

Übersetzerbau
Algebraic Compiler Construction

Peter Padawitz, TU Dortmund, Germany
1. August 2017

(actual version: <http://fdit-www.cs.uni-dortmund.de/~peter/CbauFolien.pdf>)
Webseite zur LV: fdit-www.cs.uni-dortmund.de/ueb.html

Inhalt

Mit * markierte Abschnitte bzw. Kapitel werden zur Zeit in der LV nicht behandelt.

Zur Navigation auf Titel, nicht auf Seitenzahlen klicken!

1 Einleitung	7
2 Algebraische Modellierung	14
- Mengen und Funktionen	14
- Mehrsortige Mengen	17
- Produkte und Summen	21
- Typen und Signaturen	28
- 2.4 Konstruktive Signaturen	33
- 2.5 Destruktive Signaturen	35
- 2.6 Algebren	38
- 2.7 Terme und Coterme	42
- 2.8 Beispiele	47
- 2.9 Die Algebren <i>Bool</i> und <i>Regword(BL)</i>	55
- 2.10 Die Algebren <i>Beh(X, Y)</i> , <i>Pow(X)</i> , <i>Lang(X)</i> und <i>Bro(BL)</i>	57
- 2.11 Die Erreichbarkeitsfunktion	60
- 2.12 Termfaltung und Zustandsentfaltung	61

- 2.16 Initiale Automaten	71
- 2.17 Compiler für reguläre Ausdrücke	74
- 2.18 * Minimale Automaten	77
- 2.21 * Baumautomaten	79
3 * Rechnen mit Algebren	83
- Unteralgebren	84
- Kongruenzen und Quotienten	88
- Automatenminimierung durch Quotientenbildung	92
- Transitionsmonoid und syntaktische Kongruenz	93
- 3.6 Termsubstitution	95
- Termäquivalenz und Normalformen	97
- 3.10 Normalformen regulärer Ausdrücke	102
- 3.11 Die Brzozowski-Gleichungen	103
- 3.12 Optimierter Brzozowski-Automat	105
- 3.13 Erkenner regulärer Sprachen sind endlich	106
4 Kontextfreie Grammatiken (CFGs)	109
- Linksrekursive CFGs	115
- Abstrakte Syntax	119
- Linksrekursive CFGs und abstrakte Syntax	126
- Wort- und Ableitungsbaumalgebra	136
- 4.9 Das Zustandsmodell von JavaLight	143

5 Parser und Compiler für CFGs	146
- 5.1 Funktoren und Monaden	148
- Compilermonaden	160
- Monadenbasierte Parser und Compiler	162
6 LL-Compiler	167
7 LR-Compiler	187
8 Haskell: Typen und Funktionen	215
9 Haskell: Listen	225
10 Haskell: Datentypen und Typklassen	236
11 Algebren in Haskell	250
- 11.1 Beispiele	251
- 11.2 Datentyp der JavaLight-Algebren	256
- 11.3 Die Termalgebra von JavaLight	258
- 11.4 Die Wortalgebra von JavaLight	259
- 11.5 Das Zustandsmodell von JavaLight	260
- 11.6 * Die Ableitungsbaumalgebra von JavaLight	263
12 Attributierte Übersetzung	266
- 12.1 * Binärdarstellung rationaler Zahlen	267
- 12.2 * Strings mit Hoch- und Tiefstellungen	269
- 12.3 * Übersetzung regulärer Ausdrücke in erkennende Automaten	271
- 12.4 * Darstellung von Termen als hierarchische Listen	277

-	12.5 Eine kellerbasierte Zielsprache für JavaLight	280
13	JavaLight+ = JavaLight + I/O + Deklarationen + Prozeduren	286
-	13.1 Assemblersprache mit I/O und Kelleradressierung	286
-	13.2 Grammatik und abstrakte Syntax von JavaLight+	293
-	13.3 JavaLight+-Algebra <i>javaStackP</i>	297
14 *	Mehrpassige Compiler	319
15	Funktoren und Monaden in Haskell	327
16	Monadische Compiler	335
-	16.1 Compilerkombinatoren	339
-	16.2 Monadische Scanner	341
-	16.3 Monadische LL-Compiler	343
-	16.4 Generischer JavaLight-Compiler	346
-	16.5 Korrektheit des JavaLight-Compilers	348
-	16.6 Generischer XMLstore-Compiler	352
17 *	Induktion, Coinduktion und rekursive Gleichungen	354
-	Induktion	354
-	Induktive Lösungen	356
-	Coinduktion	358
-	Coinduktive Lösungen	364
-	17.11 Cotermisierte Erkenner regulärer Sprachen	368
18 *	Iterative Gleichungen	370

- 18.2 Das iterative Gleichungssystem einer CFG	371
- 18.5 Von Erkennern regulärer Sprachen zu Erkennern kontextfreier Sprachen	377
19 * Interpretation in stetigen Algebren	387
- 19.1 Schleifensemantik	392
- 19.2 Semantik der Assemblersprache <i>StackCom</i> *	394
- Nochmal iterative Gleichungen	401
20 Literatur	410
21 Index	415

1 Einleitung

Die Folien dienen dem Vor- (!) und Nacharbeiten der Vorlesung, können und sollen aber deren regelmäßigen Besuch nicht ersetzen!

Interne Links (einschließlich der Seitenzahlen im [Index](#)) sind an ihrer **dunkelroten** Färbung, externe Links (u.a. zu Wikipedia) an ihrer **magenta**-Färbung erkennbar. Dunkelrot gefärbt ist auch das jeweils erste Vorkommen einer Notation. **Fettgedruckt** ist ein Begriff in der Regel nur an der Stelle, an der er definiert wird.

Jede Kapitelüberschrift und jede Seitenzahl in der rechten unteren Ecke einer Folie ist mit dem Inhaltsverzeichnis verlinkt. Namen von Haskell-Modulen wie [Java.hs](#) sind mit den jeweiligen Programmdateien verknüpft.

Ein **Scanner** (Programm zur **lexikalischen Analyse**) fasst Zeichen zu Symbolen zusammen, übersetzt also eine **Zeichenfolge** in eine **Symbolfolge**. Bezuglich der Grammatik, die der nachfolgenden Syntaxanalyse zugrundeliegt, heißen die Symbole auch **Terminale**. Man muss unterscheiden zwischen Symbolen, die Komponenten von **Operatoren** sind (**if**, **then**, **=**, etc.), und Symbolen wie **5** oder **True**, die der Scanner einer **Basismenge** (Int, Float, Bool, Identifier, etc.) zuordnet. In der Grammatik kommt ein solches Symbol nicht vor, jedoch der Namen der Basismenge, der es angehört.

Ein **Parser** (Programm zur **Syntaxanalyse**) übersetzt eine **Symbolfolge** in einen **Syntaxbaum**, der den für ihre Übersetzung in eine **Zielsprache** notwendigen Teil ihrer **Bedeutung (Semantik)** wiedergibt.

Der Parser **scheitert**, wenn die Symbolfolge in diesem Sinne bedeutungslos ist. Er orientiert sich dabei an einer (kontextfreien) **Grammatik**, die korrekte von bedeutungslosen Symbolfolgen trennt und damit die **Quellsprache** definiert.

Ein **Compiler** (Programm zur **semantischen Analyse** und Codeerzeugung) übersetzt ein Programm einer Quellsprache in ein semantisch äquivalentes Programm einer Zielsprache.

Ein **Interpreter**

- wertet einen Ausdruck in Abhängigkeit von einer Belegung seiner Variablen aus bzw.
- führt eine Prozedur in Abhängigkeit von einer Belegung ihrer Parameter aus.

Letzteres ist ein Spezialfall von Ersterem. Deshalb werden wir einen Interpreter stets als die **Faltung** von Termen (Ausdrücken, die aus Funktionssymbolen einer **Signatur** bestehen) in einer **Algebra** (Interpretation der Signatur) modellieren.

Auch Scanner, Parser und Interpreter sind Compiler, insofern sie *Objekte von einer Repräsentation in eine andere übersetzen*. Die Zielrepräsentation eines Interpreters ist i.d.R. eine *Funktion*, die *Eingabedaten* verarbeitet. Daher liegt es nahe, alle o.g. Programme nach denjenigen Prinzipien zu entwerfen, die bei Compilern im engeren Sinne Anwendung finden.

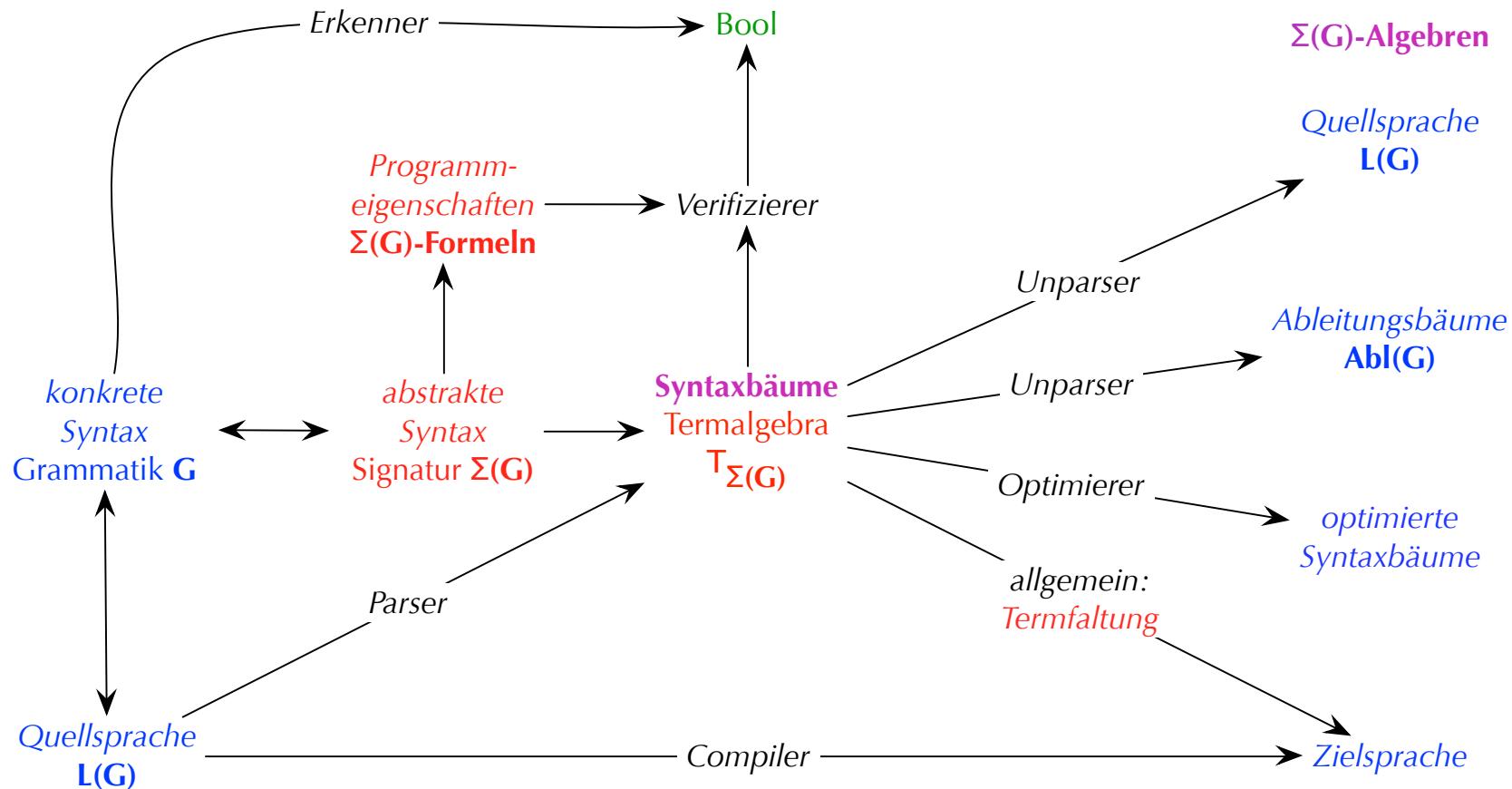
Die Entwicklung dieser Prinzipien und ihrer formalen Grundlagen wurzelt in der **mathematischen Logik** und dort vor allem in der **Theorie formaler Sprachen** und der **Universellen Algebra**, deren Anwendung im Compilerbau in dieser LV eine zentrale Rolle spielen wird. Die Aufgabe, Übersetzer zu schreiben, hat auch entscheidend die **Weiterentwicklung von Programmiersprachen**, insbesondere der logischen und funktionalen, vorangetrieben.

Insbesondere Konzepte der universellen Algebra und der funktionalen Programmierung im Übersetzerbau erlauben es, bei Entwurf und Implementierung von Scantern, Parsern, Compilern und Interpretern ähnliche Methoden einzusetzen. So ist z.B. ein wie oben definierter Interpreter gleichzeitig ein Compiler, der den Ausdruck oder die Prozedur in eine *Funktion* übersetzt, die eine Belegung auf einen Wert des Ausdrucks bzw. eine vom Aufruf der Prozedur erzeugte Zustandsfolge abbildet.

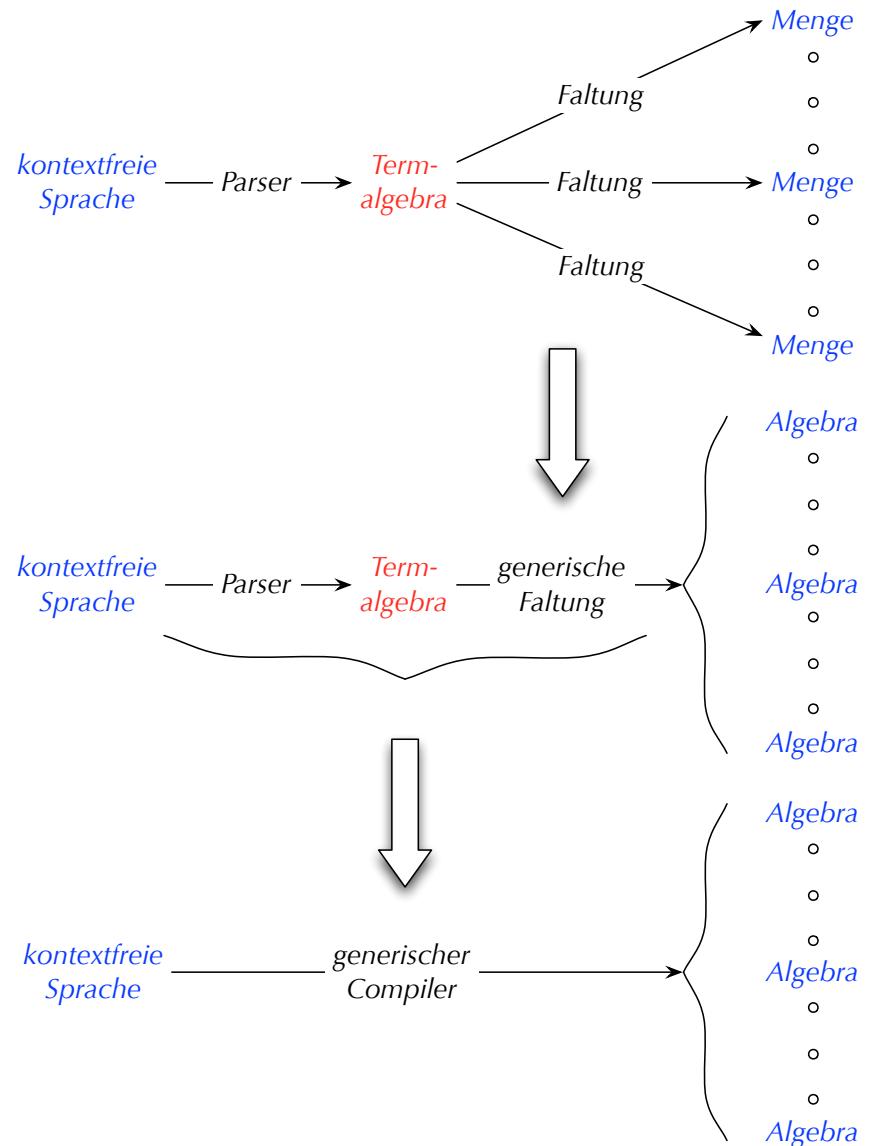
Der algebraische Ansatz klärt nicht nur die Bezüge zwischen den oben genannten Programmen, sondern auch eine Reihe weiterer in der Compilerbauliteratur meist sehr speziell definierter und daher im Kern vager Begriffe wie *attributierter Grammatiken* und *mehrpassiger Compilation*.

Mehr Beachtung als die nicht wirklich essentielle Trennung zwischen Scanner, Parser, Compiler und Interpreter verdienen die Ansätze, Compiler **generisch** (neu-informatisch: *agil*) zu entwerfen, so dass sie mit verschiedenen Quellsprachen und - wichtiger noch - mehreren Zielsprachen instanziert werden können. Es handelt sich dann im Kern um einen Parser, der keine Symbol-, sondern Zeichenfolgen verarbeitet und ohne den klassischen Umweg über Syntaxbäume gleich die gewünschten Zielobjekte/programme aufbaut. Sein *front end* kann mit verschiedenen Scannern verknüpft werden, die mehr oder weniger detaillierte Symbol- und Basistypinformation erzeugen, während sein *back end* mit verschiedenen Interpretationen ein und derselben - üblicherweise als kontextfreie Grammatik eingegebenen - *Signatur* instanziert werden kann, die verschiedenen Zielsprachen entsprechen.

Die beiden folgenden Grafiken skizzieren die Struktur des algebraischen Ansatzes und seine Auswirkung auf die Generizität der Programme. Auf die Details wird in den darauffolgenden Kapiteln eingegangen. Dabei wird die jeweilige Funktionalität der o.g. Programme in mathematisch präzise Definitionen gefasst, auf denen Entwurfsregeln aufbauen, deren Befolgung automatisch zu korrekten Compilern, Interpretern, etc. führen.



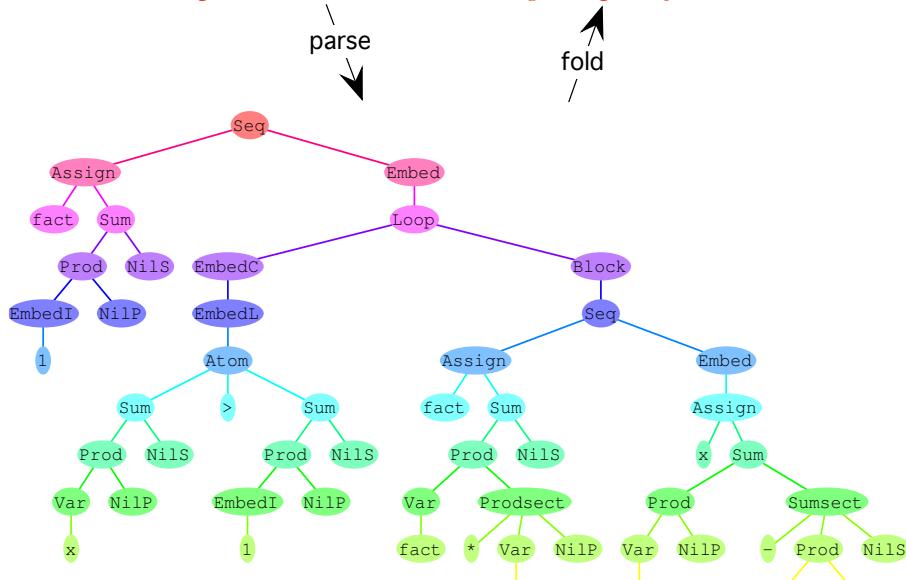
Von konkreter über abstrakte Syntax zu Algebren



Der Weg zum generischen Compiler

fact =1; while x > 1 {fact = fact*x; x = x-1;}

Eingabewort (Element der JavaLight-Algebra *javaWord*)



Syntaxbaum
(Element der JavaLight-Algebra
javaTerm)

trans : $(String \rightarrow \mathbb{Z}) \rightarrow (String \rightarrow \mathbb{Z})$

store $\mapsto \lambda str. \text{if } str = x \text{ then } 0$

else if str = fact then store(x)! else store(str)

Zustandstransitionsfunktion
(Element der JavaLight-Algebra *javaState*)

0: Push 1	9: Mul
1: Save "fact"	10: Save "fact"
2: Pop	11: Pop
3: Load "x"	12: Load "x"
4: Push 1	13: Push 1
5: Cmp ">"	14: Sub
6: JumpF 18	15: Save "x"
7: Load "fact"	16: Pop
8: Load "x"	17: Jump 3

Assemblerprogramm
(Element der JavaLight-Algebra *javaStack*)

2 Algebraische Modellierung

Mengen und Funktionen

\emptyset bezeichnet die leere Menge, 1 die Menge $\{\epsilon\}$ und 2 die Menge $\{0, 1\}$. 0 und 1 stehen hier in der Regel für die Wahrheitswerte *false* und *true*. Für alle $n > 1$, $[n] =_{def} \{1, \dots, n\}$.

Wir setzen voraus, dass die Definitionen einer **Potenzmenge**, **binären Relation** bzw. **(partiellen oder totalen) Funktion** sowie der **Vereinigung**, des **Durchschnitts**, des **Komplements**, der **Disjunktheit** und der **Mächtigkeit (Kardinalität)** von Mengen bekannt sind.

λ -Abstraktionen zur anonymen Definition und die (sequentielle) **Komposition** mehrerer Funktionen sollten ebenfalls bekannt sein.

Für jede Menge A bezeichnet $|A|$ die Kardinalität von A , $\mathcal{P}(A)$ die Potenzmenge von A und $id_A : A \rightarrow A$ die durch $id_A(a) = a$ für alle $a \in A$ definierte **Identität auf A** .

Für jede Teilmenge B von A bezeichnet $inc_B : B \rightarrow A$ die durch $inc_B(b) = b$ für alle $b \in B$ definierte **Inklusion**.

Für alle Mengen A, B bezeichnen $A \rightarrow B$ und B^A die Menge der totalen und $A \multimap B$ die Menge der partiellen Funktionen von A nach B . $def(f)$ bezeichnet den Definitionsbereich einer partiellen Funktion $f : A \multimap B$.

Für alle Mengen A, B , (totale) Funktionen $f : A \rightarrow B$, $C \subseteq A$, $D \subseteq B$ und $n > 0$,

$$\Delta_A^n =_{def} \{(a_1, \dots, a_n) \in A^n \mid \forall 1 \leq i < n : a_i = a_{i+1}\} \quad \text{Diagonale von } A$$

$$\begin{aligned} f(C) &=_{def} \{f(a) \mid a \in C\} && \text{Bild von } C \text{ unter } f \\ &= \mathcal{P}(f)(C) \quad (\text{siehe Mengenfunktor}) \end{aligned}$$

$$img(f) =_{def} f(A)$$

$$f^{-1}(D) =_{def} \{a \in A \mid f(a) \in D\} \quad \text{Urbild von } D \text{ unter } f$$

$$ker(f) =_{def} \{(a, a') \in A \times A \mid f(a) = f(a')\} \quad \text{Kern von } f$$

$$graph(f) =_{def} \{(a, f(a)) \in A \times B \mid a \in A\} \quad \text{Graph von } f$$

$$\begin{aligned} f|_C : C &\rightarrow B && \text{Einschränkung von } f \\ c &\mapsto f(c) && \text{auf } C \end{aligned}$$

$$\begin{aligned} f[b/a] : A &\rightarrow B && \text{Update von } f \\ c &\mapsto \text{if } c = a \text{ then } b \text{ else } f(c) \end{aligned}$$

$$\begin{aligned} \chi : \mathcal{P}(A) &\rightarrow 2^A && \text{charakteristische} \\ C &\mapsto \lambda a. \text{if } a \in C \text{ then } 1 \text{ else } 0 && \text{Funktion von } C \end{aligned}$$

$$f \text{ ist surjektiv} \Leftrightarrow_{def} \text{img}(f) = B$$

$$f \text{ ist injektiv} \Leftrightarrow_{def} \text{ker}(f) = \Delta_A^2$$

$$f \text{ ist bijektiv} \Leftrightarrow_{def} \exists g : B \rightarrow A : g \circ f = id_A \wedge f \circ g = id_B$$

A und B heißen **isomorph**, geschrieben: $A \cong B$, wenn es eine bijektive Funktion von A nach B gibt.

Aufgabe Zeigen Sie $\mathcal{P}(A) \cong 2^A$ und $\mathcal{P}(A \times B) \cong \mathcal{P}(B)^A$.

Aufgabe Zeigen Sie:

$$f : A \rightarrow B \text{ ist bijektiv} \Leftrightarrow f \text{ ist injektiv und surjektiv.}$$

Aufgabe Zeigen Sie: Sei f bijektiv. Dann gibt es *genau* eine Funktion $g : B \rightarrow A$ mit $g \circ f = id_A$ und $f \circ g = id_B$. g darf deshalb mit f^{-1} bezeichnet werden.

Mehrsortige Mengen

Sei S eine Menge. Eine **S -sortige** oder **S -indizierte Menge** ist ein Tupel $A = (A_s)_{s \in S}$ von Mengen. Manchmal schreiben wir A auch für die Vereinigung der Mengen A_s über alle $s \in S$.

Seien $A = (A_s)_{s \in S}$ und $B = (B_s)_{s \in S}$ S -sortige Mengen. Eine **S -sortige Funktion** $f : A \rightarrow B$ ist eine S -sortige Menge $(f_s)_{s \in S}$ derart, dass für alle $s \in S$, f_s eine Funktion von A_s nach B_s ist.

Sei $n > 0$. Eine **n -stellige S -sortige Relation auf A** ist eine S -sortige Menge $R = (R_s)_{s \in S}$ mit $R_s \subseteq A_s^n$ für alle $s \in S$. Im Fall $n = 1$ wird R auch **S -sortige Teilmenge von A** genannt.

Sei I eine Menge. Das **kartesische Produkt** (der Komponenten) einer I -sortigen Menge $A = (A_i)_{i \in I}$ ist wie folgt definiert:

$$\mathsf{X}_{i \in I} A_i = \{f : I \rightarrow \bigcup_{i \in I} A_i \mid \forall i \in I : f(i) \in A_i\}.$$

Im Fall $I = \emptyset$ besteht $\mathsf{X}_{i \in I} A_i$ aus der einzigen Funktion von \emptyset nach $\bigcup_{i \in I} A_i$.

Gibt es eine Menge A mit $A_i = A$ für alle $i \in I$, dann stimmt $\mathsf{X}_{i \in I} A_i$ offenbar mit A^I überein. Im Fall $I = [n]$ für ein $n > 1$ schreibt man auch $A_1 \times \dots \times A_n$ anstelle von $\mathsf{X}_{i \in I} A_i$ und A^n anstelle von A^I .

Sei I eine Menge. Die **disjunkte Vereinigung** (der Komponenten) einer I -sortigen Menge $A = (A_i)_{i \in I}$ ist wie folgt definiert:

$$\biguplus_{i \in I} A_i = \bigcup_{i \in I} (A_i \times \{i\}).$$

Die disjunkte Vereinigung des leeren Mengentupels wird definiert als die leere Menge.

Gibt es eine Menge A mit $A_i = A$ für alle $i \in I$, dann stimmt $\biguplus_{i \in I} A_i$ offenbar mit $A \times I$ überein.

Im Fall $I = [n]$ für ein $n > 1$ schreibt man auch $A_1 + \dots + A_n$ anstelle von $\biguplus_{i \in I} A_i$.

Der **Plusoperator** $+$ und der **Sternoperator** $*$ bilden jede Menge A , die das **leere Wort** ϵ nicht enthält, auf die Menge der (nichtleeren) **Wörter** oder **Listen über** A ab:

$$\begin{aligned} A^+ &=_{def} \bigcup_{n>0} A^n, \\ 1 &=_{def} \{\epsilon\}, \\ A^* &=_{def} 1 \cup A^+. \end{aligned}$$

Die Folge der Komponenten eines Elements von A^+ wird manchmal (z.B. in Haskell) nicht wie oben mit runden, sondern mit eckigen Klammern eingeschlossen oder (wie in der Theorie formaler Sprachen) zusammen mit den Kommas ganz weggelassen.

Teilmengen von A^* werden auch **Sprachen über** A genannt.

Die **Konkatenationen** $\cdot : A^* \times A^* \rightarrow A^*$ und $\cdot : \mathcal{P}(A^*) \times \mathcal{P}(A^*) \rightarrow \mathcal{P}(A^*)$ sind wie folgt induktiv definiert: Für alle $s, v = (a_1, \dots, a_m), w = (b_1, \dots, b_n) \in A^*$ und $B, C \subseteq A^*$,

$$\epsilon \cdot w = w,$$

$$v \cdot \epsilon = v,$$

$$v \cdot w = (a_1, \dots, a_m, b_1, \dots, b_n),$$

$$B \cdot C = \{v \cdot w \mid v \in B, w \in C\}.$$

Für alle $w \in A^+$ bezeichnet $\text{head}(w)$ das erste Element von w und $\text{tail}(w)$ die Restliste, d.h. es gilt $w = \text{head}(w) \cdot \text{tail}(w)$.

$|w|$ bezeichnet die **Länge** eines Wortes w , d.h. die Anzahl seiner Elemente.

$L \subseteq A^*$ heißt **präfixabgeschlossen**, wenn für alle $w \in A^*$ und $a \in A$ aus $wa \in L$ $w \in L$ folgt.

Eine partielle Funktion $t : A^* \rightharpoonup B$ mit präfixabgeschlossenem Definitionsbereich $\text{def}(t)$ heißt (kanten-) **markierter Baum** (*labelled tree*) über (A, B) .

t ist **wohlfundiert**, wenn es $n \in \mathbb{N}$ gibt mit $|w| \leq n$ für alle $w \in \text{def}(t)$. Anschaulich gesprochen ist t wohlfundiert, wenn t endliche Tiefe hat, wenn also alle von der Wurzel ausgehenden Pfade endlich sind.

Tatsächlich kann man sich t als (möglicherweise unendlichen) Baum mit Kantenmarkierungen aus A und Knotenmarkierungen aus B vorstellen, der keine zwei Kanten mit derselben Quelle, aber verschiedener Markierung besitzt. $t(\epsilon) \in B$ ist die Markierung der Wurzel und jedes Wort $w \in A^*$ entspricht einem von der Wurzel ausgehenden Pfad, der in einem Knoten endet, der mit $t(w)$ markiert ist.

Demnach kann ein markierter Baum als eine Art Record notiert werden:

$$t = t(\epsilon)\{x \rightarrow \lambda w. t(xw) \mid x \in \text{def}(t) \cap A\}.$$

Die Menge aller markierten Bäume über (A, B) wird mit $\text{ltr}(A, B)$ bezeichnet, die Menge der wohlfundierten darunter mit $\text{wtr}(A, B)$.

Produkte und Summen als universelle Konstruktionen

Sei I eine nichtleere Menge und A eine I -sortige Menge.

Das kartesische Produkt und die disjunkte Vereinigung eines Mengentupels A haben bestimmte **universelle Eigenschaften**, d.h. erstens, dass alle Mengen, die diese Eigenschaften haben, isomorph sind und zweitens, dass jede Menge, die zu einer anderen Menge, die diese Eigenschaften hat, isomorph ist, selbst diese Eigenschaften hat.

Jede Menge, die die universellen Eigenschaften des kartesischen Produkts bzw. der disjunktten Vereinigung von A hat, nennt man *Produkt* bzw. *Summe von A*:

Sei $A = (A_i)_{i \in I}$ ein Mengentupel, P eine Menge und $\pi = (\pi_i : P \rightarrow A_i)_{i \in I}$ ein Funktions-tupel.

Das Paar (P, π) heißt **Produkt von A**, wenn es für alle Tupel $(f_i : B \rightarrow A_i)_{i \in I}$ genau eine Funktion $f : B \rightarrow P$ gibt derart, dass für alle $i \in I$ Folgendes gilt:

$$\pi_i \circ f = f_i \tag{1}$$

π_i heißt *i-te Projektion von P* und g **Produktextension** oder **Range-Tuplung von f**. g wird mit $\langle f_i \rangle_{i \in I}$ und im Fall $I = [n]$, $n > 1$, auch mit $\langle f_1, \dots, f_n \rangle$ bezeichnet.

Demnach gilt:

$$(\forall i \in I : \pi_i \circ f = \pi_i \circ g) \Rightarrow f = g. \quad (2)$$

Mit $f = \lambda x.a : 1 \rightarrow P$ und $g = \lambda x.b : 1 \rightarrow P$ folgt aus (2), dass zwei Elemente $a, b \in P$ genau dann gleich sind, wenn für alle $i \in I$ $\pi_i(a) = \pi_i(b)$ gilt.

Beweis.

$$\forall i \in I : \pi_i(a) = \pi_i(b) \Rightarrow \forall i \in I : \pi_i \circ f = \pi_i \circ g \stackrel{(2)}{\Rightarrow} f = g \Rightarrow a = f(\epsilon) = g(\epsilon) = b. \quad \square$$

$\bigtimes_{i \in I} A_i$ ist ein Produkt von A .

Die Projektionen und Produkttextensionen für $\bigtimes_{i \in I} A_i$ sind wie folgt definiert:

- Für alle $i \in I$ und $f \in \bigtimes_{i \in I} A_i$, $\pi_i(f) =_{\text{def}} f(i)$.
- Für alle $(f_i : B \rightarrow A_i)_{i \in I}$, $b \in B$ und $i \in I$, $\langle f_i \rangle_{i \in I}(b)(i) =_{\text{def}} f_i(b)$.

Satz 2.2 Produkte sind *bis auf Isomorphie* eindeutig:

- (i) Seien (P, π) und (P', π') Produkte von A . Dann sind P und P' isomorph.
- (ii) Sei (P, π) ein Produkt von A , P' eine Menge und $h : P' \rightarrow P$ eine bijektive Abbildung. Dann ist (P', π') mit $\pi' = (\pi_i \circ h)_{i \in I}$ ebenfalls ein Produkt von A .

Beweis von (i). $\langle \rangle'$ bezeichnet den Extensionsoperator von (P', π') .

Sei $h =_{def} \langle \pi_i \rangle'_{i \in I} : P \rightarrow P'$ und $h' =_{def} \langle \pi'_i \rangle_{i \in I} : P' \rightarrow P$. Dann gilt

$$\begin{aligned}\pi_i \circ h' \circ h &= \pi_i \circ \langle \pi'_i \rangle_{i \in I} \circ \langle \pi_i \rangle'_{i \in I} \stackrel{(1)}{=} \pi'_i \circ \langle \pi_i \rangle'_{i \in I} \stackrel{(1)}{=} \pi_i, \\ \pi'_i \circ h \circ h' &= \pi'_i \circ \langle \pi_i \rangle'_{i \in I} \circ \langle \pi'_i \rangle_{i \in I} \stackrel{(1)}{=} \pi_i \circ \langle \pi'_i \rangle_{i \in I} \stackrel{(1)}{=} \pi'_i\end{aligned}$$

für alle $i \in I$, also $h' \circ h = id_P$ und $h \circ h' = id_{P'}$ wegen (2).

Beweis von (ii). Siehe [34], Kapitel 2.

□

Sei A eine Menge. Eine Funktion $f : \prod_{i \in I} B_i \rightarrow B$ wird wie folgt zur Funktion

$$lift^A(f) : \prod_{i \in I} B_i^A \rightarrow B^A$$

geliftet: Für alle Funktionstupel $(g_i : A \rightarrow B_i)_{i \in I}$,

$$lift^A(f)((g_i)_{i \in I}) =_{def} f \circ \langle g_i \rangle_{i \in I}.$$

Sei (P, π) ein Produkt von $(A_i)_{i \in I}$, (P', π') ein Produkt von $(B_i)_{i \in I}$ und

$$f = (f_i : A_i \rightarrow B_i)_{i \in I}.$$

Die Funktion

$$\prod_{i \in I} f_i =_{\text{def}} \langle f_i \circ \pi_i \rangle : P \rightarrow P'$$

heißt **Produkt von** f .

Für alle nichtleeren Mengen I , $f : A \rightarrow B$ und $n > 0$,

$$\begin{aligned} f^I &=_{\text{def}} \prod_{i \in I} f, \\ f_1 \times \cdots \times f_n &=_{\text{def}} \prod_{i \in [n]} f_i. \end{aligned}$$

Folgerungen:

Für alle $f : A \rightarrow B$, $(f_i : B \rightarrow B_i)_{i \in I}$, $(g_i : A_i \rightarrow B_i)_{i \in I}$, $k \in I$ und $(h_i : B_i \rightarrow A_i)_{i \in I}$,

$$\begin{aligned} \langle f_i \rangle_{i \in I} \circ f &= \langle f_i \circ f \rangle_{i \in I}, \\ \pi_k \circ \prod_{i \in I} g_i &= g_k \circ \pi_k, \\ (\prod_{i \in I} h_i) \circ \langle f_i \rangle_{i \in I} &= \langle h_i \circ f_i \rangle_{i \in I}. \end{aligned}$$

Vom Produkt kommt man zur Summe, indem man alle Funktionspfeile umdreht:

Sei $A = (A_i)_{i \in I}$ ein Mengentupel, S eine Menge und $\iota = (\iota_i : A_i \rightarrow S)_{i \in I}$ ein Funktionstupel.

Das Paar (S, ι) heißt **Summe** oder **Coproduct von** A , wenn es für alle Tupel $(f_i : A_i \rightarrow B)_{i \in I}$ genau eine Funktion $f : S \rightarrow B$ gibt mit

$$f \circ \iota_i = f_i \quad (3)$$

für alle $i \in I$.

ι_i heißt **i -te Injektion von** S und g **Summenextension** oder **Domain-Tuplung von** f . g wird mit $[f_i]_{i \in I}$ und im Fall $I = [n]$, $n > 1$, auch mit $[f_1, \dots, f_n]$ bezeichnet.

Demnach gilt:

$$(\forall i \in I : f \circ \iota_i = g \circ \iota_i) \Rightarrow f = g. \quad (4)$$

Aus (4) folgt, dass für alle $a \in S$ genau ein Paar (b, i) mit $\iota_i(b) = a$ existiert.

Beweis. Gäbe es $a \in S \setminus \bigcup_{i \in I} \iota_i(A_i)$, dann würden $f = \lambda x.0 : S \rightarrow 2$ und

$$g = (\lambda x. \text{if } x \in S \setminus \{a\} \text{ then } 0 \text{ else } 1) : S \rightarrow 2$$

(4) verletzen, weil für alle $i \in I$ und $b \in A_i$ Folgendes gilt:

$$(f \circ \iota_i)(b) = f(\iota_i(b)) = 0 = g(\iota_i(b)) = (g \circ \iota_i)(b).$$

Für alle $i \in I$ und $a \in A_i$ sei $f_i(a) = (a, i)$. Dann gilt für alle $i, j \in I$, $a \in A_i$ und $b \in A_j$ mit $\iota_i(a) = \iota_j(b)$:

$$(a, i) = f_i(a) \stackrel{(3)}{=} [f_i]_{i \in I}(\iota_i(a)) = [f_i]_{i \in I}(\iota_j(b)) \stackrel{(3)}{=} f_j(b) = (b, j). \quad \square$$

$\biguplus_{i \in I} A_i$ ist eine Summe von A .

Die Injektionen und Summenextensionen für $\biguplus_{i \in I} A_i$ sind wie folgt definiert:

- Für alle $i \in I$ und $a \in A_i$, $\iota_i(a) =_{\text{def}} (a, i)$.
- Für alle $(f_i : A_i \rightarrow B)_{i \in I}$, $i \in I$ und $a \in A_i$, $[f_i]_{i \in I}(a, i) =_{\text{def}} f_i(a)$.

Satz 2.3 Summen sind *bis auf Isomorphie* eindeutig:

- (i) Seien (S, ι) und (S', ι') Summen von A . Dann sind P und P' isomorph.
- (ii) Sei (S, ι) eine Summe von A , S' eine Menge und $h : S \rightarrow S'$ eine bijektive Abbildung. Dann ist (S', ι') mit $\iota' = (h \circ \iota_i)_{i \in I}$ ebenfalls eine Summe von A .

Beweis. Analog zu Satz 2.2. Es müssen nur alle Funktionspfeile umgedreht werden. □

Sei A eine Menge. Eine Funktion $f : B \rightarrow \coprod_{i \in I} B_i$ wird wie folgt zur Funktion

$$\text{lift}_A(f) : \prod_{i \in I} A^{B_i} \rightarrow A^B$$

geliftet: Für alle Funktionstupel $(g_i : B_i \rightarrow A)_{i \in I}$,

$$\text{lift}_A(f)((g_i)_{i \in I}) =_{\text{def}} [g_i]_{i \in I} \circ f.$$

Sei (S, ι) eine Summe von $(A_i)_{i \in I}$, (S', ι') eine Summe von $(B_i)_{i \in I}$ und

$$f = (f_i : A_i \rightarrow B_i)_{i \in I}.$$

Die Funktion

$$\coprod_{i \in I} f_i =_{\text{def}} [\iota'_i \circ f_i] : S \rightarrow S'$$

heißt **Summe von f** .

Für alle nichtleeren Mengen I , $f : A \rightarrow B$ und $n > 0$,

$$\begin{aligned} f \times I &=_{\text{def}} \coprod_{i \in I} f, \\ f_1 + \cdots + f_n &=_{\text{def}} \coprod_{i \in [n]} f_i, \\ f^+ &=_{\text{def}} \coprod_{n \in \mathbb{N}} f^{[n]}, \\ f^* &=_{\text{def}} 1 + f^+ =_{\text{def}} id_1 + f^+. \end{aligned}$$

Folgerungen:

Für alle $(f_i : A_i \rightarrow A)_{i \in I}$, $f : A \rightarrow B$, $(g_i : A_i \rightarrow B_i)_{i \in I}$, $k \in I$ und $(h_i : B_i \rightarrow A_i)_{i \in I}$,

$$\begin{aligned} f \circ [f_i]_{i \in I} &= [f \circ f_i]_{i \in I}, \\ (\coprod_{i \in I} g_i) \circ \iota_k &= \iota_k \circ g_k, \\ [f_i]_{i \in I} \circ \coprod_{i \in I} h_i &= [f_i \circ h_i]_{i \in I}. \end{aligned}$$

Typen und Signaturen

Sei S eine Menge von – **Sorten** genannten – Symbolen.

Die Klasse $\mathcal{T}_p(S)$ der **polynomialen Typen über S** ist wie folgt induktiv definiert:

- $S \subseteq \mathcal{T}_p(S)$.
- Jede nichtleere Menge ist ein polynomialer Typ.
- Für alle nichtleeren Mengen I und $(e_i)_{i \in I} \in \mathcal{T}_p(S)^I$, $\prod_{i \in I} e_i, \coprod_{i \in I} e_i \in \mathcal{T}_p(S)$.

Ein Typ der Form $\prod_{i \in I} e_i$ heißt **I -stelliger Produkttyp** mit den **Faktoren** e_i , $i \in I$.

Ein Typ der Form $\coprod_{i \in I} e_i$ heißt **I -stelliger Summentyp** mit den **Summanden** e_i , $i \in I$.

Für alle $n > 0$, $e_1, \dots, e_n, e \in \mathcal{T}_p(S)$ und nichtleeren Mengen I ,

$$\begin{aligned} e_1 \times \cdots \times e_n &=_{\text{def}} \prod_{i \in [n]} e_i, \\ e_1 + \cdots + e_n &=_{\text{def}} \coprod_{i \in [n]} e_i, \\ e^I &=_{\text{def}} \prod_{i \in I} e, \\ e^n &=_{\text{def}} e^{[n]}, \\ e^+ &=_{\text{def}} \coprod_{n > 0} e^n, \\ e^* &=_{\text{def}} 1 + e^+. \end{aligned}$$

Eine S -sortige Menge A wird wie folgt zur $\mathcal{T}_p(S)$ -sortigen Menge erweitert: Für alle nichtleeren Mengen I und $(e_i)_{i \in I} \in \mathcal{T}_p(S)^I$,

$$\begin{aligned} A_I &= I, \\ A_{\prod_{i \in I} e_i} &= \bigtimes_{i \in I} A_{e_i}, \\ A_{\coprod_{i \in I} e_i} &= \biguplus_{i \in I} A_{e_i}. \end{aligned}$$

Für alle $e \in \mathcal{T}_p(S)$ und $a \in A_e$ nennen wir e den **Typ von a** .

