bei imperativen Sprachen führt jede Ausführung eines Befehls zur Änderung des Zustands des Programms. Der Aufruf einer Funktion mit den gleichen Übergabeparametern kann bei einer imperativen Sprache also zu verschiedenen Ergebnissen führen – anders jedoch bei Haskell. Hier ergibt der Aufruf einer Funktion, egal an welcher Stelle des Programms, immer den gleichen Rückgabewert. In einer rein funktionalen Sprache sind Konstrukte zur Ein- und Ausgabe nicht möglich. Deshalb gibt es in Haskell Konstruktoren zur *Beschreibung* des, im allgemeinen unendlichen, Baums der möglichen Ein- und Ausgabeverhalten des Programms.

Die Nähe zur mathematischen Logik zeigt sich unter anderem in der Art der Notation, die der mathematischen Schreibweisen sehr stark ähnelt. So entspricht zum Beispiel der Ausdruck `[ a | a <- list, a > 5]` der Menge, als Liste, aller Elemente `a` in der Menge `list`, die größer als 5 sind.

Auch das Typsystem in Haskell ist sehr gut an die Bedürfnisse der mathematischen Logik angepasst. So lassen sich auf einfachem Wege eigene Datentypen erstellen, sogar wenn diese rekursiv sind. Die Programmiersprache unterstützt auch Typvariablen. Der Typ einer Variablen kann also wie eine Variable selbst benutzt werden. Ein Beispiel ist die `map`-Funktion. Sie wendet eine gegebene Funktion auf alle Listenelemente an. Ihr Typ ist also `map :: (a -> b) -> [a] -> [b]`. Dies bedeutet, dass die Funktion bei ihrem Aufruf im ersten Parameter eine Funktion vom Typ `a` zum Typ `b` erwartet und im zweiten Parameter eine Liste aus Elementen dieses Typen `a`. Für `a` und `b` setzt Haskell automatisch die richtigen Typen ein. Vor

allem der vordefinierte Datentyp Liste, der mathematisch einer geordneten Menge sehr nahe kommt, erleichtert viele Anwendungen.

Funktionen können in Haskell auch ohne Angabe der genutzten Typen geschrieben werden. Haskell »sucht« sich die passende Typendefinition aus der Funktionsdefinition heraus. So erwartet die Funktion `inc x = x + 5` als Eingabeparameter automatisch eine Variable der Klasse Zahl (`Num`), da nur auf solche Variablen der Ausdruck $x + 5$ angewendet werden kann. Eine der vorteilhaftesten Eigenschaften von Haskell ist, dass es Funktionen mit pattern matching unterstützt. Das bedeutet es können mehrere verschiedene Definitionen einer Funktion angegeben werden und Haskell versucht anhand der Übergabeparameter die passende Funktion zu finden. Ein Beispiel dafür ist die im Folgenden definierte Funktion `foo`.

```
> foo (1,y) = 1
> foo (x,y) = y
```

Wird die Funktion mit dem Paar $(1, y)$ mit y eine beliebige Variable aufgerufen, so wird 1 zurückgegeben. Wird ein anderes Paar übergeben, so wird die letzte Komponente des Paars angegeben. Haskell entscheidet also aufgrund der eingegebenen Parameter welche Funktion ausgeführt wird.

Eine weitere nützliche Eigenschaft von der Sprache Haskell ist seine *non-strictness* (auch *lazyness* genannt). Haskell berechnet nur genau die Ausdrücke, die es auch wirklich benötigt und arbeitet außerdem bevorzugt mit Zeigern als mit den Variablen selbst. Die Konstruktion `[0 ..]` erstellt die Liste alle natürlichen Zahlen angefangen bei 0. Solch eine Konstruktion wäre in einer imperativen Sprache nie möglich gewesen. Haskell jedoch wertet den Ausdruck erst gar nicht aus. Selbst wenn diese Liste als Funktionsübergabeparameter benutzt wird, wird nur der Teil ausgewertet, der gebraucht wird. So liefert `take 10 [0 ..]` die ersten 10 Elemente der Liste, ohne dass der Rest der Liste berechnet werden muss. Sogar eine Konstruktion nach dem folgenden Prinzip `take 2 [0,0, error "HALT"]` führt zu keiner Fehlermeldung, da Haskell den Ausdruck `error "HALT"` erst gar nicht auswertet.

