

# Termersetzungssysteme und Reduktion

**Definition 2.24.** Ein Term  $t$  heißt

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

Ein Termersetzungssystem  $(\Sigma, \rightarrow_0)$  heißt schwach/stark normalisierend, wenn jeder Term schwach/stark normalisierend in  $\rightarrow$  ist.

**Satz 2.28.** Sei  $>$  Reduktionsordnung und für alle Terme  $t, s$  gelte: Aus  $t \rightarrow_0 s$  folgt:  $t > s$ . Dann ist  $\rightarrow$  stark normalisierend.

**Satz 2.40.** Polynomordnungen sind Reduktionsordnungen.

**Korollar 2.41.** Sei  $\rightarrow_0$  ein Termersetzungssystem und  $>_{\mathcal{A}}$  eine Polynomordnung. Falls

$$t \rightarrow_0 s \Rightarrow t \succ_{\mathcal{A}} s \text{ für alle Terme } t, s,$$

dann ist  $\rightarrow$  stark normalisierend.

**Satz 2.46.** Sei  $T$  ein konfluentes Termersetzungssystem.

1. Für Terme  $t, s$  gilt

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

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

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

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

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

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

Ausführungsvorschrift  $\beta$ -Reduktion, die das Ausrechnen einer Funktionsanwendung modelliert:

$$(\lambda x.t)x \rightarrow_{\beta} t$$

erlaubt insbesondere:

$$(\lambda x.s)t \rightarrow_{\beta} s[t/x]$$

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

Zusätzliche Grundreduktionen wie  $\eta$ -Reduktion:  $\lambda x.yx \rightarrow_{\eta} y$ .

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

$$\lambda y.fun \rightarrow_{\delta} \delta y.(\delta x.t)$$

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

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

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

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

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

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

$$\begin{aligned}
& \underline{len} ( \underline{cons} \ \underline{two} \ \underline{nil} ) \\
\rightarrow_a & (\lambda l. l \ \underline{zero} ( \lambda x r. \underline{succ} \ r ) ) ( \underline{cons} \ \underline{two} \ \underline{nil} ) \\
\rightarrow_a & (\lambda l. l ( \lambda f a. a ) ( \lambda x r. \underline{succ} \ r ) ) ( \underline{cons} \ \underline{two} \ \underline{nil} ) \tag{1} \\
\rightarrow_a & (\lambda l. l ( \lambda f a. a ) ( \lambda x r. ( \lambda n f a. f ( n f a ) ) r ) ) ( \underline{cons} \ \underline{two} \ \underline{nil} ) \\
\rightarrow_a & (\lambda l. l ( \lambda f a. a ) ( \lambda x r. ( \lambda n f a. f ( n f a ) ) r ) ) ( ( \lambda x l u f. f x ( l u f ) ) \underline{two} \ \underline{nil} )
\end{aligned}$$

$$\begin{aligned}
& \underline{len} ( \underline{cons} \ \underline{two} \ \underline{nil} ) \\
\rightarrow_a & \underline{len} ( ( \lambda x l u f. f x ( l u f ) ) \underline{two} \ \underline{nil} ) \\
\rightarrow_a & \underline{len} ( ( \lambda x l u f. f x ( l u f ) ) ( \lambda f a. f ( f a ) ) \underline{nil} ) \tag{2} \\
\rightarrow_a & \underline{len} ( ( \lambda l u f. f ( \lambda g a. g ( g a ) ) ( l u f ) ) \underline{nil} ) \\
\rightarrow_a & \underline{len} ( ( \lambda l u f. f ( \lambda g a. g ( g a ) ) ( l u f ) ) ( \lambda u f. u ) )
\end{aligned}$$