Eine S -sortige Funktion $h : A \rightarrow B$ wird wie folgt zur $\mathcal{T}_p(S)$ -sortigen Menge erweitert: Für alle nichtleeren Mengen I und $(e_i)_{i \in I} \in \mathcal{T}_p(S)^I$,

$$\begin{aligned} h_I &= id_I, \\ h_{\prod_{i \in I} e_i} &= \prod_{i \in I} h_{e_i}, \\ h_{\coprod_{i \in I} e_i} &= \coprod_{i \in I} h_{e_i}. \end{aligned}$$

Aufgabe

Zeigen Sie, dass für alle S -sortigen Mengen A , S -sortigen Funktionen $h : A \rightarrow B$, $n > 0$, nichtleeren Mengen I und $e_1, \dots, e_n, e \in \mathcal{T}_p(S)$ Folgendes gilt:

$$A_{e_1 \times \dots \times e_n} = A_{e_1} \times \dots \times A_{e_n},$$

$$A_{e_1 + \dots + e_n} = A_{e_1} + \dots + A_{e_n},$$

$$A_{e^I} = A_e^I,$$

$$A_{e^n} = A_e^n,$$

$$A_{e^+} = A_e^+,$$

$$A_{e^*} = A_e^*,$$

$$h_{e_1 \times \dots \times e_n} = h_{e_1} \times \dots \times h_{e_n},$$

$$h_{e_1 + \dots + e_n} = h_{e_1} + \dots + h_{e_n},$$

$$h_{e^I} = h_e^I,$$

$$h_{e^n} = h_e^{[n]},$$

$$h_{e^+} = h_e^+,$$

$$h_{e^*} = h_e^*. \quad \square$$

Eine **Signatur** $\Sigma = (S, F)$ besteht aus einer Menge S von Sorten wie oben sowie einer Menge F typisierter Funktionssymbole $f : e \rightarrow e'$ mit $e, e' \in \mathcal{T}_p(S)$, den **Operationen** von Σ .

$\text{obs}(\Sigma)$ bezeichnet die Menge der **beobachtbaren Typen** (*observable types*) von Σ , das sind alle nichtleeren Mengen, die in Typen von Operationen von Σ vorkommen.

Für alle $f : e \rightarrow e' \in F$ heißt $\text{dom}(f) = e$ **Domain** und $\text{ran}(f) = e'$ **Range** von f .

Σ ist eine **Gentzen-Signatur**, falls es für alle $f \in F$ Mengen I, J gibt derart, dass $\text{dom}(f)$ ein I -stelliger Produkttyp ist, $\text{ran}(f)$ ein I -stelliger Summentyp ist und alle Faktoren von $\text{dom}(f)$ sowie alle Summanden von $\text{ran}(f)$ Sorten oder Basismengen sind. $\text{ar}(f) =_{\text{def}} I$ und $\text{coar}(f) =_{\text{def}} J$ heißen **Arität** bzw. **Coaritität** von f .

Sei Σ eine Gentzen-Signatur. $f \in F$ heißt **Konstruktor** bzw. **Destruktor**, falls $\text{ran}(f)$ bzw. $\text{dom}(f)$ eine Sorte ist. Σ ist **konstruktiv** bzw. **destruktiv**, falls F aus Konstruktoren bzw. Destruktoren besteht.

Konstruktoren dienen der **Synthese** von Elementen einer S -sortigen Menge, Destruktoren liefern Werkzeuge zu ihrer **Analyse**.

Die komponentenweise Vereinigung zweier Signaturen Σ und Σ' wird mit $\Sigma \cup \Sigma'$ bezeichnet.

Die abstrakte Syntax einer kontextfreien Grammatik (siehe Kapitel 4) ist eine konstruktive Signatur, während Parser, Interpreter und Compiler auf Automatenmodellen beruhen, die destruktive Signaturen interpretieren.

Die syntaktische Beschreibung jedes mathematischen Modells, das auf *induktiv* definierten Mengen basiert, kann als konstruktive Signatur formuliert werden.

Solchen Modellen stehen die *zustandsorientierten* gegenüber, die Objekte nicht anhand ihres Aufbaus, sondern ausschließlich durch Beobachtung ihres Verhaltens voneinander unterscheiden. Dementsprechend werden aus den Operationen der jeweils zugrundeliegenden destruktiven Signatur keine Objekte, sondern Beobachtungsinstrumente zusammengesetzt.

In den folgenden Beispielen steht hinter  die Trägermenge der initialen bzw. finalen Algebra der jeweiligen Signatur (siehe 2.6 und 2.12).

Seien $X, Y, Act, X_1, \dots, X_n, Y_1, \dots, Y_n, E_1, \dots, E_n$ beliebige Mengen, BL eine endliche Menge – **Basissprachen** (*base languages*) genannter – Mengen, die \emptyset und 1 enthält, und

$$BS = \{X_1, \dots, X_n, Y_1, \dots, Y_n, E_1, \dots, E_n\}.$$

2.4 Konstruktive Signaturen

- $\textcolor{brown}{Mon} \Leftrightarrow$ Unmarkierte binäre Bäume (Monoide sind Mon -Algebren; siehe 2.6.)

$$S = \{mon\}, \quad F = \{ \begin{aligned} & one : 1 \rightarrow mon, \\ & mul : mon \times mon \rightarrow mon \end{aligned} \}.$$

- $\textcolor{brown}{Nat} \Leftrightarrow \mathbb{N}$

$$S = \{nat\}, \quad F = \{ \begin{aligned} & zero : 1 \rightarrow nat, \\ & succ : nat \rightarrow nat \end{aligned} \}.$$

- $\textcolor{brown}{Dyn}(I, X) \Leftrightarrow I \times X^*$

$$S = \{list\}, \quad F = \{ \begin{aligned} & nil : I \rightarrow list, \\ & cons : X \times list \rightarrow list \end{aligned} \}.$$

- $\textcolor{brown}{List}(X) =_{def} Dyn(1, X) \Leftrightarrow X^*$

- $\textcolor{brown}{Bintree}(X) \Leftrightarrow$ binäre Bäume endlicher Tiefe mit Knotenmarkierungen aus X

$$S = \{btree\}, \quad F = \{ \begin{aligned} & empty : 1 \rightarrow btree, \\ & bjoin : btree \times X \times btree \rightarrow btree \end{aligned} \}.$$

- $\text{Tree}(X)$ ↪ Bäume endlicher Tiefe und endlichen Knotenausgrads mit Knotenmarkierungen aus X

$$S = \{ \text{tree}, \text{trees} \}, \quad F = \{ \begin{aligned} & \text{join} : X \times \text{trees} \rightarrow \text{tree}, \\ & \text{nil} : 1 \rightarrow \text{trees}, \\ & \text{cons} : \text{tree} \times \text{trees} \rightarrow \text{trees} \}. \end{aligned}$$

- $\text{Reg}(BL)$ ↪ reguläre Ausdrücke über BL

$$\begin{aligned} S &= \{ \text{reg} \}, \\ F &= \{ \begin{aligned} & \text{par} : \text{reg} \times \text{reg} \rightarrow \text{reg}, && \text{(parallele Komposition)} \\ & \text{seq} : \text{reg} \times \text{reg} \rightarrow \text{reg}, && \text{(sequentielle Komposition)} \\ & \text{iter} : \text{reg} \rightarrow \text{reg}, && \text{(Iteration)} \\ & \text{base} : BL \rightarrow \text{reg} \}. && \text{(Einbettung der Basissprachen)} \end{aligned} \end{aligned}$$

- $\text{CCS}(\text{Act})$ ↪ Calculus of Communicating Systems (kommunizierende Prozesse)

$$\begin{aligned} S &= \{ \text{proc} \}, \\ F &= \{ \begin{aligned} & \text{pre} : \text{Act} \times \text{proc} \rightarrow \text{proc}, && \text{(prefixing by an action)} \\ & \text{cho} : \text{proc} \times \text{proc} \rightarrow \text{proc}, && \text{(choice)} \\ & \text{par} : \text{proc} \times \text{proc} \rightarrow \text{proc}, && \text{(parallelism)} \\ & \text{res} : \text{proc} \times \text{Act} \rightarrow \text{proc}, && \text{(restriction)} \\ & \text{rel} : \text{proc} \times \text{Act}^{\text{Act}} \rightarrow \text{proc} \}. && \text{(relabelling)} \end{aligned} \end{aligned}$$

2.5 Destruktive Signaturen

- $coNat \bowtie \mathbb{N} \cup \{\infty\}$

$$S = \{nat\}, \quad F = \{pred : nat \rightarrow 1 + nat\}.$$

- $coList(X) \bowtie X^* \cup X^{\mathbb{N}}$ ($coList(1) \cong coNat$)

$$\begin{aligned} S = \{list, X \times list\}, \quad F = \{ & \text{split} : list \rightarrow 1 + X \times list, \\ & \pi_1 : X \times list \rightarrow X, \\ & \pi_2 : X \times list \rightarrow list \}. \end{aligned}$$

- $Stream(X) =_{def} DAut(1, X) \bowtie X^{\mathbb{N}}$

$$\begin{aligned} S = \{list\}, \quad F = \{ & \text{head} : list \rightarrow X, \\ & \text{tail} : list \rightarrow list \}. \end{aligned}$$

- $coBintree(X) \bowtie$ binäre Bäume beliebiger Tiefe mit Knotenmarkierungen aus X

$$\begin{aligned} S = \{ & btree, btree \times X \times btree \}, \\ F = \{ & \text{split} : btree \rightarrow 1 + btree \times X \times btree, \\ & \pi_1 : btree \times X \times btree \rightarrow btree, \\ & \pi_2 : btree \times X \times btree \rightarrow X, \\ & \pi_3 : btree \times X \times btree \rightarrow btree \}. \end{aligned}$$

- $\text{Infbintree}(X) \Leftrightarrow$ binäre Bäume unendlicher Tiefe mit Knotenmarkierungen aus X

$$S = \{btree\}, \quad F = \{ \begin{aligned} & \text{root : } btree \rightarrow X, \\ & \text{left, right : } btree \rightarrow btree \end{aligned} \}.$$

- $DAut(X, Y) \Leftrightarrow Y^{X^*}$ = Verhalten deterministischer Moore-Automaten mit Eingabe-
menge X und Ausgabemenge Y

$$\begin{aligned} S &= \{ state, state^X \}, && (\text{Zustandsmenge}) \\ F &= \{ \delta : state \rightarrow state^X, && (\text{Transitionsfunktion}) \\ &\quad \beta : state \rightarrow Y \} \cup && (\text{Ausgabefunktion}) \\ &\quad \{ \pi_x : state^X \rightarrow state \mid x \in X \}. \end{aligned}$$

- $Acc(X) =_{def} DAut(X, 2) \Leftrightarrow \mathcal{P}(X^*)$ = Wortsprachen über X

- $Proctree(Act) \Leftrightarrow$ Prozessbäume, deren Kanten mit Aktionen markiert sind

$$\begin{aligned} S &= \{ tree \} \cup \{ (Act \times tree)^n \mid n > 0 \}, \\ F &= \{ \delta : tree \rightarrow (Act \times tree)^* \} \cup \\ &\quad \{ \pi_n : (Act \times tree)^n \rightarrow Act \times tree \mid n > 0 \} \cup \\ &\quad \{ \pi_1 : Act \times tree \rightarrow Act, \\ &\quad \pi_2 : Act \times tree \rightarrow tree \}. \end{aligned}$$

- $\text{Class}(BS)$ ↳ Verhalten einer Objektklasse mit n Methoden [19]

$$S = \{ \text{ state } \},$$

$$F = \{ m_i : \text{state} \rightarrow ((Y_i \times \text{state}) + E_i)^{X_i} \mid 1 \leq i \leq n \} \cup$$

$$\{ \pi_{i,x} : ((Y_i \times \text{state}) + E_i)^{X_i} \rightarrow (Y_i \times \text{state}) + E_i \mid 1 \leq i \leq n, x \in X_i \} \cup$$

$$\{ \pi_{i,1} : Y_i \times \text{state} \rightarrow Y_i \mid 1 \leq i \leq n \} \cup$$

$$\{ \pi_{i,2} : Y_i \times \text{state} \rightarrow \text{state} \mid 1 \leq i \leq n \}.$$

2.6 Algebren

Sei $\Sigma = (S, F)$ eine Signatur. Eine Σ -**Algebra** $\mathcal{A} = (A, Op)$ besteht aus einer S -sortigen Menge A und einer F -sortigen Menge

$$Op = (f^{\mathcal{A}} : A_e \rightarrow A_{e'})_{f:e \rightarrow e' \in F}$$

von Funktionen, den **Operationen** von \mathcal{A} .

Für alle $s \in S$ heißt A_s **Trägermenge** (*carrier set*) oder **Interpretation von s in \mathcal{A}** .

Für alle $f : e \rightarrow e' \in F$ heißt $f^{\mathcal{A}} : A_e \rightarrow A_{e'}$ **Interpretation von f in \mathcal{A}** .

Alg_{Σ} bezeichnet die Klasse aller Σ -Algebren.

Algebren destruktiver Signaturen werden auch **Coalgebren** genannt.

Seien \mathcal{A}, \mathcal{B} Σ -Algebren. Eine S -sortige Funktion $h : \mathcal{A} \rightarrow \mathcal{B}$ heißt Σ -**Homomorphismus**, wenn für alle $f : e \rightarrow e' \in F$

$$h_{e'} \circ f^{\mathcal{A}} = f^{\mathcal{B}} \circ h_e$$

gilt. Ist h bijektiv, dann heißt h Σ -**Isomorphismus** und \mathcal{A} und \mathcal{B} sind Σ -**isomorph**.

h induziert die **Bildalgebra** $h(\mathcal{A})$:

- Für alle $e \in \mathcal{T}_p(S)$, $h(\mathcal{A})_e =_{def} h_e(A_e)$.
- Für alle $f : e \rightarrow e' \in F$ und $a \in A_e$, $f^{h(\mathcal{A})}(h(a)) =_{def} f^{\mathcal{B}}(h(a))$.

Algebra-Beispiele

Die Menge \mathbb{N} der natürlichen Zahlen ist Trägermenge der gleichnamigen *Nat*-Algebra \mathbb{N} , deren Operationen

$$\text{zero}^{\mathbb{N}} : 1 \rightarrow \mathbb{N}, \text{ succ}^{\mathbb{N}} : \mathbb{N} \rightarrow \mathbb{N}$$

wie folgt definiert sind: Für alle $n \in \mathbb{N}$,

$$\begin{aligned}\text{zero}^{\mathbb{N}}(\epsilon) &= 0, \\ \text{succ}^{\mathbb{N}}(n) &= n + 1.\end{aligned}$$

\mathbb{N} ist auch Trägermenge der *List(1)*-Algebra *Unary*, deren Operationen

$$\text{nil}^{\text{Unary}} : 1 \rightarrow \mathbb{N}, \text{ cons}^{\text{Unary}} : 1 \times \mathbb{N} \rightarrow \mathbb{N}$$

wie folgt definiert sind: Für alle $n \in \mathbb{N}$,

$$\begin{aligned}\text{nil}^{\text{Unary}}(\epsilon) &= 0, \\ \text{cons}^{\text{Unary}}(\epsilon, n) &= n + 1.\end{aligned}$$

Die Menge X^* der Wörter über einer Menge X ist Trägermenge der gleichnamigen *List(X)*-Algebra X^* , deren Operationen

$$\text{nil}^{X^*} : 1 \rightarrow X^*, \text{ cons}^{X^*} : X \times X^* \rightarrow X^*$$

wie folgt definiert sind: Für alle $x \in X$ und $w \in X^*$,

$$\begin{aligned} nil^{X^*}(\epsilon) &= \epsilon, \\ cons^{X^*}(x, w) &= xw, \\ one^{Word(X)}(\epsilon) &= \epsilon, \\ mul^{Word(X)}(v, w) &= vw. \end{aligned}$$

X^* ist auch Trägermenge der Mon -Algebra $\text{Word}(X)$, deren Operationen

$$one^{Word(X)} : 1 \rightarrow X^*, \quad mul^{Word(X)} : X^* \times X^* \rightarrow X^*$$

wie folgt definiert sind: Für alle $v, w \in X^*$,

$$\begin{aligned} one^{Word(X)}(\epsilon) &= \epsilon, \\ mul^{Word(X)}(v, w) &= vw. \end{aligned}$$

Die Menge X^X der Funktionen von X nach X ist Trägermenge der Mon -Algebra $\text{Endo}(X)$, deren Operationen

$$one^{Endo(X)} : 1 \rightarrow X^X, \quad mul^{Endo(X)} : X^X \times X^X \rightarrow X^X$$

wie folgt definiert ist: Für alle $f, g : X \rightarrow X$,

$$\begin{aligned} one^{Endo(X)}(\epsilon) &= id_X, \\ mul^{Endo(X)}(f, g) &= g \circ f. \end{aligned}$$

Eine Mon -Algebra \mathcal{A} heißt **Monoid**, wenn $mul^{\mathcal{A}}$ assoziativ und $one^{\mathcal{A}}$ ein links- und rechts-neutrales Element bzgl. $mul^{\mathcal{A}}$ ist. Demnach sind $\text{Word}(X)$ und $\text{Endo}(X)$ Monoide.

Die Menge der Funktionen von \mathbb{N} in eine Menge X ist Trägermenge der $\text{Stream}(X)$ -Algebra $\text{Seq}(X)$, deren Operationen

$$\text{head}^{\text{Seq}(X)} : X^{\mathbb{N}} \rightarrow X, \quad \text{tail}^{\text{Seq}(X)} : X^{\mathbb{N}} \rightarrow X^{\mathbb{N}}$$

wie folgt definiert sind: Für alle $f : \mathbb{N} \rightarrow X$,

$$\begin{aligned}\text{head}^{\text{Seq}(X)}(f) &= f(0), \\ \text{tail}^{\text{Seq}(X)}(f) &= \lambda n. f(n+1).\end{aligned}$$

Die folgende $\text{Stream}(\mathbb{Z})$ -Algebra zo repräsentiert die Ströme $0, 1, 0, 1, \dots$ und $1, 0, 1, 0, \dots$:

$$\begin{aligned}zo_{list} &= \{Blink, Blink'\}, \\ \text{head}^{zo}(Blink) &= 0, \\ \text{tail}^{zo}(Blink) &= Blink', \\ \text{head}^{zo}(Blink') &= 1, \\ \text{tail}^{zo}(Blink') &= Blink.\end{aligned}$$

Die folgende $\text{Acc}(\mathbb{Z})$ -Algebra eo dient der Erkennung der Parität der Summe von Zahlenfolgen (siehe Beispiel 2.19) und ist wie folgt definiert:

$$\begin{aligned}eo_{state} &= \{Esum, Osum\}, \\ \delta^{eo}(Esum) &= \lambda x. \text{if even}(x) \text{ then } Esum \text{ else } Osum, \\ \delta^{eo}(Osum) &= \lambda x. \text{if odd}(x) \text{ then } Esum \text{ else } Osum, \\ \beta^{eo} &= \lambda st. \text{if } st = Esum \text{ then } 1 \text{ else } 0.\end{aligned}$$

□

2.7 Terme und Coterme

Im Folgenden verwenden wir die im Abschnitt **Mengen und Typen** eingeführte Notation für deterministische Bäume.

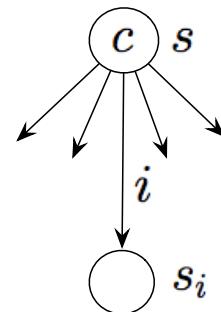
Sei $\Sigma = (S, C)$ eine **konstruktive** Signatur, $X = \bigcup obs(\Sigma)$ und V eine S -sortige Menge von “Variablen”.

Die Menge $CT_{\Sigma}(V)$ der **Σ -Terme über V** ist die **größte** $(S \cup obs(\Sigma))$ -sortige Menge M deterministische Bäume über $(X, C \cup X \cup V)$ mit folgenden Eigenschaften:

- Für alle $B \in obs(\Sigma)$, $M_B = B$. (1)
- Für alle $s \in S$ und $t \in M_s$ ist $t \in V_s$ oder gibt es $c : \prod_{i \in I} s_i \rightarrow s \in C$ and $(t_i)_{i \in I} \in \bigtimes_{i \in I} M_{s_i}$ mit $t = c\{i \rightarrow t_i \mid i \in I\}$. (2/3)

b B

x s



(1)

$$\begin{aligned} b &\in B \\ B &\in \text{obs}(\Sigma) \end{aligned}$$

(2/4)

$$\begin{aligned} s &\in S \\ x &\in V_s \end{aligned}$$

(3/5)

$$c : \prod_{i \in I} s_i \rightarrow s \in C$$

Die Elemente von $\textcolor{red}{CT}_\Sigma =_{def} CT_\Sigma(\lambda s.\emptyset)$ heißen **Σ -Grundterme**.

$T_\Sigma(V)$ bezeichnet die **kleinste** ($S \cup \text{obs}(\Sigma)$)-sortige Menge $\textcolor{blue}{M}$ wohlfundierter deterministische Bäume über $(X, C \cup X \cup V)$ mit (1) und folgenden Eigenschaften:

- Für alle $s \in S$, $V_s \subseteq \textcolor{blue}{M}_s$. (4)
- Für alle $s \in S$, $c : \prod_{i \in I} s_i \rightarrow s \in C$ und $(t_i)_{i \in I} \in \textcolor{blue}{X}_{i \in I} \textcolor{blue}{M}_{s_i}$, $c\{i \rightarrow t_i \mid i \in I\} \in \textcolor{blue}{M}_s$. (5)

Für alle $n > 0$ und $c : s_1 \times \dots \times s_n \rightarrow s \in C$ schreibt man üblicherweise $c(t_1, \dots, t_n)$ anstelle von $c\{1 \rightarrow t_1, \dots, n \rightarrow t_n\}$.

$T_\Sigma(V)$ und $CT_\Sigma(V)$ sind die Trägermengen von Σ -Algebren (s.u.). Darüberhinaus sind wohlfundierte Σ -Terme Bestandteile von Formeln wie z.B. den in Kapitel 18 iterativen Gleichungen wie (1) in Abschnitt 2.8.

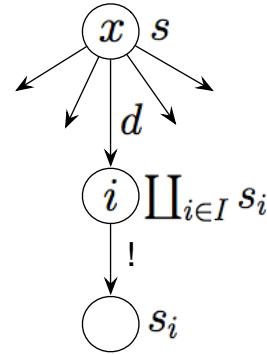
Sei $\Sigma = (S, D)$ eine **destruktive** Signatur und V eine S -sortige Menge von “Farben” oder “Covariablen”.

Die Menge $DT_{\Sigma}(V)$ der **Σ -Coterme über V** ist die **größte** ($S \cup obs(\Sigma)$)-sortige Menge M deterministische Bäume über $(D \cup \{!\}, X \cup V)$ mit (1) und folgender Eigenschaft:

- Für alle $s \in S$, $t \in M_s$ und $d : s \rightarrow \coprod_{i \in I} s_i \in D$ gibt es $x \in V_s$, $i_d \in I$ und $t_d \in M_{s_i}$ mit $t = x\{d \rightarrow i\{! \rightarrow t_d\} \mid d : s \rightarrow e \in D\}$. (6)

(b) B

$$\begin{aligned} & (1) \\ & b \in B \\ & B \in obs(\Sigma) \end{aligned}$$



$$\begin{aligned} & (6/7) \\ & x \in V_s \\ & d : s \rightarrow \coprod_{i \in I} s_i \in D \end{aligned}$$

Ist I einelementig, dann stimmt $\coprod_{i \in I} s_i$ mit s_i überein, so dass die mit $!$ markierte Kante entfällt.

Für mehrelementige Mengen I wird $(t, i) \in DT_{\Sigma}(V)_{\coprod_{i \in I} s_i}$ mit $i(t) =_{def} i\{! \rightarrow t\}$ gleichgesetzt.

Die Elemente von $\textcolor{red}{DT}_\Sigma =_{def} DT_\Sigma(\lambda s.1)$ heißen **Σ -Grundcoterme**.

$\textcolor{red}{coT}_\Sigma(V)$ bezeichnet die **kleinste** $(S \cup obs(\Sigma))$ -sortige Menge $\textcolor{blue}{M}$ wohlfundierter deterministische Bäume über $(D \cup \{!\}, X \cup V)$ mit (1) und folgender Eigenschaft:

- Für alle $s \in S, x \in V_s, d : s \rightarrow \coprod_{i \in I} s_i \in D, i_d \in I$ and $t_d \in \textcolor{blue}{M}_{s_i}$,
 $x\{d \rightarrow i(t_d) \mid d : s \rightarrow e \in D\} \in \textcolor{blue}{M}_s.$
- (7)

$\textcolor{red}{coT}_\Sigma =_{def} coT_\Sigma(\lambda s.1).$

$coT_\Sigma(V)$ und $DT_\Sigma(V)$ sind die Trägermengen von Σ -Algebren (s.u.). Im Gegensatz zu Σ -Termen kommen Σ -Coterme in Formeln nicht vor. Stattdessen werden auch im Fall einer destruktiven Signatur nur wohlfundierte Σ -Terme in Formeln verwendet wie z.B. den Gleichungen (2) und (3) von Abschnitt 2.8.

2.8 Beispiele

Sei $\Sigma = \text{Nat}$ (siehe 2.4).

$CT_{\Sigma, \text{nat}}$ ist die größte Teilmenge M von $ltr(\{1\}, C)$ mit folgender Eigenschaft:

- Für alle $t \in M$ gilt $t = \text{zero}$ oder gibt es $u \in M$ mit $t = \text{succ}(u)$.

$T_{\Sigma, \text{nat}}$ ist die kleinste Teilmenge M von $ltr(\{1\}, C)$ mit folgenden Eigenschaften:

- $\text{zero} \in M$.
- Für alle $t \in M$, $\text{succ}(t) \in M$.

Sei $\Sigma = \text{List}(X)$ (siehe 2.4).

$CT_{\Sigma, \text{list}}$ ist die größte Teilmenge M von $ltr(\{1, 2\}, C \cup X)$ mit folgender Eigenschaft:

- Für alle $t \in M$ gilt $t = \text{nil}$ oder gibt es $x \in X$ und $u \in M$ mit $t = \text{cons}(x, u)$.

$T_{\Sigma, \text{list}}$ ist die kleinste Teilmenge M von $ltr(\{1, 2\}, C \cup X)$ mit folgenden Eigenschaften:

- $\text{nil} \in M$.
- Für alle $x \in X$ und $t \in M$, $\text{cons}(x, t) \in M$.

Sei $V = \{blink, blink'\}$. Die folgenden Gleichungen zwischen $List(\mathbb{Z})$ -Termen über V haben eine eindeutige Lösung in $CT_{List(\mathbb{Z})}$ (siehe Kapitel 18):

$$\textcolor{red}{blink} = \textcolor{blue}{cons}(0, \textcolor{red}{blink'}), \quad \textcolor{red}{blink'} = \textcolor{blue}{cons}(1, \textcolor{red}{blink}). \quad (1)$$

Deshalb definiert (1) $blink$ und $blink'$ als zwei Elemente von $CT_{List(\mathbb{Z})}$. Als eindeutige Lösungen iterativer Gleichungen repräsentierbare unendliche Terme heißen **rational**. Ein rationaler Term hat nur endlich viele Teilterme.

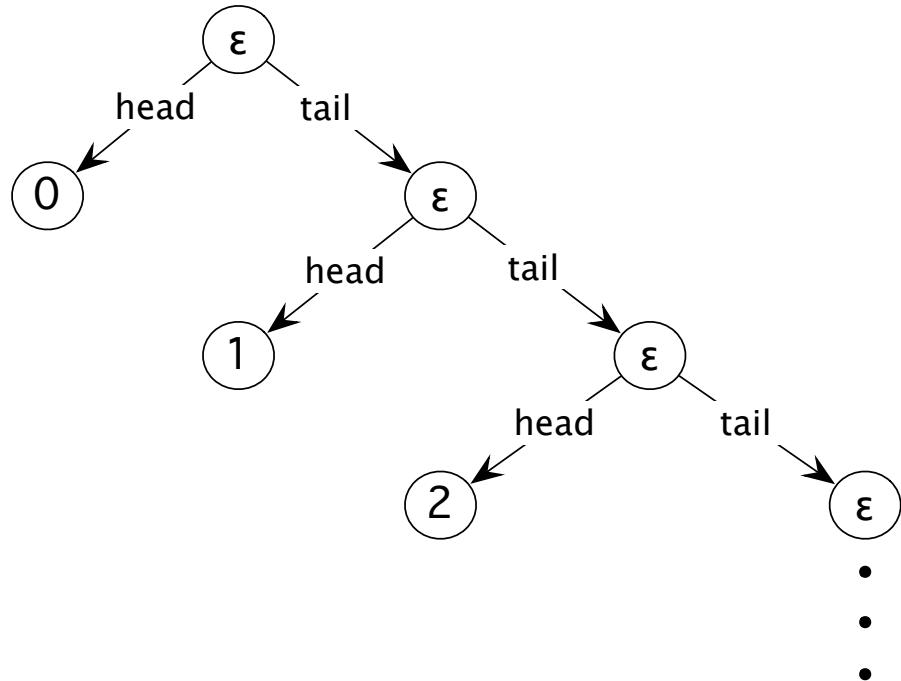
Sei $\Sigma = \textcolor{blue}{Reg(BL)}$ (siehe 2.4).

$T_{\Sigma, reg}$ ist die kleinste Teilmenge M von $ltr(\{1, 2\}, C \cup BL)$ mit folgenden Eigenschaften:

- Für alle $t, u \in M$, $\text{par}(t, u)$, $\text{seq}(t, u) \in M$.
- Für alle $t \in M$, $\text{iter}(t) \in M$.
- Für alle $B \in BL$, $\text{base}(B) \in M$.

Sei $\Sigma = \textcolor{blue}{Stream(X)}$ (siehe 2.5). $DT_{\Sigma, list}$ ist die größte Teilmenge M von $ltr(D, \{1\} \cup X \cup 1)$ mit folgender Eigenschaft:

- Für alle $t \in M$ gibt es $x \in X$ und $t' \in M$ mit $t = \epsilon\{\text{head} \rightarrow x, \text{tail} \rightarrow t'\} \in M$.



Stream(\mathbb{N})-Coterm, der den Strom aller natürlichen Zahlen darstellt

Sei $V = \{blink, blink'\}$. Da zo eine $Stream(\mathbb{Z})$ -Algebra (siehe 2.6) und $DT_{Stream(\mathbb{Z})}$ eine finale $Stream(\mathbb{Z})$ -Algebra ist (s.u.), lösen die Coterme $unfold^{zo}(Blink)$ und $unfold^{zo}(Blink')$ die folgenden Gleichungen zwischen $Stream(\mathbb{Z})$ -Termen über V eindeutig in $blink$ bzw. $blink'$:

$$blink = \epsilon\{head \rightarrow 0, tail \rightarrow blink'\}, \quad blink' = \epsilon\{head \rightarrow 1, tail \rightarrow blink\} \quad (2)$$

(siehe Kapitel 18). Deshalb definiert (2) $blink$ und $blink'$ als zwei Elemente von $DT_{Stream(\mathbb{Z})}$.

Sei $\Sigma = \text{Colist}(X)$ (siehe 2.5).

$DT_{\Sigma, list}$ ist die größte Teilmenge M von $ltr(D, \{1, 2\} \cup X \cup 1)$ mit folgender Eigenschaft:

- Für alle $t \in M$ gilt $t = \epsilon\{split \rightarrow 1(\epsilon)\}$ oder gibt es $x \in X$ und $u \in M$ mit

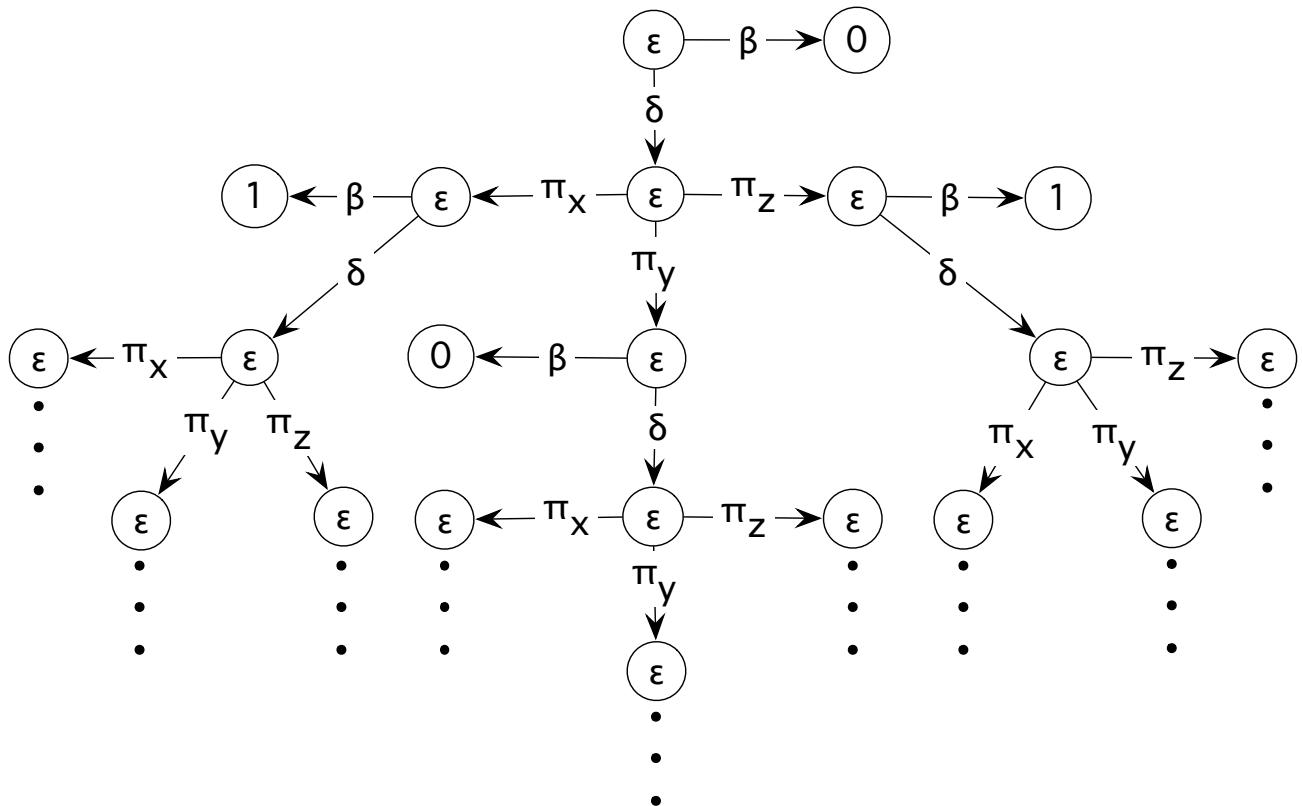
$$t = \epsilon\{split \rightarrow 2(\epsilon\{\pi_1 \rightarrow x, \pi_2 \rightarrow u\})\}.$$

Sei $\Sigma = DAut(X, Y)$ (siehe 2.5).

$DT_{\Sigma, state}$ ist die größte Teilmenge M von $ltr(D, \{1, 2\} \cup X \cup 1)$ mit folgender Eigenschaft:

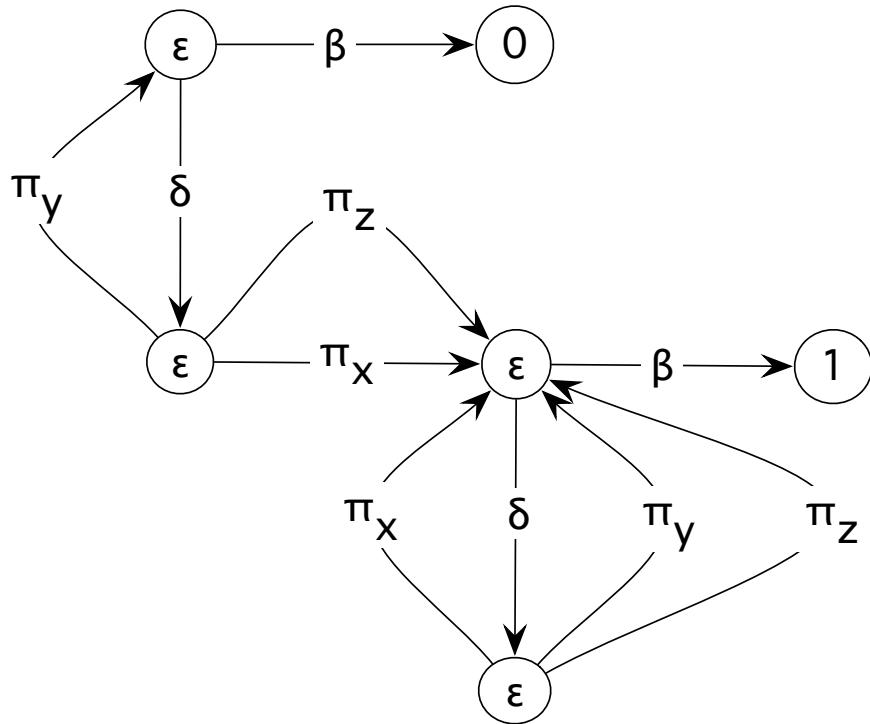
- Für alle $t \in M$ und $x \in X$ gibt es $t_x \in M$ und $y \in Y$ mit

$$t = \epsilon\{\delta \rightarrow \epsilon\{\pi_x \rightarrow t_x \mid x \in X\}, \beta \rightarrow y\}.$$



$Acc(\{x, y, z\})$ -Coterm,

der einen Akzeptor der Sprache $\{w \in \{x, y, z\}^* \mid w \text{ enthält } x \text{ oder } z\}$ repräsentiert



Darstellung des obigen $\text{Acc}(\{x, y, z\})$ -Coterms als endlicher Graph

Sei $V = \{esum, osum\}$. Da eo eine $\text{Acc}(\mathbb{Z})$ -Algebra (siehe 2.6) und $DT_{\text{Acc}(\mathbb{Z})}$ eine finale $\text{Acc}(\mathbb{Z})$ -Algebra ist (s.u.), lösen die Coterme $unfold^{eo}(Esum)$ und $unfold^{eo}(Osum)$ die folgenden iterativen Gleichungen zwischen $\text{Acc}(\mathbb{Z})$ -Coterms über V eindeutig in $esum$ bzw. $osum$:

$$\begin{aligned}
\text{esum} &= \epsilon\{\delta \rightarrow \epsilon(\{\pi_x \rightarrow \text{esum} \mid \text{even}(x)\} \cup \{\pi_x \rightarrow \text{osum} \mid \text{odd}(x)\}), \beta \rightarrow 1\}, \\
\text{osum} &= \epsilon\{\delta \rightarrow \epsilon(\{\pi_x \rightarrow \text{osum} \mid \text{even}(x)\} \cup \{\pi_x \rightarrow \text{esum} \mid \text{odd}(x)\}), \beta \rightarrow 0\}.
\end{aligned} \tag{3}$$

even und *odd* bezeichnen die Mengen der geraden bzw. ungeraden ganzen Zahlen. □

$CT_\Sigma(V)$ und $DT_\Sigma(V)$ sind Σ -Algebren

Sei $\Sigma = (S, C)$ eine konstruktive Signatur.

$CT_\Sigma(V)$ wird wie folgt zur Σ -Algebra erweitert:

Für alle $c : \prod_{i \in I} s_i \rightarrow s \in C$ and $(t_i)_{i \in I} \in \bigtimes_{i \in I} CT_\Sigma(V)_{s_i}$,

$$c^{CT_\Sigma(V)}((t_i)_{i \in I}) =_{def} c\{i \rightarrow t_i \mid i \in I\}.$$

Sei $\Sigma = (S, D)$ eine destruktive Signatur.

$DT_\Sigma(V)$ wird wie folgt zur Σ -Algebra erweitert:

Für alle $s \in S$, $t \in DT_\Sigma(V)_s$, $d : s \rightarrow e$ und $w \in (D \cup \{!\})^*$,

$$d^{DT_\Sigma(V)}(t)(w) =_{def} t(dw).$$

Die Interpretation von Destruktoren in $DT_{\Sigma}(V)$ machen Coterme zu einer Art analytischer Funktionen: Zwei Coterme $t, t' \in DT_{\Sigma}(V)_s$ sind genau dann gleich, wenn die “Anfangswerte” $t(\epsilon)$ und $t'(\epsilon)$ und für alle $d : s \rightarrow e$ die “Ableitungen” $d^{DT_{\Sigma}(V)}(t)$ und $d^{DT_{\Sigma}(V)}(t')$ miteinander übereinstimmen.

Die atomaren Formeln der Prädikatenlogik sind aus Σ -Termen und Prädikaten (= Relationssymbolen) zusammengesetzt. Prinzipiell kommt man dort mit einer einzigen Sorte aus, muss dann aber die Trennung zwischen mehreren Datenbereichen durch die Einführung eines einstelligen Prädikats für jeden Datenbereich wiedergeben.

So entspricht z.B. die Sorte nat der Signatur Nat (siehe 2.4) das Prädikat $isNat$ mit den Axiomen

$$isNat(zero), \tag{1}$$

$$isNat(x) \Rightarrow isNat(succ(x)). \tag{2}$$

Im Rahmen einer Nat umfassenden Signatur Σ wäre ein Σ -Term t genau dann ein Nat -Grundterm des Typs nat , wenn $isNat(t)$ mit Inferenzregeln der Prädikatenlogik aus (1) und (2) ableitbar ist.

2.9 Die Algebren $Bool$ und $Regword(BL)$ (siehe 2.4)

Die Menge $2 = \{0, 1\}$ ist Trägermenge der $Reg(BL)$ -Algebra $Bool$:

Für alle $x, y \in 2$ und $B \in BL \setminus 1$,

$$\begin{aligned} par^{Bool}(x, y) &= \max\{x, y\}, \\ seq^{Bool}(x, y) &= x * y, \\ iter^{Bool}(x) &= 1, \\ base^{Bool}(1) &= 1, \\ base^{Bool}(B) &= 0. \end{aligned}$$

Die üblichen Wortdarstellungen regulärer Ausdrücke e wie z.B. $aab^* + c(a\bar{\mathbb{N}})^*$ bilden die $Reg(BL)$ -Algebra $Regword(BL)$:

Sei

$$syms(BL) = \{+, ^*, (,)\} \cup \{\overline{B} \mid B \in BL\}.$$

Im Fall $|B| = 1$ setzen wir \overline{B} mit dem einen Element von B gleich. Ob ein Teilausdruck e geklammert werden muss oder nicht, hängt von der Priorität des Operationssymbols f ab, zu dessen Domain e gehört. Die Trägermenge von $Regword(BL)$ besteht deshalb aus Funktionen, die, abhängig von der Priorität von f , die Wortdarstellung von e mit oder ohne Klammern zurückgibt:

$$Regword(BL)_{reg} = \mathbb{N} \rightarrow syms(BL)^*.$$

Aus den Prioritäten 0,1,2 von *par*, *seq* bzw. *iter* ergeben sich folgende Interpretation der Operationen von *Reg(BL)*:

Für alle $f, g : \mathbb{N} \rightarrow \text{syms}(BL)^*$, $B \in BL$ und $w \in \text{syms}(BL)^*$,

$$\begin{aligned} \text{par}^{\text{Regword}(BL)}(f, g) &= \lambda n. \text{enclose}(n > 0)(f(0) + g(0)), \\ \text{seq}^{\text{Regword}(BL)}(f, g) &= \lambda n. \text{enclose}(n > 1)(f(1) \cdot g(1)), \\ \text{iter}^{\text{Regword}(BL)}(f) &= \lambda n. \text{enclose}(n > 2)(f(2)^*), \\ \text{base}^{\text{Regword}(BL)}(B) &= \lambda n. \overline{B}, \\ \text{enclose}(\text{True})(w) &= (w), \\ \text{enclose}(\text{False})(w) &= w. \end{aligned}$$

Ist die Funktion $f : \mathbb{N} \rightarrow \text{syms}(BL)^*$ das Ergebnis der Faltung eines *Reg(BL)*-Terms t in *Regword(BL)* (s.u.), dann liefert $f(0)$ eine Wortdarstellung von t , die keine überflüssigen Klammern enthält.