Die Möglichkeit mit unendlichen Listen umzugehen, das pattern matching und nicht zuletzt die selbst erstellbaren Datentypen mit dazugehörigen Konstruktor machen die Implementierung endlicher Automaten mit Haskell angenehm. Die Nähe zur mathematischen Logik erleichtert den Umgang mit Automaten ebenfalls sehr.

Die Funktionsweise von Haskell soll nun an einer Beispielfunktion veranschaulicht werden, die das Sieb des Erathostenes darstellt. Das ist eine Funktion, die von einer Liste das erste Element behält, alle Vielfachen davon aus der Liste herausfiltert und sich daraufhin auf die überbleibenden Elemente neu anwendet. Hierzu benötigen wir zunächst die Hilfsfunktion `divides`, die prüft, ob eine Zahl durch eine andere teilbar ist. Sie sieht wie folgt aus.

```
> divides a b =  (mod a b == 0)
```

Die Funktion hat zwei Parameter. a ist der Dividend, b stellt den Divisor dar. `mod` steht für Modulo und errechnet somit den Rest der Division. Das doppelte Gleichheitszeichen vergleicht zwei Variablen. Wenn die Gleichheit zutrifft, a also durch b teilbar ist, wird `True` zurückgegeben, anderenfalls `False`. Diese Funktion wird nun für eine weitere Hilfsfunktion `rmmults` gebraucht, die aus einer Liste alle Vielfachen einer Zahl herausfiltert.

```
> rmmults a (b:bs) =
>     if (divides b a)
>         then (rmmults a bs)
>         else (b:rmmults a bs)
```

Die Parameter der Funktion sind eine Variable a, die hier den Divisor darstellt, also die Zahl, deren Vielfache aus der Liste herausgefiltert werden sollen, und die Liste, aus der die Elemente gefiltert werden, bestehend aus dem ersten Element b, auch *head* genannt, und den restlichen Elementen bs, die auch *tail* genannt werden. Die Funktion besteht aus einer if-then-else-Konstruktion. Die Bedingung ist, dass b durch a teilbar ist. Ist dies der Fall, so wendet sich die Funktion auf den Rest der Liste von neuem an ohne b weiter zu beachten. Wenn die Bedingung nicht zutrifft, wird b ausgegeben und das Ergebnis der Funktion, angewendet auf den Rest der Liste, angehängt. Diese Funktion wird nun für die Hauptfunktion `sieve`, also das Sieb des Erathostenes, gebraucht.

```
> sieve (a:as) = a: sieve (rmmults a as)
```

Die Funktion wird auf eine Liste `a:as` angewendet. Das erste Element wird ausgegeben, aus dem Rest der Liste werden die Vielfachen des ersten Elements herausgefiltert und auf die übrig gebliebene Liste wird die Funktion `sieve` erneut angewandt. Schließlich liefert

```
> primes = sieve [2..]
```

die unendliche Liste aller Primzahlen.

8.4 Projektüberblick

Das Projekt beinhaltete verschiedene Module, die von verschiedenen Personen implementiert wurden, um später zusammengefügt zu werden. Zu den genannten Beispielen kamen noch weitere Module. Zunächst wird in dem Modul `Automaton` der

Begriff Automat einheitlich definiert. Das Modul `Addition` erstellt einen Automaten, der der Gleichung $x + y = z$ entspricht. Mit der `Vereinigung` werden zwei Automaten zu einem zusammengefasst, der die Sprache beider Automaten akzeptiert. Für den komplementären Automaten wurde das Modul `Komplement` verwendet. Dieses kreiert einen Automaten, der die komplementäre Sprache, jene, die alles ausser der ursprünglichen Sprache des eingegebenen Automaten akzeptiert, erkennt. Das Modul `Binaerzahlen` wandelt eine Dezimalzahl in eine Binärzahl um. Dabei wird die Ziffernfolge umgedreht, sodass die Werte der Stellen von links nach rechts ansteigen. Die `Projektion` beschreibt einen Automaten, der eine Sprache erkennt, die aus dem kartesischen Produkt zweier Alphabete besteht, bei dem eine Komponente weggelassen wird. Mit dem Modul `Shuff{}le` wurde ein Automat erstellt, der die Sprachen zweier Automaten mischt und das Ergebnis akzeptiert. Das Modul `Leerheitstest` überprüft, ob es für einen Automaten ein Wort gibt, das er akzeptiert. Sofern ein solches existiert, werden Listen der durchlaufenen Zustände und nötigen Eingaben ausgegeben, die zum Endzustand führen. Für jeden Automaten gibt es einen Automaten, der die gleiche Sprache erkennt, aber nur über eine minimale Zustandsanzahl verfügt. Dieser Automat wird mit dem Modul `MinimalizeDetAutomat` erstellt.