### Beispiel

$$\begin{aligned}
\underline{pow2} \ \underline{three} & \rightarrow_n \underline{three} ( \underline{mult} \ \underline{two} ) \ \underline{one} \\
& \rightarrow_n \lambda f a. f ( f ( f a ) ) ( \underline{mult} \ \underline{two} ) \ \underline{one} \\
& \rightarrow_n \lambda a. ( \underline{mult} \ \underline{two} ) ( ( \underline{mult} \ \underline{two} ) ( ( \underline{mult} \ \underline{two} ) a ) ) \ \underline{one} \\
& \rightarrow_n ( \underline{mult} \ \underline{two} ) ( ( \underline{mult} \ \underline{two} ) ( ( \underline{mult} \ \underline{two} ) \ \underline{one} ) ) \\
\underline{pow2} \ \underline{three} & \rightarrow_a \underline{pow2} ( \lambda f a. f ( f ( f a ) ) ) \\
& \rightarrow_a ( \lambda f a. f ( f ( f a ) ) ) ( \underline{mult} \ \underline{two} ) \ \underline{one} \\
& \rightarrow_a ( \lambda f a. f ( f ( f a ) ) ) ( \underline{mult} \ \lambda f a. f ( f a ) ) \ \underline{one} \\
& \rightarrow_a ( \lambda f a. f ( f ( f a ) ) ) ( \underline{mult} \ \lambda f a. f ( f a ) ) \ \lambda f a. f ( a )
\end{aligned}$$

### Funktionen

$flip = \lambda f x y. f y x$	$pair \ a \ b = \lambda s. s \ a \ b$	$swap \ p = p ( \lambda x y. pair \ y \ x )$
$const = \lambda x y. x$	$fst \ p = p \ \lambda x y. x$	$true = \lambda x y. x$
$twice = \lambda f x. f ( f x )$	$snd \ p = p \ \lambda x y. y$	$false = \lambda x y. y$
$zero = \lambda f a. a$	$add = n \ succ \ m$	$if\_then\_else = \lambda b x y. b x y$
$succ \ n = \lambda f a. f ( n f a )$	$sub \ n \ m = m \ pred \ n$	$isZero \ n = n \ \lambda x. false \ true$
$len = \lambda l. l \ zero ( \lambda x r. succ \ r )$	$mult = n ( add \ m ) \ zero$	
$pred \ n = fst ( n ( \lambda p. pair ( snd \ p ) ( succ ( snd \ p ) ) ) ( pair \ zero \ zero ) )$		

Applikation:  $f x y = ( f x ) y$

Abstraktion:  $\lambda x. x x = \lambda x. ( x x ) \neq ( \lambda x. x ) x$

## System F

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

$$\alpha_1 \rightarrow (\alpha_2 \rightarrow (\dots \rightarrow (\alpha_n \rightarrow \beta) \dots)),$$

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

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

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

Diese Relation ist induktiv definiert durch die folgenden Regeln:

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

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

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

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

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

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

Mehrfachinstanziierung polymorpher Funktionen ist nur per let erlaubt (daher wird ML-Polymorphie auch als *let-Polymorphie* bezeichnet):

$$(\forall_e) \frac{}{\Gamma \vdash x : \alpha[\beta_1/\alpha_1, \dots, \beta_k/\alpha_k]} \quad (x : \forall a_1, \dots, a_k. \alpha) \in \Gamma$$

$$(let) \frac{\Gamma \vdash t : \alpha \quad \Gamma, x : Cl(\Gamma, \alpha) \vdash s : \beta}{\Gamma \vdash \text{let } x = t \text{ in } s : \beta}$$

Typisierungsurteile sind weiterhin von der Form  $\Gamma \vdash t : \alpha$ , wobei wir Typvariablen, die im Typ  $\alpha$ , aber nicht im Kontext  $\Gamma$  vorkommen, als implizit allquantifiziert lesen, d. h. für  $FV(\alpha) \setminus FV(\Gamma) = \{a_1, \dots, a_k\}$  setzen wir:

$$Cl(\Gamma, \alpha) = \forall a_1 \dots \forall a_k. \alpha$$

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

$$\begin{array}{c} \frac{(Ax) \frac{}{\Gamma_2 \vdash n : \forall a. (a \rightarrow a) \rightarrow a \rightarrow a} \quad \frac{}{\Gamma_2 \vdash mult : \mathbb{N} \rightarrow \rightarrow \mathbb{N}} \quad \frac{}{\Gamma_2 \vdash two : \mathbb{N}}}{(\forall_e) \frac{\Gamma_2 \vdash n : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N} \quad \Gamma_2 \vdash mult \ two : \mathbb{N} \rightarrow \mathbb{N}}{\Gamma_2 \vdash n (mult \ two) : \mathbb{N} \rightarrow \mathbb{N}}} \quad \frac{}{\Gamma_2 \vdash one : \mathbb{N}} \\ (\rightarrow_e) \frac{}{\Gamma_2 := \{\Gamma, n : \mathbb{N}\} \vdash n (mult \ two) \ one : \mathbb{N}} \\ (\rightarrow_i) \frac{}{\{mult : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n. n (mult \ two) \ one : \mathbb{N} \rightarrow \mathbb{N}} \end{array}$$

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

1.  $PT(\Gamma; x; \alpha) = \{\alpha \doteq \beta \mid x : \beta \in \Gamma\}$

2.  $PT(\Gamma; \lambda x.t; \alpha) = PT((\Gamma, x : a); t; b) \cup \{a \rightarrow b \doteq \alpha\}$ , mit  $a, b$  frisch.

3.  $PT(\Gamma; ts; \alpha) = PT(\Gamma; t; a \rightarrow \alpha) \cup PT(\Gamma; s; a)$ , mit  $a$  frisch.

Anpassung des Algorithmus for den ML-polymorphen  $\lambda$ -Kalkül:

1.  $PT(\Gamma; x; \alpha) = \{\alpha \doteq \gamma[a'_1/a_1, \dots, a'_k/a_k]\}$  für  $(x : \forall a_1 \dots \forall a_k. \gamma) \in \Gamma$  mit  $a'_i$  frisch

2.  $PT(\Gamma; (\text{let } x = s \text{ in } t); \alpha) = PT(\Gamma\sigma, x : Cl(\Gamma\sigma, \sigma(b))); t; \alpha\sigma$  wobei  $\sigma = mgu(PT(\Gamma; s; b))$  mit  $b$  frisch

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

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

Wir erhalten

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

Also hat  $\lambda xy. xy$  den Prinzipaltyp  $mgu(\mathcal{E})(a) = (d \rightarrow e) \rightarrow d \rightarrow e$ .

**Beispiel** Wir berechnen den Prinzipaltyp von  $(x : a, x\lambda z. z)$

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

Wir erhalten

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

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

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

**Beispiel** Klausur [Probe] SoSe 2017, Aufgabe 2

$$PT(\Gamma; s; \alpha) = PT(\Gamma\sigma, f : \underbrace{Cl(\Gamma\sigma, \sigma(b))}_{\forall a. \sigma(b)}; (f \text{ or } (\text{contains false } (f \text{ cons nil}))); \alpha\sigma) \text{ und } \sigma = mgu(PT(\Gamma; \lambda t. t \text{ true}; b))$$

$$\begin{aligned} PT(\Gamma; \lambda t.t \text{ true}; b) &= PT(\Gamma, t : c; t \text{ true}; d) \cup \{c \rightarrow d \doteq b\} \\ &= PT(\Gamma, t : c; t; e \rightarrow d) \cup PT(\Gamma, t : c; \text{true}; e) \cup \{c \rightarrow d \doteq b\} \\ &= \{c \doteq e \rightarrow d, e \doteq \mathbb{B}, c \rightarrow d \doteq b\} =: \mathcal{E} \end{aligned}$$

$$mgu(\mathcal{E}) = [(\mathbb{B} \rightarrow d) \rightarrow d/b]$$

Setze  $\Gamma = \Gamma \cup (f : \forall a. (\mathbb{B} \rightarrow a) \rightarrow a)$