Regword(String) ist im Haskell-Modul **Compiler.hs** durch **regWord** implementiert. □

2.10 Die Algebren $Beh(X, Y)$, $Pow(X)$, $Lang(X)$ und $Bro(BL)$ (siehe 2.5)

Seien X und Y Mengen.

Funktionen von X^* nach Y nennen wir **Verhaltensfunktionen** (*behavior functions*). Sie bilden die Trägermenge folgender $DAut(X, Y)$ -Algebra $Beh(X, Y)$:

$$Beh(X, Y)_{state} = Y^{X^*}.$$

Für alle $f : X^* \rightarrow Y$, $x \in X$ und $w \in X^*$,

$$\delta^{Beh(X, Y)}(f)(x)(w) = f(xw) \quad \text{und} \quad \beta^{Beh(X, Y)}(f) = f(\epsilon).$$

$Beh(X, 2)$ ist im Haskell-Modul `Compiler.hs` durch `behFun` implementiert.

Eine zu $Beh(X, 2)_{state} = 2^{X^*}$ isomorphe Menge ist die Potenzmenge $\mathcal{P}(X^*)$ (siehe [Mengen und Typen](#)). Daraus ergibt sich die $Acc(X)$ -Algebra $Pow(X)$ mit

$$Pow(X)_{state} = \mathcal{P}(X^*).$$

Für alle $L \subseteq X^*$ und $x \in X$,

$$\delta^{Pow(X)}(L)(x) = \{w \in X^* \mid xw \in L\} \quad \text{und} \quad \beta^{Pow(X)}(L) = \begin{cases} 1 & \text{falls } \epsilon \in L, \\ 0 & \text{sonst.} \end{cases}$$

Aufgabe Zeigen Sie, dass die Funktion $\chi : \mathcal{P}(X^*) \rightarrow 2^{X^*}$, die jeder Teilmenge von X^* ihre charakteristische Funktion zuordnet (siehe Mengen und Typen), ein $Acc(X)$ -Homomorphismus von $Pow(X)$ nach $Beh(X, 2)$ ist. \square

Sei BL wie in 2.4 und $X = \bigcup BL \setminus 1$.

$\mathcal{P}(X^*)$ ist nicht nur die Trägermenge der $Acc(X)$ -Algebra $Pow(X)$, sondern auch der folgendermaßen definierten $Reg(BL)$ -Algebra $Lang(X)$ der Sprachen über X :

$$Lang(X)_{reg} = \mathcal{P}(X^*).$$

Für alle $B \in BL$ und $L, L' \subseteq X^*$,

$$\begin{aligned} par^{Lang(X)}(L, L') &= L \cup L', \\ seq^{Lang(X)}(L, L') &= L \cdot L', \\ iter^{Lang(X)}(L) &= L^*, \\ base^{Lang(X)}(B) &= B. \end{aligned}$$

Anstelle von $Lang(X)$ wird in `Compiler.hs` die $Reg(BL)$ -Algebra $regB$ mit der Trägermenge 2^{X^*} implementiert. Sie macht χ zum $Reg(BL)$ -Homomorphismus.

Aufgabe Zeigen Sie, dass χ ein $Reg(BL)$ -Homomorphismus von $Lang(X)$ nach $regB$ ist. \square

Die Menge der $Reg(BL)$ -Grundterme ist nicht nur eine Trägermenge der $Reg(BL)$ -Algebra $T_{Reg(BL)}$, sondern auch der wie folgt definierten $Acc(X)$ -Algebra $Bro(BL)$ (**accT** in **Compiler.hs**; siehe [3, 17]), die **Brzozowski-Automat** genannt wird:

$$Bro(BL)_{state} = T_{Reg(BL), reg}.$$

Die Interpretationen von δ und β in $Bro(BL)$ werden induktiv über dem Aufbau von $Reg(BL)$ -Grundterminen definiert:

Für alle $t, u \in T_{Reg(BL)}$ und $B \in BL \setminus 1$,

$$\begin{aligned}\delta^{Bro(BL)}(par(t, u)) &= \lambda x. par(\delta^{Bro(BL)}(t)(x), \delta^{Bro(BL)}(u)(x)), \\ \delta^{Bro(BL)}(seq(t, u)) &= \lambda x. par(seq(\delta^{Bro(BL)}(t)(x), u), \\ &\quad \text{if } \beta^{Bro(BL)}(t) = 1 \text{ then } \delta^{Bro(BL)}(u)(x) \text{ else } base(\emptyset)), \\ \delta^{Bro(BL)}(iter(t)) &= \lambda x. seq(\delta^{Bro(BL)}(t)(x), iter(t)), \\ \delta^{Bro(BL)}(base(1)) &= base(\emptyset), \\ \delta^{Bro(BL)}(base(B)) &= \lambda x. if \ x \in B \text{ then } base(1) \text{ else } base(\emptyset), \\ \beta^{Bro(BL)}(par(t, u)) &= max\{\beta^{Bro(BL)}(t), \beta^{Bro(BL)}(u)\}, \\ \beta^{Bro(BL)}(seq(t, u)) &= \beta^{Bro(BL)}(t) * \beta^{Bro(BL)}(u), \\ \beta^{Bro(BL)}(iter(t)) &= 1, \\ \beta^{Bro(BL)}(base(1)) &= 1, \\ \beta^{Bro(BL)}(base(B)) &= 0.\end{aligned}$$

2.11 Die Erreichbarkeitsfunktion

Sei $\mathcal{A} = (A, Op)$ eine $DAut(X, Y)$ -Algebra und $Z = A_{state}$. Die **Erreichbarkeitsfunktion**

$$\text{reach}^{\mathcal{A}} : X^* \rightarrow Z^Z$$

von \mathcal{A} ist wie folgt induktiv definiert: Für alle $x \in X$ und $w \in X^*$,

$$\begin{aligned}\text{reach}^{\mathcal{A}}(\epsilon) &= id_A, \\ \text{reach}^{\mathcal{A}}(xw) &= \text{reach}^{\mathcal{A}}(w) \circ \lambda a. \delta^{\mathcal{A}}(a)(x).\end{aligned}$$

Der Definitionsbereich X^* von $\text{reach}^{\mathcal{A}}$ ist Trägermenge der *Mon*-Algebra $Word(X)$, der Wertebereich Z^Z ist Trägermenge der *Mon*-Algebra $Endo(Z)$ (siehe 2.6).

$\text{reach}^{\mathcal{A}}$ ist *Mon*-homomorph:

$$\text{reach}^{\mathcal{A}}(\text{one}^{Word(X)}) = \text{reach}^{\mathcal{A}}(\epsilon) = id_A = \text{one}^{FM(Z)}.$$

Für alle $x \in X$ und $v, w \in X^*$,

$$\begin{aligned}\text{reach}^{\mathcal{A}}(\text{mul}^{Word(X)}(\epsilon, w)) &= \text{reach}^{\mathcal{A}}(\epsilon w) = \text{reach}^{\mathcal{A}}(w) = \text{reach}^{\mathcal{A}}(w) \circ id_A \\ &= \text{reach}^{\mathcal{A}}(w) \circ \text{reach}^{\mathcal{A}}(\epsilon) = \text{mul}^{FM(Z)}(\text{reach}^{\mathcal{A}}(\epsilon), \text{reach}^{\mathcal{A}}(w)), \\ \text{reach}^{\mathcal{A}}(\text{mul}^{Word(X)}(xv, w)) &= \text{reach}^{\mathcal{A}}(xvw) = \text{reach}^{\mathcal{A}}(x(\text{mul}^{Word(X)}(v, w))) \\ &= \text{reach}^{\mathcal{A}}(\text{mul}^{Word(X)}(v, w)) \circ \lambda a. \delta^{\mathcal{A}}(a)(x) \\ &\stackrel{\text{ind. hyp.}}{=} (\text{mul}^{FM(Z)}(\text{reach}^{\mathcal{A}}(v), \text{reach}^{\mathcal{A}}(w)) \circ \lambda a. \delta^{\mathcal{A}}(a)(x)\end{aligned}$$

$$\begin{aligned}
&= (\text{reach}^{\mathcal{A}}(w) \circ \text{reach}^{\mathcal{A}}(v)) \circ \lambda a. \delta^{\mathcal{A}}(a)(x) = \text{reach}^{\mathcal{A}}(w) \circ (\text{reach}^{\mathcal{A}}(v) \circ \lambda a. \delta^{\mathcal{A}}(a)(x)) \\
&= \text{reach}^{\mathcal{A}}(w) \circ \text{reach}^{\mathcal{A}}(xv) = \text{mul}^{FM(Z)}(\text{reach}^{\mathcal{A}}(xv), \text{reach}^{\mathcal{A}}(w)).
\end{aligned}$$
□

2.12 Termfaltung und Zustandsentfaltung

Sei $\Sigma = (S, C)$ eine konstruktive Signatur, V eine S -sortige Menge, $\mathcal{A} = (A, Op)$ eine Σ -Algebra und $g : V \rightarrow A$ eine – **Variablenbelegung** (*valuation*) genannte – S -sortige Funktion.

In Abhängigkeit von g wertet die wie folgt induktiv definierte S -sortige Funktion

$$g^* : T_{\Sigma}(V) \rightarrow A,$$

die **Extension von** g , jeden wohlfundierten Σ -Term in \mathcal{A} aus:

- Für alle $s \in S$ und $x \in V_s$, $g_s^*(x) = g_s(x)$. (1)

- Für alle $c : \prod_{i \in I} s_i \rightarrow s \in C$ und $(t_i)_{i \in I} \in \bigtimes_{i \in I} T_{\Sigma}(V)_{s_i}$,

$$g_s^*(c\{i \rightarrow t_i \mid i \in I\}) = c^{\mathcal{A}}((g_{s_i}^*(t_i))_{i \in I}).$$
 (2)

Insbesondere führt $\text{id}_A^* : T_{\Sigma}(A) \rightarrow \mathcal{A}$ die Operationen von $t \in T_{\Sigma}(A)$ bottom-up auf den Blättern von t aus.

Satz 2.13 ([34], Theorem FREE)

g^* ist Σ -homomorph und der einzige Σ -Homomorphismus von $T_\Sigma(V)$ nach \mathcal{A} , der (1) erfüllt. \square

Wegen dieser universellen Eigenschaft wird $T_\Sigma(V)$ als **freie Σ -Algebra über V** bezeichnet.

Alle freien Σ -Algebren über V sind isomorph zueinander. Alle zu einer freien Σ -Algebra über V isomorphen Σ -Algebren sind frei über V .

Substitutionslemma ([34], Lemma SUBST)

Für alle Σ -Algebren $\mathcal{A} = (A, Op)$, Variablenbelegungen $g : V \rightarrow A$ und Σ -Homomorphismen $h : \mathcal{A} \rightarrow \mathcal{B}$,

$$(h \circ g)^* = h \circ g^*. \quad \square$$

Offenbar hängt die Einschränkung von g^* auf Grundterme nicht von g ab. Sie wird **Termfaltung** genannt und mit $\text{fold}^{\mathcal{A}}$ bezeichnet.

Aus Satz 2.13 folgt sofort:

$\text{fold}^{\mathcal{A}}$ ist der einzige Σ -Homomorphismus von T_{Σ} nach \mathcal{A} . (4)

Freie Σ -Algebren über \emptyset heißen **initial**.

Wegen der universellen Eigenschaft freier Algebren, wird der eindeutige Σ -Homomorphismus von T_{Σ} nach \mathcal{A} immer mit $\text{fold}^{\mathcal{A}}$ bezeichnet, egal welche isomorphe Darstellung von T_{Σ} gerade verwendet wird.

Z.B. ist neben T_{Nat} auch \mathbb{N} eine initiale Nat -Algebra und neben $T_{\text{List}(X)}$ auch X^* eine initiale $\text{List}(X)$ -Algebra (siehe 2.6).

Aufgabe Wie lauten die Isomorphismen von T_{Nat} nach \mathbb{N} bzw. von $T_{\text{List}(X)}$ nach X^* ? □

Die Faltung $\text{fold}^{\text{Lang}(X)}(t)$ eines $\text{Reg}(BL)$ -Grundterms – also eines regulären Ausdrucks – t in $\text{Lang}(X)$ heißt **Sprache von t** (siehe 2.10).

Umgekehrt nennt man $L \subseteq X^*$ **regulär**, wenn L zum Bild von $\text{fold}^{\text{Lang}(X)}$ gehört.

Die Haskell-Funktion foldReg von **Compiler.hs** implementiert die Faltung

$$\text{fold}^{\mathcal{A}} : T_{\text{Reg}(BL)} \rightarrow \mathcal{A}$$

in einer beliebigen $\text{Reg}(BL)$ -Algebra \mathcal{A} .

Aufgabe Zeigen Sie $\text{fold}^{\text{Bool}} = \beta^{\text{Bro}(BL)}$. □

Aufgabe Zeigen Sie durch Induktion über den Aufbau von $\text{Reg}(BL)$ -Grundterminen, dass für alle $t \in T_{\text{Reg}(BL)}$ die folgende Äquivalenz gilt:

$$\epsilon \in \text{fold}^{\text{Lang}(X)}(t) \Leftrightarrow \text{fold}^{\text{Bool}}(t) = 1. \quad \square$$

Sei $\Sigma = (S, D)$ eine **destruktive** Signatur, V eine S -sortige Menge, $\mathcal{A} = (A, \text{Op})$ eine Σ -Algebra und $g : A \rightarrow V$ eine – **Färbung** (*coloring*) genannte – S -sortige Funktion.

In Abhängigkeit von g entfaltet die wie folgt induktiv (!) definierte S -sortige Funktion

$$g^\# : A \rightarrow DT_\Sigma(V),$$

die **Coextension von** g , jeden Zustand $a \in A$ in den Σ -Coterm, der das *Verhalten* von a (im Kontext von \mathcal{A}) repräsentiert:

- Für alle $s \in S$ und $a \in A_s$, $g_s^\#(a)(\epsilon) = g_s(a)$.
- Für alle $s \in S$, $a \in A_s$, $d : s \rightarrow \coprod_{i \in I} s_i \in D$ und $w \in (D \cup \{!\})^*$,
 $d^\mathcal{A}(a) = (b, i)$ impliziert $g_s^\#(a)(dw) = i(g_{s_i}^\#(b))(w)$.

Insbesondere berechnet $\text{id}_A^\# : \mathcal{A} \rightarrow DT_\Sigma(A)$ für jeden Anfangszustand $a \in A$ die Entfaltung des Transitionsteilgraphen von \mathcal{A} mit Wurzel a .

Satz 2.14 ([34], Theorem COFREE)

$g^\#$ ist Σ -homomorph und der einzige Σ -Homomorphismus von \mathcal{A} nach $DT_\Sigma(V)$, der für alle $a \in A$ $g^\#(a)(\epsilon) = g(a)$ erfüllt. \square

Wegen dieser universellen Eigenschaft wird $DT_\Sigma(V)$ als **cofreie Σ -Algebra über V** bezeichnet.

Alle cofreien Σ -Algebren über V sind Σ -isomorph zueinander. Alle zu einer cofreien Σ -Algebra über V isomorphen Σ -Algebren sind cofrei über V .

Substitutionslemma Für alle Σ -Algebren $\mathcal{A} = (A, Op)$, Färbungen $g : A \rightarrow V$ und Σ -Homomorphismen $h : \mathcal{B} \rightarrow \mathcal{A}$,

$$(g \circ h)^\# = g^\# \circ h. \quad \square$$

Offenbar hängt die Einschränkung von $g^\#$ auf Grundcoterme nicht von g ab. Sie wird **Zustandsentfaltung** genannt und mit $\text{unfold}^{\mathcal{A}} : \mathcal{A} \rightarrow DT_\Sigma$ bezeichnet.

Aus Satz 2.14 folgt sofort:

$unfold^{\mathcal{A}}$ ist der einzige Σ -Homomorphismus von \mathcal{A} nach DT_{Σ} . (5)

Cofreie Σ -Algebren über 1 heißen **final**.

Wegen der universellen Eigenschaft cofreier Algebren, wird der eindeutige Σ -Homomorphismus von \mathcal{A} nach DT_{Σ} immer mit $fold^{\mathcal{A}}$ bezeichnet, egal welche isomorphe Darstellung von DT_{Σ} gerade verwendet wird.

Z.B. gibt es folgende $Stream(X)$ -, $Beh(X, Y)$ - bzw. $Acc(X)$ -Isomorphismen (siehe 2.10):

$$DT_{DAut(X, Y)} \cong Beh(X, Y), \quad (1)$$

$$DT_{Acc(X)} \cong Pow(X), \quad (2)$$

$$DT_{Stream(X)} \cong Seq(X). \quad (3)$$

Beispiel 2.15

Sei $\Sigma = DAut(X, Y)$. Die Trägermengen von DT_{Σ} und $Beh(X, Y)$ sind isomorph:

Sei $L = \{(\delta, x) \mid x \in X\}$. DT_{Σ} besteht aus allen Funktionen von $L^* + L^*\beta$ nach $1 + Y$, die für alle $w \in L^*$ w auf ϵ und $w\beta$ auf ein Element von Y abbilden, m.a.W.:

$$DT_{\Sigma} \cong 1^{L^*} \times Y^{L^*\beta} \cong Y^{L^*\beta} \xrightarrow{L^*\beta \cong X^*} Y^{X^*}. \quad \square$$

Aufgabe Wie lautet der Isomorphismus zwischen $DT_{Acc(X)}$ und $Pow(X)$? □

Die $Acc(X)$ -Algebra ist $DT_{Acc(X)}$ ist im Haskell-Modul **Compiler.hs** durch **accC** implementiert.

Aufgabe Zeigen Sie, dass $\chi : \mathcal{P}(X) \rightarrow 2^X$ ein $Acc(X)$ -Homomorphismus von $Pow(X)$ nach $Beh(X, 2)$ ist. □

Für alle destruktiven Signaturen Σ und Σ -Algebren \mathcal{A} wird der eindeutige Σ -Homomorphismus von $\mathcal{A} = (A, Op)$ nicht nur nach DT_Σ , sondern auch in andere isomorphe finale Σ -Algebren mit **unfold^A** bezeichnet.

Deshalb kann $unfold^\mathcal{A} : A \rightarrow DT_\Sigma$ in den Fällen (1)-(3) als Funktion von A in die jeweilige Repräsentation von DT_Σ formuliert werden:

(1) $\Sigma = DAut(X, Y)$

$$unfold^\mathcal{A} : A \rightarrow Y^{X^*}$$
$$a \mapsto \lambda w. if\ w = \epsilon\ then\ \beta^\mathcal{A}(a)\ else\ unfold^\mathcal{A}(\delta^\mathcal{A}(a)(head(w)))(tail(w))$$

(2) $\Sigma = Acc(X)$

$$unfold^{\mathcal{A}} : A \rightarrow \mathcal{P}(X^*)$$

$$a \mapsto \{xw \mid w \in unfold^{\mathcal{A}}(\delta^{\mathcal{A}}(a)(x))\} \cup (\text{if } \beta^{\mathcal{A}}(a) = 1 \text{ then } \{\epsilon\} \text{ else } \emptyset)$$

(3) $\Sigma = Stream(X)$

$$unfold^{\mathcal{A}} : A \rightarrow X^{\mathbb{N}}$$

$$a \mapsto \lambda n. \text{if } n = 0 \text{ then } head^{\mathcal{A}}(a) \text{ else } unfold^{\mathcal{A}}(tail^{\mathcal{A}}(a))(n - 1)$$

Aus der Initialität von $T_{Reg(BL)}$ und der $Pow(X)$ ergeben sich folgende Äquivalenzen:

$$fold^{Lang(X)} \text{ ist } Acc(X)\text{-homomorph} \tag{1}$$

$$\Leftrightarrow fold^{Lang(X)} = unfold^{Bro(BL)} : T_{Reg(BL)} \rightarrow \mathcal{P}(X^*) \tag{2}$$

$$\Leftrightarrow unfold^{Bro(BL)} \text{ ist } Reg(BL)\text{-homomorph.} \tag{3}$$

Aufgabe Zeigen Sie (2), indem Sie (1) oder (3) beweisen. □

(2) folgt auch aus der Form der Gleichungen, die $Bro(BL)$ definieren (siehe Beispiel 17.10).

Streicht man die Ausgabefunktion $\beta : state \rightarrow Y$ aus $DAut(X, Y)$ heraus, dann erhält man die Signatur **Med(X)** der **Medvedev-Automaten**.

Demnach ist jede $DAut(X, Y)$ -Algebra $\mathcal{A} = (A, Op)$ auch eine $Med(X)$ -Algebra. Umgekehrt ist die cofreie $Med(X)$ -Algebra über Y eine finale $DAut(X, Y)$ -Algebra mit

$$\textcolor{blue}{unfold}^{\mathcal{A}} = (\beta^{\mathcal{A}})^{\#} : \mathcal{A} \rightarrow DT_{Med(X)}(Y). \quad (4)$$

Die Interpretation von β in \mathcal{A} wird hier offenbar als Färbung der Zustände von \mathcal{A} verwendet. Aus (4) und einer allgemeinen Beziehung zwischen Färbungen und ihren Coextensionen erhält man für initiale Automaten (\mathcal{A}, a) :

$$\textcolor{blue}{unfold}^{\mathcal{A}}(a) = \beta^{\mathcal{A}} \circ id_A^{\#}(a). \quad (5)$$

Demnach ordnet $unfold^{\mathcal{A}}(a)$ diejenige Verhaltensfunktion zu, die für jedes Eingabewort $w \in X^*$ den Ausgabewert des Zustandes zurückgibt, in dem Verarbeitung von w durch \mathcal{A} endet.

Da $id_A^{\#} : A \rightarrow DT_{Med(X)}(A) \cong Beh(X, A)$ (siehe 2.10) nur von $\delta^{\mathcal{A}}$ abhängt, wird diese Funktion auch Fortsetzung von $\delta^{\mathcal{A}}$ auf Wörter genannt und mit $(\delta^{\mathcal{A}})^*$ bezeichnet.

$id_A^{\#}$ entspricht der Erreichbarkeitsfunktion von 2.11 mit vertauschten Argumenten: Für alle $w \in X^*$,

$$id_A^{\#}(a)(w) = \textcolor{blue}{reach}^{\mathcal{A}}(w)(a).$$

Insbesondere gilt für alle $x \in X$ und $w \in X^*$:

$$id_A^{\#}(a)(\epsilon) = \textcolor{blue}{root}(id_A^{\#}(a)) = id_A(a) = a, \quad (6)$$

$$id_A^{\#}(a)(xw) = \delta^{Beh(X, A)}(id_A^{\#}(a))(x)(w) \stackrel{id_A^{\#} \text{ Med}(X)-hom.}{=} id_A^{\#}(\delta^{\mathcal{A}}(a)(x))(w). \quad (7)$$

Außerdem gilt für alle $v, w \in X^*$:

$$id_A^\#(a)(vw) = id_A^\#(id_A^\#(a)(v))(w). \quad (8)$$

Für eine beliebige Zustandsmenge Q und ein beliebiges Monoid M mit Multiplikation $*$ und neutralem Element e heißt eine Funktion $(\cdot) : Q \times M \rightarrow Q$ **Aktion von M** , wenn für alle $q \in Q$ und $m, m' \in M$ Folgendes gilt:

$$q \cdot e = q, \quad (9)$$

$$q \cdot (m * m') = (q \cdot m) \cdot m'. \quad (10)$$

Wegen (6) und (8) ist (die dekaskadierte Version von) $id_A^\#$ eine Aktion des Monoids X^* .

2.16 Initiale Automaten

Sei $\Sigma = (S, D)$ eine destruktive Signatur. Für alle Σ -Algebren \mathcal{A} , $s \in S$ und $a \in A_s$,

$$(\mathcal{A}, a) \text{ realisiert } t \in DT_{\Sigma, s} \Leftrightarrow_{def} \text{unfold}_s^{\mathcal{A}}(a) = t.$$

Im Fall $\Sigma = DAut(X, Y)$ nennt man das Paar (\mathcal{A}, a) einen **initialen Automaten**, der die Verhaltensfunktion $\text{unfold}^{\mathcal{A}}(a) : X^* \rightarrow Y$ realisiert.

Im Fall $\Sigma = Acc(X)$ nennt man das Paar (\mathcal{A}, a) einen **initialen Automaten**, der die Sprache $\text{unfold}^{\mathcal{A}}(a) \subseteq X^*$ erkennt oder akzeptiert..

Beispiel

Let eo die wie im Abschnitt 2.6 definierte $Acc(\mathbb{Z})$ -Algebra.

$$(eo, Esum) \text{ erkennt } L = \{(x_1, \dots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist gerade}\}. \quad (11)$$

$$(eo, Osum) \text{ erkennt } L' = \{(x_1, \dots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist ungerade}\}. \quad (12)$$

Beweis. Sei $h : eo \rightarrow Pow(\mathbb{Z})$ wie folgt definiert: $h(1) = L$ and $h(0) = L'$.

Da $Pow(\mathbb{Z})$ eine finale $Acc(\mathbb{Z})$ -Algebra ist, stimmen alle $Acc(\mathbb{Z})$ -Homomorphismen von eo nach $Pow(\mathbb{Z})$ mit $\text{unfold}^{eo} : eo \rightarrow Pow(\mathbb{Z})$ überein. (11) und (12) folgen demnach aus der $Acc(\mathbb{Z})$ -Homomorphie von h .

Aufgabe

Zeigen Sie, dass $h \ Acc(\mathbb{Z})$ -homomorph ist, also für alle $x \in \mathbb{Z}$ die folgenden Gleichungen erfüllt:

$$h(\delta^{eo}(Esum)(x)) = \delta^{Pow}(h(Esum))(x), \quad (13)$$

$$h(\delta^{eo}(Osum)(x)) = \delta^{Pow}(h(Osum))(x), \quad (14)$$

$$\beta^{eo}(Esum) = \beta^{Pow}(h(Esum)), \quad (15)$$

$$\beta^{eo}(Osum) = \beta^{Pow}(h(Osum)). \quad (16)$$

□

Sei $t \in T_{Reg(BL)}$. Aus (2) folgt, dass der initiale Automat $(Bro(BL), t)$ die Sprache von t erkennt:

$$unfold^{Bro}(BL)(t) = fold^{Lang(X)}(t)$$

(siehe 2.10). Damit liefert $(Bro(BL), t)$ eine einfache Alternative zum klassischen Potenzautomat, der über den Umweg eines nichtdeterministischen Automaten mit ϵ -Übergängen aus t gebildet wird (siehe Beispiel 12.3).

Übersicht über die oben definierten $Reg(BL)$ - bzw. $DAut(X, Y)$ -Algebren und ihre jeweiligen Implementierungen in [Compiler.hs](#):

	Signatur	$Reg(BL)$	$Acc(X) = DAut(X, 2)$	$DAut(X, Y)$
Trägermenge	Algebra			
$T_{Reg(BL)}$ RegT		$T_{Reg(BL)}$ regT initial	$Bro(BL)$ accT	
Y^{X^*}				$Beh(X, Y)$ final
2^{X^*}		$\chi(Lang(X))$ regB	$\chi(Pow(X))$ $Beh(X, 2)$ behFun final	
$\mathcal{P}(X^*)$		$Lang(X)$	$Pow(X)$ final	
$\mathbb{N} \rightarrow syms(BL)^*$		$Regword(BL)$ regWord		
2		$Bool$		

2.17 Compiler für reguläre Ausdrücke

Im Folgenden werden wir einen Parser für die Wortdarstellung eines regulären Ausdrucks t mit dessen Faltung in einer beliebigen $Reg(BL)$ -Algebra \mathcal{A} verknüpfen und damit ein erstes Beispiel für einen Compiler erhalten, der die Elemente einer Wortmenge ohne den Umweg über $Reg(BL)$ -Terme nach \mathcal{A} – übersetzt.

Werden reguläre Ausdrücke als Wörter $syms(BL)$ (siehe 2.9) eingelesen, dann muss dem Erkennern $fold^{\chi(Lang(X))}(t)$ eine Funktion

$$\textcolor{red}{parse}_{REG} : syms(BL)^* \rightarrow M(T_{Reg(BL)})$$

vorgeschaltet werden, die jedes korrekte Eingabewort w in einen oder mehrere $Reg(BL)$ -Terme t mit $fold^{Regword(BL)}(t)(0) = w$ überführt. M ist ein monadischer Funktor (siehe 5.1), der die Ausgabe des Parsers steuert und insbesondere Fehlermeldungen erzeugt, falls w keinem $Reg(BL)$ -Term entspricht.

REG steht hier für eine konkrete Syntax, d.h. eine kontextfreie Grammatik, deren Sprache mit dem Bild von $T_{Reg(BL)}$ unter $flip(fold^{Regword(BL)})$ (0) übereinstimmt (siehe Beispiel 4.1).

Sei X eine Menge, G eine kontextfreie Grammatik mit abstrakter Syntax Σ und Sprache $L \subseteq X^*$. In Kapitel 5 werden wir einen generischen Compiler für L , der jedes Wort von L in Elemente einer Σ -Algebra A übersetzt, als *natürliche Transformation* compile_G des konstanten Funktors $const(X^*)$ nach M formulieren.

parse_G ist diejenige Instanz von compile_G , die Wörter von L in Syntaxbäume ($= \Sigma$ -Grundterme) übersetzt: $\text{parse}_G = \text{compile}_G^{T_\Sigma}$.

Im Fall $G = REG$, $\Sigma = \text{Reg}(BL)$, $X = \text{syms}(BL)$ und $M(A) = A+1$ für alle Σ -Algebren $\mathcal{A} = (A, Op)$ stimmt $\text{compile}_G^{\mathcal{A}}(w)$ mit der Faltung des durch w dargestellten regulären Ausdrucks t in der $\text{Reg}(BL)$ -Algebra \mathcal{A} überein:

$$\begin{aligned} \text{compile}_G^{\mathcal{A}}(w) &\stackrel{\text{compile}_G \text{ ist nat. Transformation}}{=} M(fold^{\mathcal{A}})(\text{compile}_G^{T_{\text{Reg}(BL)}}(w)) \\ &= M(fold^{\mathcal{A}})(\text{parse}_G(w)) = (fold^{\mathcal{A}} + id_1)(\text{parse}_G(w)) \\ &= (fold^{\mathcal{A}} + id_1)(\text{parse}_G(fold^{\text{Regword}(BL)}(t)(0))) = (fold^{\mathcal{A}} + id_1)(t) = \text{fold}^{\mathcal{A}}(t). \end{aligned}$$

Im Haskell-Modul `Compiler.hs` ist compile_{REG} durch die Funktion `regToAlg` implementiert, deren Zahlparameter eine von 8 Zielalgebren auswählt.

Das folgende Diagramm zeigt die Beziehungen zwischen parse_G und den oben behandelten $\text{Reg}(BL)$ - bzw. $\text{Acc}(X)$ -Algebren:

$$\begin{array}{ccccc}
 \text{sym}(BL)^+ & \xrightarrow{\text{parse}_{\text{REG}}} & T_{\text{Reg}(BL)} + 1 & \xrightarrow{\text{fold}^{\text{Lang}(X)} + \text{id}_1} & \mathcal{P}(X^*) + 1 \\
 \pi_1 \circ \text{fold}^{\text{Regword}(BL)} & = & \text{inc} & = & \text{inc} \\
 & & \downarrow & & \downarrow \\
 & & T_{\text{Reg}(BL)} & \xrightarrow{\text{fold}^{\text{Lang}(X)}} & \text{Lang}(X) \\
 & & \parallel & = & \parallel \\
 & & \text{fold}^{\text{Bool}} & & \\
 & = & & & \\
 \text{Bool} & \xleftarrow{\beta^{\text{Bro}(BL)}} & \text{Bro}(BL) & \xrightarrow{\text{unfold}^{\text{Bro}(BL)}} & \text{Pow}(X) \\
 & & \downarrow & & \downarrow \\
 & & \delta^{\text{Bro}(BL)} & = & \delta^{\text{Pow}(X)} \\
 & & & & \\
 & & \text{Bro}(BL)^X & & \text{Pow}(X)^X
 \end{array}$$

2.18 Minimale Automaten

Wie zeigt man, dass initiale Automaten bestimmte Verhaltensfunktionen realisieren bzw. Sprachen erkennen?

Sei $\mathcal{A} = (A, Op)$ eine $Acc(X)$ -Algebra, und $f : A \rightarrow \mathcal{P}(X^*)$.

Da $Pow(X)$ eine finale $Acc(X)$ -Algebra ist (s.o.), erkennt für alle $a \in A_{state}$ der initiale Automat (\mathcal{A}, a) genau dann die Sprache $f(a)$, wenn f $Acc(X)$ -homomorph ist.

Beispiel 2.19 Sei

$$\begin{aligned} L &= \{(x_1, \dots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist gerade}\}, \\ L' &= \{(x_1, \dots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist ungerade}\}. \end{aligned}$$

Die Funktion $h : eo \rightarrow Pow(X)$ (siehe 2.6) mit $h(Esum) = L$ und $h(Osum) = L'$ ist $Acc(\mathbb{Z})$ -homomorph. Also wird L vom initialen Automaten $(eo, Esum)$ und L' vom initialen Automaten $(eo, Osum)$ erkannt. \square

Sei (\mathcal{A}, a) ein initialer Automat. Die Elemente der Menge $\langle a \rangle =_{def} id_A^\#(a)(X^*)$ heißen **Folgezustände von a in A** .

Satz 2.20 Sei $\mathcal{A} = (A, Op)$ eine $DAut(X, Y)$ -Algebra.

- (i) Für alle $a \in A$ ist $\langle a \rangle$ die kleinste $DAut(X, Y)$ -Unteralgebra von \mathcal{A} , die a enthält.
- (ii) Für alle Σ -Homomorphismen $h : \mathcal{A} \rightarrow B$ gilt $h(\langle a \rangle) = \langle h(a) \rangle$.
- (iii) Für alle $f : X^* \rightarrow Y$ ist $(\langle f \rangle, f)$ eine minimale Realisierung von f .

Beweis.

(i): Wir zeigen zunächst durch Induktion über $|w|$, dass für alle $a \in A$, $x \in X$ und $w \in X^*$ Folgendes gilt:

$$id_A^\#(a)(wx) = \delta^{\mathcal{A}}(id_A^\#(a)(w))(x). \quad (11)$$

Beweis von (11):

$$id_A^\#(a)(\epsilon x) = id_A^\#(a)(x\epsilon) \stackrel{(7)}{=} id_A^\#(\delta^{\mathcal{A}}(a)(x))(\epsilon) \stackrel{(6)}{=} \delta^{\mathcal{A}}(a)(x).$$

Für alle $y \in X$ und $w \in X^*$,

$$id_A^\#(a)(ywx) \stackrel{(7)}{=} id_A^\#(\delta^{\mathcal{A}}(a)(y))(wx) \stackrel{ind. \ hyp.}{=} \delta^{\mathcal{A}}(id_A^\#(\delta^{\mathcal{A}}(a)(y))(w))(x) \stackrel{(7)}{=} \delta^{\mathcal{A}}(id_A^\#(yw))(x). \quad \square$$

Wegen (11) gilt $\delta^{\mathcal{A}}(b)(x) \in \langle a \rangle$ für alle $b \in \langle a \rangle$ und $x \in X$. Also ist $\langle a \rangle$ eine Unteralgebra von A .

Sei $\mathcal{B} = (B, Op')$ eine Unteralgebra von \mathcal{A} , die a enthält. Durch Induktion über $|w|$ erhält man aus (6) und (7), dass $id_A^\#(a)(w)$ für alle $w \in X^*$ zu B gehört. Also ist $\langle a \rangle$ die kleinste Unteralgebra von \mathcal{A} , die a enthält.

(ii): Da h Σ -homomorph ist, ist h auch mit $id_A^\#$ bzw. $id_B^\#$ verträglich (s.o.), d.h. für alle $w \in X^*$ gilt $h(id_A^\#(a)(w)) = id_B^\#(h(a))(w)$. Daraus folgt sowohl $h(\langle a \rangle) \subseteq \langle h(a) \rangle$ als auch $\langle h(a) \rangle \subseteq h(\langle a \rangle)$.

(iii):

Da $Beh(X, Y)$ final ist, stimmt $unfold^{Beh(X, Y)}$ (s.o.) mit der Identität auf Y^{X^*} überein. Also ist $unfold^{Beh(X, Y)}(f) = f$ und damit $(\langle f \rangle, f)$ wegen (i) ein initialer Automat, der f realisiert.

Sei $(\mathcal{A} = (A, Op), a)$ ein initialer Automat, der f realisiert.

Wegen (ii) ist $h : \langle a \rangle \rightarrow \langle unfold^{\mathcal{A}}(a) \rangle$ mit $h(b) =_{def} unfold^{\mathcal{A}}(b)$ für alle $b \in \langle a \rangle$ wohldefiniert und surjektiv. Daraus folgt $|\langle f \rangle| = |\langle unfold^{\mathcal{A}}(a) \rangle| \leq |\langle a \rangle| \leq |A|$. □

2.21 Baumautomaten [1, 4, 5, 35, 36, 44]

Sei $\Sigma = (S, C)$ eine konstruktive Signatur, Y eine Menge und

$$\Sigma_Y = (S, C \cup \{out : s \rightarrow Y \mid s \in S\}).$$

Eine Σ_Y -Algebra \mathcal{B} heißt (bottom-up) **Σ -Baumautomat**. Sei $out^{\mathcal{B}} = (out_s^{\mathcal{B}})_{s \in S}$ und \mathcal{A} das Σ -Redukt $\mathcal{B}|_\Sigma$ von \mathcal{B} , also die Σ -Algebra, deren Trägermengen und Operationen mit denen von \mathcal{B} übereinstimmen.

$$\textcolor{blue}{out}^{\mathcal{B}} \circ \text{fold}^{\mathcal{A}} : T_{\Sigma} \rightarrow Y$$

heißt **Verhaltensfunktion von \mathcal{B}** .

Im Fall $Y = 2$ nennt man

$$\textcolor{red}{L(\mathcal{B})} =_{def} (\text{out}^{\mathcal{B}} \circ \text{fold}^{\mathcal{A}})^{-1}(1) \subseteq T_{\Sigma}$$

die **von \mathcal{B} erkannte Baumsprache**.

Wie man leicht sieht, ist die Funktion

$$\begin{aligned} h : T_{List(X)} &\rightarrow X^* \\ nil &\mapsto \epsilon \\ cons(x, t) &\mapsto x \cdot h(t) \end{aligned}$$

bijektiv. Folglich verallgemeinert der Begriff einer Baumsprache $L \subseteq T_{\Sigma}$ den einer Wortsprache $L \subseteq X^*$.

Dementsprechend nennt man $L \subseteq T_{\Sigma}$ **regulär**, wenn es einen endlichen (!) Σ -Baumautomaten \mathcal{B} gibt, der L erkennt, der also die Gleichung $\chi(L) = \text{out}^{\mathcal{B}} \circ \text{fold}^{\mathcal{A}}$ erfüllt.

Sei $\mathcal{A} = (A, Op)$ eine $DAut(X, Y)$ -Algebra und $a \in A$. Der initiale Automat (\mathcal{A}, a) liefert den $List(X)$ -Baumautomaten \mathcal{B} mit

$$\mathcal{B}_{list} = \mathcal{A}_{state}, \quad nil^{\mathcal{B}} = a, \quad cons^{\mathcal{B}} = \lambda(x, a). \delta^{\mathcal{A}}(a)(x) \text{ und } out^{\mathcal{B}} = \beta^{\mathcal{A}}.$$

\mathcal{B} realisiert $unfold^{\mathcal{A}}(a) \circ h : T_{List(X)} \rightarrow Y$ (siehe Abschnitt 2.12).

Umgekehrt liefert ein $List(X)$ -Baumautomat \mathcal{B} den initialen Automaten $(\mathcal{A}, nil^{\mathcal{B}})$ mit

$$\mathcal{A}_{state} = \mathcal{B}_{list}, \quad \delta^{\mathcal{A}} = \lambda a. \lambda x. cons^{\mathcal{B}}(x, a) \text{ und } \beta^{\mathcal{A}} = out^{\mathcal{B}}$$

$(\mathcal{A}, nil^{\mathcal{B}})$ realisiert $out^{\mathcal{B}} \circ fold^{\mathcal{A}} \circ h^{-1} : X^* \rightarrow Y$.

Insbesondere ist eine Baumsprache $L \subseteq T_{List(X)}$ genau dann regulär, wenn es einen endlichen initialen Akzeptor (\mathcal{A}, a) gibt, der $h(L)$ erkennt, was wiederum – wegen 3.13 oder 12.3 – genau dann gilt, wenn $h(L)$ im Sinne von Wortsprachen regulär ist, also zum Bild von $fold^{Lang(X)} : T_{Reg(BL)} \rightarrow \mathcal{P}(X^*)$ gehört.

Sei $f : T_\Sigma \rightarrow Y$ und \sim die **Nerode-Relation von** f , d.h. die größte Σ -Kongruenz (siehe Kapitel 3), die im Kern von f enthalten ist.

Ein Σ -Baumautomat heißt **Realisierung von** f , wenn seine Verhaltensfunktion mit f übereinstimmt.

Laut Kapitel 3 ist der Quotient $Q = T_\Sigma / \sim$ eine Σ -Algebra. Folglich ist \mathcal{F} mit $\mathcal{F}|_\Sigma = Q$ und $out^{\mathcal{F}} \circ nat_\sim = f$ wegen $fold^Q = nat_\sim$ eine Realisierung von f . $out^{\mathcal{F}}$ ist wohldefiniert, weil der Kern von $f \sim$ enthält.

Baumautomaten werden zum Beispiel zur Erkennung von Mengen von Σ -Termen eingesetzt, die XML-Dokumente mit bestimmten Eigenschaften repräsentieren (siehe Beispiel 4.3).

3 Rechnen mit Algebren

In diesem Abschnitt werden die beiden wichtigsten einstelligen Algebratransformationen: die Bildung von Unteralgebren bzw. Quotienten, und ihr Zusammenhang zu Homomorphismen vorgestellt. Unteralgebren und Quotienten modellieren *Restriktionen* bzw. *Abstraktionen* eines gegebenen Modells. Beide Konstrukte sind aus der Mengenlehre bekannt. Sie werden hier auf S -sortige Mengen fortgesetzt.

Sei $A = (A_e)_{e \in \mathcal{T}_p(S)}$ eine S -sortige Menge und $n > 0$. Eine n -stellige S -sortige Relation R auf A wird wie folgt zur $\mathcal{T}_p(S)$ -sortigen Relation geliftet: Sei I eine nichtleere Menge.