8.5 Modul `Automaton`

Das Modul `Automaton` definiert einen Automaten in Haskell.

```
> module Automaton where
> data (Eq a, Eq b) => Automaton a b =
>     Automaton { states ::  [a],
>                 initialstates :: [a],
>                 transitions :: [(a,b,a)],
>                 goodstates :: [a] }
>               deriving Show
```

Die Parameter des Typkonstruktes `Automaton` sind die Zustände `states`, die Anfangszustände `initialstates`, dann die Transitionsrelationen `transitions` und die Endzustände `goodstates`. Dabei werden die Zustands-, Anfangszustands- und Endzustandsmenge durch Listen der Form `[a]`, die Transitionsrelationsmenge durch eine Liste der Form `[(a,b,a)]` festgelegt.

8.6 Modul `Schnitt`

Wenn wir zwei Automaten mit ihren jeweiligen Sprachen betrachten, verwendet man das Modul `Schnitt`, um aus ihnen einen neuen Automaten $\mathfrak{A}$ zu konstruieren, dessen Sprache $\mathcal{L}(\mathfrak{A})$ der Schnitt aus den Sprachen eben dieser beiden Automaten ist. Was nun folgt, sind die Programmzeilen, die bei der Umsetzung benötigt werden. So werden am Anfang des Moduls `Schnitt` zunächst mehrere Module importiert.

```
> import Automaton
> import Utils
```

Im Modul `Schnitt` wird die Funktion `times` verwendet, welche im Modul `Utils` definiert wird und das kartesische Produkt von zwei Mengen bildet. Die beiden Mengen `x` und `y` als Listen sind die Parameter der Funktion `times`.

```
> times []     _   = []
> times (x:xs) y   =
>    map (\ a -> [x,a]) y ++ times xs y
```

Ist einer der Parameter der Funktion `times` eine leere Menge und der andere eine beliebig, so wird eine leere Menge zurückgegeben. Ansonsten nimmt die Funktion `times` ein Element `x` aus der Liste `x:xs` und wendet die Funktion `map (\a -> [x,a])` auf das andere Argument an. Dabei verwendet `map(\a -> [x,a]) y` eine Liste aus allen Paaren, die aus dem Element `x` und allen Elementen der anderen Liste bestehen. Die Listenoperation `++` vereinigt zwei Listen. Die Funktion `times xs y` besagt, dass die Funktion `times` auch auf alle anderen `x` angewandt wird. Die Funktion ist also rekursiv aufgebaut.

Die Parameter der, in Abbildung 2.7 dargestellten, Funktion `schnitt` sind zwei Automaten $\mathfrak{A}_1$ und $\mathfrak{A}_2$. Dementsprechend werden die Zustands-, Anfangszustands- und Endzustandsmenge des neuen Automaten durch `times q1 q2`, beziehungsweise `times i1 i2` und `times f1 f2` und dessen Transitionsrelation als `schnittDelta delta1 delta2` definiert.