$$\begin{aligned} PT(\Gamma; s; \alpha) &= PT(\Gamma; (f \text{ or } (\text{contains false } (f \text{ cons nil}))); \alpha) \\ &= PT(\Gamma; f \text{ or } a \rightarrow \alpha) \cup PT(\Gamma; \text{contains false } (f \text{ cons nil}); a) \\ &= PT(\Gamma; f; b \rightarrow (a \rightarrow \alpha)) \cup PT(\Gamma; \text{or}; b) \cup PT(\Gamma; \text{contains false}; c \rightarrow a) \cup PT(\Gamma; f \text{ cons nil}; c) \\ &= \{b \rightarrow (a \rightarrow \alpha) \doteq (\mathbb{B} \rightarrow a_1) \rightarrow a_1\} \cup \{b \doteq \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}\} \\ &\quad \cup PT(\Gamma; \text{contains}; d \rightarrow c \rightarrow a) \cup PT(\Gamma; \text{false}; d) \cup PT(\Gamma; f \text{ cons}; e \rightarrow c) \cup PT(\Gamma; \text{nil}; e) \\ &= \{b \rightarrow (a \rightarrow \alpha) \doteq (\mathbb{B} \rightarrow a_1) \rightarrow a_1\} \cup \{b \doteq \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}\} \\ &\quad \cup \{d \rightarrow c \rightarrow a \doteq a_2 \rightarrow \mathbb{L}a_2 \rightarrow \mathbb{B}\} \cup \{d \doteq \mathbb{B}\} \cup PT(\Gamma; f; g \rightarrow e \rightarrow c) \cup PT(\Gamma; \text{cons}; g) \cup \{e \doteq \mathbb{L}a_3\} \\ &= \{b \rightarrow (a \rightarrow \alpha) \doteq (\mathbb{B} \rightarrow a_1) \rightarrow a_1, b \doteq \mathbb{B} \rightarrow \mathbb{B} \rightarrow \mathbb{B}, d \rightarrow c \rightarrow a \doteq a_2 \rightarrow \mathbb{L}a_2 \rightarrow \mathbb{B}, d \doteq \mathbb{B}, \\ &\quad g \rightarrow e \rightarrow c \doteq (\mathbb{B} \rightarrow a_3) \rightarrow a_3, g \doteq a_4 \rightarrow \mathbb{L}a_4 \rightarrow \mathbb{L}a_4, e \doteq \mathbb{L}a_5\} \end{aligned}$$

## Strukturelle Induktion

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

### Beispiel Allgemeine Listen

```
data List a where  
  Nil : () → List a  
  Cons : a → List a → List a
```

Dazugehörige *fold*-Funktion:

$$\begin{aligned} \text{fold } a \ c \ \text{Nil} &= a \\ \text{fold } a \ c \ (\text{Cons } x \ xs) &= c \ x \ (\text{fold } a \ c \ xs) \end{aligned}$$

Für  $h : M \rightarrow A, t : M \rightarrow M$  wird außerdem eine Funktion  $\text{unfold } h \ t : M \rightarrow A^\omega$  definiert per

$$\begin{aligned} \text{hd} (\text{unfold } h \ t \ x) &= h \ x \\ \text{tl} (\text{unfold } h \ t \ x) &= \text{unfold } h \ t \ (t \ x) \end{aligned}$$

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

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

### Funktionen

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

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

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

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

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

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

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

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

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

### Beispiel Klausur [Probe] SoSe 2017, Aufgabe 3

1. Beweisen Sie mittels struktureller Induktion, dass

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

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

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

IV) Sei Aussage für  $as$  bereits bewiesen.