- $R_I = \Delta_I^n$.
- Für alle $\{e_i\}_{i \in I} \subseteq \mathcal{T}_p(S)$,

$$R_{\prod_{i \in I} e_i} = \{(a_1, \dots, a_n) \in A_{\prod_{i \in I} e_i}^n \mid \forall i \in I : (\pi_i(a_1), \dots, \pi_i(a_n)) \in R_{e_i}\},$$

$$R_{\coprod_{i \in I} e_i} = \{(\iota_i(a_1), \dots, \iota_i(a_n)) \in A_{\coprod_{i \in I} e_i}^n \mid (a_1, \dots, a_n) \in R_{e_i}, i \in I\}.$$

Lemma LIFT Seien $g, h : A \rightarrow B$ typverträgliche Funktionen und R eine typverträgliche Teilmenge von A und R' eine typverträgliche zweistellige Relation auf B mit

$$R_s = \{a \in A_s \mid g(a) = h(a)\} \quad \text{und} \quad R'_s = \{(g(a), h(a)) \mid a \in A_s\}$$

für alle $s \in S$. Dann gilt

$$R_e = \{a \in A_e \mid g_e(a) = h_e(a)\}, \tag{1}$$

$$R'_e = \{(g_e(a), h_e(a)) \mid a \in A_e\} \tag{2}$$

für alle $e \in \mathcal{T}_p(S)$. □

Unteralgebren

Sei $\Sigma = (S, F)$ eine Signatur und $\mathcal{A} = (A, Op)$ eine Σ -Algebra.

Eine S -sortige Teilmenge $B = (B_s)_{s \in S}$ von A heißt **Σ -Invariante**, wenn für alle $f : e \rightarrow e' \in F$ und $a \in B_e$ $f^{\mathcal{A}}(a) \in B_{e'}$ gilt.

B heißt Σ -Invariante, weil die Zugehörigkeit von Elementen von A zu B invariant gegenüber der Anwendung von Operationen von Σ ist.

B induziert die **Σ -Unteralgebra** $\mathcal{A}|_B$ von \mathcal{A} :

- Für alle $s \in S$, $(\mathcal{A}|_B)_s =_{def} B_s$.
- Für alle $f : e \rightarrow e' \in F$ und $a \in B_e$, $f^{\mathcal{A}|_B}(a) =_{def} f^{\mathcal{A}}(a)$.

Beispiel 3.1

Sei $X = \bigcup BL$. Die Menge $\text{fold}^{\text{Lang}(X)}(T_{\text{Reg}(BL)})$ der regulären Sprachen über X ist eine $\text{Reg}(BL)$ -Invariante von $\text{Lang}(X)$. Das folgt allein aus der Verträglichkeit von $\text{fold}^{\text{Lang}(X)}$ mit den Operationen von $\text{Reg}(BL)$.

Für jede Signatur Σ , jede Σ -Algebra $\mathcal{A} = (A, Op)$ und jeden Σ -Homomorphismus $h : \mathcal{A} \rightarrow \mathcal{B}$ ist $(h(A_s))_{s \in S}$ eine Σ -Invariante. Die von ihr induzierte Unteralgebra von \mathcal{A} stimmt mit der weiter oben definierten Bildalgebra $h(\mathcal{A})$ überein.

Für jede konstruktive Signatur Σ ist $T_\Sigma(V)$ eine Σ -Unteralgebra von $CT_\Sigma(V)$.

Für jede destruktive Signatur Σ ist $coT_\Sigma(V)$ eine Σ -Unteralgebra von $DT_\Sigma(V)$. □

Satz 3.2

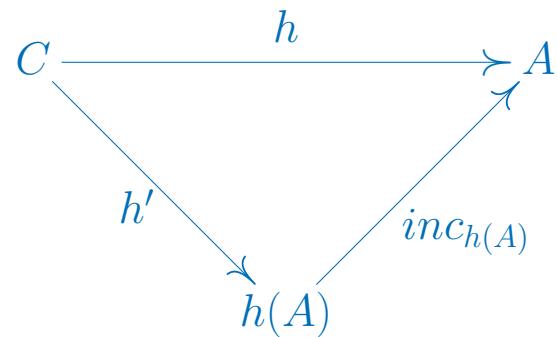
- (1) Sei $\mathcal{B} = (B, Op')$ eine Unteralgebra von $\mathcal{A} = (A, Op)$. Die S -sortige Funktion $\text{inc}_{\mathcal{B}} = (inc_{B_s} : B_s \rightarrow A_s)_{s \in S}$ ist Σ -homomorph.
- (2) (Homomorphiesatz) Sei $h : \mathcal{C} \rightarrow \mathcal{A}$ ein Σ -Homomorphismus. $h(\mathcal{C})$ ist eine Unteralgebra von A .

$$\begin{aligned} h' : \mathcal{C} &\rightarrow h(\mathcal{C}) \\ c &\mapsto h(c) \end{aligned}$$

ist ein surjektiver Σ -Homomorphismus.

Ist h injektiv, dann ist h' bijektiv.

Alle Σ -Homomorphismen $h'' : \mathcal{C} \rightarrow h(\mathcal{C})$ mit $inc_{h(\mathcal{C})} \circ h'' = h$ stimmen mit h' überein.



(3) Sei Σ eine konstruktive Signatur und A eine Σ -Algebra. $fold^A(T_\Sigma)$ ist die kleinste Σ -Unteralgebra von A und T_Σ ist die einzige Σ -Unteralgebra von T_Σ .

Demnach erfüllen alle zu T_Σ isomorphen Σ -Algebren A das **Induktionsprinzip**, d.h. eine prädikatenlogische Formel φ gilt für alle Elemente von A , wenn φ von allen Elementen einer Σ -Unteralgebra von A erfüllt wird.

(4) Sei A eine Σ -Algebra und die A die einzige Σ -Unteralgebra von A . Dann gibt es für alle Σ -Algebren B höchstens einen Σ -Homomorphismus $h : A \rightarrow B$.

Beweis von (3). Sei B eine Unteralgebra von A . Da T_Σ initial und inc_B Σ -homomorph ist, kommutiert das folgende Diagramm:

$$\begin{array}{ccc}
 T_\Sigma & \xrightarrow{\text{fold}^A} & A \\
 & \searrow \text{fold}^B & \nearrow inc_B \\
 & = &
 \end{array}$$

Daraus folgt für alle $t \in T_\Sigma$,

$$\text{fold}^A(t) = inc_B(\text{fold}^B(t)) = \text{fold}^B(t) \in B.$$

Also ist das Bild von fold^A in B enthalten.

Sei B eine Unteralgebra von T_Σ . Da T_Σ initial ist, folgt $inc_B \circ \text{fold}^B = id_{T_\Sigma}$ aus (1). Da id_{T_Σ} surjektiv ist, ist auch inc_B surjektiv. Da inc_B auch injektiv ist, sind B und T_Σ isomorph. Also kann B nur mit T_Σ übereinstimmen.

Beweis von (4). Seien $g, h : A \rightarrow B$ Σ -Homomorphismen. Dann ist

$$C = \{a \in A \mid g(a) = h(a)\}$$

eine Σ -Unteralgebra von A : Sei $f : e \rightarrow e' \in F$ und $a \in A_e$ mit $g_e(a) = h_e(a)$. Da g und h Σ -homomorph sind, gilt $g_{e'}(f^{\mathcal{A}}(a)) = f^{\mathcal{B}}(g_e(a)) = f^{\mathcal{B}}(h_e(a)) = h_{e'}(f^{\mathcal{A}}(a))$. Da g und h S -sortig sind, folgt $f^{\mathcal{A}}(a) \in C_e$ aus Lemma LIFT.

Da A die einzige Σ -Unteralgebra von A ist, stimmt C mit A überein, d.h. für alle $a \in A$ gilt $g(a) = h(a)$. Also ist $g = h$. \square

Beispiel 3.3 (siehe Beispiel 3.1) Nach Satz 3.2 (3) ist $\text{fold}^{\text{Lang}(X)}(T_{\text{Reg}(BL)})$ die *kleinste* Unteralgebra von $\text{Lang}(X)$. \square

Kongruenzen und Quotienten

Sei $\Sigma = (S, F)$ eine Signatur und $\mathcal{A} = (A, Op)$ eine Σ -Algebra.

Eine S -sortige binäre Relation R auf A heißt **Σ -Kongruenz**, wenn für alle $s \in S$ R_s eine Äquivalenzrelation ist und für alle $f : e \rightarrow e' \in F$ und $(a, b) \in R_e$ $(f^{\mathcal{A}}(a), f^{\mathcal{A}}(b)) \in R_{e'}$ gilt.

R induziert die **Σ -Quotientenalgebra** \mathcal{A}/R von \mathcal{A} :

- Für alle $s \in S$, $(\mathcal{A}/R)_s =_{\text{def}} \{[a]_R \mid a \in A_s\}$, wobei $[a]_R = \{b \in A_s \mid (a, b) \in R_s\}$.
- Für alle $f : e \rightarrow e' \in F$ und $a \in A_e$, $f^{\mathcal{A}/R}([a]_R) =_{\text{def}} [f^{\mathcal{A}}(a)]_R$.

Satz 3.4 (ist dual zu Satz 3.2)

- (1) Sei A eine Σ -Algebra und R eine Σ -Kongruenz auf A . Die **natürliche Abbildung** $nat_R : \mathcal{A} \rightarrow \mathcal{A}/R$, die jedes Element $a \in A$ auf seine Äquivalenzklasse $[a]_R$ abbildet, ist Σ -homomorph.
- (2) (Homomorphiesatz) Sei $h : A \rightarrow B$ ein Σ -Homomorphismus und $ker(h) \subseteq A^2$ der **Kern** von h , d.h. $ker(h) = \{(a, b) \mid h(a) = h(b)\}$.

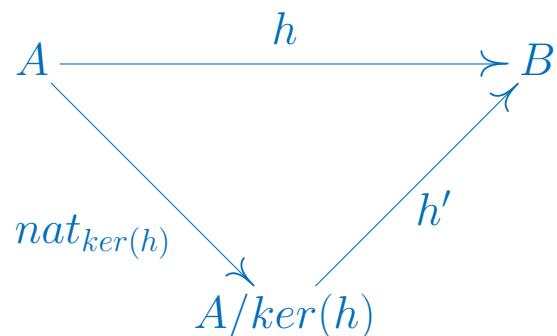
$ker(h)$ ist eine Σ -Kongruenz.

$$\begin{aligned} h' : A/ker(h) &\rightarrow B \\ [a]_{ker(h)} &\mapsto h(a) \end{aligned}$$

ist ein injektiver Σ -Homomorphismus.

Ist h surjektiv, dann ist h' bijektiv.

Alle Σ -Homomorphismen $h'' : A/ker(h) \rightarrow B$ mit $h'' \circ nat = h$ stimmen mit h' überein.



(3) Sei Σ eine **destruktive** Signatur, A eine Σ -Algebra und Fin eine finale Σ -Algebra (s.o.). Der – **Verhaltenskongruenz von A** genannte – Kern des eindeutigen Σ -Homomorphismus $unfold^A : A \rightarrow Fin$ ist die größte Σ -Kongruenz auf A und die Diagonale von Fin^2 die einzige Σ -Kongruenz R auf Fin .

Demnach erfüllen alle zu Fin isomorphen Σ -Algebren A das **Coinduktionsprinzip**: Sei E eine Menge von Σ -**Gleichungen** $t = u$ zwischen Σ -Termen über V , die denselben Typ haben. E gilt in A , wenn es eine Σ -Kongruenz R auf A gibt, die für alle $t = u \in E$ und $g : V \rightarrow A$ das Paar $(g^*(t), g^*(u))$ enthält.

(4) Sei B eine Σ -Algebra und die Diagonale von B^2 die einzige Σ -Kongruenz R auf B . Dann gibt es für alle Σ -Algebren A höchstens einen Σ -Homomorphismus $h : A \rightarrow B$.

Beweis von (3). Sei R eine Kongruenz auf A . Da Fin final und nat_R Σ -homomorph ist, kommutiert das folgende Diagramm:

$$\begin{array}{ccc}
 A & \xrightarrow{\quad unfold^A \quad} & Fin \\
 & \searrow nat_R & \nearrow unfold^{A/R} \\
 & A/R &
 \end{array}$$

Daraus folgt für alle $a, b \in A$,

$$\begin{aligned}(a, b) \in R &\Rightarrow [a]_R = [b]_R \\ &\Rightarrow \text{unfold}^{\mathcal{A}}(a) = \text{unfold}^{\mathcal{A}/R}([a]_R) = \text{unfold}^{\mathcal{A}/R}([b]_R) = \text{unfold}^{\mathcal{A}}(b).\end{aligned}$$

Also ist R im Kern von $\text{unfold}^{\mathcal{A}}$ enthalten.

Sei R eine Kongruenz auf Fin . Da Fin final ist, folgt $\text{unfold}^{\text{Fin}/R} \circ \text{nat}_R = \text{id}_{\text{Fin}}$ aus (1). Da id_{Fin} injektiv ist, ist auch nat_R injektiv. Da nat_R auch surjektiv ist, sind Fin und Fin/R isomorph. Also kann R nur die Diagonale von Fin^2 sein.

Beweis von (4). Seien $g, h : A \rightarrow B$ Σ -Homomorphismen. Dann ist

$$R = \{(g(a), h(a)) \mid a \in A\}$$

eine Σ -Kongruenz auf B : Sei $f : e \rightarrow e' \in F$, $a \in A_e$ und $(g_e(a), h_e(a)) \in R_e$. Da g und h Σ -homomorph sind, gelten $f^{\mathcal{B}}(g_e(a)) = g_{e'}(f^{\mathcal{A}}(a))$ und $f^{\mathcal{B}}(h_e(a)) = h_{e'}(f^{\mathcal{A}}(a))$.

Da g und h S -sortig sind, folgt

$$(f^{\mathcal{B}}(g_e(a)), f^{\mathcal{B}}(h_e(a))) = (g_{e'}(f^{\mathcal{A}}(a)), h_{e'}(f^{\mathcal{A}}(a))) \in R_{e'}$$

aus Lemma LIFT. Da Δ_B^2 die einzige Σ -Kongruenz auf B ist, stimmt R mit Δ_B^2 überein, d.h. für alle $a \in A$ gilt $g(a) = h(a)$. Also ist $g = h$. □

Automatenminimierung durch Quotientenbildung

Eine minimale Realisierung einer Verhaltensfunktion $f : X^* \rightarrow Y$ erhält man auch als Quotienten eines gegebenen initialen Automaten A :

Im Beweis von Satz 2.20 (iii) wurde der $DAut(X, Y)$ -Homomorphismus $h : \langle a \rangle \rightarrow \langle f \rangle$ definiert. Wegen der Surjektivität von h ist $\langle a \rangle / \ker(h)$ nach Satz 3.4 (2) $DAut(X, Y)$ -isomorph zu $\langle f \rangle$. Nach Definition von h ist $\ker(h) = \ker(unfold^A) \cap \langle a \rangle^2$.

Nach Satz 3.4 (3) ist $\ker(unfold^A)$ die größte Σ -Kongruenz auf A , also die größte binäre Relation R auf A , die für alle $a, b \in A_{state}$ folgende Bedingung erfüllt:

$$(a, b) \in R \quad \Rightarrow \quad \beta^A(a) = \beta^A(b) \wedge \forall w \in X^* : (\delta^A(a)(w), \delta^A(b)(w)) \in R. \quad (1)$$

Ist A_{state} endlich, dann lässt sich $\ker(unfold^A)$ schnell und elegant mit dem in Abschnitt 2.3 von [29] und in [33] beschriebene (und in Haskell implementierte) **Paull-Unger-Verfahren** berechnen. Es startet mit $R' = A_{state}^2$ und benutzt die Kontraposition

$$(a, b) \notin R \quad \Leftarrow \quad \beta^A(a) \neq \beta^A(b) \vee \forall w \in X^* : (\delta^A(a)(w), \delta^A(b)(w)) \notin R \quad (2)$$

von (1), um R' schrittweise auf den Kern von $unfold^A$ zu reduzieren.

Transitionsmonoid und syntaktische Kongruenz

Sei $\mathcal{A} = (A, Op)$ eine $DAut(X, Y)$ -Algebra. Das Bild der Mon -homomorphen Erreichbarkeitsfunktion

$$\text{reach}^{\mathcal{A}} : X^* \rightarrow (A_{\text{state}} \rightarrow A_{\text{state}})$$

von \mathcal{A} heißt **Transitionsmonoid von A** .

Satz 3.5 (Transitionsmonoide und endliche Automaten)

Für alle $a \in A_{\text{state}}$ ist $\langle a \rangle$ genau dann endlich, wenn das Transitionsmonoid der Unteralgebra von \mathcal{A} mit Trägermenge $\langle a \rangle$ endlich ist.

Beweis. Sei R der Kern von $\text{reach}^{\langle a \rangle} : X^* \rightarrow (\langle a \rangle \rightarrow \langle a \rangle)$. Nach Satz 3.4 (2) ist das Transitionsmonoid von $\langle a \rangle$ Mon -isomorph zum Quotienten X^*/R . Außerdem ist die Abbildung

$$\begin{aligned} h : X^*/R &\rightarrow \langle a \rangle \\ [w]_R &\mapsto \delta_{A^*}(a)(w) \end{aligned}$$

wohldefiniert: Sei $(v, w) \in R$. Dann gilt $\text{reach}^{\langle a \rangle}(v) = \text{reach}^{\langle a \rangle}(w)$, also insbesondere

$$h([v]_R) = id_A^\#(a)(v) = \text{reach}^{\langle a \rangle}(v)(a) = \text{reach}^{\langle a \rangle}(w)(a) = id_A^\#(a)(w) = h([w]_R).$$

Da h surjektiv ist, liefert die Komposition $h \circ g$ mit dem Isomorphismus

$$g : \text{reach}^{\langle a \rangle}(X^*) \rightarrow X^*/R$$

eine surjektive Abbildung von $\text{reach}^{\langle a \rangle}(X^*)$ nach $\langle a \rangle$. Also überträgt sich die Endlichkeit des Transitionsmonoids von $\langle a \rangle$ auf $\langle a \rangle$ selbst. Umgekehrt ist das Transitionsmonoid jedes endlichen Automaten A endlich, weil es eine Teilmenge von $A_{\text{state}} \rightarrow A_{\text{state}}$ ist. \square

Sei $L \subseteq X^*$. Das Transitionsmonoid von $\langle L \rangle$ heißt **syntaktisches Monoid** und der Kern von $\text{reach}^{\langle L \rangle}$ **syntaktische Kongruenz von L** .

Letztere lässt sich als Menge aller Paare $(v, w) \in X^* \times X^*$ mit $uvu' \in L \Leftrightarrow uwu' \in L$ für alle $u, u' \in X^*$ charakterisieren. Satz 3.5 impliziert, dass sie genau dann endlich viele Äquivalenzklassen hat, wenn L von einem endlichen initialen Automaten (A, a) erkannt wird, was wiederum zur Regularität von L äquivalent ist (siehe 3.13 für einen direkten Beweis oder Beispiel 12.3 für den klassischen über die Konstruktion eines nichtdeterministischen Akzeptors von L).

Folglich kann die Nichtregularität einer Sprache L oft gezeigt werden, indem aus der Endlichkeit der Zustandsmenge eines Akzeptors von L ein Widerspruch hergeleitet wird.

Wäre z.B. $L = \{x^n y^n \mid n \in \mathbb{N}\}$ regulär, dann gäbe es einen endlichen Automaten (A, a) mit $\text{unfold}^A(a) = L$.

Demnach müsste für alle $n \in \mathbb{N}$ ein Zustand $b_n \in A_{\text{reg}}$ existieren mit $\text{id}_A^\#(a)(x^n) = b_n$ und $\beta^A(\text{id}_A^\#(b_n)(y^n)) = 1$. Da A endlich ist, gäbe es $i, j \in \mathbb{N}$ mit $i \neq j$ und $b_i = b_j$.

Daraus würde jedoch

$$\begin{aligned} \text{unfold}^{\mathcal{A}}(x^i y^j) &= \beta^{\mathcal{A}}(\text{id}_A^\#(a)(x^i y^j)) = \beta^{\mathcal{A}}(\text{id}_A^\#(\text{id}_A^\#(a)(x^i))(y^j)) = \beta^{\mathcal{A}}(\text{id}_A^\#(\textcolor{blue}{b}_i)(y^j)) \\ &= \beta^{\mathcal{A}}(\text{id}_A^\#(\textcolor{blue}{b}_j)(y^j)) = 1 \end{aligned}$$

folgen, im Widerspruch dazu, dass $x^i y^j$ nicht zu L gehört. Also ist L nicht regulär.

3.6 Termsubstitution

Sei $\Sigma = (S, F)$ eine Signatur. Belegungen von Variablen durch Σ -Terme heißen **Substitutionen** und werden üblicherweise mit kleinen griechischen Buchstaben bezeichnet.

Im Gegensatz zu Kapitel 2 erlauben wir hier auch λ -Abstraktionen, Fallunterscheidungen, etc. in den Termen, deren Variablen substituiert werden sollen.

Um ungewollte Bindungen von Variablen zu vermeiden, muss die Fortsetzung

$$\sigma^* : T_{\Sigma}(V) \rightarrow T_{\Sigma}(V)$$

von σ auf Terme anders als die Auswertung $g^* : T_{\Sigma}(V) \rightarrow A$ einer Belegung $g : V \rightarrow A$ von Kapitel 2 definiert werden:

- Für alle $X \in obs(\Sigma)$, $e \in \mathcal{T}_p(S)$, $x \in V_X$ und $t \in T_\Sigma(V)_e$,
$$\sigma^*(\lambda x.t) = \begin{cases} \lambda x'.\sigma[x'/x]^*(t) & \text{falls } x \in var(\sigma(free(t) \setminus \{x\})), \\ \lambda x.\sigma[x/x]^*(t) & \text{sonst.} \end{cases}$$
- Für alle $e \in \mathcal{T}_p(S)$ $t \in T_\Sigma(V)_2$ und $u, v \in T_\Sigma(V)_e$,
$$\sigma^*(ite(t, u, v)) = ite(\sigma^*(t), \sigma^*(u), \sigma^*(v)).$$

In den restlichen Fällen ist σ^* genauso definiert wie g^* :

- Für alle $x \in V$, $\sigma^*(x) = \sigma(x)$.
- Für alle $x \in X \in obs(\Sigma)$, $\sigma^*(x) = x$.
- Für alle $n > 1$ und $t_1, \dots, t_n \in T_\Sigma(V)$, $\sigma^*(t_1, \dots, t_n) = (\sigma^*(t_1), \dots, \sigma^*(t_n))$.
- Für alle $f : e \rightarrow e' \in F$ und $t \in T_\Sigma(V)_e$, $\sigma^*(ft) = f\sigma^*(t)$.
- Für alle $X \in obs(\Sigma)$, $e \in \mathcal{T}_p(S)$, $t \in T_\Sigma(V)_{e^X}$ und $u \in T_\Sigma(V)_X$,
$$\sigma^*(t(u)) = \sigma^*(t)(\sigma^*(u)).$$

$\sigma^*(t)$ heißt **σ -Instanz** von t und **Grundinstanz**, falls σ alle Variablen von t auf Grundterme abbildet.

Man schreibt häufig $t\sigma$ anstelle von $\sigma^*(t)$ sowie $\{t_1/x_1, \dots, t_n/x_n\}$ für die Substitution σ mit $\sigma(x_i) = t_i$ für alle $1 \leq i \leq n$ und $\sigma(x) = x$ für alle $x \in V \setminus \{x_1, \dots, x_n\}$.

Satz 3.7 (Auswertung und Substitution)

(1) Für alle Belegungen $g : V \rightarrow A$ und Σ -Homomorphismen $h : A \rightarrow B$ gilt:

$$(h \circ g)^* = h \circ g^*.$$

(2) Für alle Substitutionen $\sigma, \tau : V \rightarrow T_\Sigma(V)$ gilt:

$$(\sigma^* \circ \tau)^* = \sigma^* \circ \tau^*.$$

Beweis. (1) Induktion über den Aufbau von Σ -Termen. (2) folgt aus (1), weil σ^* ein Σ -Homomorphismus ist. □

Termäquivalenz und Normalformen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur. In diesem Abschnitt geht es um Σ -Algebren A , die eine gegebene Menge E von Σ -Gleichungen (s.o.) **erfüllen**, d.h. für die Folgendes gilt:

- für alle $t = t' \in E$ und $g : V \rightarrow A$ gilt $g^*(t) = g^*(t')$.

$\text{Alg}_{\Sigma, E}$ bezeichnet die Klasse aller Σ -Algebren, die E erfüllen.

Aus Satz 3.7 (1) folgt sofort, dass $Alg_{\Sigma, E}$ unter Σ -homomorphen Bildern abgeschlossen, d.h. für alle Σ -Homomorphismen $h : A \rightarrow B$ gilt:

$$A \in Alg_{\Sigma, E} \quad \Rightarrow \quad h(A) \in Alg_{\Sigma, E}.$$

Die Elemente der Menge

$$Inst(E) =_{def} \{(t\sigma, t'\sigma) \mid (t, t') \in E, \sigma : V \rightarrow T_{\Sigma}(V)\} \quad (6)$$

heißen **Instanzen von E** .

Die E -Äquivalenz \equiv_E ist definiert als kleinste Σ -Kongruenz, die $Inst(E)$ enthält.

Aufgabe Zeigen Sie, dass der Quotient $T_{\Sigma}(V)/\equiv_E E$ erfüllt. □

Satz 3.8

Für alle Belegungen $g : V \rightarrow A$ in eine Σ -Algebra A , die E erfüllt, faktorisiert $g^* : T_{\Sigma}(V) \rightarrow A$ durch $T_{\Sigma}(V)/\equiv_E$, d.h. es gibt einen Σ -Homomorphismus $h : T_{\Sigma}(V)/\equiv_E \rightarrow A$ mit $h \circ nat_{\equiv_E} = g^*$.

$$\begin{array}{ccccc}
 V & \xrightarrow{\textcolor{blue}{inc_V}} & T_\Sigma(V) & \xrightarrow{\textcolor{blue}{nat_{\equiv_E}}} & T_\Sigma(V)/\equiv_E \\
 & \searrow g & \downarrow g^* & \nearrow (7) & \nearrow h \\
 & & A & &
 \end{array}$$

Beweis. Da der Kern von g^* eine Σ -Kongruenz ist, die $Inst(E)$ enthält (siehe [31], Satz GLK), \equiv_E aber die kleinste derartige Relation auf $T_\Sigma(V)$ ist, ist letztere im Kern von g^* enthalten. Folglich ist $h : T_\Sigma(V)/\equiv_E \rightarrow A$ mit $h([t]_{\equiv_E}) = h(t)$ für alle $t \in T_\Sigma(V)$ wohldefiniert. (7) kommutiert, was zusammen mit der Surjektivität und Σ -Homomorphie von nat_{\equiv_E} impliziert, dass auch h Σ -homomorph ist. \square

Die durch den gestrichelten Pfeil angedeutete Eigenschaft von h , der einzige Σ -Homomorphismus zu sein, der (3) kommutativ macht, folgt ebenfalls aus der Surjektivität und Σ -Homomorphie von nat_{\equiv_E} .

Zusammen mit $T_\Sigma/\equiv_E \in Alg_{\Sigma,E}$ impliziert Satz 3.8, dass T_Σ/\equiv_E initial in $Alg_{\Sigma,E}$ ist, dass es also für alle $A \in Alg_{\Sigma,E}$ genau einen Σ -Homomorphismus $h : T_\Sigma/\equiv_E \rightarrow A$ gibt.

Da g^* im Fall $V = \emptyset$ mit $\text{fold}^{\mathcal{A}}$ übereinstimmt, reduziert sich (7) zu folgendem Diagramm:

$$\begin{array}{ccc}
 T_{\Sigma} & \xrightarrow{\text{fold}^{\mathcal{A}}} & A \\
 & \searrow \text{nat}_{\equiv_E} & \swarrow h \\
 & T_{\Sigma}/\equiv_E &
 \end{array}$$

Beispiel 3.9 Sei $\Sigma = \text{Mon}$, $x, y, z \in V$ und

$$E = \{ \text{mul}(\text{one}, x) = x, \text{mul}(x, \text{one}) = x, \text{mul}(\text{mul}(x, y), z) = \text{mul}(x, \text{mul}(y, z)) \}.$$

Die freie Mon -Algebra $T_{\text{Mon}}(V)$ ist kein Monoid, wohl aber ihr Quotient $T_{\text{Mon}}(V)/\equiv_E$. Man nennt ihn das **freie Monoid** (über V).

Aufgabe Zeigen Sie, dass das freie Monoid tatsächlich ein Monoid ist, also E erfüllt, und Mon -isomorph zu V^* ist. □

Bei der Implementierung von Quotienten werden isomorphe Darstellungen bevorzugt, deren Elemente keine Äquivalenzklassen sind, diese aber eindeutig repräsentieren. Z.B. sind die Wörter über V eindeutige Repräsentanten der Äquivalenzklassen von $T_{\text{Mon}}(V)/\equiv_E$.

Bei der Berechnung äquivalenter Normalformen beschränkt man sich oft auf die folgende Teilrelation von \equiv_E , die nur “orientierte” Anwendungen der Gleichungen von E zulässt:

Die **E -Reduktionsrelation** \rightarrow_E besteht aus allen Paaren

$$(u\{t\sigma/x\}, u\{t'\sigma/x\})$$

von Σ -Termen mit $u \in T_\Sigma(V)$, $t = t' \in E$ und $\sigma : V \rightarrow T_\Sigma(V)$.

Die kleinste transitive Relation auf $T_\Sigma(V)$, die \rightarrow_E enthält, wird mit $\stackrel{+}{\rightarrow}_E$ bezeichnet.

Aufgabe Zeigen Sie, dass $\stackrel{+}{\rightarrow}_E$ die kleinste transitive und mit Σ verträgliche Relation auf $T_\Sigma(V)$ ist, die $Inst(E)$ enthält. Folgern Sie daraus, dass $\stackrel{+}{\rightarrow}_E$ eine Teilmenge von \equiv_E ist. \square

Sei $t \in T_\Sigma(V)$. $u \in T_\Sigma(V)$ heißt **E -Normalform von t** , wenn $t \stackrel{+}{\rightarrow}_E u$ gilt und u zu einer vorgegebenen Teilmenge von $T_\Sigma(V)$ gehört.

Eine Funktion $reduce : T_\Sigma(V) \rightarrow T_\Sigma(V)$ heißt **E -Reduktionsfunktion**, wenn für alle $t \in T_\Sigma(V)$, $reduce(t)$ eine E -Normalform von t ist.

Sei $A \in Alg_{\Sigma,E}$ und $g : V \rightarrow A$. Wegen $\stackrel{+}{\rightarrow}_E \subseteq \equiv_E \subseteq ker(g^*)$ (siehe obige Aufgabe und den Beweis von Satz 3.8) gilt

$$g^* \circ reduce = g^*. \tag{8}$$

Allgemeine Methoden zur Berechnung von Normalformen werden in [31], §5.7, behandelt.

Beispiel 3.10 Normalformen regulärer Ausdrücke

E bestehe aus folgenden $Reg(BL)$ -Gleichungen:

$f(f(x, y), z) = f(x, f(y, z))$	(Assoziativität von $f \in \{par, seq\}$)
$par(x, y) = par(y, x)$	(Kommutativität von par)
$seq(x, par(y, z)) = par(seq(x, y), seq(x, z))$	(Linksdistributivität von seq über par)
$seq(par(x, y), z) = par(seq(x, z), seq(y, z))$	(Rechtsdistributivität von seq über par)
$par(x, x) = x$	(Idempotenz von par)
$par(mt, x) = x$	$par(x, mt) = x$ (Neutralität von mt bzgl. par)
$seq(eps, x) = x$	$seq(x, eps) = x$ (Neutralität von eps bzgl. seq)
$seq(mt, x) = mt$	$seq(x, mt) = mt$ (Annihilation)
$f(ite(x, y, z), z') = ite(x, f(y, z'), f(z, z'))$	($f \in \{par, seq\}$)
$f(z', ite(x, y, z)) = ite(x, f(z', y), f(z', z))$	($f \in \{par, seq\}$)

Demnach entfernt eine E -Reduktionsfunktion mt und Mehrfachkopien von Summanden aus Summen sowie eps aus Produkten, ersetzt alle Produkte, die mt enthalten, durch mt , distribuiert seq über par und linearisiert geschachtelte Summen und Produkte rechtsassoziativ. Angewendet auf $Reg(BL)$ -Grundterme entspricht sie deren Faltung in der $Reg(BL)$ -Algebra **regNorm** (siehe **Compiler.hs**).

Sei $X = \bigcup BL$. Da $\text{Lang}(X)$ E erfüllt, gilt (8) für $A = \text{Lang}(X)$.

Algebren, die E erfüllen, heißen idempotente **Semiringe**.

Kleene-Algebren sind Semiringe, auf denen ein Sternoperator definiert ist und die neben E weitere (den Sternoperator betreffende) Gleichungen erfüllen. Die Elemente von $\text{Beh}(X, Y)$ (siehe 2.10) heißen **formale Potenzreihen**, wenn Y ein Semiring ist (siehe [39], Kapitel 9).

3.11 Die Brzozowski-Gleichungen

Die folgende Menge **BRE** von – **Brzozowski-Gleichungen** genannten – $\text{Reg}(BL)$ -Gleichungen hat in $T_{\text{Reg}(BL)}$ genau eine Lösung, d.h. es gibt genau eine Erweiterung von $T_{\text{Reg}(BL)}$ zur $\text{Acc}(X)$ -Algebra, die **BRE** erfüllt (siehe Beispiel 17.4).

In Abschnitt 2.10 wurde diese Lösung unter dem Namen $\text{Bro}(BL)$ eingeführt. Existenz und Eindeutigkeit folgen aus Satz 17.1 und werden dort aus dem in Satz 3.2 (3) eingeführten Induktionsprinzip abgeleitet.

$$\begin{aligned}
\delta(\text{base}(B)) &= \lambda x.\text{ite}(x \in B, \text{base}(1), \text{base}(\emptyset)) \\
\delta(\text{par}(t, u)) &= \lambda x.\text{par}(\delta(t)(x), \delta(u)(x)) \\
\delta(\text{seq}(t, u)) &= \lambda x.\text{par}(\text{seq}(\delta(t)(x), u), \text{ite}(\beta(t), \delta(u)(x), \text{mt})) \\
\delta(\text{iter}(t)) &= \lambda x.\text{seq}(\delta(t)(x), \text{iter}(t)) \\
\beta(\text{base}(B)) &= \text{ite}(B = 1, 1, 0) \\
\beta(\text{par}(t, u)) &= \max\{\beta(t), \beta(u)\} \\
\beta(\text{seq}(t, u)) &= \beta(t) * \beta(u) \\
\beta(\text{iter}(t)) &= 1
\end{aligned}$$

t und u sind hier *Variablen* der Sorte *reg*.

Nach obiger Lesart definiert *BRE* Destruktoren (δ und β) auf der Basis von Konstruktoren von *Reg(BL)*. Umgekehrt lässt sich *BRE* auch als Definition der Konstruktoren auf der Basis der Destruktoren δ und β auffassen: Nach Satz 17.7 hat *BRE* nämlich in der finalen *Acc(X)*-Algebra *Pow(X)* genau eine *coinduktive* Lösung, d.h. es gibt genau eine Erweiterung von *Lang(X)* zur *Reg(BL)*-Algebra, die *BRE* erfüllt (siehe Beispiel 17.10). \square

3.12 Optimierter Brzozowski-Automat

Der Erkenner $\text{Bro}(BL)$ regulärer Sprachen (siehe Abschnitt 2.6) benötigt viel Platz, weil die wiederholten Aufrufe von $\delta^{\text{Bro}(BL)}$ aus t immer größere Ausdrücke erzeugen. Um das zu vermeiden, ersetzen wir $\text{Bro}(BL)$ durch die $\text{Acc}(X)$ -Algebra $\text{Norm}(BL)$ (**norm** in `Compiler.hs`), die bis auf die Interpretation von δ mit $\text{Bro}(BL)$ übereinstimmt. $\delta^{\text{Norm}(BL)}$ normalisiert die von $\delta^{\text{Bro}(BL)}$ berechneten Folgezustände mit der in Beispiel 3.10 beschriebenen Reduktionsfunktion

$$\text{reduce} : T_{\text{Reg}(BL)}(V) \rightarrow T_{\text{Reg}(BL)}(V).$$

Für alle $t \in T_{\text{Reg}(BL)}$,

$$\delta^{\text{Norm}(BL)}(t) =_{\text{def}} \text{reduce} \circ \delta^{\text{Bro}(BL)}(t). \quad (1)$$

Gemäß Beispiel 3.10 gilt

$$\text{fold}^{\text{Lang}(X)} \circ \text{reduce} = \text{fold}^{\text{Lang}(X)}. \quad (2)$$

Es bleibt zu zeigen, dass für alle $t \in T_{\text{Reg}(BL)}$ die initialen Automaten $(\text{Bro}(BL), t)$ und $(\text{Norm}(BL), t)$ dieselben Sprachen erkennen, d.h.

$$\text{unfold}^{\text{Norm}(BL)}(t) = \text{unfold}^{\text{Bro}(BL)}(t). \quad (3)$$

Wir beweisen (3) durch Induktion über die Länge der Wörter über X .

$$\beta^{\text{Norm}(BL)}(\text{id}_T^\#(t)(\epsilon)) = \beta^{\text{Norm}(BL)}(t) = \beta^{\text{Bro}(BL)}(t) = \beta^{\text{Bro}(BL)}(\text{id}_T^\#(t)(\epsilon)).$$

Für alle $x \in X$ und $w \in X^*$,

$$\begin{aligned}
& \beta^{Norm(BL)}(id_T^\#(t)(xw)) = \beta^{Norm(BL)}(id_T^\#(\delta^{Norm(BL)}(t)(x))(w)) \\
&= unfold^{Norm(BL)}(\delta^{Norm(BL)}(t)(x))(w) \stackrel{ind. \ hyp.}{=} unfold^{Bro(BL)}(\delta^{Norm(BL)}(t)(x))(w) \\
&\stackrel{(1)}{=} unfold^{Bro(BL)}(reduce(\delta^{Bro(BL)}(t))(x))(w) = fold^{Lang(X)}(reduce(\delta^{Bro(BL)}(t))(x))(w) \\
&\stackrel{(2)}{=} fold^{Lang(X)}(\delta^{Bro(BL)}(t)(x))(w) = unfold^{Bro(BL)}(\delta^{Bro(BL)}(t)(x))(w) \\
&= \beta^{Bro(BL)}(id_T^\#(\delta^{Bro(BL)}(t)(x))(w)) = \beta^{Bro(BL)}(id_T^\#(t)(xw)).
\end{aligned}$$

Also gilt (3):

$$\begin{aligned}
unfold^{Norm(BL)}(t) &= \{w \in X^* \mid \beta^{Norm(BL)}(id_T^\#(t)(w)) = 1\} \\
&= \{w \in X^* \mid \beta^{Bro(BL)}(id_T^\#(t)(w)) = 1\} = unfold^{Bro(BL)}(t).
\end{aligned}$$

Im Haskell-Modul `Compiler.hs` entspricht der Erkenner $unfold^{Norm(BL)}(t)$ dem Aufruf `regToAlg "" w 4`, wobei `w` die Wortdarstellung von t ist. □

3.13 Erkenner regulärer Sprachen sind endlich

Aus der Gültigkeit von BRE in $Pow(X)$ lassen sich die folgenden Gleichungen für

$$id_{\mathcal{P}(X^*)}^\# : \mathcal{P}(X^*) \rightarrow \mathcal{P}(X^*)^{X^*}$$

ableiten (siehe 2.4): Für alle $w \in X^*$, $B \in BL$ und $L, L' \subseteq \mathcal{P}(X^*)$,

$$id_{\mathcal{P}(X^*)}^\#(\textcolor{red}{eps}^{Lang(X)})(w) = \begin{cases} 1 & \text{falls } w = \epsilon, \\ \emptyset & \text{sonst,} \end{cases} \quad (4)$$

$$id_{\mathcal{P}(X^*)}^\#(\textcolor{red}{mt}^{Lang(X)})(w) = \emptyset, \quad (5)$$

$$id_{\mathcal{P}(X^*)}^\#(\overline{B}^{Lang(X)})(w) = \begin{cases} C & \text{falls } w = \epsilon, \\ 1 & \text{falls } w \in C, \\ \emptyset & \text{sonst,} \end{cases} \quad (6)$$

$$id_{\mathcal{P}(X^*)}^\#(\textcolor{red}{par}^{Lang(X)}(L, L'))(w) = id_{\mathcal{P}(X^*)}^\#(L)(w) \cup id_{\mathcal{P}(X^*)}^\#(L')(w), \quad (7)$$

$$\begin{aligned} id_{\mathcal{P}(X^*)}^\#(\textcolor{red}{seq}^{Lang(X)}(L, L'))(w) &= \{uv \mid u \in id_{\mathcal{P}(X^*)}^\#(L)(w), v \in L'\} \\ &\cup \bigcup_{uv=w} (\text{if } \epsilon \in id_{\mathcal{P}(X^*)}^\#(L)(u) \\ &\quad \text{then } id_{\mathcal{P}(X^*)}^\#(L')(v) \text{ else } \emptyset), \end{aligned} \quad (8)$$

$$\begin{aligned} id_{\mathcal{P}(X^*)}^\#(\textcolor{red}{iter}^{Lang(X)}(L))(w) &= \{uv \mid u \in id_{\mathcal{P}(X^*)}^\#(L)(w), v \in iter^{Pow(X)}(L)\} \\ &\cup \bigcup_{u_1 \dots u_n v=w} (\text{if } \epsilon \in \bigcap_{i=1}^n id_{\mathcal{P}(X^*)}^\#(L)(u_i) \\ &\quad \text{then } \{uv' \mid u \in id_{\mathcal{P}(X^*)}^\#(L)(v), \\ &\quad v' \in iter^{Pow(X)}(L)\} \\ &\quad \text{else } \emptyset) \end{aligned} \quad (9)$$

(siehe z.B. [39], Theorem 10.1).

Wir erinnern an Satz 2.20 (iii), aus dem folgt, dass für alle $L \subseteq X^*$ der Unterautomat $(\langle L \rangle, L)$ von $(\text{Pow}(X), L)$ ein minimaler Erkennender von L ist. Ist L die Sprache eines regulären Ausdrucks t , ist also $L = \text{fold}^{\text{Lang}(X)}(t)$, dann ist

$$\langle L \rangle = \{id_{\mathcal{P}(X^*)}^\#(\text{fold}^{\text{Lang}(X)}(t))(w) \mid w \in X^*\}.$$

Daraus folgt durch Induktion über den Aufbau von t , dass $\langle L \rangle$ endlich ist:

Im Fall $t \in \{\text{eps}, mt\} \cup \{\overline{B} \mid B \in BL\}$ besteht $\langle L \rangle$ wegen (4), (5) und (6) aus zwei, einem bzw. drei Zuständen.

Im Fall $t = \text{par}(t', t'')$ folgt $|\langle L \rangle| \leq |\langle \text{fold}^{\text{Lang}(X)}(t') \rangle| * |\langle \text{fold}^{\text{Lang}(X)}(t'') \rangle|$ aus (7).

Im Fall $t = \text{seq}(t', t'')$ folgt $|\langle L \rangle| \leq |\langle \text{fold}^{\text{Lang}(X)}(t') \rangle| * 2^{|\langle \text{fold}^{\text{Lang}(X)}(t'') \rangle|}$ aus (8).

Im Fall $t = \text{iter}(t')$ folgt $|\langle L \rangle| \leq 2^{|\langle \text{fold}^{\text{Lang}(X)}(t') \rangle|}$ aus (9).

Damit ist ohne den üblichen Umweg über Potenzautomaten (siehe Beispiel 12.3) gezeigt, dass reguläre Sprachen von endlichen Automaten erkannt werden.