Es folgt die Definition der Funktion `schnittDelta`. Die Parameter der Funktion `schnittDeltaSolo` sind Elemente `d@(q1, a, q1')` einer bestimmten Transitionsrelation aus der Transitionrelationsmenge `delta1` des Automaten $\mathfrak{A}_1$, und die gesamte Menge an Transitionsrelationen `delta2` des Automaten $\mathfrak{A}_2$. Die Anwendung der Funktion `schnittDeltaSolo` auf eine Menge von Transitionsrelationen mit einer leeren Menge ergibt wiederum eine leere Menge. Die Funktion `schnittDeltaSolo` erstellt bei der Anwendung auf eine nicht-leere Mengen eine Liste mit allen Tripeln `([q1,q2],a1,[q1',q2'])`, bei denen die Aktionen `a1` und `a2` identisch sind, `a1==a2`, für eine bestimmte Transitionsrelation aus `d`, also ein festes Tripel `(q1,a1,q1')` und der gesamten Transitionsrelationsmenge `e`, sprich allen Tripel `(q2,a2,q2')`.

```
> schnitt a1@(Automaton {states     =q1, initialstates =i1,
>                        transitions=delta1, goodstates=f1})
>         a2@(Automaton {states     =q2, initialstates =i2,
>                        transitions=delta2, goodstates=f2})
>         = Automaton   {states     =times q1 q2,
>                        initialstates=times i1 i2,
>                        goodstates   =times f1 f2,
>                        transitions  =(schnittDelta delta1 delta2)}

> schnittDeltaSolo _ [] = []
> schnittDeltaSolo d@(q1,a1,q1')
>                  e@((q2,a2,q2'):es)
>                       = if(a1 == a2)
>                           then [([q1,q2],a1,[q1',q2'])]
>                                ++ schnittDeltaSolo d es
>                           else    schnittDeltaSolo d es
> schnittDelta [] _     = []
> schnittDelta (d:ds) e = schnittDeltaSolo d e ++ schnittDelta ds e
```

Abb. 2.7: Die Funktion ***schnittDelta*** *zur Bestimmung der Transitionsrelation des Automaten für den Durchschnitt zweier Sprachen*

Auch die Funktion `schnittDelta` angewandt auf eine Menge von Transitionsrelationen mit einer leeren Menge ergibt eine leere Menge. Dahingegen wird bei nicht-leeren Mengen die Funktion `schnittDeltaSolo` auf alle Elemente der Transitionsrelationsmenge `d` angewandt.

8.7 **Modul** `Determinisierung`

Das Modul `Determinisierung` wandelt einen nicht-deterministischen Automaten in einen deterministischen Automaten um.

Wie in Abbildung 2.8 gezeigt, erzeugt die Funktion `determinate` einen neuen Automaten, der die deterministierte Form des eingegebenen Automaten darstellt. Sie erwartet den Konstruktor `Automaton`, also einen Automaten mit den Parametern `states` (Zustandsmenge), `init` (Anfangszustandsmenge), `transitions` (Transitionsrelationsmenge) und `goodstates` (Endzustandsmenge). Die Namen der Variablen sprechen für sich, `newstates` sind also die Zustände im neuen Automaten, `newgoods` die neuen Endzustände und `newtrans` die neue Transitionrelationsmenge.

```
> genpower [] = [[]]
> genpower (x:xs) =
>      let p = genpower xs
>      in p ++ [ x:teil | teil <- p ]
```

Die Funktion `genpower` erstellt die Potenzmenge der Eingabemenge und gibt diese als Liste zurück. Als Parameter wird die Eingabeliste erwartet. Um zum gewünschten Ergebnis zu kommen, wird das erste Element abgetrennt und aus dem Rest die Potenzmenge erzeugt. Anschließend wird das jeweils zuletzt abgetrennte Element der Potenzmenge hinzugefügt und dann mit der Potenzmenge vereinigt.

Mit der Funktion `findTransitions` in Abbildung 2.9 wird eine Liste mit allen neuen Transitionsrelationen zu dem Zustand `currstate` für den determinisierten Automaten erzeugt. Als Parameter werden die Transitionrelationsmenge `transitions` und der aktuelle Zustand `currState` erwartet.

Ist die eingegebene Transitionsrelationsmenge `transitions` leer, dann gibt die Funktion `findTransitions` eine leere Liste zurück. Weiterhin überprüft die Funktion `findTransitions` für eine bestimmte Transition `(_,a',_)` mit einer bestimmten Aktion `a'` und allen Zuständen `q2` aus der Transitionsrelationsmenge `transitions`, ob es eine Transition `(q1,a,q2)` mit `q1` in der Menge `currState` und der Aktion `a` gleich der Aktion `a'` gibt. Diese Transitionen der Form `(currState,a',q2)` werden in die neue Liste aufgenommen und die Funktion `findTransitions` wird auf alle anderen Aktionen `ac` ausgeführt, während die Funktion `filter (\(os,ac,ns) -> not (ac==a'))` alle Transitionsrelationen mit der Aktion `a'` herausfiltert.

```
> buildDeterministicTransitions [] _ = []
> buildDeterministicTransitions
>     (state:reststates) transitions
>   = findTransitions transitions state
>     ++
```

```
> determinate  (Automaton states init transitions goodstates) =
>    let newinit = [init]
>        newstates   = genpower states
>        newtrans    = buildDeterministicTransitions newstates transitions
>        newgoods    = filter (\ state -> null (intersect state goodstates)) newstates
>    in (Automaton newstates newinit newtrans newgoods)
```