IS) Sei  $xs = Cons a as$

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

*q.e.d*

2. Geben Sie die *fold*-Funktion für Twins, *foldtw*, inklusive ihres Typs an:

$$foldtw : (a \rightarrow a \rightarrow b) \rightarrow (a \rightarrow b \rightarrow a \rightarrow b) \rightarrow Twins a \rightarrow b$$

$$foldtw f g (End x y) = f x y$$

$$foldtw f g (More x T y) = g x (foldtw f g T) y$$

$$identicalTwins a = foldtw (if x == y then Cons x Nil else Nil) (\lambda x l y. if x == y then Cons x l else l) a$$

## Korekursion und Koinduktion

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

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

und

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

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

**Beispiel 4.37.** Wir definieren:

$$\begin{aligned}blink &: 2 \times a \times Stream\ a \rightarrow Stream\ a \\blink\ c\ x\ (a_0, a_1, \dots) &= \begin{cases} (x, a_0, x, a_1, \dots) & \text{falls } c = 0 \\ (a_0, x, a_1, x, \dots) & \text{falls } c = 1 \end{cases}\end{aligned}$$

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

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

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

$$\begin{aligned}hd\ (even\ s) &= hd\ s \\tl\ (even\ s) &= even\ tl\ (tl\ (s)) \\hd\ (odd\ s) &= hd\ tl\ (s) \\tl\ (odd\ s) &= odd\ tl\ (tl\ (s))\end{aligned}$$

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

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

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

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

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

**Beispiel** Klausur [Probe] SoSe 2017, Aufgabe 3

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

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

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

Funktion in Codata-Funktionen einsetzen und Verhalten anwenden:

$$\begin{aligned} \text{currentSample } (\text{sampler } t \ s) &= \text{if } (\text{currentSample } t > 0) \text{ then } (\text{currentSample } s) \text{ else } 0 \\ \text{discardSample } (\text{sampler } t \ s) &= \text{sampler } (\text{discardSample } t) (\text{discardSample } s) \end{aligned}$$

2. Beweis der Aussagen per Koinduktion:

$$\begin{aligned} (1): \text{currentSample } (\text{sampler } (\text{square } 0 \ 1) (\text{square } x \ 0)) &= \\ &= \text{if } \text{currentSample } (\text{square } 0 \ 1) > 0 \text{ then } \text{currentSample } (\text{square } x \ 0) \text{ else } 0 \\ &= \text{if } 0 > 0 \text{ then } x \text{ else } 0 \\ &= 0 \\ &= \text{currentSample } (\text{flat } 0) \checkmark \end{aligned}$$

$$\begin{aligned} \text{discardSample } (\text{sampler } (\text{square } 0 \ 1) (\text{square } x \ 0)) &= \\ &= \text{sampler } (\text{discardSample } (\text{square } 0 \ 1)) (\text{discardSample } (\text{square } x \ 0)) \\ &= \text{sampler } (\text{square } 1 \ 0) (\text{square } 0 \ x) \\ \rightarrow \text{currentSample } (\text{sampler } (\text{square } 1 \ 0) (\text{square } 0 \ x)) &= \\ &= \text{if } \text{currentSample } (\text{square } 1 \ 0) > 0 \text{ then } \text{currentSample } (\text{square } 0 \ x) \text{ else } 0 \\ &= \text{if } 1 > 0 \text{ then } 0 \text{ else } 0 \\ &= 0 \\ &= \text{currentSample } (\text{discardSample } (\text{flat } 0)) \checkmark \end{aligned}$$

$$\begin{aligned} (2): \text{currentSample } (\text{sampler } (\text{square } 1 \ 0) (\text{flat } x)) &= \\ &= \text{if } \text{currentSample } (\text{square } 1 \ 0) > 0 \text{ then } \text{currentSample } (\text{flat } x) \text{ else } 0 = \\ &= \text{if } 1 > 0 \text{ then } x \text{ else } 0 \\ &= x \\ &= \text{currentSample } (\text{square } x \ 0) \checkmark \end{aligned}$$

$$\begin{aligned} \text{discardSample } (\text{sampler } (\text{square } 1 \ 0) (\text{flat } x)) &= \\ &= \text{sampler } (\text{discardSample } (\text{square } 1 \ 0)) (\text{discardSample } (\text{flat } x)) \\ &= \text{sampler } (\text{square } 0 \ 1) (\text{flat } x) \\ \rightarrow \text{currentSample } (\text{sampler } (\text{square } 0 \ 1) (\text{flat } x)) &= \\ &= \text{if } \text{currentSample } (\text{square } 0 \ 1) > 0 \text{ then } \text{currentSample } (\text{flat } x) \text{ else } 0 \\ &= \text{if } 0 > 0 \text{ then } x \text{ else } 0 \\ &= 0 \\ &= \text{currentSample } (\text{discardSample } (\text{square } x \ 0)) \checkmark \end{aligned}$$