4 Kontextfreie Grammatiken (CFGs)

Sie ordnen einer konstruktiven Signatur Σ eine **konkrete Syntax** und damit eine vom Compiler verstehbare **Quellsprache** zu, so dass er diese in eine als Σ -Algebra formulierte **Zielsprache** übersetzen kann. Auch wenn die Quellsprache bereits als konstruktive Signatur Σ und die Zielsprache als Σ -Algebra gegeben sind, benötigt der Compiler eine kontextfreie Grammatik, um Zeichenfolgen in Elemente der Algebra zu übersetzen.

Eine **kontextfreie Grammatik (CFG)**

$$G = (S, BS, R)$$

besteht aus

- einer endlichen Menge S von **Sorten**, die auch *Nichtterminale* oder *Variablen* genannt werden,
- einer endlichen Menge BS nichtleerer **Basismengen**,
- einer endlichen Menge R von **Regeln** $s \rightarrow w$ mit $s \in S$ und $w \in (S \cup BS)^*$ \ { s }, die auch genannt werden.

$Z(G)$ bezeichnet die Menge der **Terminale** von G , das sind die Singletons (einelementigen Mengen) von BS . Sie werden oft mit ihrem jeweiligen einzigen Element gleichgesetzt.

$n > 0$ Regeln $s \rightarrow w_1, \dots, s \rightarrow w_n$ mit derselben linken Seite s werden oft zu der einen Regel $s \rightarrow w_1 | \dots | w_n$ zusammengefasst.

Beispiel 4.1

Sei BL wie in Abschnitt 2.4, $\text{syms}(BL)$ wie in 2.9 und

$$\begin{aligned} R = & \{ \text{reg} \rightarrow \text{reg} + \text{reg}, \text{ reg} \rightarrow \text{reg reg}, \text{ reg} \rightarrow \text{reg}^*, \text{ reg} \rightarrow (\text{reg}) \} \\ & \cup \{ \text{reg} \rightarrow \overline{B} \mid B \in BL \}. \end{aligned}$$

$\text{REG} =_{\text{def}} (\{\text{reg}\}, \text{syms}(BL), R)$ ist eine CFG. Offenbar sind hier alle Basismengen Singletons, also Terminale. Für jede Basissprache $B \in BL$ liefert \overline{B} einen eindeutigen Namen für B . □

Oft genügt es, bei der Definition einer CFG nur deren Regeln und Basismengen anzugeben. Automatisch bilden dann die Symbole, die auf der linken Seite einer Regel vorkommen, die Menge S der Sorten, während alle anderen Wörter und Symbole (außer dem senkrechten Strich $|$; s.o.) auf der rechten Seite einer Regel terminale oder Namen mehrelementiger Basismengen sind.

Beispiel 4.2

Die Regeln der CFG **JavaLight** für imperative Programme mit Konditionalen und Schleifen lauten wie folgt:

$$\begin{array}{lcl} Commands & \rightarrow & Command \; Commands \mid Command \\ Command & \rightarrow & \{Commands\} \mid String = Sum; \mid \\ & & \text{if } Disjunct \; Command \; \text{else } Command \mid \\ & & \text{if } Disjunct \; Command \mid \text{while } Disjunct \; Command \\ Sum & \rightarrow & Sum + Prod \mid Sum - Prod \mid Prod \\ Prod & \rightarrow & Prod * Factor \mid Prod / Factor \mid Factor \\ Factor & \rightarrow & \mathbb{Z} \mid String \mid (Sum) \\ Disjunct & \rightarrow & Conjunct \parallel Disjunct \mid Conjunct \\ Conjunct & \rightarrow & Literal \&& Conjunct \mid Literal \\ Literal & \rightarrow & !Literal \mid Sum \; Rel \; Sum \mid 2 \mid (Disjunct) \end{array}$$

String, *Z*, *Rel* und *2* (als Menge Boolescher Werte) sind die mehrelementigen Basismengen von JavaLight.

String bezeichnet die Menge aller Zeichenfolgen außer Elementen anderer Basismengen von JavaLight.

Rel bezeichnet eine Menge nicht näher spezifizierter binärer Relationen auf \mathbb{Z} .

Die Verwendung von jeweils drei Sorten für arithmetische bzw. Boolesche Ausdrücke berücksichtigt die üblichen Prioritäten arithmetischer bzw. Boolescher Operationen und erlaubt daher die Vermeidung überflüssiger Klammern.

Ein aus der Sorte *Commands* ableitbares JavaLight-Programm ist z.B.

```
fact = 1; while x > 1 {fact = fact*x; x = x-1;} □
```

Beispiel 4.3

XMLstore (siehe [21], Abschnitt 2)

<i>Store</i>	$\rightarrow \langle store \rangle \langle stock \rangle Stock \langle /stock \rangle \langle /store \rangle \mid \langle store \rangle Orders \langle stock \rangle Stock \langle /stock \rangle \langle /store \rangle$
<i>Orders</i>	$\rightarrow Order\ Orders \mid Order$
<i>Order</i>	$\rightarrow \langle order \rangle \langle customer \rangle Person \langle /customer \rangle Items \langle /order \rangle$
<i>Person</i>	$\rightarrow \langle name \rangle String \langle /name \rangle \mid \langle name \rangle String \langle /name \rangle Emails$
<i>Emails</i>	$\rightarrow Email\ Emails \mid \epsilon$
<i>Email</i>	$\rightarrow \langle email \rangle String \langle /email \rangle$
<i>Items</i>	$\rightarrow Item\ Items \mid Item$
<i>Item</i>	$\rightarrow \langle item \rangle Id \langle price \rangle String \langle /price \rangle \langle /item \rangle$
<i>Stock</i>	$\rightarrow ItemS\ Stock \mid ItemS$
<i>ItemS</i>	$\rightarrow \langle item \rangle Id \langle quantity \rangle \mathbb{Z} \langle /quantity \rangle Suppliers \langle /item \rangle$
<i>Suppliers</i>	$\rightarrow \langle supplier \rangle Person \langle /supplier \rangle \mid Stock$
<i>Id</i>	$\rightarrow \langle id \rangle String \langle /id \rangle$

Die Sprache von XMLstore beschreibt XML-Dokumente wie z.B. das folgende:

```
<store> <order> <customer> <name> John Mitchell </name>
          <email> j.mitchell@yahoo.com </email>
        </customer>
        <item> <id> I18F </id> <price> 100 </price> </item>
      </order>
      <stock>
        <item> <id> IG8 </id> <quantity> 10 </quantity>
          <supplier> <name> Al Jones </name>
                      <email> a.j@gmail.com </email>
                      <email> a.j@dot.com </email>
        </supplier>
      </item>
      <item> <id> J38H </id> <quantity> 30 </quantity>
        <item> <id> J38H1 </id> <quantity> 10 </quantity>
          <supplier> <name> Richard Bird </name> </supplier>
        </item>
        <item> <id> J38H2 </id> <quantity> 20 </quantity>
          <supplier> <name> Mick Taylor </name> </supplier>
        </item>
      </item>
    </stock>
</store>
```

Linksrekursive CFGs

Sei $G = (S, BS, R)$ eine CFG und $X = \bigcup BS$.

X ist die Menge der *Eingabesymbole*, die Compiler für G verarbeiten. Demgegenüber tauchen auf den rechten Seiten der *Regeln* von G nur (Namen für) komplette Basismengen auf!

Die klassische Definition einer linksrekursiven Grammatik verwendet die **direkte Ableitungsrelation**

$$\rightarrow_G = \{(vsw, v\alpha w) \mid s \rightarrow \alpha \in R, v, w \in (S \cup BS)^*\}.$$

\rightarrow_G^+ und \rightarrow_G^* bezeichnen den transitiven bzw. reflexiv-transitiven Abschluss von \rightarrow_G .

G heißt **linksrekursiv**, falls es eine **linksrekursive Ableitung** $s \xrightarrow{+}_G sv$ gibt.

G heißt **LL-kompilierbar**, falls es eine partielle Ordnung \leq auf S gibt mit $s' \leq s$ für alle Ableitungen $sv \xrightarrow{+}_G s'w$.

Z.B. ist *REG* LL-kompilierbar, *JavaLight* und *XMLstore* jedoch nicht (siehe Beispiele 4.1-4.3).

Aufgabe

Zeigen sie, dass eine LL-kompilierbare Grammatik genau dann linksrekursiv ist, wenn sie eine **linksrekursive Regel** $s \rightarrow sw$ enthält. □

Linksassoziative Auswertung erzwingt keine Linksrekursion

Sind alle Regeln, deren abstrakte Syntax binäre Operationen darstellen, dann werden aus diesen Operationen bestehende Syntaxbäume rechtsassoziativ ausgewertet. Bei ungeklammerten Ausdrücke wie $x + y - z - z'$ oder $x/y * z/z'$ führt aber nur die linksassoziative Faltung zum gewünschten Ergebnis. Die gegebene Grammatik muss deshalb um Sorten und Regeln erweitert werden, die bewirken, dass aus der linksassoziativen eine semantisch äquivalente rechtsassoziative Faltung wird.

Im Fall von **JavaLight** bewirken das die Sorten *Sumsect* und *Prodsect* und deren Regeln. Die Namen der Sorten weisen auf deren Interpretation als Mengen von **Sektionen** hin, also von Funktionen wie z.B. $(*)^5 : \mathbb{Z} \rightarrow \mathbb{Z}$. Angewendet auf eine Zahl x , liefert $(*)^5$ das Fünffache von x . Die linksassoziative Faltung $((x+y)-z)-z'$ bzw. $((x/y)*z)/z'$ der obigen Ausdrücke entspricht daher der – von den Regeln für *Sumsect* und *Prodsect* bewirkten – rechtsassoziativen Faltung $((-z') \circ ((-z) \circ (+y)))(x)$ bzw. $((/z') \circ ((*z) \circ (/y)))(x)$.

Die Sektionssorten von JavaLight werden automatisch eingeführt, wenn man die folgende Entrekursivierung auf JavaLight anwendet.

Verfahren zur Eliminierung von Linksrekursion

Sei $G = (S, BS, R)$ eine CFG und $S = \{s_1, \dots, s_n\}$.

Führe für alle $1 \leq i \leq n$ die beiden folgenden Schritte in der angegebenen Reihenfolge durch:

- Für alle $1 \leq j < i$ und Regelpaare $(s_i \rightarrow s_j v, s_j \rightarrow w)$ ersetze die Regel $s_i \rightarrow s_j v$ durch die neue Regel $s_i \rightarrow wv$.
- Falls vorhanden, streiche die Regel $s_i \rightarrow s_i$.
- Für alle Regelpaare $(s_i \rightarrow v, s_i \rightarrow s_i w)$ mit $v \notin \{s_i\} \times (S \cup BS)^*$ ersetze beide Regeln durch die drei neuen Regeln $s_i \rightarrow vs'_i$, $s'_i \rightarrow ws'_i$ und $s'_i \rightarrow \epsilon$.

Die ersten beiden Schritte dienen der Überführung von G in eine LL-kompilierbare CFG. Der dritte Schritt transformiert diese in eine nicht-linksrekursive Grammatik, die – beschränkt auf die Sorten von G – zu G äquivalent ist, d.h. dieselbe Sprache wie G hat (s.u.).

□

Beispiel 4.2 (Fortsetzung)

Eine mit obigem Algorithmus erzeugte nicht-linksrekursive Version von JavaLight hat folgende Regeln:

$$\begin{array}{lcl} Commands & \rightarrow & Command\ Commands \mid Command \\ Command & \rightarrow & \{Commands\} \mid \textcolor{red}{String} = Sum; \mid \\ & & \text{if } Disjunct\ Command \text{ else } Command \mid \\ & & \text{if } Disjunct\ Command \mid \text{while } Disjunct\ Command \\ Sum & \rightarrow & Prod\ Sumsect \\ Sumsect & \rightarrow & +Prod\ Sumsect \mid -Prod\ Sumsect \mid \epsilon \\ Prod & \rightarrow & Factor\ Prodsect \\ Prodsect & \rightarrow & *Factor\ Prodsect \mid /Factor\ Prodsect \mid \epsilon \\ Factor & \rightarrow & \mathbb{Z} \mid \textcolor{red}{String} \mid (Sum) \\ Disjunct & \rightarrow & Conjunct \parallel Disjunct \mid Conjunct \\ Conjunct & \rightarrow & Literal\ \&\&\ Conjunct \mid Literal \\ Literal & \rightarrow & !Literal \mid Sum\ \textcolor{red}{Rel}\ Sum \mid 2 \mid (Disjunct) \end{array}$$



Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG,

$$\begin{aligned} \text{recs}(S) &= \{s \in S \mid \exists s \rightarrow sw \in R\}, \\ \text{nonrecs}(R) &= \{s \rightarrow v \in R \mid v \notin \{s\} \times (S \cup BS)^*\} \end{aligned}$$

und $G' = (S', BS, R')$ die nach obigem Verfahren aus G gebildete nicht-linksrekursive CFG.

Dann gilt

$$\begin{aligned} S' &= S \cup \{s' \mid s \in \text{recs}(S)\}, \\ R' &= R \setminus \{s \rightarrow w \in R \mid s \in \text{recs}(S)\} \\ &\quad \cup \{s \rightarrow vs' \mid s \in \text{recs}(S), s \rightarrow v \in \text{nonrecs}(R)\} \\ &\quad \cup \{s' \rightarrow ws' \mid s \in \text{recs}(S), s \rightarrow sw \in R\} \\ &\quad \cup \{s' \rightarrow \epsilon \mid s \in \text{recs}(S)\}. \end{aligned}$$

Offenbar haben alle in G oder G' aus s ableitbaren terminalen Wörter die Form $vw_1 \dots w_n$. Außerdem ist jedes Wort genau dann in G aus s ableitbar, wenn es in G' aus s ableitbar ist.

Abstrakte Syntax

Sei $G = (S, BS, R)$ eine CFG.

Die folgende Funktion $\text{typ} : (S \cup BS)^* \rightarrow \mathcal{T}_p(S)$ streicht alle Elemente von $Z(G)$ aus Wörtern über $S \cup BS$ und überführt diese in die durch sie bezeichneten Produkttypen:

- $\text{typ}(\epsilon) = 1$.
- Für alle $s \in S \cup BS \setminus Z(G)$ und $w \in (S \cup BS)^*$, $\text{typ}(sw) = s \times \text{typ}(w)$.
- Für alle $x \in Z$ und $w \in (S \cup BS)^*$, $\text{typ}(xw) = \text{typ}(w)$.

Die konstruktive Signatur

$$\Sigma(G) = (S, BS, \{f_{s \rightarrow w} : \text{typ}(w) \rightarrow s \mid s \rightarrow w \in R\})$$

heißt **abstrakte Syntax von G** . Demnach gilt:

$$\text{obs}(\Sigma(G)) \subseteq BS \setminus Z(G).$$

$\Sigma(G)$ -Grundterme werden **Syntaxbäume von G** genannt.

Beispiel 4.4

Die Signatur $Reg(BL)$ ist eine Teilsignatur der abstrakten Syntax der CFG REG von Beispiel 4.1.

Aufgabe

Zu welcher Regel von REG fehlt in $Reg(BL)$ der entsprechende Konstruktor? □

Die Konstruktoren von $\Sigma(G)$ lassen sich i.d.R. direkt aus den Terminalen von G basteln. So wird manchmal für den aus der Regel $r = (s \rightarrow w_0e_1w_1 \dots e_nw_n)$ mit $w_0, \dots, w_n \in Z^*$ und $e_1, \dots, e_n \in S \cup BS \setminus Z(G)$ entstandenen Konstruktor f_r die (Mixfix-)Darstellung $w_0_w_1 \dots _w_n$ gewählt.

So könnte beispielsweise der Konstruktor

$$f : Disjunct \times Command \times Command \rightarrow Commands$$

für die JavaLight-Regel

$$Commands \rightarrow \text{if } Disjunct \text{ Command else Command}$$

(siehe Beispiel 4.2) **if_else** genannt werden. Um Verwechslungen zwischen Terminalen und Konstruktoren vorzubeugen, werden wir hier jedoch keine Terminalen als Namen für Konstruktoren verwenden.

Umgekehrt kann jede endlich verzweigende konstruktive Signatur $\Sigma = (S, F)$ in eine CFG

$$G(\Sigma) = (S, obs(\Sigma) \cup \{(,), , \}, R)$$

mit $\Sigma(G(\Sigma)) = \Sigma$ überführt werden, wobei

$$R = \{s \rightarrow f(e_1, \dots, e_n) \mid f : e_1 \times \dots \times e_n \rightarrow s \in F\}.$$

Beispiel 4.5 SAB

Die Grammatik SAB besteht aus den Sorten S, A, B , den Terminalen a, b und den Regeln

$$\begin{aligned} r_1 &= S \rightarrow aB, & r_2 &= S \rightarrow bA, & r_3 &= S \rightarrow \epsilon, \\ r_4 &= A \rightarrow aS, & r_5 &= A \rightarrow bAA, & r_6 &= B \rightarrow bS, & r_7 &= B \rightarrow aBB. \end{aligned}$$

Demnach lauten die Konstruktoren der abstrakten Syntax von SAB wie folgt:

$$\begin{aligned} f_1 &: B \rightarrow S, & f_2 &: A \rightarrow S, & f_3 &: 1 \rightarrow S, \\ f_4 &: S \rightarrow A, & f_5 &: A \times A \rightarrow A, & f_6 &: S \rightarrow B, & f_7 &: B \times B \rightarrow B. \end{aligned}$$

Die folgende SAB-Algebra $SABcount$ berechnet die Anzahl $\#a(w)$ bzw. $\#b(w)$ der Vorkommen von a bzw. b im Eingabewort w :

$$SABcount_S = SABcount_A = SABcount_B = \mathbb{N}^2,$$

$$f_1^{SABcount} = f_4^{SABcount} = \lambda(i, j).(i + 1, j),$$

$$f_2^{SABcount} = f_6^{SABcount} = \lambda(i, j).(i, j + 1),$$

$$f_3^{SABcount} = (0, 0),$$

$$f_5^{SABcount} = \lambda((i, j), (k, l)).(i + k, j + l + 1),$$

$$f_7^{SABcount} = \lambda((i, j), (k, l)).(i + k + 1, j + l).$$

□

Beispiel 4.6 Die abstrakte Syntax (S, F) von JavaLight lautet wie folgt:

$$\begin{aligned} S &= \{ \text{Commands}, \text{Command}, \text{Sum}, \text{Prod}, \text{Factor}, \text{Disjunct}, \text{Conjunct}, \text{Literal} \} \\ I &= \{ \mathbb{Z}, \text{String}, \text{Rel}, 2 \} \\ F &= \{ \begin{aligned} &\text{seq} : \text{Command} \times \text{Commands} \rightarrow \text{Commands}, \\ &\text{embed} : \text{Command} \rightarrow \text{Commands}, \\ &\text{block} : \text{Commands} \rightarrow \text{Command}, \\ &\text{assign} : \text{String} \times \text{Sum} \rightarrow \text{Command}, \\ &\text{cond} : \text{Disjunct} \times \text{Command} \times \text{Command} \rightarrow \text{Command}, \\ &\text{cond1}, \text{loop} : \text{Disjunct} \times \text{Command} \rightarrow \text{Command}, \\ &\underline{\text{sum}} : \text{Prod} \rightarrow \text{Sum}, \\ &\underline{\text{plus}}, \underline{\text{minus}} : \text{Sum} \times \text{Prod} \rightarrow \text{Sum}, \\ &\underline{\text{prod}} : \text{Factor} \rightarrow \text{Prod}, \\ &\underline{\text{times}}, \underline{\text{div}} : \text{Prod} \times \text{Factor} \rightarrow \text{Prod}, \\ &\text{embedI} : \mathbb{Z} \rightarrow \text{Factor}, \\ &\text{var} : \text{String} \rightarrow \text{Factor}, \\ &\text{encloseS} : \text{Sum} \rightarrow \text{Factor}, \\ &\text{disjunct} : \text{Conjunct} \times \text{Disjunct} \rightarrow \text{Disjunct}, \\ &\text{embedC} : \text{Conjunct} \rightarrow \text{Disjunct}, \\ &\text{conjunct} : \text{Literal} \times \text{Conjunct} \rightarrow \text{Conjunct}, \\ &\text{embedL} : \text{Literal} \rightarrow \text{Conjunct}, \\ &\text{not} : \text{Literal} \rightarrow \text{Literal}, \end{aligned} \quad (1) \end{aligned}$$

$$\begin{aligned}
 atom : Rel \times Sum \times Sum &\rightarrow Literal, \\
 embedB : 2 &\rightarrow Literal, \\
 encloseD : Disjunct &\rightarrow Literal \}
 \end{aligned}$$

Beim Typ von *atom* wurden die ersten beiden Faktoren gegenüber der zugrundeliegenden JavaLight-Regel *Literal* \rightarrow *Sum Rel Sum* vertauscht.

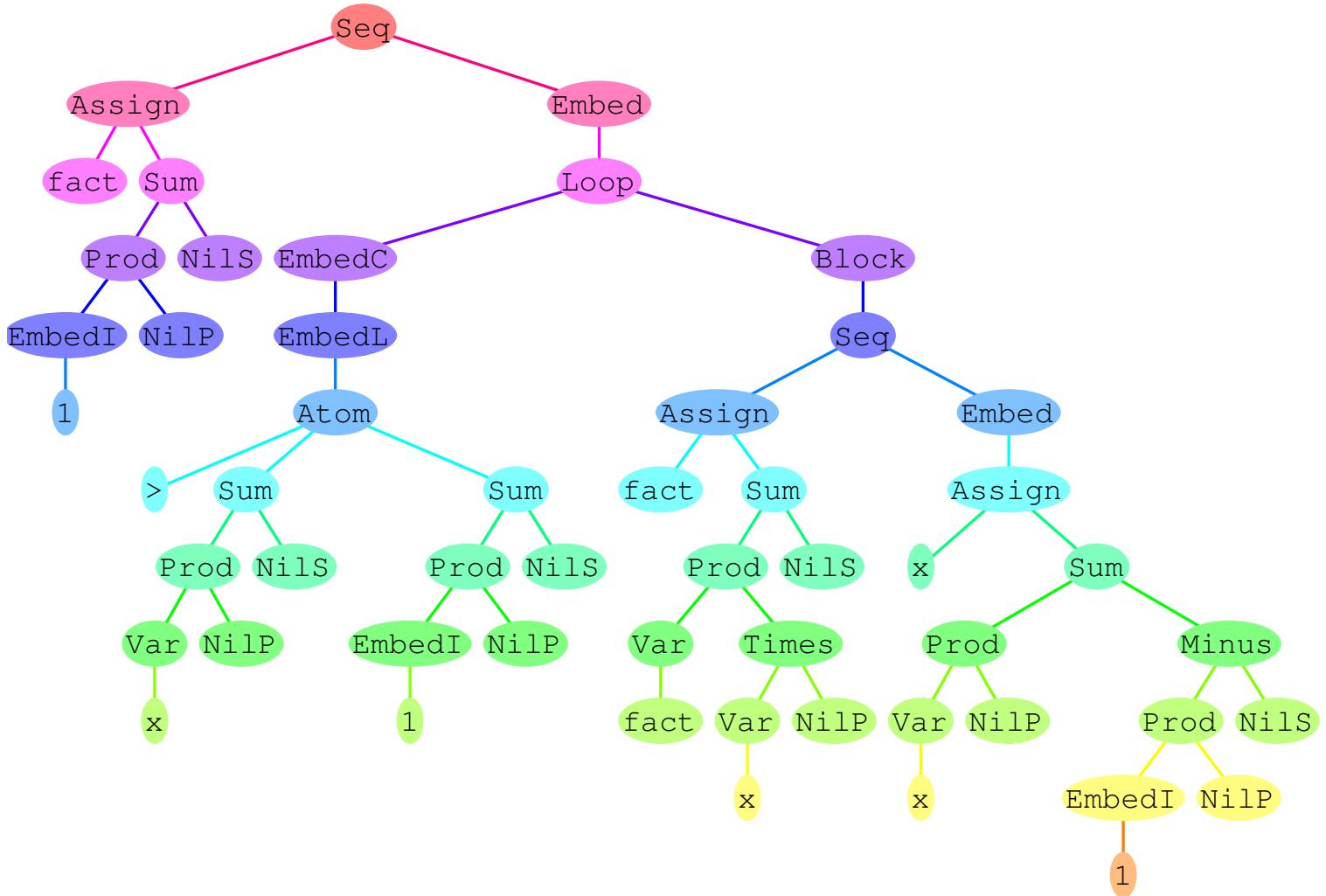
Die abstrakte Syntax der nach obigem Verfahren aus JavaLight gebildeten nicht-linksrekursiven Grammatik *JavaLight'* enthält zusätzlich die Sorten *Sumsect* und *Prodsect* sowie folgende Konstruktoren anstelle von (1)-(4):

$$\begin{aligned}
 sum : Prod \times Sumsect &\rightarrow Sum, \\
 plus, minus : Prod \times Sumsect &\rightarrow Sumsect, \\
 nilS : 1 &\rightarrow Sumsect, \\
 prod : Factor \times Prodsect &\rightarrow Prod, \\
 times, div : Factor \times Prodsect &\rightarrow Prodsect, \\
 nilP : 1 &\rightarrow Prodsect.
 \end{aligned}$$

Z.B. hat das JavaLight-Programm

```
fact = 1; while x > 1 {fact = fact * x; x = x - 1; }
```

folgenden JavaLight'-Syntaxbaum:



Der Anfangsbuchstabe eines Konstruktors ist hier Haskell-gemäß großgeschrieben.

`javaToAlg "prog" 1` (siehe `Java.hs`) übersetzt das JavaLight-Programm in der Datei `prog` in den zugehörigen Syntaxbaum und schreibt diesen in die Datei `Pix/javaterm.svg`. Mit <https://cloudconvert.org> kann diese in ein anderes Format konvertiert werden. \square

Linksrekursive CFGs und abstrakte Syntax

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG und $G' = (S', BS, R')$ die wie oben aus G gebildete nicht-linksrekursive CFG.

Die Faltung eines Syntaxbaums t von G in folgender $\Sigma(G)$ -Algebra $derec(G)$ liefert den t entsprechenden Syntaxbaum von G' :

- Für alle $s \in S \cup BS \setminus Z(G)$, $derec(G)_s = T_{\Sigma(G'),s}$. (1)

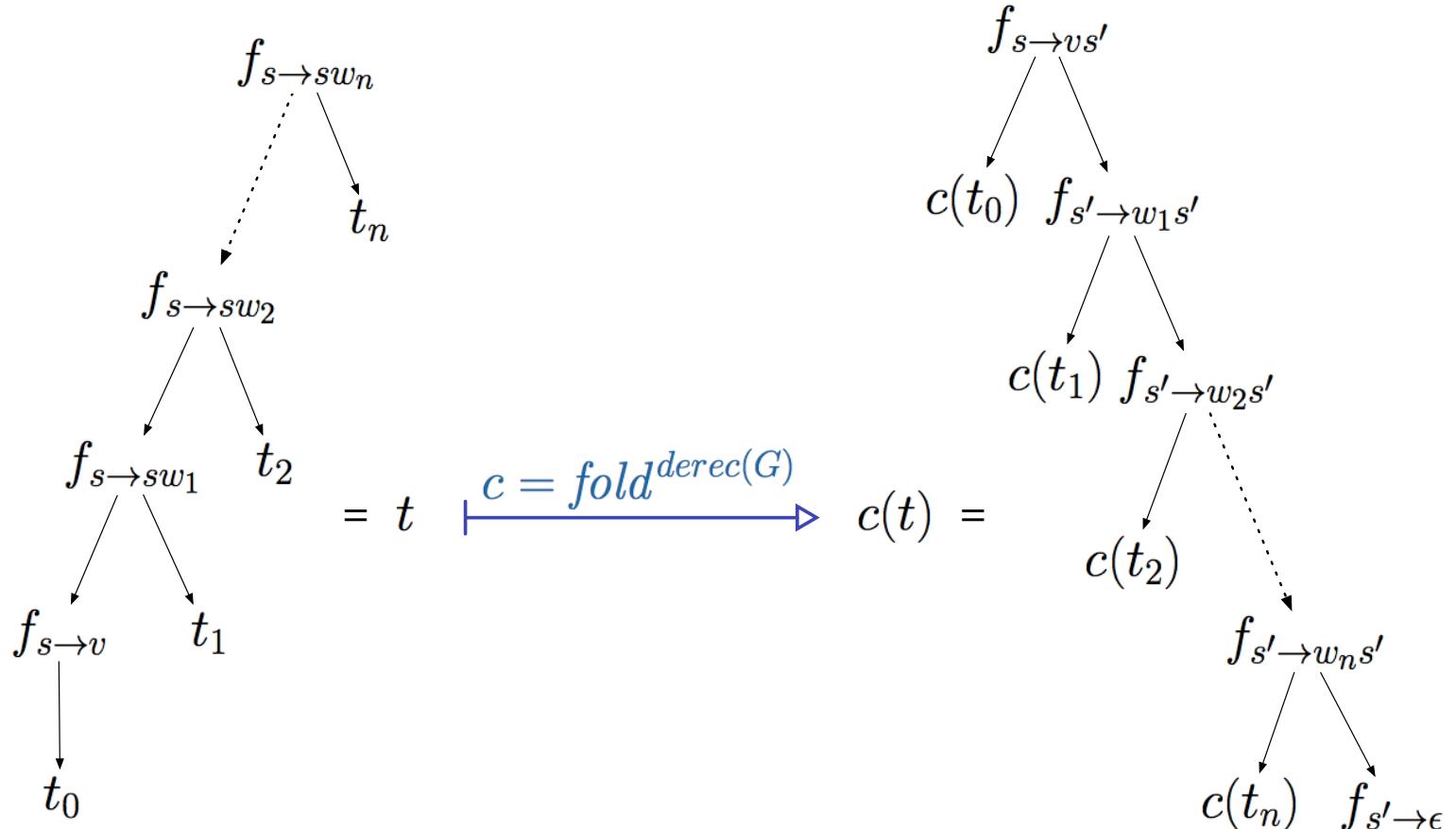
- Für alle $s \in S \setminus recs(S)$ und $s \rightarrow v \in R$ und $t \in T_{\Sigma(G'),typ(v)}$, $f_{s \rightarrow v}^{derec(G)}(t) = f_{s \rightarrow v}(t)$. (2)

- Für alle $s \rightarrow v \in nonrecs(R)$, $s \rightarrow sw \in R$, $t \in T_{\Sigma(G'),typ(v)}$, $t' \in T_{\Sigma(G'),s'}$ und $u \in T_{\Sigma(G'),typ(w)}$,

$$f_{s \rightarrow v}^{derec(G)}(t) = f_{s \rightarrow vs'}(t, f_{s' \rightarrow \epsilon}), \quad (3)$$

$$f_{s \rightarrow sw}^{derec(G)}(f_{s \rightarrow vs'}(t, t'), u) = f_{s \rightarrow vs'}(t, f_{s' \rightarrow ws'}(u, t')). \quad (4)$$

Die Faltung von $t \in T_{\Sigma(G),s}$ in $derec(G)$ lässt sich graphisch wie folgt darstellen:



Zur Faltung der von (einem Compiler nach) $derec(G)$ erzeugten Syntaxbäume von G' in einer $\Sigma(G)$ -Algebra $\mathcal{A} = (A, Op)$ muss \mathcal{A} in eine $\Sigma(G')$ -Algebra $derec(\mathcal{A}) = (A', Op')$ transformiert werden. Sie ergibt sich aus \mathcal{A} wie folgt:

- Für alle $s \in S \cup BS \setminus Z(G)$, $A'_s = A_s$. (5)

$$\bullet \text{ Für alle } s \in S \setminus recs(S) \text{ und } s \rightarrow v \in R, f_{s \rightarrow v}^{derec(\mathcal{A})} = f_{s \rightarrow v}^{\mathcal{A}}. \quad (6)$$

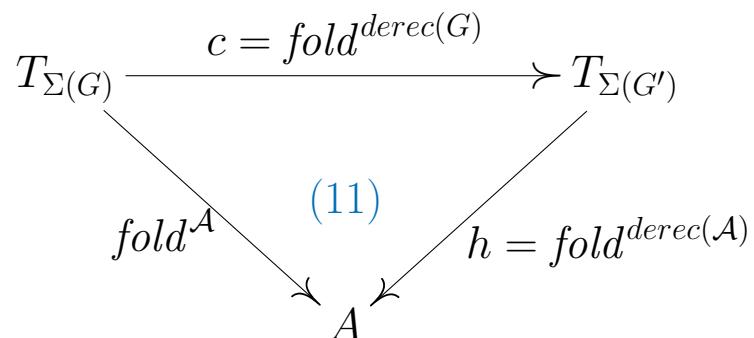
- Für alle $s \rightarrow v \in nonrecs(R)$, $s \rightarrow sw \in R$, $a \in \mathcal{A}_{typ(v)}$, $b \in \mathcal{A}_{typ(w)}$, $g : A_s \rightarrow A_s$ und $x \in A_s$,

$$A'_{s'} = (A_s \rightarrow A_s), \quad (7)$$

$$f_{s' \rightarrow \epsilon}^{derec(\mathcal{A})}(x) = x, \quad (8)$$

$$f_{s \rightarrow vs'}^{derec(\mathcal{A})}(a, g) = g(f_{s \rightarrow v}^{\mathcal{A}}(a)), \quad (9)$$

$$f_{s' \rightarrow ws'}^{derec(\mathcal{A})}(b, g)(x) = g(f_{s \rightarrow sw}^{\mathcal{A}}(x, b)). \quad (10)$$



Beweis der Kommutativität von (11) durch Induktion über die Anzahl der Symbole eines $\Sigma(G)$ -Terms.

Sei $s \in S \setminus \text{recons}(S)$ und $t \in T_{\Sigma(G),s}$. Dann gibt es o.B.d.A. $s \rightarrow v \in R$ und $u \in T_{\Sigma(G),\text{typ}(v)}$ mit $t = f_{s \rightarrow v}(u)$. Daraus folgt

$$\begin{aligned} h(c(t)) &= h(c(f_{s \rightarrow v}(u))) \stackrel{c \text{ hom.}}{=} h(f_{s \rightarrow v}^{\text{deref}(G)}(c(u))) \stackrel{(2)}{=} h(f_{s \rightarrow v}(c(u))) \\ &\stackrel{h \text{ hom.}}{=} f_{s \rightarrow v}^{\text{deref}(\mathcal{A})}(h(c(u))) \stackrel{(6)}{=} f_{s \rightarrow v}^{\mathcal{A}}(h(c(u))) \stackrel{\text{ind. hyp.}}{=} f_{s \rightarrow v}^{\mathcal{A}}(\text{fold}^{\mathcal{A}}(u)) \\ &\stackrel{\text{fold}^{\mathcal{A}} \text{ hom.}}{=} \text{fold}^{\mathcal{A}}(f_{s \rightarrow v}(u)) = \text{fold}^{\mathcal{A}}(t). \end{aligned}$$

Sei $s \in \text{recons}(S)$ und $t \in T_{\Sigma(G),s}$. Dann gibt es $n \in \mathbb{N}$, $s \rightarrow v \in \text{nonrecons}(R)$, $t_0 \in T_{\Sigma(G),\text{typ}(v)}$ und für alle $1 \leq i \leq n$ $s \rightarrow sw_i \in R$ und $t_i \in T_{\Sigma(G),\text{typ}(w_i)}$ mit

$$t = f_{s \rightarrow sw_n}(\dots(f_{s \rightarrow sw_1}(f_{s \rightarrow v}(t_0), t_1) \dots), t_n). \quad (12)$$

Daraus folgt

$$\begin{aligned} h(c(t)) &\stackrel{(12)}{=} h(c(f_{s \rightarrow sw_n}(\dots(f_{s \rightarrow sw_1}(f_{s \rightarrow v}(t_0), t_1) \dots), t_n))) \\ &\stackrel{c \text{ hom.}}{=} h(f_{s \rightarrow sw_n}^{\text{deref}(G)}(\dots(f_{s \rightarrow sw_1}^{\text{deref}(G)}(f_{s \rightarrow v}^{\text{deref}(G)}(c(t_0))), c(t_1)) \dots), c(t_n)) \\ &\stackrel{(3)}{=} h(f_{s \rightarrow sw_n}^{\text{deref}(G)}(\dots(f_{s \rightarrow sw_1}^{\text{deref}(G)}(f_{s \rightarrow vs'}(c(t_0), f_{s' \rightarrow \epsilon})), c(t_1)) \dots), c(t_n)) \\ &\stackrel{(4)}{=} h(f_{s \rightarrow sw_n}^{\text{deref}(G)}(\dots(f_{s \rightarrow vs'}(c(t_0), f_{s' \rightarrow w_1 s'}(c(t_1), f_{s' \rightarrow \epsilon}))) \dots), c(t_n)) = \dots \end{aligned}$$

$$\begin{aligned}
&\stackrel{(4)}{=} h(f_{s \rightarrow vs'}(c(t_0), f_{s' \rightarrow w_1 s'}(c(t_1), \dots, f_{s' \rightarrow w_n s'}(c(t_n), f_{s' \rightarrow \epsilon}) \dots))) \\
&\stackrel{h \text{ hom.}}{=} f_{s \rightarrow vs'}^{deref(\mathcal{A})}(h(c(t_0)), f_{s' \rightarrow w_1 s'}^{deref(\mathcal{A})}(h(c(t_1)), \dots, f_{s' \rightarrow w_n s'}^{deref(\mathcal{A})}(h(c(t_n)), f_{s' \rightarrow \epsilon}^{deref(\mathcal{A})}) \dots))) \\
&\stackrel{(8)}{=} f_{s \rightarrow vs'}^{deref(\mathcal{A})}(h(c(t_0)), \textcolor{blue}{f_{s' \rightarrow w_1 s'}^{deref(\mathcal{A})}(h(c(t_1)), \dots, f_{s' \rightarrow w_n s'}^{deref(\mathcal{A})}(h(c(t_n)), id) \dots)}) \\
&\stackrel{(9)}{=} \textcolor{blue}{f_{s' \rightarrow w_1 s'}^{deref(\mathcal{A})}(h(c(t_1)), \dots, f_{s' \rightarrow w_n s'}^{deref(\mathcal{A})}(h(c(t_n)), id) \dots)}(f_{s \rightarrow v}^{\mathcal{A}}(h(c(t_0)))) \\
&\stackrel{(10)}{=} (\dots, f_{s' \rightarrow w_n s'}^{deref(\mathcal{A})}(h(c(t_n)), id) \dots)(f_{s \rightarrow sw_1}^{\mathcal{A}}(f_{s \rightarrow v}^{\mathcal{A}}(h(c(t_0))), h(c(t_1)))) = \dots \\
&\stackrel{(10)}{=} f_{s' \rightarrow w_n s'}^{deref(\mathcal{A})}(h(c(t_n)), \textcolor{blue}{id})(\dots (f_{s \rightarrow sw_1}^{\mathcal{A}}(f_{s \rightarrow v}^{\mathcal{A}}(h(c(t_0))), h(c(t_1)))) \dots) \\
&\stackrel{(10)}{=} \textcolor{blue}{id}(f_{s \rightarrow sw_n}^{\mathcal{A}}(\dots (f_{s \rightarrow sw_1}^{\mathcal{A}}(f_{s \rightarrow v}^{\mathcal{A}}(h(c(t_0))), h(c(t_1)))) \dots, h(c(t_n)))) \\
&= f_{s \rightarrow sw_n}^{\mathcal{A}}(\dots (f_{s \rightarrow sw_1}^{\mathcal{A}}(f_{s \rightarrow v}^{\mathcal{A}}(h(c(t_0))), h(c(t_1)))) \dots, h(c(t_n))) \\
&\stackrel{ind. \ hyp.}{=} f_{s \rightarrow sw_n}^{\mathcal{A}}(\dots (f_{s \rightarrow sw_1}^{\mathcal{A}}(f_{s \rightarrow v}^{\mathcal{A}}(fold^{\mathcal{A}}(t_0)), fold^{\mathcal{A}}(t_1))) \dots, fold^{\mathcal{A}}(t_n)) \\
&\stackrel{fold^{\mathcal{A}} \text{ hom.}}{=} fold^{\mathcal{A}}(f_{s \rightarrow sw_n}(\dots (f_{s \rightarrow sw_1}(f_{s \rightarrow v}(t_0), t_1) \dots), t_n)) \stackrel{(12)}{=} fold^{\mathcal{A}}(t). \quad \square
\end{aligned}$$

Beispiel 4.6 (Fortsetzung) Für $G = \text{JavaLight}$ und $G' = \text{JavaLight}'$ gilt:

$$S' = S \cup \{\text{Sumsect}, \text{Prodsect}\},$$

$$F' = F \setminus \{\underline{\text{sum}}, \underline{\text{plus}}, \underline{\text{minus}}, \underline{\text{prod}}, \underline{\text{times}}, \underline{\text{div}}\}$$

$$\cup \{\text{sum}, \text{plus}, \text{minus}, \text{nilS}, \text{prod}, \text{times}, \text{div}, \text{nilP}\}.$$

Damit ist die $\Sigma(G)$ -Algebra $\text{derek}(G)$ – neben (1) und (2) – wie folgt definiert:

Für alle $t, u \in T_{\Sigma(G), \text{Prod}}$ und $t' \in T_{\Sigma(G'), \text{Sumsect}}$,

$$\underline{\text{sum}}^{\text{derek}(G)}(t) = \text{sum}(t, \text{nilS}),$$

$$\underline{\text{plus}}^{\text{derek}(G)}(\text{sum}(t, t'), u) = \text{sum}(t, \text{plus}(u, t')),$$

$$\underline{\text{minus}}^{\text{derek}(G)}(\text{sum}(t, t'), u) = \text{sum}(t, \text{minus}(u, t')).$$

Für alle $t, u \in T_{\Sigma(G), \text{Factor}}$ und $t' \in T_{\Sigma(G'), \text{Prodsect}}$,

$$\underline{\text{prod}}^{\text{derek}(G)}(t) = \text{prod}(t, \text{nilP}),$$

$$\underline{\text{times}}^{\text{derek}(G)}(\text{prod}(t, t'), u) = \text{prod}(t, \text{times}(u, t')),$$

$$\underline{\text{div}}^{\text{derek}(G)}(\text{prod}(t, t'), u) = \text{prod}(t, \text{div}(u, t'))$$

Sei $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$ -Algebra. Die $\Sigma(G')$ -Algebra $derec(\mathcal{A}) = (A', Op')$ ist – neben (3) und (4) – wie folgt definiert:

- $A'_{Sumsect} = A_{Sum} \rightarrow A_{Sum}$ und $A'_{Prodsect} = A_{Prod} \rightarrow A_{Prod}$.