Abb. 2.8: Funktion `determinate` *im Modul* `Deterministisierung`

```
> findTransitions [] _ = []
> findTransitions transitions@((_,a',_):ts) currState =
>    let newTarget = [ q2 | (q1,a,q2) <- transitions, q1 `elem` currState, a == a' ]
>    in  (currState,a',newTarget):
>         findTransitions (filter (\(os,ac,ns) -> not (ac==a')) ts) currState
```

Abb. 2.9: Funktion `findTransitions` *im Modul* `Deterministisierung`

```
>     buildDeterministicTransitions
>           reststates transitions
```

Zum Schluss erzeugt die Funktion `buildDeterministicTransitions` eine Liste mit allen neuen Transitionsrelationen für den deterministierten Automaten mit einer Zustandsmenge `states` und einer Transitionsrelationsmenge `transitions` als Parameter.

Wird die Funktion `buildDeterministicTransitions` auf eine leere Zustandsmenge `states` angewandt, wird eine leere Liste zurückgegeben. Bei nicht-leeren Mengen wird die Funktion `findTransitions` auf einen bestimmten Zustand `state` angewandt. Auf die restlichen Zustände wird die Funktion `buildDeterministicTransitions` angewandt, also wird die Funktion `findTransitions` auf alle Zustände angewandt und alle Listen werden vereinigt.

9 Erfahrungsberichte

9.1 Erfahrungen mit CVS

Zu Beginn der Arbeit mit CVS kristallisierte sich heraus, dass unser Kurs grob in zwei Gruppen einzuteilen war Eine, die schon einmal mit Unix oder Programmierung zu tun hatte und die, die das erste Mal in ihrem Leben auf dem Bildschirm vor sich ein schwarzes Fenster mit weißer Schrift sah und damit arbeiten sollte. Dementsprechend unterschiedlich wurde auch die Einführung in CVS, die die Kursleiter gaben, aufgenommen.

Entweder man verstand die Befehle und dargelegten Wege sofort, oder suchte sich im Anflug von Panik und Bedenken in Richtung weiterer Kursarbeit jemanden, der einen nochmal den letzten Schritt bereitwillig erklärte. Die erstgenannte Gruppe stellte auch munter Fachfragen, die zwar beantwortet wurden, aber beim Rest der Teilnehmer eher zu weiteren Fragezeichen führten. Als es nicht besser wurde, bat relativ schnell ein Teilnehmer, ob man die ganzen Befehle, die einem als »Ersthörer« um den Kopf schwirrten, samt ihren Funktionen einmal schwarz auf weiß, sprich als Handout, auflisten könne, wozu sich glücklicherweise ein anderer Kursteilnehmer auch sofort bereit erklärte. Damit ließ es sich dann auch als Neuling arbeiten – zumindest besser als zuvor –, wozu auch der Hinweis, dass man über Unix sogenannte »manpages« aufrufen kann, seinen Teil beigetragen hat. Auf diesen erklärt sich das Programm praktisch von selbst, indem die jeweiligen Befehle und deren Funktionen erläutert werden. Interessant ist in diesem Zusammenhang auch das Zitat »If you are not familiar with manpages, type '`man man`'.« aus der (default) »message of the day«.

Wie auch immer, darüber hätte sich ein gewisser Teil des Kurses nicht den Kopf zerbrechen müssen, denn für diejenigen, die mit Unix Probleme hatten, wurde am darauf folgenden Tag ohne Vorankündigung eine andere Möglichkeit eröffnet. Ein typischer Fall von »Hätten wir das vorher gewusst!«. Es stellte sich nämlich heraus, dass alles, was vorher abstrakt weiß auf schwarz stand, auch visuell bearbeitet werden kann, mit Windows und mit Fenstern, die sich öffnen, wo man »OK« oder »Abbrechen« anklicken kann, Verzeichnisse, Ordner und Dateien, die man – wie gewohnt – über den Arbeitsplatz aufrufen, verschieben, kopieren und öffnen kann. Damit war es nicht mehr zwingend notwendig über `putty` das »schwarze Befehlfenster« zu öffnen, sondern man konnte nun über die rechte Maustaste den Befehl »CVS Auschecken« aufrufen, seinen Benutzernamen und Pass-

wort in ein Fenster eingeben, sich Arbeitskopien per Mausklick »auschecken« und die jeweiligen Dateien, zum Beispiel Übungsaufgaben in sein Home-Verzeichnis kopieren und dort aufrufen, bearbeiten und »einchecken«.