# Reguläre Sprachen

**Definition 6.6.** Reguläre Ausdrücke  $r, s, \dots$  sind durch die Grammatik

$$r, s ::= 1 \mid 0 \mid r + 1 \mid rs \mid r^* \mid a \quad (a \in \Sigma)$$

definiert. Die Semantik eines regulären Ausdrucks  $r$  ist die wie folgt rekursiv definierte Sprache  $L(r)$ :

- $L(a) = \{a\}$
- $L(1) = \{\varepsilon\}$
- $L(0) = \emptyset$
- $L(r + s) = L(r) \cup L(s)$
- $L(rs) = \{uv \mid u \in L(r), v \in L(s)\}$
- $L(r^*) = \{u_1 \dots u_n \mid n \geq 0, u_i \in L(r) \forall i\}$

**Satz 6.8** (Kleene). Eine Sprache  $L$  ist genau dann regulär, wenn es einen regulären Ausdruck  $r$  gibt mit  $L = L(r)$ .

**Bemerkung** (zu Satz 6.8).

- $L(r_{s,t}^R) = \{u \in \Sigma \mid s \xrightarrow{u} t \text{ mit Zwischenzuständen nur aus } R\}$
- insbesondere  $r_{s,t}^\emptyset$ : von  $s$  nach  $t$  ohne Zwischenzustand
- $r_{i,j}^{\{p,q\}} := r_{i,j}^{\{p\}} + r_{i,q}^{\{p\}} \left( r_{q,q}^{\{p\}} \right)^* r_{q,j}^{\{p\}}$

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

**Satz 6.18** (Myhill und Nerode). Eine Sprache  $L$  ist genau dann regulär, wenn die durch

$$v \sim_L w \Leftrightarrow \forall u \in \Sigma^*. (vu \in L \Leftrightarrow wu \in L)$$

definierte Äquivalenzrelation  $\sim_L$  endlichen Index hat.

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

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

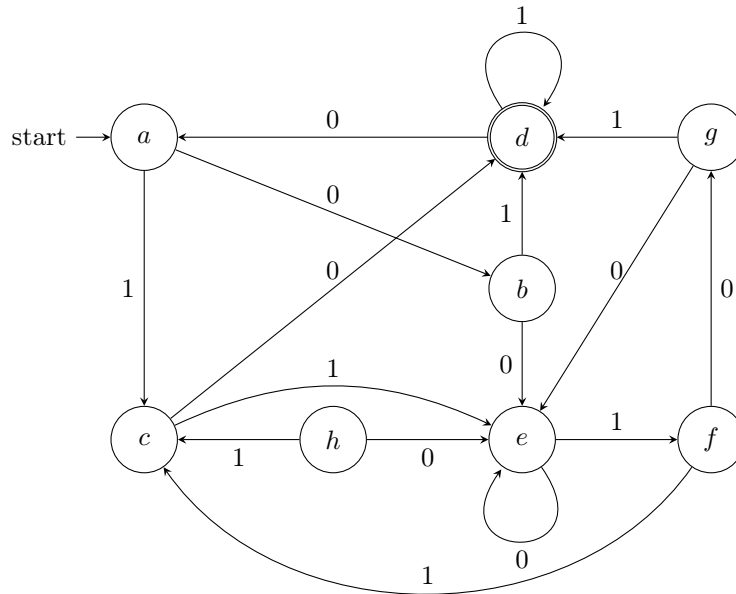
$$(\delta(a, q_1), \delta(a, q_2)) \notin R$$

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

4. Identifiziere alle Zustandspaare in  $R$ .

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

**Beispiel** Klausur [Probe] SoSe 2017, Aufgabe 3



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

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

Beschreibung von  $R$ :

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