- Für alle $a \in A_{Prod}$, $g : A_{Sum} \rightarrow A_{Sum}$ und $x \in A_{Sum}$,

$$\text{sum}^{derec(\mathcal{A})}(a, g) = g(\underline{\text{sum}}^{\mathcal{A}}(a)),$$

$$\text{plus}^{derec(\mathcal{A})}(a, g)(x) = g(\underline{\text{plus}}^{\mathcal{A}}(x, a)),$$

$$\text{minus}^{derec(\mathcal{A})}(a, g)(x) = g(\underline{\text{minus}}^{\mathcal{A}}(x, a)),$$

$$\text{nilS}^{derec(\mathcal{A})}(x) = x.$$

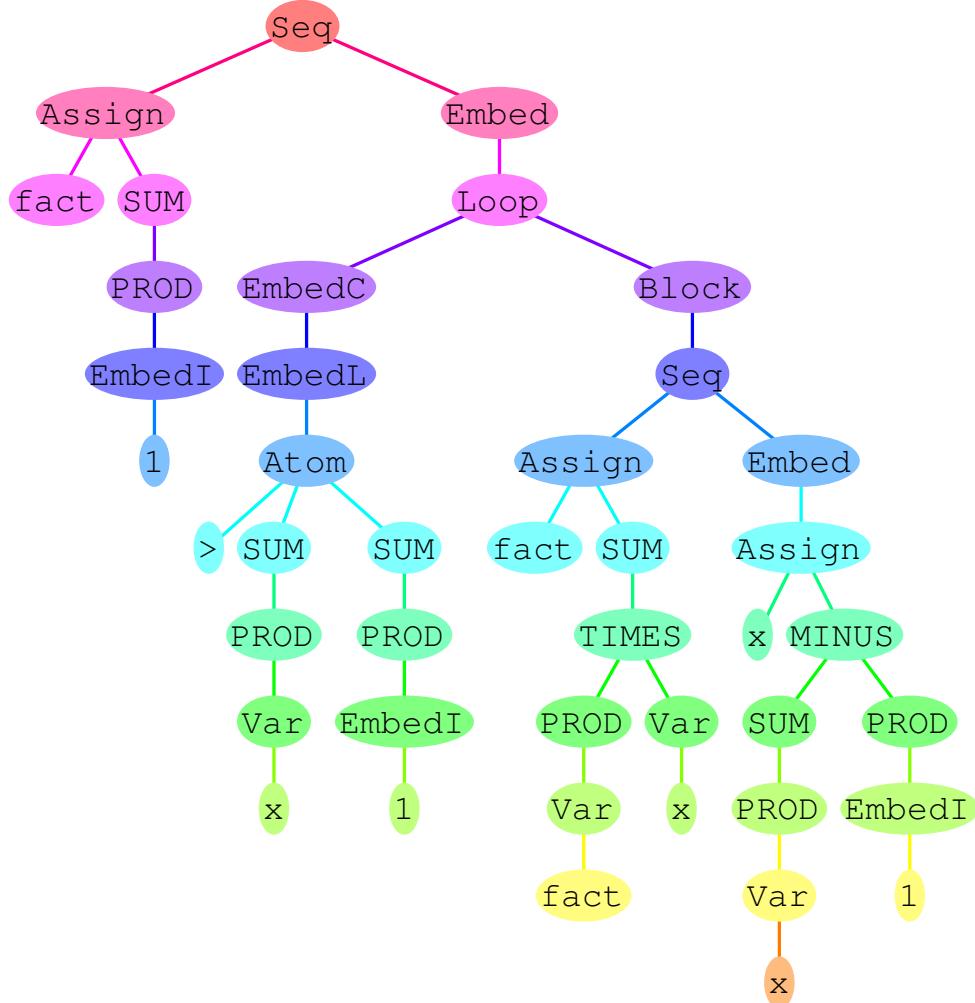
- Für alle $a \in A_{Factor}$, $g : A_{Prod} \rightarrow A_{Prod}$ und $x \in A_{Prod}$,

$$\text{prod}^{derec(\mathcal{A})}(a, g) = g(\underline{\text{prod}}^{\mathcal{A}}(a)),$$

$$\text{times}^{derec(\mathcal{A})}(a, g)(x) = g(\underline{\text{times}}^{\mathcal{A}}(x, a)),$$

$$\text{div}^{derec(\mathcal{A})}(a, g)(x) = g(\underline{\text{div}}^{\mathcal{A}}(x, a)),$$

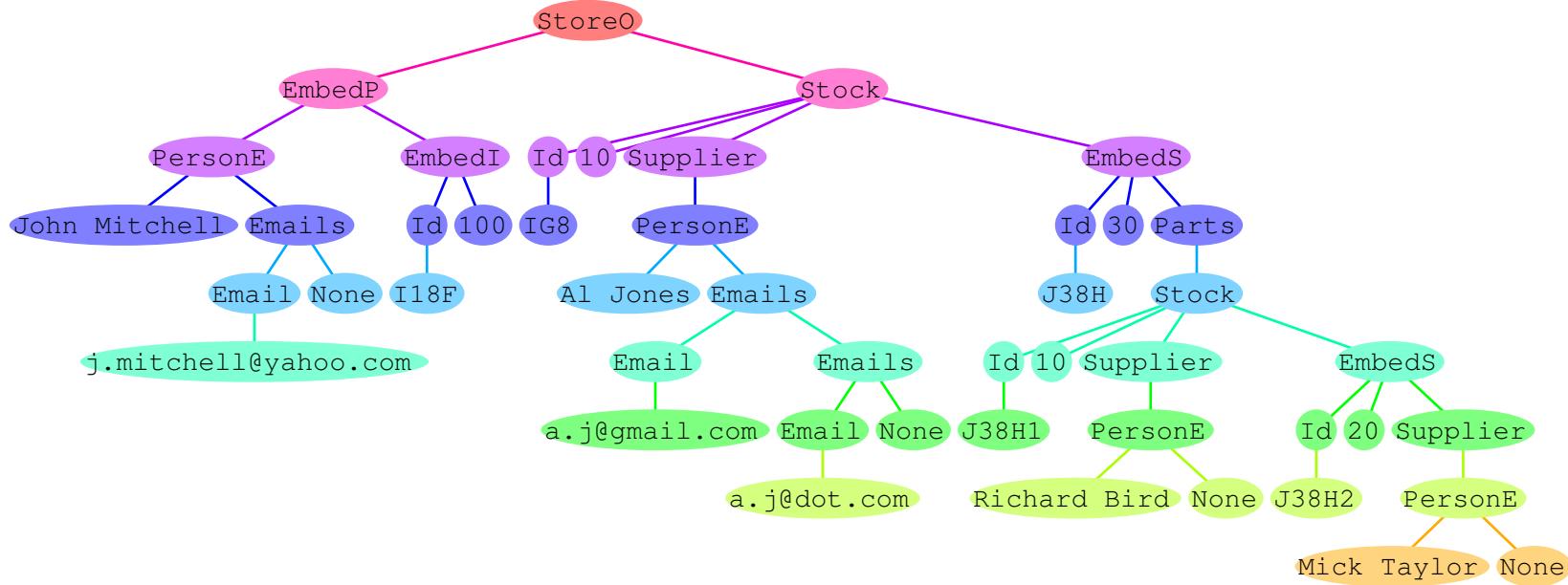
$$\text{nilP}^{derec(\mathcal{A})}(x) = x.$$



*Syntaxbaum von G , dessen G' -Version (= Faltung in $\text{derec}(G)$) weiter oben steht.
 Die Konstruktoren von $F \setminus F'$ sind hier großgeschrieben.*

Beispiel 4.7 Die abstrakte Syntax (S, F) von XMLstore lautet wie folgt:

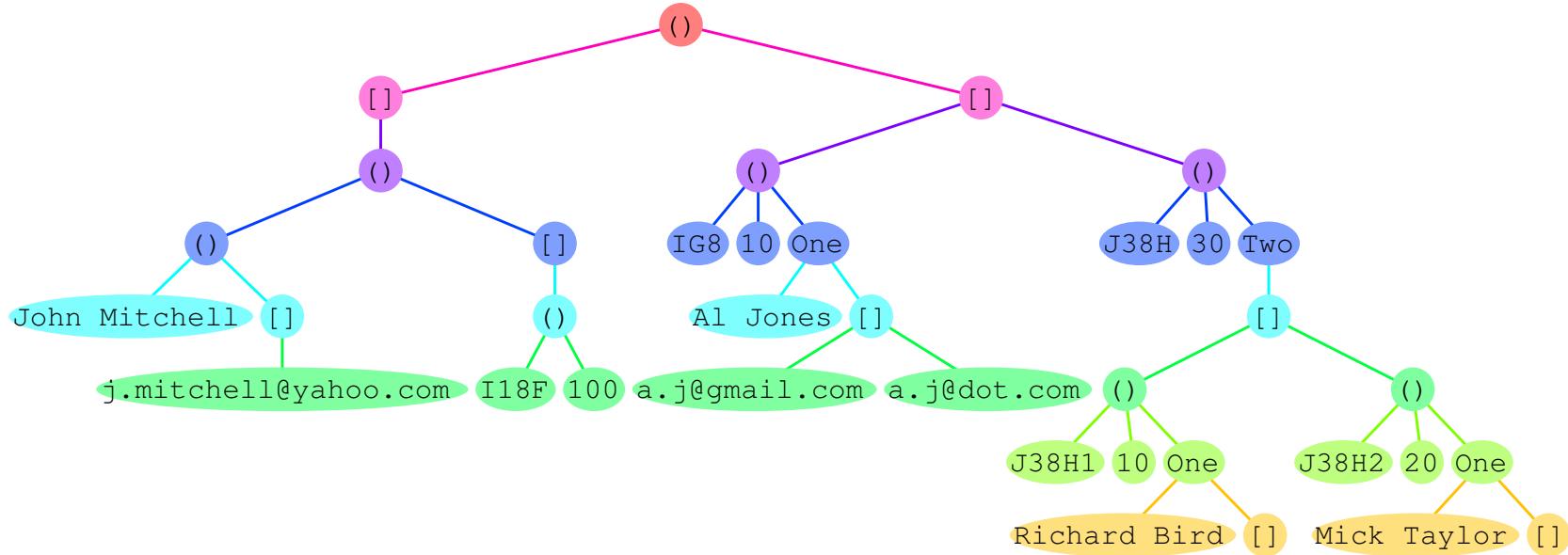
$$\begin{aligned} S &= \{ \textit{Store}, \textit{Orders}, \textit{Order}, \textit{Person}, \textit{Emails}, \textit{Email}, \textit{Items}, \textit{Item}, \textit{Stock}, \\ &\quad = \{ \textit{ItemS}, \textit{Suppliers}, \textit{Id} \} \} \\ \mathcal{I} &= \{ 1, \textit{String}, \textit{Id}, \mathbb{Z} \} \\ F &= \{ \textit{store} : \textit{Stock} \rightarrow \textit{Store}, \\ &\quad \textit{storeO} : \textit{Orders} \times \textit{Stock} \rightarrow \textit{Store}, \\ &\quad \textit{orders} : \textit{Person} \times \textit{Items} \times \textit{Orders} \rightarrow \textit{Orders}, \\ &\quad \textit{embedP} : \textit{Person} \times \textit{Items} \rightarrow \textit{Orders}, \\ &\quad \textit{person} : \textit{String} \rightarrow \textit{Person}, \\ &\quad \textit{personE} : \textit{String} \times \textit{Emails} \rightarrow \textit{Person}, \\ &\quad \textit{emails} : \textit{Email} \times \textit{Emails} \rightarrow \textit{Emails}, \\ &\quad \textit{none} : 1 \rightarrow \textit{Emails}, \\ &\quad \textit{email} : \textit{String} \rightarrow \textit{Email}, \\ &\quad \textit{items} : \textit{Id} \times \textit{String} \times \textit{Items} \rightarrow \textit{Items}, \\ &\quad \textit{embedI} : \textit{Id} \times \textit{String} \rightarrow \textit{Items}, \\ &\quad \textit{stock} : \textit{Id} \times \mathbb{Z} \times \textit{Supplier} \times \textit{Stock} \rightarrow \textit{Stock}, \\ &\quad \textit{embedS} : \textit{Id} \times \mathbb{Z} \times \textit{Supplier} \rightarrow \textit{Stock}, \\ &\quad \textit{supplier} : \textit{Person} \rightarrow \textit{Suppliers}, \\ &\quad \textit{parts} : \textit{Stock} \rightarrow \textit{Suppliers}, \\ &\quad \textit{id} : \textit{String} \rightarrow \textit{Id} \} \end{aligned}$$



Syntaxbaum des XML-Dokumentes von Beispiel 4.3

`xmlToAlg "xmldoc" 1` (siehe `Compiler.hs`) übersetzt das XMLstore-Dokument in der Datei `xmldoc` in den zugehörigen Syntaxbaum übersetzt und schreibt diesen in die Datei `Pix/xmlterm.svg`.

`xmlToAlg "xmldoc" 2` (siehe `Compiler.hs`) übersetzt `xmldoc` in eine Listen-Produkte-Summen-Darstellung und schreibt diese in die Datei `Pix/xmllist.svg`. Für Beispiel 4.3 sieht sie folgendermaßen aus:



□

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG und $X = \bigcup BS$.

Wort- und Ableitungsbaumalgebra

Neben $T_{\Sigma(G)}$ lassen sich auch die Menge der Wörter über X und die Menge der Ableitungsbäume von G zu $\Sigma(G)$ -Algebren erweitern.

$\text{Word}(G)$, die Wortalgebra von G

- Für alle $s \in S$, $\text{Word}(G)_s =_{\text{def}} X^*$.
 - Für alle $w_0 \dots w_n \in Z^*$, $s_1, \dots, s_n \in S \cup BS \setminus Z(G)$, $r = (s \rightarrow w_0 s_1 w_1 \dots s_n w_n) \in R$ und $(v_1, \dots, v_n) \in \text{Word}(G)_{s_1 \times \dots \times s_n} = (X^*)^n$,
- $$f_r^{\text{Word}(G)}(v_1, \dots, v_n) =_{\text{def}} w_0 v_1 w_1 \dots v_n w_n.$$

$t \in T_{\Sigma(G)}$ heißt **G -Syntaxbaum für $w \in X^*$** , falls $\text{fold}^{\text{Word}(G)}(t) = w$ gilt.

Beispiel 4.8 Nochmal die Regeln von **SAB** (siehe Beispiel 4.5):

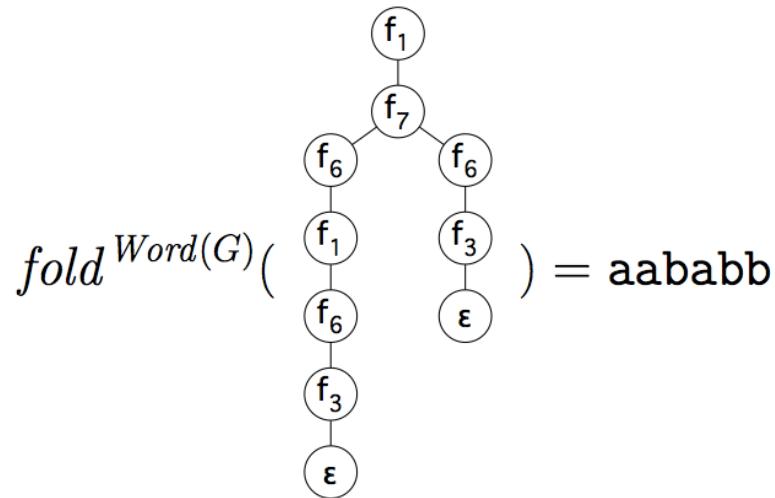
$$\begin{aligned} r_1 &= S \rightarrow aB, & r_2 &= S \rightarrow bA, & r_3 &= S \rightarrow \epsilon, \\ r_4 &= A \rightarrow aS & r_5 &= A \rightarrow bAA, & r_6 &= B \rightarrow bS, & r_7 &= B \rightarrow aBB. \end{aligned}$$

Alle drei Trägermengen der Wortalgebra $\text{Word}(\text{SAB})$ sind durch $\{a, b\}^*$ gegeben.

Die Konstruktoren von $\Sigma(\text{SAB})$ werden in $\text{Word}(\text{SAB})$ wie folgt interpretiert: Für alle $v, w \in \{a, b\}^*$,

$$\begin{aligned} f_1^{\text{Word}(\text{SAB})}(w) &= f_4^{\text{Word}(\text{SAB})}(w) = aw, \\ f_2^{\text{Word}(\text{SAB})}(w) &= f_6^{\text{Word}(\text{SAB})}(w) = bw, \\ f_3^{\text{Word}(\text{SAB})} &= \epsilon, \\ f_5^{\text{Word}(\text{SAB})}(v, w) &= bvw, \\ f_7^{\text{Word}(\text{SAB})}(v, w) &= avw. \end{aligned}$$

Syntaxbaum für aababb:



Mit ϵ markierte Blätter eines Syntaxbaums werden künftig weglassen. □

G ist **eindeutig**, wenn $fold^{Word(G)}$ injektiv ist.

Die **Sprache $L(G)$ von G** ist die Menge der Wörter über X , die sich aus der Faltung eines Syntaxbaums in $Word(G)$ ergeben:

$$L(G) =_{def} fold^{Word(G)}(T_{\Sigma(G)}).$$

Die Theorie formaler Sprachen liefert eine äquivalente Definition von $L(G)$, die die oben definierte Ableitungsrelation \rightarrow_G verwendet: Für alle $s \in S$,

$$L(G)_s = \{w \in BS^* \mid s \xrightarrow{+}_G w\}.$$

Z.B. ist $L(REG)$ (siehe 4.1) die Menge aller Wörter über $syms(BL)$ (siehe 2.9), die reguläre Ausdrücke über BL darstellen, formal:

$$L(REG) = \{w \in syms(BL)^* \mid \exists t \in T_{Reg(BL)} : fold^{Regword(BL)}(t)(0) = w\}.$$

Zwei Grammatiken G und G' heißen **äquivalent**, wenn ihre Sprachen übereinstimmen.

Für eine LL-kompilierbare CFG G und die daraus gebildete nicht-linksrekursive CFG G' lässt sich die Kommutativität des folgenden Diagramms analog zu der von (11) durch Induktion über die Anzahl der Symbole eines $\Sigma(G)$ -Terms zeigen:

$$\begin{array}{ccc}
 T_{\Sigma(G)} & \xrightarrow{c = fold^{derec(G)}} & T_{\Sigma(G')} \\
 & \searrow g = fold^{Word(G)} & \swarrow h = fold^{Word(G')} \\
 & X^* &
 \end{array}$$

(13)

Aufgabe

Aus (der Kommutativität von) (13) und der Bijektivität von c (!) folgt

$$g \circ c^{-1} = h \circ c \circ c^{-1} = h, \quad (14)$$

Schließen Sie $L(G) = L(G')$ aus (13) und (14). □

`javaToAlg "prog" 2` (siehe `Java.hs`) übersetzt das JavaLight-Programm in der Datei `prog` in die Interpretation des zugehörigen Syntaxbaums in $Word(JavaLight)$ und schreibt diese in die Datei `javadoc`. Die Inhalte von `prog` und `javadoc` stimmen also miteinander überein!

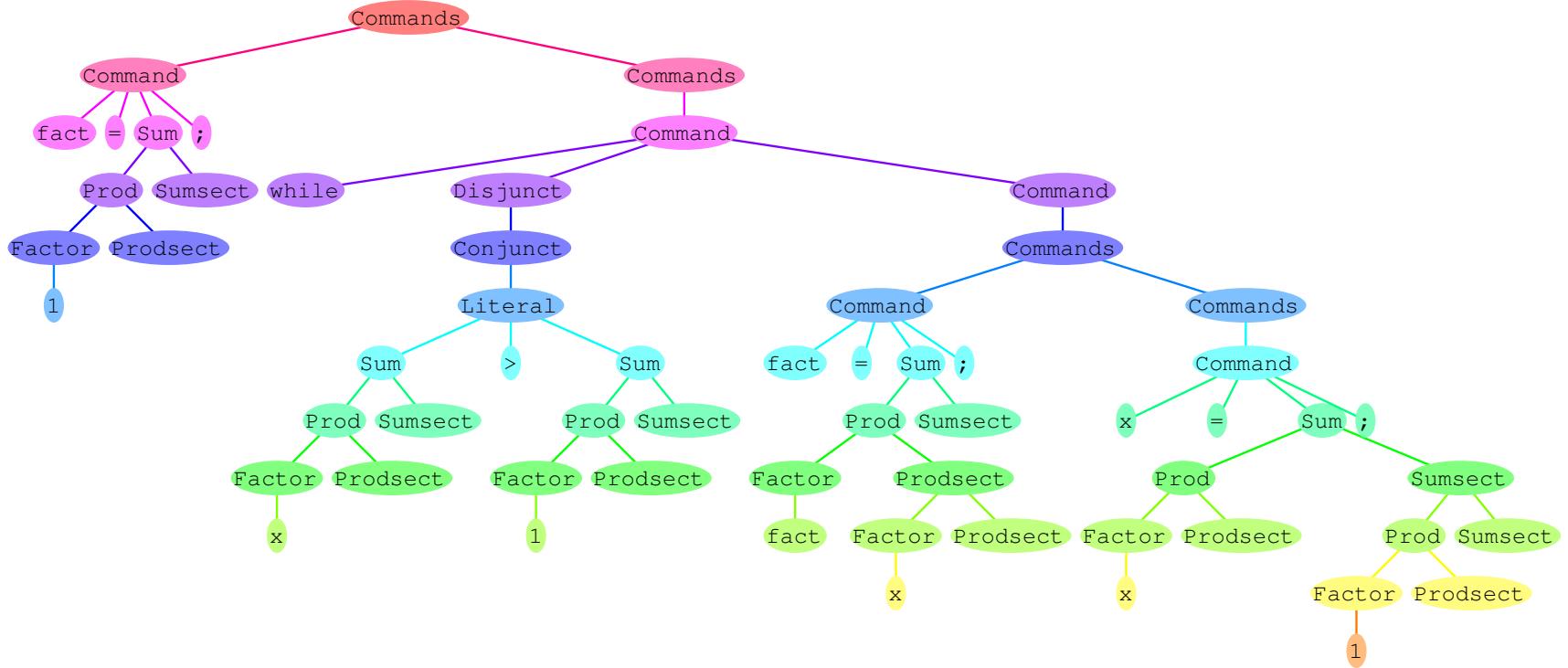
$Abl(G)$, die Ableitungsbaumalgebra von G

- Für alle $s \in S$, $Abl(G)_s = wtr(\mathbb{N}, S \cup BS)$ (siehe Kapitel 2).
- Für alle $z_0, \dots, z_{n_0}, z_{n_0+2}, \dots, z_{n_{k-1}+2}, \dots, z_{n_k} \in Z$, $s_1, \dots, s_k \in S \cup BS \setminus Z(G)$,

$$r = (s \rightarrow z_0 \dots z_{n_0} \textcolor{blue}{s}_1 z_{n_0+2} \dots z_{n_{k-1}} \textcolor{blue}{s}_k z_{n_{k-1}+2} \dots z_{n_k}) \in R$$

und $(t_1, \dots, t_k) \in Abl(G)_{s_1 \times \dots \times s_k} = wtr(\mathbb{N}, S \cup BS)^k$,

$$\begin{aligned} f_r^{Abl(G)}(t_1, \dots, t_k) &=_{def} s \{ &0 \rightarrow z_0, \dots, \\ &n_0 \rightarrow z_{n_0}, n_0 + 1 \rightarrow \textcolor{blue}{t}_1, n_0 + 2 \rightarrow z_{n_0+2}, \\ &\dots, \\ &n_{k-1} \rightarrow z_{n_{k-1}}, n_{k-1} + 1 \rightarrow \textcolor{blue}{t}_k, n_{k-1} + 2 \rightarrow z_{n_{k-1}+2}, \\ &\dots, n_k \rightarrow z_{n_k} \} \end{aligned}$$



Ableitungsbaum von `fact = 1; while x > 1 {fact = fact*x; x = x-1;}`

`javaToAlg "prog" 3` (siehe `Java.hs`) übersetzt das JavaLight-Programm in der Datei `prog` in den zugehörigen Ableitungsbaum und schreibt diesen in die Datei `Pix/javaderi.svg`.

4.9 Das Zustandsmodell von JavaLight

Sei $\text{Store} = \text{String} \rightarrow \mathbb{Z}$ (Menge der Speicherzustände; hier: Belegungen von Programmvariablen durch ganze Zahlen).

Das Zustandsmodell von JavaLight ist durch folgende $\Sigma(\text{JavaLight})$ -Algebra $\mathcal{A} = (A, Op)$ gegeben (siehe Beispiel 4.6):

$$A_{Commands} = A_{Command} = \text{Store} \multimap \text{Store} \quad (\text{Menge der partiellen Funktionen von } \text{Store} \text{ nach } \text{Store})$$

$$A_{Sum} = A_{Prod} = A_{Factor} = \text{Store} \rightarrow \mathbb{Z}$$

$$A_{Disjunct} = A_{Conjunct} = A_{Literal} = \text{Store} \rightarrow 2$$

Für alle $f, g : \text{Store} \multimap \text{Store}$, $x \in \text{String}$, $e : \text{Store} \rightarrow \mathbb{Z}$, $st \in \text{Store}$ und $p : \text{Store} \rightarrow 2$,

$$\text{seq}^{\mathcal{A}}(f, g) = g \circ f,$$

$$\text{embed}^{\mathcal{A}}(f) = \text{block}^{\mathcal{A}}(f) = f,$$

$$\text{assign}^{\mathcal{A}}(x, e)(st) = st[e(st)/x],$$

$$\text{cond}^{\mathcal{A}}(p, f, g)(st) = \text{if } p(st) \text{ then } f(st) \text{ else } g(st),$$

$$\text{cond1}^{\mathcal{A}}(p, f)(st) = \text{if } p(st) \text{ then } f(st) \text{ else } st,$$

$$\text{loop}^{\mathcal{A}}(p, f)(st) = \text{if } p(st) \text{ then } \text{loop}^{\mathcal{A}}(p, f)(f(st)) \text{ else } st.$$

Offenbar können mit der letzten Gleichung unendliche Berechnungen erzeugt werden. In solchen Fällen bleibt $\text{loop}^{\mathcal{A}}(p, f)(st)$ undefiniert. Aus diesem Grund mussten wir *Commands* und *Command* durch Mengen *partieller* Funktionen interpretieren.

Darüberhinaus liefert die Gleichung für $\text{loop}^{\mathcal{A}}$ keine Definition, sondern nur eine Anforderung an eine Funktion, deren Existenz noch zu zeigen ist. Der Beweis wendet den Fixpunkt-Satz von Kleene auf den CPO $\text{Store} \rightarrow \text{Store}$ an (siehe Kapitel 19).

Für alle $f, g : \text{Store} \rightarrow \mathbb{Z}$, $st \in \text{Store}$, $x \in \text{String}$ und $i \in \mathbb{Z}$,

$$\begin{aligned}
\text{sum}^{\mathcal{A}}(f) &= \text{prod}^{\mathcal{A}}(f) = f, \\
\text{plus}^{\mathcal{A}}(f, g) &= \text{lift}^{\text{Store}}(+)(f, g) = \lambda st. f(st) + g(st), \quad (\text{siehe Kapitel 2}) \\
\text{minus}^{\mathcal{A}}(f, g) &= \text{lift}^{\text{Store}}(-)(f, g), \\
\text{times}^{\mathcal{A}}(f, g) &= \text{lift}^{\text{Store}}(*) (f, g), \\
\text{div}^{\mathcal{A}}(f, g) &= \text{lift}^{\text{Store}}(/)(f, g), \\
\text{embedI}^{\mathcal{A}}(i)(st) &= i, \\
\text{var}^{\mathcal{A}}(x)(st) &= st(x), \\
\text{encloseS}^{\mathcal{A}}(f) &= f.
\end{aligned}$$

Für alle $f, g : Store \rightarrow 2$, $rel \in Rel$, $e, e' : Store \rightarrow \mathbb{Z}$ und $b \in 2$,

$$\begin{aligned} disjunct^{\mathcal{A}}(f, g) &= lift^{Store}(\vee)(f, g), \\ embedC^{\mathcal{A}}(f) &= embedL^{\mathcal{A}}(f) = encloseD^{\mathcal{A}}(f) = f, \\ conjunct^{\mathcal{A}}(f, g) &= lift^{Store}(\wedge)(f, g), \\ not^{\mathcal{A}}(f) &= \neg \circ f, \\ atom^{\mathcal{A}}(rel, e, e') &= lift^{Store}(rel)(e, e'), \\ embedB^{\mathcal{A}}(b)(st) &= b. \end{aligned}$$

In Abschnitt 11.5 wird \mathcal{A} unter dem Namen *javaState* in Haskell implementiert.

5 Parser und Compiler für CFGs

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG, $X = \bigcup BS$ und *Ziel* eine $\Sigma(G)$ -Algebra, in die Wörter über X übersetzt werden sollen.

In Anlehnung an den klassischen Begriff eines semantisch korrekten Compilers [24, 43] verlangen wir, dass die Semantiken *Sem* und *Mach* seiner Quell- bzw. Zielsprache in der durch das folgende Funktionsdiagramm ausgedrückten Weise miteinander zusammenhängen:

$$\begin{array}{ccc} T_{\Sigma(G)} & \xrightarrow{\text{fold}^{\text{Ziel}}} & \text{Ziel} \\ \downarrow \text{fold}^{\text{Sem}} & (1) & \downarrow \text{evaluate} \\ \text{Sem} & \xrightarrow{\text{encode}} & \text{Mach} \end{array}$$

Hierbei sind

- *Sem* die ebenfalls als $\Sigma(G)$ -Algebra gegebene Semantik der Quellsprache $L(G)$,
- *Mach* ein in der Regel unabhängig von $\Sigma(G)$ definiertes Modell der Zielsprache, meist in Form einer abstrakten Maschine,
- *evaluate* ein Interpreter, der Zielprogramme in der abstrakten Maschine *Mach* ausführt,

- *encode* eine Funktion, die *Sem* auf *Mach* abbildet und die gewünschte Arbeitsweise des Compilers auf semantischer Ebene reflektiert.

Die Initialität der Termalgebra $T_{\Sigma(G)}$ erlaubt es uns, den Beweis der Kommutativität von (1) auf die Erweiterung von *encode* und *evaluate* zu $\Sigma(G)$ -Homomorphismen zu reduzieren. Dazu muss zunächst *Mach* zu einer $\Sigma(G)$ -Algebra gemacht werden, was z.B. bedeuten kann, die elementaren Funktionen der Zielsprache so in einer Signatur Σ' zusammenzufassen, dass $T_{\Sigma'}$ mit *Ziel* übereinstimmt, jeder Konstruktor von $\Sigma(G)$ einem Σ' -Term entspricht, *Sem* eine Σ' -Algebra ist und *evaluate* Σ' -Terme in *Sem* faltet. Die Darstellung von $\Sigma(G)$ -Konstruktoren durch Σ' -Terme bestimmt dann möglicherweise eine Definition von *encode*, die sowohl *encode* als auch *evaluate* $\Sigma(G)$ -homomorph macht. So wurde z.B. in [43] die Korrektheit eines Compilers gezeigt, der imperative Programme in Datenflussgraphen übersetzt.

Damit sind alle vier Abbildungen in Diagramm (1), also auch und die beiden Kompositionen $\text{evaluate} \circ \text{fold}^{\text{Ziel}}$ und $\text{encode} \circ \text{fold}^{\text{Mach}}$ $\Sigma(G)$ -homomorph. Da $T_{\Sigma(G)}$ eine initiale $\Sigma(G)$ -Algebra ist, sind beide Kompositionen gleich. Also kommutiert (1).

Während $\text{fold}^{\text{Ziel}}$ Syntaxbäume in der *Zielsprache* auswertet, ordnet $\text{fold}^{\text{Word}(G)}$ (siehe Kapitel 4) einem Syntaxbaum t das Wort der *Quellsprache* zu, aus dem ein Parser für G t berechnen soll.

In der Theorie formaler Sprachen ist der Begriff Parser auf Entscheidungsalgorithmen beschränkt, die anstelle von Syntaxbäumen lediglich einen Booleschen Wert liefern, der angibt, ob ein Eingabewort zur Sprache der jeweiligen Grammatik gehört oder nicht.

Demgegenüber definieren wir einen **Parser für G** als eine S -sortige Funktion

$$\text{parse}_G : X^* \rightarrow M(T_{\Sigma(G)}),$$

die entweder Syntaxbäume oder, falls das Eingabewort nicht zur Sprache von G gehört, Fehlermeldungen erzeugt. Welche Syntaxbäume bzw. Fehlermeldungen ausgegeben werden sollen, wird durch eine *Monade M* festgelegt.

5.1 Funktoren und Monaden

Funktoren, natürliche Transformationen und Monaden sind *kategorientheoretische* Grundbegriffe, die heutzutage jeder Softwaredesigner kennen sollte, da sie sich in den Konstruktions- und Transformationsmustern jedes denkbaren statischen, dynamischen oder hybriden Systemmodells wiederfinden. Die hier benötigten kategorientheoretischen Definitionen lauten wie folgt:

Eine **Kategorie \mathcal{K}** besteht aus

- einer – ebenfalls mit \mathcal{K} bezeichneten – Klasse von **\mathcal{K} -Objekten**,
- für alle $A, B \in \mathcal{K}$ einer Menge $\mathcal{K}(A, B)$ von **\mathcal{K} -Morphismen**,

- einer assoziativen **Komposition**

$$\circ : \mathcal{K}(A, B) \times \mathcal{K}(B, C) \rightarrow \mathcal{K}(A, C)$$

$$(f, g) \longmapsto g \circ f,$$

- einer **Identität** $id_A \in \mathcal{K}(A, A)$, die bzgl. \circ neutral ist, d.h. für alle $B \in \mathcal{K}$ und $f \in \mathcal{K}(A, B)$ gilt $f \circ id_A = f = id_B \circ f$.

Im Kontext einer festen Kategorie \mathcal{K} schreibt man meist $f : A \rightarrow B$ anstelle von $f \in \mathcal{K}(A, B)$.

Wir haben hier mit vier Kategorien zu tun: $\textcolor{red}{Set}$, $\textcolor{red}{Set}^2$ und $\textcolor{red}{Set}^S$, deren Objekte alle Mengen, Mengenpaare bzw. S -sortigen Mengen und deren Morphismen alle Funktionen, Funktionspaare bzw. S -sortigen Funktionen sind, sowie die Unterkategorie $\textcolor{red}{Alg}_\Sigma$ von $\textcolor{red}{Set}^S$, deren Objekte alle Σ -Algebren und deren Morphismen alle Σ -Homomorphismen sind.

Seien \mathcal{K}, \mathcal{L} Kategorien. Ein **Funktor** $\textcolor{red}{F} : \mathcal{K} \rightarrow \mathcal{L}$ ist eine Funktion, die jedem \mathcal{K} -Objekt ein \mathcal{L} -Objekt und jedem \mathcal{K} -Morphismus $f : A \rightarrow B$ einen \mathcal{L} -Morphismus $F(f) : F(A) \rightarrow F(B)$ zuordnet sowie folgende Gleichungen erfüllt:

- Für alle \mathcal{K} -Objekte A , $\textcolor{blue}{F}(id_A) = id_{F(A)}$, (2)

- Für alle \mathcal{K} -Morphismen $f : A \rightarrow B$ und $g : B \rightarrow C$, $\textcolor{blue}{F}(g \circ f) = F(g) \circ F(f)$. (3)

Zwei Funktoren $F : \mathcal{K} \rightarrow \mathcal{L}$ und $G : \mathcal{L} \rightarrow \mathcal{M}$ kann man wie andere Funktionen sequentiell zu weiteren Funktoren komponieren: Für alle \mathcal{K} -Objekte und -Morphismen A , $(GF)(A) =_{\text{def}} G(F(A))$.

Beispiele

Sei $B \in \mathcal{L}$. Der **konstante Funktor** $\text{const}(B) : \mathcal{K} \rightarrow \mathcal{L}$ ordnet jedem \mathcal{K} -Objekt das \mathcal{L} -Objekt B zu und jedem \mathcal{K} -Morphismus die Identität auf B .

Der **Identitätsfunktor** $\text{Id}_{\mathcal{K}} : \mathcal{K} \rightarrow \mathcal{K}$ ordnet jedem \mathcal{K} -Objekt und jedem \mathcal{K} -Morphismus sich selbst zu.

Der **Diagonalfunktor** $\Delta_{\mathcal{K}} : \mathcal{K} \rightarrow \mathcal{K}^2$ ordnet jedem \mathcal{K} -Objekt A das Objektpaar (A, A) und jedem \mathcal{K} -Morphismus f das Morphismenpaar (f, f) zu.

Der **Produktfunktor** $_ \times _ : \text{Set}^2 \rightarrow \text{Set}$ ordnet jedem Mengenpaar (A, B) die Menge $A \times B$ und jedem Funktionspaar $(f : A \rightarrow B, g : C \rightarrow D)$ die Funktion $f \times g =_{\text{def}} \lambda(a, c).(f(a), g(c))$ zu.

Der **Listenfunktor** $\underline{*} : \text{Set} \rightarrow \text{Set}$ ordnet jeder Menge A die Menge A^* der Wörter über A zu und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} f^* = \text{map}(f) : A^* &\rightarrow B^* \\ \epsilon &\mapsto \epsilon \\ (a_1, \dots, a_n) &\mapsto (f(a_1), \dots, f(a_n)) \end{aligned}$$

Der **Mengenfunktor** $\mathcal{P} : \text{Set} \rightarrow \text{Set}$ ordnet jeder Menge A die Potenzmenge $\mathcal{P}(A)$ zu und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} \mathcal{P}(f) : \mathcal{P}(A) &\rightarrow \mathcal{P}(B) \\ C &\mapsto \{f(c) \mid c \in C\} \end{aligned}$$

Sei E eine Menge von Fehlermeldungen, Ausnahmewerten o.ä.

Der **Ausnahmefunktor** $\underline{+ E} : \text{Set} \rightarrow \text{Set}$ ordnet jeder Menge A die Menge $A + E$ zu (siehe Kapitel 2) und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} f + E : A + E &\rightarrow B + E \\ (a, 1) &\mapsto (f(a), 1) \\ (e, 2) &\mapsto (e, 2) \end{aligned}$$

Sei S eine Menge.

Der **Potenz-** oder **Leserfunktor** $\underline{}^S : \text{Set} \rightarrow \text{Set}$ ordnet jeder Menge A die Menge $S \rightarrow A$ zu und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} f^S : A^S &\rightarrow B^S \\ g &\mapsto f \circ g \end{aligned}$$

Der **Copotenz-** oder **Schreiberfunktor** $\underline{} \times S : \text{Set} \rightarrow \text{Set}$ kombiniert den Identitätsfunktor mit einem Produktfunktor: Er ordnet jeder Menge A die Menge $A \times S$ zu und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} f \times S : A \times S &\rightarrow B \times S \\ (a, s) &\mapsto (f(a), s) \end{aligned}$$

Der **Transitionsfunktor** $(\underline{} \times S)^S : \text{Set} \rightarrow \text{Set}$ kombiniert einen Leser- mit einem Schreiberfunktor: Er ordnet jeder Menge A die Menge $(A \times S)^S$ zu und jeder Funktion $f : A \rightarrow B$ die Funktion

$$\begin{aligned} (f \times S)^S : (A \times S)^S &\rightarrow (B \times S)^S \\ g &\mapsto (\lambda(a, s). (f(a), s)) \circ g \end{aligned}$$

Aufgabe Zeigen Sie, dass die hier definierten Funktionen tatsächlich Funktoren sind, also (2) und (3) erfüllen, und dass sie sich von Set auf Set^S fortsetzen lassen. □

Seien $F, G : \mathcal{K} \rightarrow \mathcal{L}$ Funktoren. Eine **natürliche Transformation** $\tau : F \rightarrow G$ ordnet jedem \mathcal{K} -Objekt A einen \mathcal{L} -Morphismus $\tau_A : F(A) \rightarrow G(A)$ derart, dass für alle \mathcal{K} -Morphismen $f : A \rightarrow B$ folgendes Diagramm kommutiert:

$$\begin{array}{ccc} F(A) & \xrightarrow{\tau_A} & G(A) \\ F(f) \downarrow & & \downarrow G(f) \\ F(B) & \xrightarrow{\tau_B} & G(B) \end{array}$$

Ein Funktor $M : \mathcal{K} \rightarrow \mathcal{K}$ heißt **Monade**, wenn es zwei natürliche Transformationen $\eta : \text{Id}_{\mathcal{K}} \rightarrow M$ (**Einheit**) und $\mu : M \circ M \rightarrow M$ (**Multiplikation**) gibt, die für alle $A \in \mathcal{K}$ das folgende Diagramm kommutativ machen:

$$\begin{array}{ccccc}
 M(A) & \xrightarrow{M(\eta_A)} & M(M(A)) & \xleftarrow{\eta_{M(A)}} & M(A) \\
 & \searrow (4) & \downarrow \mu_A & \swarrow (5) & \\
 M(id_A) & & M(A) & & M(id_A) \\
 & & \downarrow & & \\
 & & M(A) & &
 \end{array}
 \qquad
 \begin{array}{ccccc}
 M(M(M(A))) & \xrightarrow{\mu_{M(A)}} & M(M(A)) & & \\
 \downarrow M(\mu_A) & & \downarrow \mu_A & & \\
 M(M(A)) & \xrightarrow[\mu_A]{} & M(A) & &
 \end{array}
 \quad (6)$$

Beispiele

Viele der o.g. Funktoren sind Monaden. Einheit bzw. Multiplikation sind wie folgt definiert:
Seien A, E, S Mengen.

- Identitätsfunktor: $\eta_A = \mu_A = id_A$.
- Listenfunktor:

$$\begin{array}{ll}
 \eta_A : A \rightarrow A^* & \mu_A : (A^*)^* \rightarrow A^* \\
 a \mapsto (a) & (w_1, \dots, w_n) \mapsto w_1 \cdot \dots \cdot w_n
 \end{array}$$

- Mengenfunktor:

$$\begin{array}{ll}
 \eta_A : A \rightarrow \mathcal{P}(A) & \mu_A : \mathcal{P}(\mathcal{P}(A)) \rightarrow \mathcal{P}(A) \\
 a \mapsto \{a\} & S \mapsto \bigcup S
 \end{array}$$

- Ausnahmefunktor:

$$\begin{array}{ll}
 \eta_A : A \rightarrow A + E & \mu_A : (A + E) + E \rightarrow A + E \\
 a \mapsto (a, 1) & ((a, 1), 1) \mapsto (a, 1) \\
 & ((e, 2), 1) \mapsto (e, 2) \\
 & (e, 2) \mapsto (e, 2)
 \end{array}$$

- Leserfunktor:

$$\begin{array}{ll}
 \eta_A : A \rightarrow A^S & \mu_A : (A^S)^S \rightarrow A^S \\
 a \mapsto \lambda s. a & f \mapsto \lambda s. f(s)(s)
 \end{array}$$

- Schreiberfunktor: Hier muss S die Trägermenge eines Monoids $M = (S, *, e)$ sein.

$$\begin{array}{ll}
 \eta_A : A \rightarrow A \times S & \mu_A : (A \times S) \times S \rightarrow A \times S \\
 a \mapsto (a, e) & ((a, s), s') \mapsto (a, s * s')
 \end{array}$$

- Transitionsfunktor:

$$\begin{array}{ll}
 \eta_A : A \rightarrow (A \times S)^S & \mu_A : ((A \times S)^S \times S)^S \rightarrow (A \times S)^S \\
 a \mapsto \lambda s. (a, s) & f \mapsto (\lambda(h, s). h(s)) \circ f
 \end{array}$$

Aufgabe Zeigen Sie, dass für diese Beispiele (4)-(6) gilt. □

Seien A und B Mengen. Der **bind-Operator**

$$\gg= : M(A) \times (A \rightarrow M(B)) \rightarrow M(B)$$

wird wie folgt aus M und der Multiplikation von M abgeleitet:

Für alle $A \in \mathcal{K}$ und $f : A \rightarrow M(B)$,

$$(\gg= f) = \mu_B \circ M(f). \quad (7)$$

Intuitiv stellt man sich ein monadisches Objekt $m \in M(A)$ als Berechnung vor, die eine – evtl. leere – Menge von Werten in A erzeugt. Ein Ausdruck der Form $\textcolor{blue}{m} \gg= \textcolor{blue}{f}$ wird dann wie folgt ausgewertet: Die von m berechneten Werte $a \in A$ werden als Eingabe an die Berechnung f übergeben und von $f(a)$ verarbeitet.

Die Betrachtung der Elemente von $M(A)$ als Berechnungen, die eine Ausgabe in A produzieren, lässt sich besonders gut am Beispiel der Transitionsmonade begründen.