Hätten wir im Kurs es von Anfang an auf diese Art und Weise gemacht, wäre meiner Meinung nach überhaupt kein Problem entstanden, denn alle, denen am Anfang bei Unix der Kopf geraucht hatte, arbeiteten fortan über Windows, der Rest blieb bei Unix. So gab es in Bezug auf CVS im weiteren Verlauf des Kurses keine großen Probleme oder allzu verzweifelte Gesichter mehr.

9.2 Erfahrungen mit Haskell

Es hatten nur wenige Teilnehmer Erfahrungen mit der Programmiersprache Haskell vor der DSA gesammelt, so dass fast alle Teilnehmer Neuland betraten. Wir hatten also mal wieder den Computerraum für uns reserviert und kopierten – wie uns gesagt wurde – eine Übungsaufgabendatei zu Haskell in unser Verzeichnis. Dann wurde uns Teilnehmern zunächst einmal das erzählt, was man grundsätzlich zu dieser Programmiersprache wissen sollte. Nachdem uns dann einzelne Module vorgestellt wurden, machten wir uns an die Arbeit, die Aufgaben zu bearbeiten. Einige waren dabei zwar wesentlich schneller fertig als andere, aber im Großen und Ganzen waren wir allesamt in der Lage diese zu lösen. Wie wir im Kurs dann später auch merken sollten, eigneten sich diese Module, als sie alle zusammengefügt waren, gut für die Presburger Arithmetik.

10 Abschließende Bemerkungen

Diese Dokumentation soll eine kurze Zusammenfassung der im Kurs behandelten Themen liefern. Die Herleitung und der Beweis der mathematischen Konzepte und Grundlagen war ein wichtiger Teil der vielen Lerneinheiten. Doch auch die praktische Anwendung der Automatenkonzepte, die Implementierung der Modelle in Haskell und das Erlernen der, für viele doch neuen, funktionalen Programmiersprache war ein wesentlicher Kursinhalt. Die Konzepte zu den endlichen Automaten spielen in der Informatik eine große Rolle. Vor allem aufgrund ihrer guten Abschlusseigenschaften und ihrer einfachen Implementierung in die verschiedensten Programmiersprachen wird oft versucht, ein Problem auf einen endlichen Automaten zurückzuführen.

Die Arbeit im Kurs hat, trotz der vielleicht oft schwierigen mathematischen Konzepte, allen Kursteilnehmern viel Spaß gemacht. Schnell entwickelte sich im Kurs ein ganz eigenes System die Aufgaben in der Gruppe zu lösen. Mithilfe der stetigen Anregungen der Kursleiter und der gegenseitigen Hilfe wurde das Kursziel, die Implementierung der Presburger Arithmetik, erreicht.

Endliche Automaten sind ein spannendes und weitreichendes wissenschaftliches Feld, das auf viele Themengebiete anwendbar ist. Ohne die hinter den Automaten stehende Logik und Konzepte wäre das Leben mit dem Computer sicher nicht so einfach und effizient.

Literatur

[1] *CVS – Concurrent Versions System.* Dokumentation und Software verfügbar auf `http://www.nongnu.org/cvs/`.

[2] *Haskell.* Dokumentation und Software verfügbar auf `http://www.haskell.org/`.

[3] J. E. Hopcroft and J. D. Ullman. *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Addison-Wesley (Deutschland) GmbH, zweite Auflage, 1990.

[4] M. Lange and M. Hofmann. Automatentheorie. Vorlesungsmanusskript an der Ludwig-Maximilians-Universität München, `http://www.tcs.ifi.lmu.de/lehre/SS08/Automat/`, 2008.

[5] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, Editoren, *Handbook of Formal Languages – Volume 1: Word, Language, Grammar*, Seiten 41–110. Springer-Verlag, 1997.