Aus der Multiplikation der Transitionsmonade und der allgemeinen Definition des bind-Operators ergibt sich nämlich die folgende Charakterisierung des bind-Operators der Transitionsmonade:

Für alle $g : S \rightarrow A \times S$ und $f : A \rightarrow (S \rightarrow B \times S)$,

$$\begin{aligned} g \gg= f &= \mu_B(M(f)(g)) = \mu_B((f \times S)^S(g)) = \mu_B((\lambda(a, s).(f(a), s)) \circ g) \\ &= (\lambda(h, s).h(s)) \circ (\lambda(a, s).(f(a), s)) \circ g = (\lambda(a, s).f(a)(s)) \circ g. \end{aligned}$$

Die Anwendung der Funktion $(g \gg= f) : S \rightarrow B \times S$ auf einen Zustand s' besteht demnach

- in der Anwendung von g auf s' , die eine Ausgabe a und einen Folgezustand s liefert,
- und der darauffolgenden Anwendung von $f(a)$ auf s .

Seien A, B, C Mengen, $a \in A$, $m \in M(A)$, $m' \in M(M(A))$, $f : A \rightarrow M(B)$, $g : B \rightarrow M(C)$ und $h : A \rightarrow B$. Aus (4)-(7) erhält man die folgenden Eigenschaften von $\gg=$:

$$m \gg= \eta_A = m, \tag{8}$$

$$\eta_A(a) \gg= f = f(a), \tag{9}$$

$$(m \gg= f) \gg= g = m \gg= \lambda a. f(a) \gg= g. \tag{10}$$

(2)-(4) und (7) implizieren die folgende Charakterisierung von $M(h)$ bzw. μ_A :

$$M(h)(m) = m \gg= \eta_B \circ h, \tag{11}$$

$$\mu_A(m') = m' \gg= id_{M(A)}.$$

Mit dem bind-Operator werden monadische Objekte *sequentiell* verknüpft. **Plusmonaden** haben zusätzlich eine **parallele Komposition**, die als natürliche Transformation

$$\oplus : M \times M =_{\text{def}} - \times - \circ \Delta \circ M \rightarrow M$$

definiert werden kann. Damit lassen sich u.a. Backtracking und Nichtdeterminismus monadischer Compiler realisieren. Wir werden \oplus deshalb in die Definition einer Compilermonade aufnehmen (s.u.).

Die Komposition von Monaden führt zu weiteren Monaden: Sei M eine Monade mit Einheit η , Multiplikation μ und paralleler Komposition \oplus und S eine Menge.

- Lesermonade über (M, \oplus) :

$$M' =_{\text{def}} \lambda A. M(A)^S$$

Die Morphismenabbildung von M , η , μ und \oplus werden wie folgt auf M' fortgesetzt:

Sei $h : A \rightarrow B$.

$$\begin{aligned} M'(h) : M'(A) &\rightarrow M'(B) \\ f &\mapsto M(h) \circ f \\ \eta'_A : A &\rightarrow M'(A) \\ a &\mapsto \lambda s. a \\ \mu'_A : M'(M'(A)) &\rightarrow M'(A) \\ f &\mapsto \lambda s. f(s) \gg= \lambda g. g(s) \end{aligned}$$

$$\begin{aligned}\oplus' : M'(A) \times M'(A) &\rightarrow M'(A) \\ (f, g) &\mapsto \lambda s. f(s) \oplus g(s)\end{aligned}$$

- Transitionsmonade über (M, \oplus) :

$$M' =_{\text{def}} \lambda A. M(A \times S)^S$$

Die Morphismenabbildung von M , η , μ und \oplus werden wie folgt auf M' fortgesetzt:
Sei $h : A \rightarrow B$.

$$\begin{aligned}M'(h) : M'(A) &\rightarrow M'(B) \\ f &\mapsto \lambda s. f(s) \gg= \lambda(a, s). \eta(h(a), s) \\ \eta'_A : A &\rightarrow M'(A) \\ a &\mapsto \lambda s. \eta(a, s) \\ \mu'_A : M'(M'(A)) &\rightarrow M'(A) \\ f &\mapsto \lambda s. f(s) \gg= \lambda(g, s). g(s) \\ \oplus' : M'(A) \times M'(A) &\rightarrow M'(A) \\ (f, g) &\mapsto \lambda s. f(s) \oplus g(s)\end{aligned}$$

Die Implementierung von Monaden in Haskell wird in Kapitel 15 behandelt, monadische Compiler in Kapitel 16. Letztere sind Transitionsmonaden über Ausnahmemonaden, wobei S die Menge X^* der zu verarbeitenden Wörter ist.

Compilermonaden

Sei $M : Set^S \rightarrow Set^S$ eine Monade mit Einheit η , bind-Operator $\gg=$ und paralleler Komposition \oplus , $set : M \rightarrow \mathcal{P}$ eine weitere natürliche Transformation und

$$E = \{m \in M(A) \mid set_A(m) = \emptyset, A \in Set^S\}$$

(“Menge der Ausnahmewerte”).

M heißt **Compilermonade**, wenn für alle Mengen A und B , $m, m', m'' \in M(A)$, $e \in E$, $f : A \rightarrow M(B)$, $h : A \rightarrow B$ und $a \in A$ Folgendes gilt:

$$\begin{aligned} (m \oplus m') \oplus m'' &= m \oplus (m' \oplus m''), \\ M(h)(e) &= e, \\ M(h)(m \oplus m') &= M(h)(m) \oplus M(h)(m'), \\ set_A(m \oplus m') &= set_A(m) \cup set_A(m'), \\ set_A(\eta_A(a)) &= \{a\}, \\ set_B(m \gg= f) &= \bigcup \{set_B(f(a)) \mid a \in set_A(m)\}. \end{aligned} \quad \begin{array}{l} (\text{CM1}) \\ (\text{CM2}) \\ (\text{CM3}) \end{array}$$

Nach Definition der Einheit $\eta^{\mathcal{P}}$ und des bind-Operators $\gg=^{\mathcal{P}}$ der Mengenmonade \mathcal{P} machen die letzten beiden Gleichungen set zum **Monadenmorphismus**.

Aus ihnen folgt nämlich:

$$\begin{aligned} \text{set}_A \circ \eta_A &= \eta_A^{\mathcal{P}}, \\ \text{set}_B(m \gg= f) &= \text{set}_A(m) \gg=^{\mathcal{P}} \text{set}_B \circ f. \end{aligned}$$

Außerdem erhält man für alle S -sortigen Mengen A , S -sortigen Funktionen $h : A \rightarrow B$ und $m \in M(A)$:

$$\begin{aligned} \text{set}_B(M(h)(m)) &= \text{set}_B(m \gg= \eta_B \circ h) = \bigcup \{ \text{set}_B(\eta_B(h(a))) \mid a \in \text{set}_A(m) \} \\ &= \bigcup \{ \{h(a)\} \mid a \in \text{set}_A(m) \} = h(\text{set}_A(m)). \end{aligned}$$

Satz 5.2 Listen-, Mengen- und Ausnahmefunktoren sind Compilermonaden.

Beweis. Seien A und E disjunkte Mengen.

Sei $\oplus = \cdot$ und $\text{set}_A : A^* \rightarrow \mathcal{P}(A)$ definiert durch $\text{set}_A(\epsilon) = \emptyset$ und $\text{set}_A(s) = \{a_1, \dots, a_n\}$ für alle $s = (a_1, \dots, a_n) \in A^+$. Damit ist \cdot^* eine Compilermonade.

Sei $\oplus = \cup$ und $\text{set}_A : \mathcal{P}(A) \rightarrow \mathcal{P}(A)$ definiert durch $\text{id}_{\mathcal{P}(A)}$. Damit ist \mathcal{P} eine Compilermonade.

Seien $\oplus_A : (A + E) \times (A + E) \rightarrow A + E$ und $\text{set}_A : A + E \rightarrow \mathcal{P}(A)$ definiert durch $\oplus_A((a, 1), x) = (a, 1)$ und $\oplus_A((e, 2), x) = x$ bzw. $\text{set}_A(a, 1) = \{a\}$ und $\text{set}_A(e, 2) = \emptyset$ für alle $a \in A$, $e \in E$ und $x \in A + E$. Damit ist $\lambda A. A + E$ eine Compilermonade. □

Monadenbasierte Parser und Compiler

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG, G' die daraus gebildete nicht-linksrekursive CFG, $X = \bigcup BS$, $\Sigma(G) = (S, F)$ und $M : Set^S \rightarrow Set^S$ eine Compilermonade.

Wie bereits in Kapitel 1 informell beschrieben wurde, soll ein Compiler für G in die als $\Sigma(G)$ -Algebra \mathcal{A} formulierten Zielsprache einen Parser für G mit der Faltung in \mathcal{A} der vom Parser erzeugten Syntaxbäume komponieren. Da der Parser dem Compiler in die Termalgebra $T_{\Sigma(G)}$ entsprechen soll, stellen wir folgende Anforderung an die Übersetzungsfunktion:

$$\text{compile}_G^{\mathcal{A}} = X^* \xrightarrow{\text{compile}_G^{T_{\Sigma(G)}}} M(T_{\Sigma(G)}) \xrightarrow{M(\text{fold}^{\mathcal{A}})} M(A). \quad (12)$$

Ist G linksrekursiv, dann wird $\text{parse}_G =_{\text{def}} \text{compile}_G^{T_{\Sigma(G)}}$ nicht immer terminieren. Deshalb fordern wir in diesem Fall

$$\text{compile}_G^{\mathcal{A}} = X^* \xrightarrow{\text{parse}_G} M(T_{\Sigma(G')}) \xrightarrow{M(\text{fold}^{\text{deref}(\mathcal{A})})} M(A)$$

anstelle von (12).

Da $T_{\Sigma(G)}$ eine initiale eine $\Sigma(G)$ -Algebra ist, stimmt $\text{fold}^{T_{\Sigma(G)}}$ mit der Identität auf $T_{\Sigma(G)}$ überein.

Also gilt (12) stets für $\mathcal{A} = T_{\Sigma(G)}$:

$$\begin{aligned} \text{compile}_G^{T_{\Sigma(G)}} &\stackrel{(12)}{=} id_{M(T_{\Sigma(G)})} \circ \text{parse}_G \stackrel{M \text{ ist Funktor}}{=} M(id_{T_{\Sigma(G)}}) \circ \text{parse}_G \\ &= M(fold^{T_{\Sigma(G)}}) \circ \text{parse}_G. \end{aligned}$$

Da $fold^{\mathcal{A}} : T_{\Sigma(G)} \rightarrow \mathcal{A}$ $\Sigma(G)$ -homomorph ist, folgt (12) aus folgender Verträglichkeit von Homomorphismen mit Compilerinstanzen: Für alle $\Sigma(G)$ -Homomorphismen $h : \mathcal{A} \rightarrow \mathcal{B}$,

$$M(h) \circ \text{compile}_G^{\mathcal{A}} = \text{compile}_G^{\mathcal{B}}. \quad (13)$$

(13) gilt genau dann, wenn

$$\text{compile}_G : \text{const}(X^*) \rightarrow M \circ U$$

eine natürliche Transformation ist, wobei

$$\text{const}(X^*), U : \text{Alg}_{\Sigma(G)} \rightarrow \text{Set}^S$$

die Funktoren bezeichnen, die

- jeder $\Sigma(G)$ -Algebra \mathcal{A} die S -sortige Menge $(X^*)_{s \in S}$ bzw. die Trägermenge von \mathcal{A} und
- jedem $\Sigma(G)$ -Homomorphismus h die S -sortige Funktion $(id_{X^*})_{s \in S}$ bzw. die h zugrundeliegende S -sortige Funktion

zuordnen.

Gilt (12), dann überträgt sich die Kommutativität von Diagramm (1) auf das folgende Diagramm:

$$\begin{array}{ccc}
 X^* & \xrightarrow{\text{compile}_G^A} & M(A) \\
 \downarrow \text{compile}_G^{Sem} & (14) & \downarrow M(\text{evaluate}) \\
 M(Sem) & \xrightarrow[M(\text{encode})]{} & M(Mach)
 \end{array}$$

$$\begin{aligned}
 M(\text{evaluate}) \circ \text{compile}_G^A &\stackrel{(12)}{=} M(\text{evaluate}) \circ M(\text{fold}^A) \circ \text{parse}_G \\
 M \text{ ist } \underset{\text{Funktör}}{=} & M(\text{evaluate} \circ \text{fold}^A) \circ \text{parse}_G \stackrel{(1)}{=} M(\text{encode} \circ \text{fold}^{Sem}) \circ \text{parse}_G \\
 M \text{ ist } \underset{\text{Funktör}}{=} & M(\text{encode}) \circ M(\text{fold}^{Sem}) \circ \text{parse}_G \stackrel{(12)}{=} M(\text{encode}) \circ \text{compile}_G^{Sem}.
 \end{aligned}$$

Sei $\text{unparse}_G =_{\text{def}} \text{fold}^{\text{Word}(G)}$.

parse_G ist **korrekt**, wenn für alle $w \in X^*$ die Faltung jedes von $\text{parse}_G(w)$ berechneten Syntaxbaums in der Algebra $\text{Word}(G)$ mit w übereinstimmt, formal:

$$\mathcal{P}(\text{unparse}_G)(\text{set}_{T_{\Sigma(G)}}(\text{parse}_G(w))) \subseteq \{w\}, \quad (15)$$

parse_G ist **vollständig**, wenn für alle $w \in L(G)$ $\text{parse}_G(w)$ kein Ausnahmewert ist, formal:

$$\text{set}_{\text{Word}(G)}(\text{compile}_G^{\text{Word}(G)}(w)) \neq \emptyset. \quad (16)$$

Wegen

$$\begin{aligned} \mathcal{P}(\text{unparse}_G) \circ \text{set}_{T_{\Sigma(G)}} \circ \text{parse}_G &= \text{set}_{\text{Word}(G)} \circ M(\text{unparse}_G) \circ \text{parse}_G \\ &= \text{set}_{\text{Word}(G)} \circ \text{compile}_G^{\text{Word}(G)} \end{aligned}$$

ist (15) äquivalent zu

$$\text{set}_{\text{Word}(G)}(\text{compile}_G^{\text{Word}(G)}(w)) \subseteq \{w\}. \quad (17)$$

Zusammenfassend ergibt sich folgende Definition:

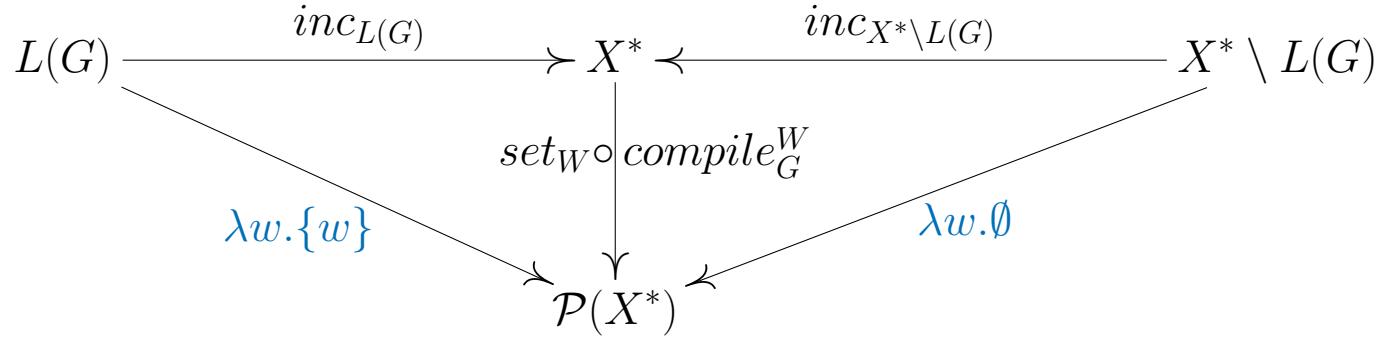
Ein **generischer Compiler für G** ist eine $\text{Alg}_{\Sigma(G)}$ -sortige Menge

$$\text{compile}_G = (\text{compile}_G^{\mathcal{A}} : X^* \rightarrow M(A))_{\mathcal{A}=(A,\text{Op}) \in \text{Alg}_{\Sigma(G)}},$$

die (13), (16) und (17) erfüllt.

Sei $W = Word(G)$. (16) und (17) machen $set_W \circ compile_G^W$ zu der Summenextension (siehe Kapitel 2)

$$[\lambda w.\{w\}, \lambda w.\emptyset] : L(G) + X^* \setminus L(G) \rightarrow \mathcal{P}(X^*).$$



U.a. ist dann die Einschränkung von $parse_G$ auf $L(G)$ rechtsinvers zu $unparse_G$:

$$\begin{aligned} set_W \circ M(unparse_G) \circ parse_G \circ inc_{L(G)} &= set_W \circ M(fold^W) \circ parse_G \circ inc_{L(G)} \\ \stackrel{(12)}{=} set_W \circ compile_G^W \circ inc_{L(G)} &= \lambda w.\{w\}. \end{aligned}$$

6 LL-Compiler

Der in diesem Kapitel definierte generische Compiler überträgt die Arbeitsweise eines klassischen LL-Parsers auf Compiler mit Backtracking. Das erste L steht für seine Verarbeitung des Eingabewortes von links nach rechts, das zweite L für seine – implizite – Konstruktion einer [Linksableitung](#). Da diese Arbeitsweise dem *mit der Wurzel beginnenden* schrittweisen Aufbau eines Syntaxbaums entspricht, werden LL-Parser auch **top-down-Parser** genannt.

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG, $G' = (S', BS, R')$ die daraus gebildete nicht-linksrekursive CFG, $X = \bigcup BS$, M eine Compilermonade, $\text{errmsg} : X^* \rightarrow E$ und $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$ -Algebra. compile_G heißt **LL-Compiler**, wenn

$$(\text{compile}_{G,s}^{\mathcal{A}} : X^* \rightarrow M(A_s))_{s \in S}$$

wie folgt definiert ist: Für alle $w \in X^*$,

$$\text{compile}_{G,s}^{\mathcal{A}}(w) = \text{trans}_s^{\mathcal{A}}(w) \gg= \lambda(a, w). \text{if } w = \epsilon \text{ then } \eta_A(a) \text{ else } \text{errmsg}(w), \quad (1)$$

wobei für alle für alle $s \in S' \cup BS$

$$\text{trans}_s^{\mathcal{A}} : X^* \rightarrow M(A_s \times X^*)$$

wie folgt definiert ist:

Fall 1: $s \in BS$. Für alle $x \in X$ und $w \in X^*$,

$$\begin{aligned} trans_s^{\mathcal{A}}(xw) &= \text{if } x \in s \text{ then } \eta_{A \times X^*}(x, w) \text{ else } errmsg(xw) \\ trans_s^{\mathcal{A}}(\epsilon) &= errmsg(\epsilon) \end{aligned}$$

Fall 2: $s \in S'$. Für alle $w \in X^*$,

$$trans_s^{\mathcal{A}}(w) = \bigoplus_{r=(s \rightarrow e) \in R} try_r^{\mathcal{A}}(w). \quad (2)$$

Für alle $r = (s \rightarrow (e_1, \dots, e_n)) \in R'$ und $w \in X^*$,

$$try_r^{\mathcal{A}}(w) = \begin{cases} trans_{e_1}^{\mathcal{A}}(w) \gg= \lambda(a_1, w_1). \\ trans_{e_2}^{\mathcal{A}}(w_1) \gg= \lambda(a_2, w_2). \\ \vdots \\ trans_{e_n}^{\mathcal{A}}(w_{n-1}) \gg= \lambda(a_n, w_n). \eta_{A \times X^*}(f_r^{deref(\mathcal{A})}(a_{i_1}, \dots, a_{i_k}), w_n), \end{cases} \quad (3)$$

wobei $\{i_1, \dots, i_k\} = \{1 \leq i \leq n \mid e_i \in S' \cup BS \setminus Z(G)\}$.

Während alle Summanden von (2) auf das gesamte Eingabewort w angewendet werden, wird es von $try_r^{\mathcal{A}}$ Symbol für Symbol verarbeitet: Zunächst wird $trans_{e_1}^{\mathcal{A}}$ auf w angewendet, dann $trans_{e_2}^{\mathcal{A}}$ auf das von $trans_{e_1}^{\mathcal{A}}$ nicht verarbeitete Suffix w_1 von w , $trans_{e_3}^{\mathcal{A}}$ auf das von $trans_{e_2}^{\mathcal{A}}$ nicht verarbeitete Suffix w_2 von w_1 , usw.

Gibt es $1 \leq j \leq n$ derart, dass $\text{trans}_{e_j}^{\mathcal{A}}(w)$ scheitert, d.h. existiert kein aus e_j ableitbares Präfix von w , dann übergibt $\text{try}_r^{\mathcal{A}}$ das gesamte Eingabewort zur Parsierung an den nächsten Teilcompiler $\text{try}_{r'}^{\mathcal{A}}$ der Argumentliste von \oplus in (2) (**Backtracking**).

Ist $\text{trans}_{e_i}^{\mathcal{A}}(w)$ jedoch für alle $1 \leq i \leq n$ erfolgreich, dann schließt der Aufruf von $\text{try}_r^{\mathcal{A}}$ mit der Anwendung von $f_r^{\text{deref}(\mathcal{A})}$ auf die Zwischenergebnisse $a_{i_1}, \dots, a_{i_k} \in A$.

Klassische LL-PARSER sind deterministisch, d.h. sie erlauben kein Backtracking, sondern setzen voraus, dass die ersten k Symbole des Eingabewortes w bestimmen, welcher der Teilcompiler $\text{try}_{s \rightarrow e}^{\mathcal{A}}$ aufgerufen werden muss, um zu erkennen, dass w zu $L(G)_s$ gehört. CFGs, die diese Voraussetzung erfüllen, heißen **LL(k)-Grammatiken**.

Im Gegensatz zur Nicht-Linksrekursivität lässt sich die $LL(k)$ -Eigenschaft nicht immer durch eine Transformation der Grammatik erzwingen, auch dann nicht, wenn sie eine von einem deterministischen Kellerautomaten erkennbare Sprache erzeugt!

Monadische Ausdrücke werden in Haskell oft in der **do-Notation** wiedergegeben. Sie verdeutlicht die Korrespondenz zwischen monadischen Berechnungen einerseits und imperativen Programmen andererseits. Die Rückübersetzung der do-Notation in die ursprünglichen monadischen Ausdrücke ist induktiv wie folgt definiert:

Sei $m \in M(A)$, $a \in A$ und $m' \in M(B)$.

$$\textcolor{red}{do} \ a \leftarrow m; \ m' = m \gg= \lambda a. \textcolor{red}{do} \ m'$$

$$\textcolor{red}{do} \ m; \ m' = m \gg \textcolor{red}{do} \ m'$$

Gleichung (10) von Kapitel 5 impliziert die Assoziativitat des Sequentialisierungsoperators (\cdot) , d.h. $\textcolor{blue}{do} \ m; m'; m''$ ist gleichbedeutend mit $\textcolor{blue}{do} \ (\textcolor{blue}{do} \ m; m'); m''$ und $\textcolor{blue}{do} \ m; (\textcolor{blue}{do} \ m'; m'')$.

Beispiel 6.1 SAB (siehe Beispiel 4.5)

Die Regeln

$$\begin{aligned} r_1 &= S \rightarrow aB, & r_2 &= S \rightarrow bA, & r_3 &= S \rightarrow \epsilon, \\ r_4 &= A \rightarrow aS, & r_5 &= A \rightarrow bAA, & r_6 &= B \rightarrow bS, \\ r_7 &= B \rightarrow aBB \end{aligned}$$

von SAB liefern nach obigem Schema die folgende Transitionsfunktion des LL-Compilers in eine beliebige SAB-Algebra alg :

Fur alle $x, z \in \{a, b\}$ und $w \in \{a, b\}^*$,

```
trans_z(xw) = if x = z then η(z,w) else errmsg(xw)
trans_z(ε)   = errmsg(ε)
```

```

trans_S(w) = try_r1(w) ⊕ try_r2(w) ⊕ try_r3(w)
trans_A(w) = try_r4(w) ⊕ try_r5(w)
trans_B(w) = try_r6(w) ⊕ try_r7(w)

try_r1(w) = do (x,w) <- trans_a(w);
               (c,w) <- trans_B(w); η(f_r1^alg(c),w)
try_r2(w) = do (x,w) <- trans_b(w);
               (c,w) <- trans_A(w); η(f_r2^alg(c),w)
try_r3(w) = η(f_r3^alg,w)
try_r4(w) = do (x,w) <- trans_a(w);
               (c,w) <- trans_S(w); η(f_r4^alg(c),w)
try_r5(w) = do (x,w) <- trans_b(w);
               (c,w) <- trans_A(w);
               (d,w) <- trans_A(w); η(f_r5^alg(c,d),w)
try_r6(w) = do (x,w) <- trans_b(w);
               (c,w) <- trans_S(w); η(f_r6^alg(c),w)
try_r7(w) = do (x,w) <- trans_a(w);
               (c,w) <- trans_B(w);
               (d,w) <- trans_B(w); η(f_r7^alg(c,d),w)

```

□

Sei $W = Word(G)$. Um später zeigen zu können, dass der LL-Compiler vollständig ist, setzen wir voraus, dass

- für alle $s \in S$, $v \in L(G)_s$ und $w \in X^*$

$$r = (s \rightarrow (e_1, \dots, e_n)) \in R$$

und für alle $1 \leq i \leq n$ $v_i \in L(G)_{e_i}$ mit $v_1 \dots v_n = v$ und

$$set_{W^2}(try_r^W(w)) \subseteq set_{W^2}(trans_s^W(w))$$

existieren. **(LLC)**

Gibt es keine Anordnung der Regeln in (2), die diese Bedingung erfüllt, dann müssen ggf. neue Sorten für die Vorkommen einer Sorte in bestimmten Kontexten eingeführt und die Regeln von R entsprechend modifiziert werden.

Beispiel 6.2

Ein solcher Fall würde z.B. in **JavaLight+** auftreten (siehe Kapitel 13), wenn wir dort anstelle der drei Sorten *ExpSemi*, *ExpBrac* und *ExpComm* die zunächst naheliegende eine Sorte *Exp* und die folgenden Regeln verwenden würden:

Command \rightarrow *String* = *Exp*; | write *Exp*; | ... (A)

Exp \rightarrow *Sum* | *Disjunct* (B)

Sum \rightarrow ... | *Prod*

Prod \rightarrow ... | *Factor*

Factor \rightarrow *String* | *String Actuals* | ...

Disjunct \rightarrow ... | *Conjunct*

Conjunct \rightarrow ... | *Literal*

Literal \rightarrow *String* | *String Actuals* | ...

Actuals \rightarrow () | (*Actuals'*)

Actuals' \rightarrow *Exp*) | *Exp, Actuals'* (C)

Ein LL-Compiler für *Command* würde z.B. die Eingabe `z = x<=11;` als syntaktisch inkorrekt betrachten, weil der Teilcompiler für *Exp* zunächst nach dem längsten aus *Sum* ableitbaren Präfix von `x<=11` sucht, `x` als solches erkennt und deshalb das Wort `x<=11` nicht bis zum Ende liest.

Die Ersetzung von Regel (B) durch $Exp \rightarrow Disjunct \mid Sum$ löst das Problem nicht, denn nun würde der Compiler z.B. die Eingabe `z = x+11;` als syntaktisch inkorrekt betrachten, weil der Teilcompiler für *Exp* zunächst nach dem längsten aus *Disjunct* ableitbaren Präfix von `x+11` sucht, `x` als solches erkennt und deshalb das Wort `x+11` nicht bis zum Ende liest.

Ersetzt man hingegen (A)-(C) durch

$$\begin{aligned}
 \textit{Command} &\rightarrow \textcolor{brown}{String} = \textit{ExpSemi} \mid \textit{write ExpSemi} \mid \dots \\
 \textit{ExpSemi} &\rightarrow \textit{Sum;} \mid \textit{Disjunct;} \\
 \textit{ExpBrac} &\rightarrow \textit{Sum}) \mid \textit{Disjunct}) \\
 \textit{ExpComm} &\rightarrow \textit{Sum,} \mid \textit{Disjunct,} \\
 \textit{Actuals}' &\rightarrow \textit{ExpBrac} \mid \textit{ExpComm Actuals}'
 \end{aligned}$$

dann wird alles gut: Jetzt erzwingt das auf *Sum* oder *Disjunct* folgende Terminal (Semicolon, schließende Klammer bzw. Komma), dass die Compiler für *ExpSemi*, *ExpBrac* und *ExpComm* die jeweilige Resteingabe stets bis zu diesem Terminal verarbeiten.

Natürlich genügt ein Compiler für alle drei Sorten. Man muss ihn lediglich mit dem jeweiligen Terminal parametrisieren (siehe [Java2.hs](#)). □

Satz 6.4 $\textit{trans}^{\mathcal{A}}$ ist wohldefiniert.

Beweis durch Noethersche Induktion. Da S endlich und G nicht linksrekursiv ist, ist die folgende Ordnung $>$ auf $S \cup BS$ **Noethersch**, d.h., es gibt keine unendlichen Ketten $s_1 > s_2 > s_3 > \dots$:

$$s > s' \Leftrightarrow_{\text{def}} \exists w \in (S \cup BS)^*: s \xrightarrow{+}_G s'w.$$

Wir erweitern $>$ zu einer ebenfalls Noetherschen Ordnung \succ auf $X^* \times (S \cup BS)$:

$$(v, s) \succ (w, s') \Leftrightarrow_{def} |v| > |w| \text{ oder } (|v| = |w| \text{ und } s > s').$$

Nach Definition von $trans_s(w)$ gibt es $r = (s \rightarrow e_1 \dots e_n) \in R$ mit

$$trans_s^A(w) = \dots trans_{e_1}^A(w) \dots trans_{e_2}^A(w_1) \dots trans_{e_n}^A(w_{n-1}) \dots$$

Da w_1, \dots, w_n echte Suffixe von w sind, gilt $s > e_1$ und $|w| > |w_i|$ für alle $1 \leq i < n$, also $(w, s) \succ (w, e_1)$ und $(w, s) \succ (w_{i-1}, e_i)$ für alle $1 < i \leq n$. Die rekursiven Aufrufe von $trans^A$ haben also bzgl. \succ kleinere Argumente.

Folglich terminiert jeder Aufruf von $trans^A$, m.a.W.: $trans^A$ ist wohldefiniert. □

Der Beweis, dass $compile_G$ Gleichung (15) von Kapitel 5 erfüllt, verwendet die folgenden Zusammenhänge zwischen einer Monade M und ihrem bind-Operator:

Lemma 6.5

Sei $M : Set^S \rightarrow Set^S$ eine Monade und $h : A \rightarrow B$ eine S -sortige Funktion. Für alle $m \in M(A)$, $f : A \rightarrow M(A)$ und $g : B \rightarrow M(B)$,

$$M(h)(m \gg= f) = m \gg= M(h) \circ f, \tag{4}$$

$$M(h)(m) \gg= g = m \gg= g \circ h. \tag{5}$$

Sei Σ eine Signatur, $m_1, \dots, m_n \in M(A)$, $h : A \rightarrow B$ Σ -homomorph, für alle $1 \leq i \leq n$,

$$f_i : A \rightarrow \dots \rightarrow A \rightarrow M(A), \quad g_i : B \rightarrow \dots \rightarrow B \rightarrow M(B),$$

und $t \in T_\Sigma(V)$ mit $var(t) = \{x_1, \dots, x_n\}$,

$$\begin{aligned} f_i(a_1) \dots (a_{i-1}) &= m_i \gg f_{i+1}(a_1) \dots (a_{i-1}), \\ f_{n+1}(a_1) \dots (a_n) &= \eta_A(f_a^*(t)) \end{aligned}$$

für alle $a = (a_1, \dots, a_n) \in A$ und

$$\begin{aligned} g_i(b_1) \dots (b_{i-1}) &= M(h)(m_i) \gg g_{i+1}(b_1) \dots (b_{i-1}), \\ g_{n+1}(b_1) \dots (b_n) &= \eta_B(g_b^*(t)) \end{aligned}$$

für alle $b = (b_1, \dots, b_n) \in B$, wobei $f_a : V \rightarrow A$ und $g_b : V \rightarrow B$ durch $f_a(x_i) = a_i$ und $g_b(x_i) = b_i$ für alle $1 \leq i \leq n$ definiert sind. Dann gilt

$$M(h)(f_1) = g_1. \tag{6}$$

Beweis von (4).

$$\begin{aligned} M(h)(m \gg= f) &\stackrel{(11)}{=} \stackrel{in \ Kap. \ 5}{=} (m \gg= f) \gg= \eta_B \circ h \\ &\stackrel{(10)}{=} \stackrel{in \ Kap. \ 5}{=} m \gg= \lambda a. f(a) \gg= \eta_B \circ h \\ &\stackrel{(11)}{=} \stackrel{in \ Kap. \ 5}{=} m \gg= \lambda a. M(h)(f(a)) = m \gg= M(h) \circ f. \end{aligned}$$

Beweis von (5).

$$\begin{aligned} M(h)(m) \gg= g &= (m \gg= \eta_B \circ h) \gg= g \stackrel{(10) \text{ in } Kap. 5}{=} m \gg= \lambda a. \eta_B(h(a)) \gg= g \\ &\stackrel{(9) \text{ in } Kap. 5}{=} m \gg= \lambda a. g(h(a)) = m \gg= g \circ h. \end{aligned}$$

Beweis von (6).

$$\begin{aligned} M(h)(f_1) &= M(h)(m_1 \gg f_2) \stackrel{(4)}{=} m_1 \gg= M(h) \circ f_2 = m_1 \gg= \lambda a_1. M(h)(f_2(a_1)) \\ &= m_1 \gg= \lambda a_1. M(h)(m_2 \gg= f_3(a_1)) \stackrel{(4)}{=} m_1 \gg= \lambda a_1. m_2 \gg= M(h) \circ f_3(a_1) \\ &= m_1 \gg= \lambda a_1. m_2 \gg= \lambda a_2. M(h)(f_3(a_1)(a_2)) \\ &= \dots = m_1 \gg= \lambda a_1. m_2 \gg= \dots \gg= \lambda a_n. M(h)(f_{n+1}(a_1) \dots (a_n)) \\ &= m_1 \gg= \lambda a_1. m_2 \gg= \dots \gg= \lambda a_n. M(h)(\eta_A(f_a^*(t))) \\ &= m_1 \gg= \lambda a_1. m_2 \gg= \dots \gg= \lambda a_n. \eta_B(h(f_a^*(t))) \\ &= m_1 \gg= \lambda a_1. m_2 \gg= \dots \gg= \lambda a_n. \eta_B((h \circ f_a)^*(t)) \\ &\stackrel{h \circ f_a = g_{h(a)}}{=} m_1 \gg= \lambda a_1. m_2 \gg= \dots \gg= \lambda a_n. \eta_B(g_{h(a)}^*(t)) \end{aligned} \tag{7}$$

$$\begin{aligned}
g_1 &= M(h)(m_1) \gg= g_2 \stackrel{(5)}{=} m_1 \gg= g_2 \circ h = m_1 \gg= \lambda a_1. g_2(h(a_1)) \\
&= m_1 \gg= \lambda a_1. M(h)(m_2) \gg= g_3(h(a_1)) = m_1 \gg= \lambda a_1. m_2 \gg= g_3(h(a_1)) \circ h \\
&= m_1 \gg= \lambda a_1. m_2 \gg= \lambda a_2. g_3(h(a_1))(h(a_2)) \\
&= \cdots = m_1 \gg= \lambda a_1. m_2 \gg= \cdots \gg= \lambda a_n. g_{n+1}(h(a_1)) \dots, (h(a_n)) \\
&= \textcolor{blue}{m_1 \gg= \lambda a_1. m_2 \gg= \cdots \gg= \lambda a_n. \eta_B(g_{h(a)}^*(t))} \tag{8}
\end{aligned}$$

Aus (7) und (8) folgt (6). □

Satz 6.6

Sei $h : \mathcal{A} \rightarrow \mathcal{B} = (B, Op')$ ein $\Sigma(G)$ -Homomorphismus. Der oben definierte LL-Compiler ist mit Homomorphismen verträglich, d.h.

$$\textcolor{blue}{compile}_G^{\mathcal{B}} = M(h) \circ \textcolor{blue}{compile}_G^{\mathcal{A}}. \tag{9}$$

Beweis. Sei

$$\textcolor{blue}{trans}^{\mathcal{B}} = M(h \times X^*) \circ \textcolor{blue}{trans}^{\mathcal{A}}. \tag{10}$$

Dann gilt (9): Für alle $w \in X^*$,

$$\textcolor{blue}{compile}_G^{\mathcal{B}}(w) \stackrel{(1)}{=} \textcolor{blue}{trans}^{\mathcal{B}}(w) \gg= \lambda(b, w). \textit{if } w = \epsilon \textit{ then } \eta_B(b) \textit{ else } \textcolor{red}{errmsg}(w)$$

$$\begin{aligned}
& \stackrel{(10)}{=} M(h \times X^*)(trans^A(w)) \gg= \lambda(b, w).if\ w = \epsilon\ then\ \eta_B(b)\ else\ errmsg(w) \\
& \stackrel{(11)\ in\ Kap.\ 5}{=} (trans^A(w) \gg= (\eta_{B \times X^*} \circ (h \times X^*))) \\
& \qquad \gg= \lambda(b, w).if\ w = \epsilon\ then\ \eta_B(b)\ else\ errmsg(w) \\
& = (trans^A(w) \gg= \lambda(a, w).\eta_{B \times X^*}(h(a), w)) \\
& \qquad \gg= \lambda(b, w).if\ w = \epsilon\ then\ \eta_B(b)\ else\ errmsg(w) \\
& \stackrel{(10)\ in\ Kap.\ 5}{=} trans^A(w) \gg= \lambda(a, w).\eta_{B \times X^*}(h(a), w) \\
& \qquad \gg= \lambda(b, w).if\ w = \epsilon\ then\ \eta_B(b)\ else\ errmsg(w) \\
& \stackrel{(9)\ in\ Kap.\ 5}{=} trans^A(w) \gg= \lambda(a, w).if\ w = \epsilon\ then\ \eta_B(h(a))\ else\ errmsg(w) \\
& \stackrel{(CM1)}{=} trans^A(w) \gg= \lambda(a, w).if\ w = \epsilon\ then\ \eta_B(h(a))\ else\ M(h)(errmsg(w)) \\
& \stackrel{(9),(11)\ in\ Kap.\ 5}{=} trans^A(w) \gg= \lambda(a, w).if\ w = \epsilon\ then\ \eta_A(a) \gg= \eta_B \circ h \\
& \qquad \qquad \qquad else\ errmsg(w) \gg= \eta_B \circ h \\
& = trans^A(w) \gg= \lambda(a, w).(if\ w = \epsilon\ then\ \eta_A(a)\ else\ errmsg(w)) \gg= \eta_B \circ h \\
& \stackrel{(10)\ in\ Kap.\ 5}{=} (trans^A(w) \gg= \lambda(a, w).if\ w = \epsilon\ then\ \eta_A(a)\ else\ errmsg(w)) \gg= \eta_B \circ h \\
& = compile_G^A(w) \gg= \eta_B \circ h \stackrel{(11)\ in\ Kap.\ 5}{=} M(h)(compile_G^A(w)).
\end{aligned}$$

Beweis von (10) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung \succ :

Fall 1: $s \in BS$. Für alle $x \in X$ und $w \in X^*$,

$$M(h \times X^*)(trans_s^{\mathcal{A}}(xw)) = \begin{cases} \text{if } x \in s \text{ then } M(h \times X^*)(\eta_{A \times X^*}(x, w)) \\ \text{else } M(h \times X^*)(errmsg(xw)) \end{cases}$$

$$\eta \text{ ist nat. Transformation, (CM1)} \underset{=} \rightarrow \begin{cases} \text{if } x \in s \text{ then } \eta_{B \times X^*}((h \times X^*)(x, w)) \\ \text{else } errmsg(xw) \end{cases}$$

$$(h \times X^*)(x, w) = (h(x), w) = (x, w) \underset{=} \rightarrow \begin{cases} \text{if } x \in s \text{ then } \eta_{B \times X^*}(x, w) \\ \text{else } errmsg(xw) \end{cases} = trans_s^{\mathcal{B}}(xw),$$

$$M(h \times X^*)(trans_s^{\mathcal{A}}(\epsilon)) = M(h \times X^*)(errmsg(\epsilon)) \stackrel{(CM1)}{=} errmsg(\epsilon) = trans_s^{\mathcal{B}}(\epsilon).$$

Fall 2: $s \in S$. Sei $r = (s \rightarrow w') \in R$. Für alle $w \in X^*$,

$$M(h \times X^*)(try_r^{\mathcal{A}}(w))$$

$$\stackrel{(3)}{=} M(h \times X^*)\left(\left\{ \begin{array}{l} trans_{e_1}^{\mathcal{A}}(w) \gg= \lambda(a_1, w_1). \\ \vdots \\ trans_{e_n}^{\mathcal{A}}(w_{n-1}) \gg= \lambda(a_n, w_n). \eta_{A \times X^*}(f_r^{deref(\mathcal{A})}(a_{j_1}, \dots, a_{j_k}), w_n) \end{array} \right\}\right)$$

$$= M(h \times X^*)\left(\left\{ \begin{array}{l} trans_{e_1}^{\mathcal{A}}(w) \gg= \lambda x_1. \\ \vdots \\ trans_{e_n}^{\mathcal{A}}(w_{n-1}) \gg= \lambda x_n. \eta_{A \times X^*}(f_r^{deref(\mathcal{A})}(\pi_1(x_{j_1}), \dots, \pi_1(x_{j_k})), \pi_2(x_n)) \end{array} \right\}\right)$$

$$\begin{aligned}
& \stackrel{6.5}{=} \left\{ \begin{array}{l} M(h \times X^*)(trans_{e_1}^{\mathcal{A}}(w)) \gg= \lambda x_1. \\ \vdots \\ M(h \times X^*)(trans_{e_n}^{\mathcal{A}}(w_{n-1})) \\ \gg= \lambda x_n. \eta_{B \times X^*}(f_r^{deref(\mathcal{B})}(\pi_1(x_{j_1}), \dots, \pi_1(x_{j_k})), \pi_2(x_n)) \end{array} \right\} \\
& = \left\{ \begin{array}{l} M(h \times X^*)(trans_{e_1}^{\mathcal{A}}(w)) \gg= \lambda(b_1, w_1). \\ \vdots \\ M(h \times X^*)(trans_{e_n}^{\mathcal{A}}(w_{n-1})) \gg= \lambda(b_n, w_n). \eta_{B \times X^*}(f_r^{deref(\mathcal{B})}(b_{j_1}, \dots, b_{j_k}), w_n) \end{array} \right\} \\
& \stackrel{ind. \ hyp.}{=} \left\{ \begin{array}{l} trans_{e_1}^{\mathcal{B}}(w) \gg= \lambda(b_1, w_1). \\ \vdots \\ trans_{e_n}^{\mathcal{B}}(w_{n-1}) \gg= \lambda(b_n, w_n). \eta_{B \times X^*}(f_r^{deref(\mathcal{B})}(b_{j_1}, \dots, b_{j_k}), w_n) \end{array} \right\} \\
& \stackrel{(3)}{=} try_r^{\mathcal{B}}(w). \tag{11}
\end{aligned}$$

Daraus folgt (10):

$$\begin{aligned}
& M(h \times X^*)(trans_s^{\mathcal{A}}(w)) \stackrel{(2)}{=} M(h \times X^*)(\bigoplus_{r=(s \rightarrow e) \in R} try_r^{\mathcal{A}}(w)) \\
& \stackrel{(CM2)}{=} \bigoplus_{r=(s \rightarrow e) \in R} M(h \times X^*)(try_r^{\mathcal{A}}(w)) \stackrel{(11)}{=} \bigoplus_{r=(s \rightarrow e) \in R} try_r^{\mathcal{B}}(w) \stackrel{(2)}{=} trans_s^{\mathcal{B}}(w). \quad \square
\end{aligned}$$

Lemma 6.7

Sei $M : Set^S \rightarrow Set^S$ eine Compilermonade, $f : A^N \rightarrow A$ und $m_1, \dots, m_n \in M(A)$.

$$\begin{aligned} & set_A(m_1 \gg= \lambda a_1 \dots m_n \gg= \lambda a_n. \eta_A(f(a_1, \dots, a_n))) \\ & = \{f(a_1, \dots, a_n) \mid \bigwedge_{i=1}^n a_i \in set_A(m_i)\}. \end{aligned}$$

Beweis.

$$\begin{aligned} & set_A(m_1 \gg= \lambda a_1 \dots m_n \gg= \lambda a_n. \eta_A(f(a_1, \dots, a_n))) \\ & = \bigcup \{set_A(m_2 \gg= \lambda a_2 \dots m_n \gg= \lambda a_n. \eta_A(f(a_1, \dots, a_n))) \mid a_1 \in set_A(m_1)\} \\ & = \bigcup \{\bigcup \{set_A(m_3 \gg= \lambda a_3 \dots m_n \gg= \lambda a_n. \eta_A(f(a_1, \dots, a_n))) \mid a_2 \in set_A(m_2)\} \\ & \quad \mid a_1 \in set_A(m_1)\} \\ & = \bigcup \{set_A(m_3 \gg= \lambda a_3 \dots m_n \gg= \lambda a_n. \eta_A(f(a_1, \dots, a_n))) \\ & \quad \mid a_2 \in set_A(m_2), a_1 \in set_A(m_1)\} \\ & = \dots \\ & = \bigcup \{set_A(\eta_A(f(a_1, \dots, a_n))) \mid \bigwedge_{i=1}^n a_i \in set_A(m_i)\} \\ & = \bigcup \{\{f(a_1, \dots, a_n)\} \mid \bigwedge_{i=1}^n a_i \in set_A(m_i)\} \\ & = \{f(a_1, \dots, a_n) \mid \bigwedge_{i=1}^n a_i \in set_A(m_i)\}. \quad \square \end{aligned}$$

Satz 6.8

Der oben definierte LL-Compiler ist korrekt, d.h. für alle $w \in X^*$ gilt:

$$set_{Word(G)}(compile_G^{Word(G)}(w)) \subseteq \{w\}. \quad (12)$$

Beweis. Sei $W = Word(G)$. Für alle $w \in X^*$ und $s \in S \cup BS$ gelte folgende Bedingung:

$$\forall (v, v') \in set_{W^2}(trans_s^W(w)) : vv' = w. \quad (13)$$

Dann gilt (12):

$$\begin{aligned} & set_W(compile_{G,s}^W(w)) \\ & \stackrel{(1)}{=} set_W(trans_s^W(w) \gg= \lambda(v, v').if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v')) \\ & = \bigcup \{set_W(if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v')) \mid (v, v') \in set_{W^2}(trans_s^W(w))\} \\ & \stackrel{(13)}{\subseteq} \bigcup \{set_W(if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v')) \mid vv' = w\} \\ & = \bigcup \{if\ v' = \epsilon\ then\ set_W(\eta_W(v))\ else\ set_W(errmsg(v')) \mid vv' = w\} \\ & = \bigcup \{if\ v' = \epsilon\ then\ \{v\}\ else\ \emptyset \mid vv' = w\} = \{w\}. \end{aligned}$$

Beweis von (15) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung \succ .

Fall 1: $s \in BS$. Sei $x \in X$, $w \in X^*$ und $(v, v') \in set_{W^2}(trans_s^W(xw))$. Dann gilt

$$\begin{aligned} (v, v') &\in set_{W^2}(\text{if } x \in s \text{ then } \eta_{W^2}(x, w) \text{ else } errmsg(xw)) \\ &= \text{if } x \in s \text{ then } set_{W^2}(\eta_{W^2}(x, w)) \text{ else } set_{W^2}(errmsg(xw)) \\ &= \text{if } x \in s \text{ then } \{(x, w)\} \text{ else } \emptyset, \end{aligned}$$

also $vv' = xw$.

Fall 2: $s \in S$. Sei $w \in X^*$ und $(v, v') \in set_{W^2}(trans_s^W(w))$. Dann gilt

$$\begin{aligned} (v, v') &\in set_{W^2}(trans_s^W(w))) \stackrel{(2)}{=} set_{W^2}(\bigoplus_{r=(s \rightarrow e) \in R} try_r^W(w)) \\ &\stackrel{(CM3)}{\subseteq} \bigcup_{r=(s \rightarrow e) \in R} set_{W^2}(try_r^W(w)). \end{aligned}$$

Also gibt es $r = (s \rightarrow e_1 \dots e_n) \in R$ mit $(v, v') \in set_{W^2}(try_r^W(w))$. Sei $w_0 = w$. Dann gilt

$$\begin{aligned} (v, v') &\in set_{W^2}(try_r^W(w_0)) \\ &\stackrel{(3)}{=} set_{W^2}(trans_{e_1}^W(w_0) \gg= \lambda(v_1, w_1). \\ &\quad \vdots \\ &\quad trans_{e_n}^W(w_{n-1}) \gg= \lambda(v_n, w_n). \eta_{W^2}(f_r^W(v_{j_1}, \dots, v_{j_k}), w_n)) \\ &\stackrel{Lemma}{=} \stackrel{6.7}{=} \{(f_r^W(v_{j_1}, \dots, v_{j_k}), w_n) \mid \bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\} \\ &= \{(v_1 \dots v_n, w_n) \mid \bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\}. \end{aligned} \tag{14}$$

Also gibt es $v_1, \dots, v_n, w_1, \dots, w_n \in X^*$ mit $\bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))$, $v = v_1 \dots v_n$ und $v' = w_n$. Nach Induktionsvoraussetzung folgt $\bigwedge_{i=1}^n v_i w_i = w_{i-1}$, also

$$vv' = v_1 \dots v_n w_n = w_0 = w. \quad \square$$

Satz 6.9 Der oben definierte LL-Compiler ist vollständig, d.h. für alle $w \in L(G)$ gilt:

$$set_W(\text{compile}_G^{Word(G)}(w)) \neq \emptyset. \quad (15)$$

Beweis. Sei $W = Word(G)$. Für alle $s \in S \cup BS$, $v \in L(G)_s$ und $w \in X^*$ gelte:

$$(v, w) \in set_{W^2}(trans_s^W(vw)). \quad (16)$$

Dann gilt (15): Sei $w \in L(G)$.

$$\begin{aligned} & set_W(\text{compile}_{G,s}^{Word(G)}(w)) \\ & \stackrel{(1)}{=} set_W(trans_s^W(w) \gg= \lambda(v, v'). if \ v' = \epsilon \ then \ \eta_W(v) \ else \ errmsg(v'))} \\ & = \bigcup \{ set_W(if \ v' = \epsilon \ then \ \eta_W(v) \ else \ errmsg(v')) \mid (v, v') \in set_{W^2}(trans_s^W(w)) \} \\ & = \bigcup \{ set_W(\eta_W(v)) \mid (v, \epsilon) \in set_{W^2}(trans_s^W(w)) \} \cup \\ & \quad \bigcup \{ set_W(errmsg(v')) \mid (v, v') \in set_{W^2}(trans_s^W(w)), \ v' \neq \epsilon \} \\ & = \bigcup \{ \{v\} \mid (v, \epsilon) \in set_{W^2}(trans_s^W(w)) \} \cup \bigcup \{ \emptyset \mid (v, v') \in set_{W^2}(trans_s^W(w)), \ v' \neq \epsilon \} \\ & \stackrel{(16)}{=} \{v \mid (v, \epsilon) \in set_{W^2}(trans_s^W(w))\} \neq \emptyset. \end{aligned}$$

Beweis von (16) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung \succ .

Fall 1: $s \in BS$. Dann ist $v \in X$ und damit

$$set_{W^2}(trans_s^W(vw)) = set_{W^2}(\eta_{W^2}(v, w)) = \{(v, w)\}.$$

Fall 2: $s \in S$. Wegen (LLC) gibt es $r = (s \rightarrow (e_1, \dots, e_n)) \in R$ und für alle $1 \leq i \leq n$ $v_i \in L(G)_{e_i}$ mit $v_1 \dots v_n = v$ und

$$set_{W^2}(try_r^W(w)) \subseteq set_{W^2}(trans_s^W(w)). \quad (17)$$

Nach Induktionsvoraussetzung folgt

$$\bigwedge_{i=1}^n (v_i, v_{i+1} \dots v_n w) \in set_{W^2}(trans_{e_i}^W(v_i \dots v_n w)). \quad (18)$$

Mit $w_i =_{def} v_{i+1} \dots v_n w$ für alle $0 \leq i \leq n$ ist (18) äquivalent zu:

$$\bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1})). \quad (19)$$

Aus (14), (17) und (19) folgt schließlich

$$(v, w) = (v_1 \dots v_n, w_n) \in set_{W^2}(try_r^W(w_0)) = set_{W^2}(try_r^W(w)) \subseteq set_{W^2}(trans_s^W(w)). \quad \square$$

7 LR-Compiler

LR-Compiler lesen ein Wort w wie LL-Compiler von links nach rechts, konstruieren dabei jedoch eine **Rechtsreduktion** von w , d.h. die Umkehrung einer **Rechtsableitung**. Da dies dem schrittweisen, *mit den Blättern beginnenden* Aufbau eines Syntaxbaums entspricht, werden LR-Compiler auch **bottom-up-Compiler** genannt.

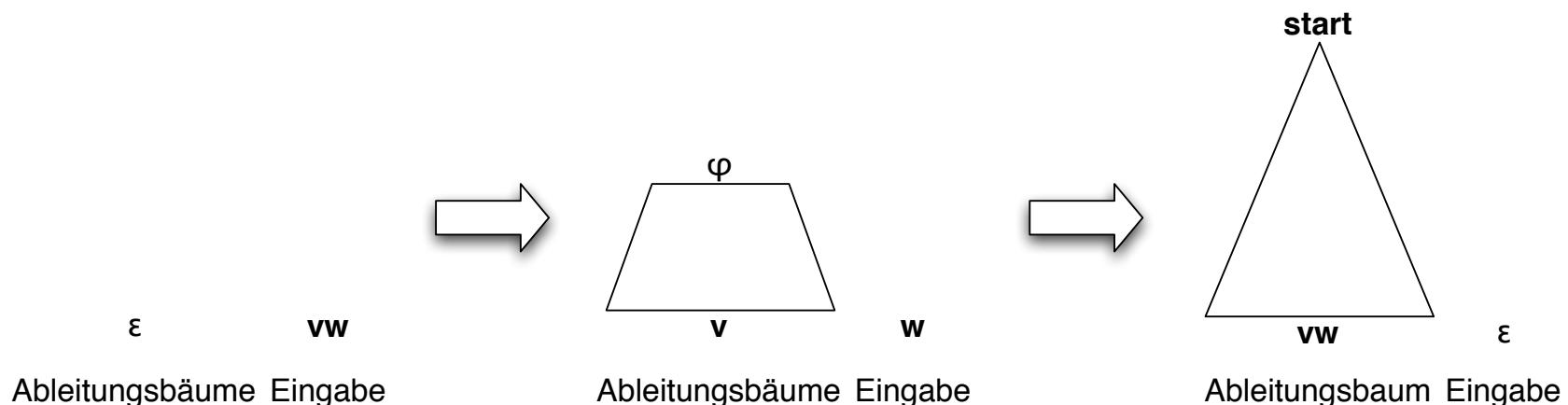
Im Gegensatz zum LL-Compiler ist die entsprechende Übersetzungsfunktion eines LR-Compilers iterativ. Die Umkehrung der Ableitungsrichtung (Reduktion statt Ableitung) macht es möglich, die Compilation durch einen Automaten, also eine iterativ definierte Funktion, zu steuern.

Während LL-Compiler keine linksrekursiven CFGs verarbeiten können, sind LR-Compiler auf LR(k)-Grammatiken beschränkt (s.u.).

Sei $G = (S, BS, R)$ eine CFG und $X = \bigcup BS$.

Wir setzen jetzt voraus, dass S das Symbol **start** enthält und dieses nicht auf der rechten Seite einer Regel von R auftritt.

Ablauf der LR-Übersetzung auf Ableitungsbäumen



G heißt **LR(k)-Grammatik**, falls das Vorauslesen von k noch nicht verarbeiteten Eingabesymbolen genügt, um zu entscheiden, ob ein weiteres Zeichen gelesen oder eine Reduktion durchgeführt werden muss, und, wenn ja, welche. Außerdem müssen die Basismengen von BS paarweise disjunkt sein.

Beispiel 7.1 Die CFG $G = (\{S, A\}, \{*, b\}, \{S \rightarrow A, A \rightarrow A * A, A \rightarrow b\})$ ist für kein k eine LR(k)-Grammatik.

Nach der Reduktion des Präfixes $b * b$ des Eingabewortes $b * b * b$ zu $A * A$ liegt ein **shift-reduce-Konflikt** vor. Soll man $A * A$ mit der Regel $A \rightarrow A * A$ zu A reduzieren oder erst die Resteingabe $*b$ lesen und dann b mit $A \rightarrow b$ zu A reduzieren? Die Resteingabe bestimmt keine eindeutige Antwort. □

first- und follow-Wortmengen

Sei $k > 0$, $\alpha \in (S \cup BS)^*$ und $s \in S$.

$$\begin{aligned} \textcolor{red}{first}_k(\alpha) &= \{\beta \in BS^k \mid \exists \gamma \in BS^* : \alpha \xrightarrow{*} \beta \gamma\} \cup \{\beta \in BS^{<k} \mid \alpha \xrightarrow{*} \beta\} \\ \textcolor{red}{follow}_k(s) &= \{\beta \in BS^k \mid \exists \alpha, \gamma \in BS^* : \text{start} \xrightarrow{*} \alpha s \beta \gamma\} \cup \\ &\quad \{\beta \in BS^{<k} \mid \exists \alpha \in BS^* : \text{start} \xrightarrow{*} \alpha s \beta\} \\ \textcolor{brown}{first}(\alpha) &= \textcolor{brown}{first}_1(\alpha) \\ \textcolor{brown}{follow}(s) &= \textcolor{brown}{follow}_1(s) \end{aligned}$$

Simultane induktive Definition der first-Mengen

$$\textcolor{brown}{first}(\epsilon) = \{\epsilon\} \tag{1}$$

$$B \in BS \wedge \alpha \in (S \cup BS)^* \Rightarrow \textcolor{brown}{first}(B\alpha) = \{B\} \tag{2}$$

$$(s \rightarrow \alpha) \in R \wedge \beta \in (S \cup BS)^* \wedge B \in \textcolor{brown}{first}(\alpha\beta) \Rightarrow B \in \textcolor{brown}{first}(s\beta) \tag{3}$$

Sei G eine CFG. Wir definieren den LR(1)-Compiler für G in mehreren Schritten. Zunächst definieren wir einen **Erkenner für die Sprache** $L(G)_{start}$. Während Erkenner für reguläre Sprachen reguläre Ausdrücke auf Funktionen des Typs $X^* \rightarrow 2$ abbilden (siehe Kapitel 2 und 3), bildet der folgende Erkenner (*recognizer*) Wörter über den Sorten und Basismengen von G auf jene Funktionen ab:

$$recog_1 : (S \cup BS)^* \rightarrow 2^{X^*}$$

formuliert. Sei $\gamma, \alpha, \varphi \in (S \cup BS)^*$, $x \in X$ und $w \in X^*$.

$$\begin{aligned} recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha x)(w) \quad \text{falls } \exists s \rightarrow \alpha\beta \in R, v \in (S \cup BS)^* : \\ &\qquad\qquad\qquad start \xrightarrow{*} \gamma sv, \beta \neq \epsilon, x \in first(\beta) \cap Z \\ &\qquad\qquad\qquad \text{shift (read)} \\ recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha B)(w) \quad \text{falls } \exists s \rightarrow \alpha\beta \in R, B \in BS, v \in (S \cup BS)^* : \\ &\qquad\qquad\qquad start \xrightarrow{*} \gamma sv, \beta \neq \epsilon, \textcolor{red}{x} \in \textcolor{red}{B} \in first(\beta) \\ &\qquad\qquad\qquad \text{shift (read)} \end{aligned}$$

$$\begin{aligned}
recog_1(\gamma\alpha)(w) &= recog_1(\gamma s)(w) \text{ falls } \exists s \rightarrow \alpha \in R, B \in BS, v \in (S \cup BS)^* : \\
&\quad start \xrightarrow{*} \gamma sv, s \neq start, \\
&\quad w = \epsilon \in follow(s) \vee \\
&\quad head(w) \in follow(s) \cap Z \vee \\
&\quad head(w) \in B \in follow(s)
\end{aligned}$$

reduce

$$recog_1(\varphi)(\epsilon) = 1 \text{ falls } start \rightarrow \varphi \in R$$

accept

$$recog_1(\varphi)(w) = 0 \text{ sonst}$$

reject

$recog_1$ ist genau dann wohldefiniert, wenn G eine LR(1)-Grammatik ist.

$recog_1$ ist korrekt bzgl. $L(G)_{start}$, d.h. für alle $w \in X^*$ gilt:

$$recog_1(\epsilon) = \chi(L(G)_{start}). \tag{1}$$

Die Funktion $recog_1$ ist *iterativ* definiert. Um (1) benötigt man daher eine *Invariante*. Sie lautet wie folgt: Für alle $w \in X^*$,

$$recog_1(\varphi)(w) = 1 \iff start \xrightarrow{*} \varphi w. \tag{2}$$

(2) zeigt man durch Induktion über $|w|$. Aus (2) folgt sofort (1).

Im zweiten Schritt wird das erste Argument von $recog_1$ durch den Inhalt eines Zustandskellers ersetzt. Die zugrundeliegende endliche (!) Zustandsmenge $\textcolor{red}{Q}$ ist die Bildmenge der Funktion

$$state : (S \cup BS)^* \rightarrow \mathcal{P}(\text{Quad})$$

$$\varphi \mapsto \{(\textcolor{red}{s}, \alpha, \beta, u) \in \text{Quad} \mid \exists \gamma, v : start \xrightarrow{*}_G \gamma sv, \varphi = \gamma\alpha, \\ u = v = \epsilon \vee u = head(v) \in BS\},$$

wobei Quad die Menge aller Quadrupel $(s, \alpha, \beta, u) \in S \times ((S \cup BS)^*)^2 \times (BS \cup 1)$ mit $s \rightarrow \alpha\beta \in R$ ist.

Die Bedingungen in der Definition von $recog_1$ werden durch Abfragen der Form $(s, \alpha, \beta, x) \in state(\varphi)$ ersetzt:

$$\begin{aligned} recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha x)(w) \quad \text{falls } \exists (s, \alpha, \beta, \epsilon) \in state(\gamma\alpha) : \\ &\qquad \beta \neq \epsilon, x \in first(\beta) \cap Z \\ recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha B)(w) \quad \text{falls } \exists (s, \alpha, \beta, \epsilon) \in state(\gamma\alpha), B \in BS : \\ &\qquad \beta \neq \epsilon, x \in B \in first(\beta) \\ recog_1(\gamma\alpha)(w) &= recog_1(\gamma s)(w) \quad \text{falls } \exists (s, \alpha, \epsilon, u) \in state(\gamma\alpha) : s \neq start, \\ &\qquad w = \epsilon = u \vee head(w) = u \in Z(G) \vee \\ &\qquad head(w) \in u \in BS \end{aligned}$$

$$recog_1(\varphi)(\epsilon) = 1 \text{ falls } (start, \varphi, \epsilon, \epsilon) \in state(\varphi)$$

$$recog_1(\varphi)(w) = 0 \text{ sonst}$$

Der **LR-Automat für G** hat die Eingabemenge $S \cup BS$, die Zustandsmenge

$$Q_G = \{state(\varphi) \mid \varphi \in (S \cup BS)^*\}$$

und die (**goto-Tabelle** genannte) partielle (!) Transitionsfunktion

$$\begin{aligned} \delta_G : Q_G &\longrightarrow Q_G^{S \cup BS} \\ state(\varphi) &\mapsto \lambda s. state(\varphi s) \end{aligned}$$

Simultane induktive Definition von Q_G und δ_G

$$start \rightarrow \alpha \in R \Rightarrow (start, \epsilon, \alpha, \epsilon) \in q_0 \quad (3)$$

$$(s, \alpha, s'\beta, u) \in q \wedge s' \rightarrow \gamma \in R \wedge v \in first(\beta u) \Rightarrow (s', \epsilon, \gamma, v) \in q \quad (4)$$

$$(s, \alpha, s'\beta, u) \in q, s' \in S \cup BS \Rightarrow (s, \alpha s', \beta, u) \in \delta_G(q)(s') \quad (5)$$

In der Definition von $recog_1$ wird jedes Wort von $(S \cup BS)^*$ durch eine gleichlange Liste von Zuständen des LR-Automaten ersetzt:

$$h : (S \cup BS)^* \rightarrow Q_G^*$$

$$\epsilon \mapsto q_0 =_{def} state(\epsilon)$$

$$(s_1, \dots, s_n) \mapsto (state(s_1 \dots s_n), state(s_1 \dots s_{n-1}), \dots, state(s_1), state(\epsilon))$$

Im Folgenden benutzen wir gelegentlich die Haskell-Funktionen `(:)` und `(++)`, um auf Wörtern zu operieren (siehe Kapitel 9).

Aus $recog_1$ wird $recog_2 : Q_G^* \rightarrow 2^{X^*}$:

Für alle $q, q_i \in Q_G$, $qs \in Q_G^*$, $x \in X$ und $w \in X^*$,

$$recog_2(q : qs)(xw) = recog_2(\delta_G(q)(x) : q : qs)(w)$$

falls $\exists (s, \alpha, \beta, \epsilon) \in q : \beta \neq \epsilon$, $x \in first(\beta) \cap Z(G)$

$$recog_2(q : qs)(xw) = recog_2(\delta_G(q)(B) : q : qs)(w)$$

falls $\exists (s, \alpha, \beta, \epsilon) \in q$, $B \in BS$:

$$\beta \neq \epsilon, x \in B \in first(\beta)$$

$$\begin{aligned}
recog_2(q_1 : \cdots : q_{|\alpha|} : q : qs)(w) &= recog_2(\delta_G(q)(s) : q : qs)(w) \\
&\quad \text{falls } \exists (s, \alpha, \epsilon, u) \in q_1 : s \neq start, \\
&\quad w = \epsilon = u \vee head(w) = u \in Z(G) \vee \\
&\quad head(w) \in u \in BS \\
recog_2(q : qs)(\epsilon) &= 1 \quad \text{falls } \exists \varphi : (start, \varphi, \epsilon, \epsilon) \in q \\
recog_2(qs)(w) &= 0 \quad \text{sonst}
\end{aligned}$$

Offenbar gilt $recog_1 = recog_2 \circ h$, also insbesondere

$$recog_2(q_0) = \chi(L(G)_{start}) \tag{6}$$

wegen (1).

Um die Suche nach bestimmten Elementen eines Zustands in den Iterationsschritten von $recog_2$ zu vermeiden, verwenden wir die – **Aktionstabellen für G** genannte – Funktion

$$act_G : Q_G \times (BS \cup 1) \rightarrow R \cup \{shift, error\},$$

die wie folgt definiert ist:

Für alle $u \in BS \cup 1$,

$$act_G(q, u) = \begin{cases} shift & \text{falls } \exists (s, \alpha, \beta, \epsilon) \in q : \beta \neq \epsilon, u \in first(\beta), \\ s \rightarrow \alpha & \text{falls } \exists (s, \alpha, \epsilon, u) \in q, \\ error & \text{sonst.} \end{cases}$$

Unter Verwendung von δ_G und act_G erhält man eine kompakte Definition von $recog_2$:

Für alle $q, q_i \in Q_G$, $qs \in Q_G^*$, $x \in X$ und $w \in X^*$,

$$\begin{aligned} recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(x) : q : qs)(w) \\ &\quad \text{falls } x \in Z(G), act_G(q, x) = shift \\ recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(B) : q : qs)(w) \\ &\quad \text{falls } \exists B \in BS : x \in B, act_G(q, B) = shift \\ recog_2(q_1 : \cdots : q_{|\alpha|} : q : qs)(w) &= recog_2(\delta_G(q)(s) : q : qs)(w) \\ &\quad \text{falls } \exists u \in BS \cup 1 : act_G(q_1, u) = s \rightarrow \alpha, \\ &\quad w = \epsilon = u \vee head(w) = u \in Z(G) \vee \\ &\quad head(w) \in u \in BS \end{aligned}$$

$$\begin{aligned}
 recog_2(q : qs)(\epsilon) &= 1 \quad \text{falls } act_G(q, \epsilon) = start \rightarrow \alpha \\
 recog_2(qs)(w) &= 0 \quad \text{sonst}
 \end{aligned}$$

Beispiel 7.2 SAB2

$$\begin{aligned}
 G &= (\{S, A, B\}, \{c, d, *\}, R) \\
 R &= \{S \rightarrow A, \quad A \rightarrow A * B, \quad A \rightarrow B, \quad B \rightarrow c, \quad B \rightarrow d\}
 \end{aligned}$$

LR-Automat für G

$$\begin{aligned}
 q_0 &= \{(S, \epsilon, A, \epsilon), (A, \epsilon, A * B, \epsilon), (A, \epsilon, B, \epsilon), (A, \epsilon, A * B, *), (A, \epsilon, B, *), \\
 &\quad (B, \epsilon, c, \epsilon), (B, \epsilon, d, \epsilon), (B, \epsilon, c, *), (B, \epsilon, d, *)\} \\
 q_1 &= \delta_G(q_0)(A) = \{(S, A, \epsilon, \epsilon), (A, A, *B, \epsilon), (A, A, *B, *)\} \\
 q_2 &= \delta_G(q_0)(B) = \{(A, B, \epsilon, \epsilon), (A, B, \epsilon, *)\} \\
 q_3 &= \delta_G(q_0)(c) = \{(B, c, \epsilon, \epsilon), (B, c, \epsilon, *)\} = \delta_G(q_5)(c) \\
 q_4 &= \delta_G(q_0)(d) = \{(B, d, \epsilon, \epsilon), (B, d, \epsilon, *)\} = \delta_G(q_5)(d) \\
 q_5 &= \delta_G(q_1)(*) = \{(A, A*, B, \epsilon), (A, A*, B, *), \\
 &\quad (B, \epsilon, c, \epsilon), (B, \epsilon, d, \epsilon), (B, \epsilon, c, *), (B, \epsilon, d, *)\} \\
 q_6 &= \delta_G(q_5)(B) = \{(A, A * B, \epsilon, \epsilon), (A, A * B, \epsilon, *)\}
 \end{aligned}$$

goto-Tabelle für G

	A	B	c	d	*
q_0	q_1	q_2	q_3	q_4	
q_1					q_5
q_5		q_6	q_3	q_4	

Aktionstabellen für G

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
c	<i>shift</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>shift</i>	<i>error</i>
d	<i>shift</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>error</i>	<i>shift</i>	<i>error</i>
$*$	<i>error</i>	<i>shift</i>	$A \rightarrow B$	$B \rightarrow c$	$B \rightarrow d$	<i>error</i>	$A \rightarrow A * B$
ϵ	<i>error</i>	$S \rightarrow A$	$A \rightarrow B$	$B \rightarrow c$	$B \rightarrow d$	<i>error</i>	$A \rightarrow A * B$

Ein Erkennerauf

$$\begin{aligned}
& \text{recog}_2(q_0)(c * d) \\
= & \text{recog}_2(q_3q_0)(*d) \quad \text{wegen } \text{act}_G(q_0, c) = \text{shift} \quad \text{und } \delta_G(q_0)(c) = q_3 \\
= & \text{recog}_2(q_2q_0)(*d) \quad \text{wegen } \text{act}_G(q_3, *) = B \rightarrow c \quad \text{und } \delta_G(q_0)(B) = q_2 \\
= & \text{recog}_2(q_1q_0)(*d) \quad \text{wegen } \text{act}_G(q_2, *) = A \rightarrow B \quad \text{und } \delta_G(q_0)(A) = q_1 \\
= & \text{recog}_2(q_5q_1q_0)(d) \quad \text{wegen } \text{act}_G(q_1, *) = \text{shift} \quad \text{und } \delta_G(q_1)(*) = q_5 \\
= & \text{recog}_2(q_4q_5q_1q_0)(\epsilon) \quad \text{wegen } \text{act}_G(q_5, d) = \text{shift} \quad \text{und } \delta_G(q_5)(d) = q_4 \\
= & \text{recog}_2(q_6q_5q_1q_0)(\epsilon) \quad \text{wegen } \text{act}_G(q_4, \epsilon) = B \rightarrow d \quad \text{und } \delta_G(q_5)(B) = q_6 \\
= & \text{recog}_2(q_1q_0)(\epsilon) \quad \text{wegen } \text{act}_G(q_6, \epsilon) = A \rightarrow A * B \quad \text{und } \delta_G(q_0)(A) = q_1 \\
= & 1 \quad \text{wegen } \text{act}_G(q_1, \epsilon) = S \rightarrow A \quad \square
\end{aligned}$$

Für den dritten und letzten Schritt zum LR(1)-Compiler benötigen wir die abstrakte Syntax $\Sigma(G)$ von G (siehe Kapitel 4).

Sei $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$ -Algebra. Aus recog_2 wird der generische Compiler

$$\text{compile}_G^{\mathcal{A}} : Q_G^* \times A^* \rightarrow M(A)^{X^*}.$$

Er startet mit $q_0 = \text{state}(\epsilon)$ im ansonsten leeren Zustandskeller. Die Bedeutung der Liste von A -Elementen im Definitionsbereich von compile_G^A entnimmt man am besten der folgenden Formalisierung der Korrektheit von compile_G :

- Für alle $w \in X^*$ und $t \in T_{\Sigma(G), \text{start}}$,

$$\text{compile}_G^A(q_0, \epsilon)(w) = \eta(\text{fold}^A(t)) \Rightarrow \text{fold}^{\text{Word}(G)}(t) = w, \quad (1)$$

$$\text{compile}_G^A(q_0, \epsilon)(w) = \text{error}(w) \Rightarrow w \notin L(G)_{\text{start}}. \quad (2)$$

Die Invariante von compile_G , aus der (1) folgt, ist komplizierter als die des Erkenners recog_1 (s.o.):

- Für alle $\alpha = w_0 s_1 w_1 \dots s_n w_n$ mit $w_i \in Z(G)^*$ und $s_i \in S \cup BS$, $qs \in Q_G^*$, $a_i \in A$, $w \in X^*$, und $t \in T_{\Sigma(G)}(\{x_1, \dots, x_n\})$,

$$\text{compile}_G^A(\text{state}(\alpha) : qs, [a_1, \dots, a_n])(w) = \eta(g_A^*(t)) \Rightarrow g_W^*(t) = \alpha w, \quad (3)$$

wobei $g_A : \{x_1, \dots, x_n\} \rightarrow A$ und $g_W : \{x_1, \dots, x_n\} \rightarrow \text{Word}(G)$ x_i auf a_i bzw. s_i abbilden.

Aus (3) ergibt sich die folgende iterative Definition von compile_G^A :

Für alle $n \in \mathbb{N}$, $q, q_i \in Q_G$, $qs \in Q_G^*$, $a_i \in A$, $as \in A^*$, $x \in X$ und $w \in X^*$,

$$\text{compile}_G^A(q : qs, \textcolor{red}{as})(xw) = \text{compile}_G^A(\delta_G(q)(x) : q : qs, \textcolor{red}{as})(w)$$

falls $x \in Z(G)$, $\text{act}_G(q, x) = \text{shift}$

$$\text{compile}_G^A(q : qs, \textcolor{red}{as})(xw) = \text{compile}_G^A(\delta_G(q)(B) : q : qs, \textcolor{red}{as} ++ [\textcolor{red}{x}])(w)$$

falls $\exists B \in BS : x \in B$, $\text{act}_G(q, B) = \text{shift}$

$$\text{compile}_G^A(q_1 : \dots : q_n : q : qs, \textcolor{red}{as} ++ [a_{i_1}, \dots, a_{i_k}])(w) =$$

$$\text{compile}_G^A(\delta_G(q)(s) : q : qs, \textcolor{red}{as} ++ [f_r^A(a_{i_1}, \dots, a_{i_k})])(w)$$

falls $\exists u \in BS \cup 1 : \text{act}_G(q_1, u) = r = (s \rightarrow e_1 \dots e_n)$,

$$\{i_1, \dots, i_k\} = \{1 \leq i \leq n \mid e_i \in S \cup BS\},$$

$$w = \epsilon = u \vee \text{head}(w) = u \in Z(G) \vee \text{head}(w) \in u \in E$$

$$\text{compile}_G^A(q : qs, [a_{i_1}, \dots, a_{i_k}])(\epsilon) = \eta(f_r^A(a_{i_1}, \dots, a_{i_k}))$$

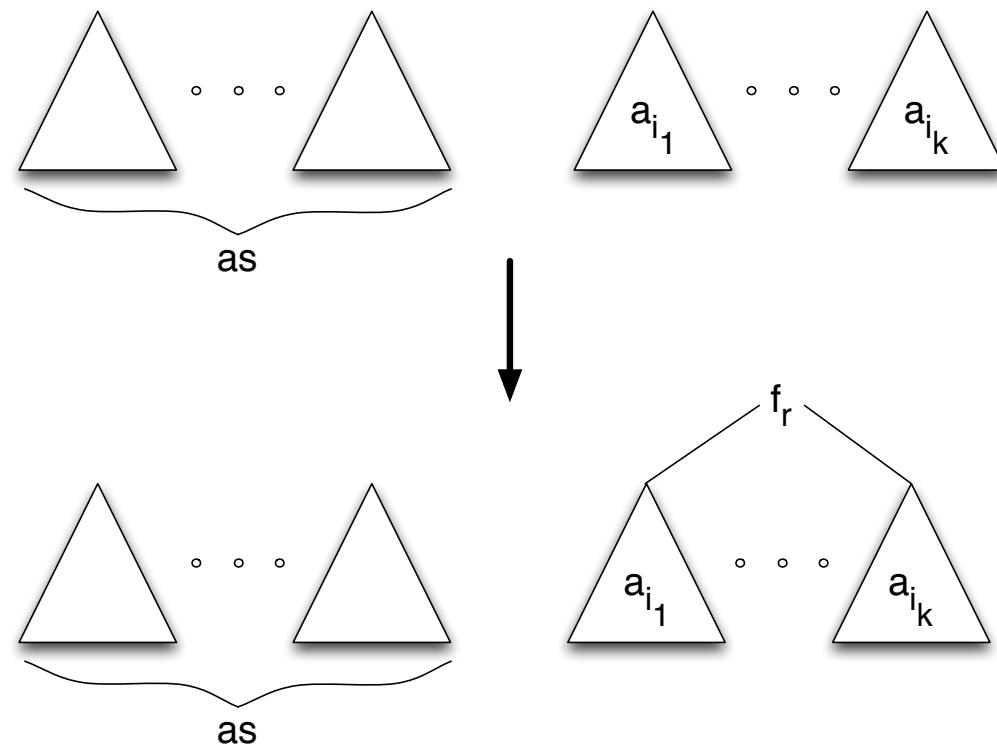
falls $\text{act}_G(q, \epsilon) = r = (\text{start} \rightarrow e_1 \dots e_n)$,

$$\{i_1, \dots, i_k\} = \{1 \leq i \leq n \mid e_i \in S \cup BS\}$$

$$\text{compile}_G^A(qs, \textcolor{red}{as})(w) = \text{errormsg}(w)$$

sonst

Im Fall $A = T_{\Sigma(G)}$ bestehen die Listen as , $[a_{i_1}, \dots, a_{i_k}]$ und $[f_r^{\mathcal{A}}(a_{i_1}, \dots, a_{i_k})]$ aus Syntaxbäumen. Die zweite Gleichung zeigt ihren schrittweisen Aufbau:



Beispiel 7.3 SAB2

Hier ist der um die schrittweise Konstruktion des zugehörigen Syntaxbaums erweiterte Erkennerlauf von Beispiel 7.2:

$$\begin{aligned} & \text{compile}_G^{T_\Sigma(G)}([0], [])(c * d) \\ = & \text{compile}_G^{T_\Sigma(G)}([3, 0], [])(\ast d) \\ = & \text{compile}_G^{T_\Sigma(G)}([2, 0], [f_{B \rightarrow c}])(\ast d) \\ = & \text{compile}_G^{T_\Sigma(G)}([1, 0], [f_{A \rightarrow B}(f_{B \rightarrow c})])(\ast d) \\ = & \text{compile}_G^{T_\Sigma(G)}([5, 1, 0], [f_{A \rightarrow B}(f_{B \rightarrow c})])(d) \\ = & \text{compile}_G^{T_\Sigma(G)}([4, 5, 1, 0], [f_{A \rightarrow B}(f_{B \rightarrow c})])(\epsilon) \\ = & \text{compile}_G^{T_\Sigma(G)}([6, 5, 1, 0], [f_{A \rightarrow B}(f_{B \rightarrow c}), f_{B \rightarrow d}])(\epsilon) \\ = & \text{compile}_G^{T_\Sigma(G)}([1, 0], [f_{A \rightarrow A \ast B}(f_{A \rightarrow B}(f_{B \rightarrow c}), f_{B \rightarrow d})])(\epsilon) \\ = & \eta(f_{S \rightarrow A}(f_{A \rightarrow A \ast B}(f_{A \rightarrow B}(f_{B \rightarrow c}), f_{B \rightarrow d}))) \end{aligned}$$

In der **graphischen Darstellung** dieses Parserlaufs sind die Knoten der Syntaxbäume nicht mit Konstruktoren, sondern mit den Regeln markiert, aus denen sie hervorgehen, wobei der Pfeil zwischen der linken und rechten Seite einer Regel durch den Unterstrich $_$ ersetzt wurde.



Beispiel 7.4 Auf den folgenden Seiten steht die vom C-Compiler-Generator **yacc** (“yet another compiler-compiler”) aus der folgenden LR(1)-Grammatik G erzeugte Zustandsmenge – deren Elemente hier nur aus den ersten drei Komponenten der o.g. Quadrupel bestehen – zusammen mit der auf die einzelnen Zustände verteilten Einträge der goto- und Aktionstabelle:

```
$accept -> prog $end
prog      -> statems exp .
statems   -> statems assign ; | ε
assign    -> ID : = exp
exp       -> exp + exp | exp * exp | (exp) | NUMBER | ID
```

So besteht z.B. der Zustand 0 aus den beiden Tripeln $(\$accept, \epsilon, \$end)$ und $(statems, \epsilon, \epsilon)$. Die beim Lesen eines Punktes im Zustand 0 ausgeführte Aktion ist eine Reduktion mit Regel 3, also mit $statems \rightarrow \epsilon$. Der Folgezustand von 0 ist bei Eingabe von *prog* bzw. *statems* der Zustand 1 bzw. 2.

[Link](#) zur graphischen Darstellung des Parserlaufs auf dem Eingabewort

```
ID : = NUM ; ID : = ID + NUM ; NUM * ID + ID .
```

Den Sonderzeichen entsprechen in der input-Spalte des Parserlaufs und den u.a. Tabellen passende Wortsymbole. Die Knoten der Syntaxbäume sind wie in Beispiel 7.3 mit Regeln markiert.

```

state 0
$accept : _prog $end
statems : _ (3)

. reduce 3

prog goto 1
statems goto 2

state 1
$accept : prog $end

$end accept
. error



---


state 2
prog : statems_exp .
statems : statems_assign ;

ID shift 7
NUMBER shift 6
( shift 5
. error

exp goto 3
assign goto 4

state 3
prog : statems_exp_.
exp : exp_+ exp
exp : exp_* exp

+ shift 9
* shift 10
. shift 8
. error

state 4
statems : statems_assign_;

; shift 11
. error

state 5
exp : (_exp )

ID shift 13
NUMBER shift 6
( shift 5
. error

exp goto 12

```

```

state 6
exp : NUMBER_ (8)

. reduce 8

state 7
assign : ID_ = exp
exp : ID_ (9)

: shift 14
. reduce 9

state 8
prog : statems_exp . (1)

. reduce 1

state 9
exp : exp +_exp
ID shift 13
NUMBER shift 6
( shift 5
. error

exp goto 15

state 10
exp : exp *_exp

ID shift 13
NUMBER shift 6
( shift 5
. error

exp goto 16

state 11
statems : statems_assign ; (2)

. reduce 2

state 12
exp : exp_+ exp
exp : exp_* exp
exp : ( exp_ )

+ shift 9
* shift 10
) shift 17
. error

state 13
exp : ID_ (9)

. reduce 9

```

```

state 14
    assign : ID := exp
        = shift 18
        . error

state 15
    exp : exp_+ exp
    exp : exp + exp_      (5)
    exp : exp_* exp

        * shift 10
        . reduce 5

state 16
    exp : exp_+ exp
    exp : exp_* exp
    exp : exp * exp_      (6)

        . reduce 6

state 17
    exp : ( exp )_      (7)
        . reduce 7

state 18
    assign : ID : = _exp
        ID shift 13
        NUMBER shift 6
        ( shift 5
        . error

        exp goto 19

state 19
    assign : ID : = exp_      (4)
    exp : exp_+ exp
    exp : exp_* exp

        + shift 9
        * shift 10
        . reduce 4

```

12/300 terminals, 4/300 nonterminals
 10/600 grammar rules, 20/1000 states
 0 shift/reduce, 0 reduce/reduce conflicts
 reported

Im Folgenden sind die goto- bzw. Aktionstabelle noch einmal getrennt wiedergegeben. **op** (*open*) und **cl** (*close*) bezeichnen eine öffnende bzw. schließende Klammer. **end** steht für das leere Wort ϵ . Wieder wird auf den Pfeil zwischen linker und rechter Seite einer Regel verzichtet.

	prog	stmts	exp	assign	op	ID	NUM	dot	add	mul	semi	col	cl	eq
0	1	2												
1														
2			3	4	5	7	6							
3								8	9	10				
4											11			
5			12		5	13	6							
7												14		
6														
8														
9			15		5	13	6							
10			16		5	13	6							
11														
12								9	10			17		
13														
14													18	
15										10				
16														
17														
18			19		5	13	6							
19									9	10				

	end	semi	col	dot	op	cl
0	stmts	stmts	stmts	stmts	stmts	stmts
1	S prog	error	error	error	error	error
2	error	error	error	error	shift	error
3	error	error	error	shift	error	error
4	error	shift	error	error	error	error
5	error	error	error	error	shift	error
7	exp ID	exp ID	shift	exp ID	exp ID	exp ID
6	exp NUM					
8	prog stmts exp dot					
9	error	error	error	error	shift	error
10	error	error	error	error	shift	error
11	stmts stmts assign semi					
12	error	error	error	error	error	shift
13	exp ID					
14	error	error	error	error	error	error
15	exp exp add exp					
16	exp exp mul exp					
17	exp op exp cl					
18	error	error	error	error	shift	error
19	assign ID col eq exp					

	ID	NUM	add	mul	eq
0	stmts	stmts	stmts	stmts	stmts
1	error	error	error	error	error
2	shift	shift	error	error	error
3	error	error	shift	shift	error
4	error	error	error	error	error
5	shift	shift	error	error	error
7	exp ID				
6	exp NUM				
8	prog stmts exp dot				
9	shift	shift	error	error	error
10	shift	shift	error	error	error
11	stmts stmts assign semi				
12	error	error	shift	shift	error
13	exp ID				
14	error	error	error	error	shift
15	exp exp add exp	exp exp add exp	exp exp add exp	shift	exp exp add exp
16	exp exp mul exp				
17	exp op exp cl				
18	shift	shift	error	error	error
19	assign ID col eq exp	assign ID col eq exp	shift	shift	assign ID col eq exp

Im Folgenden werden die Regeln von G um C-Code erweitert, der eine $\Sigma(G)$ -Algebra implementiert, die dem Zustandsmodell von JavaLight (siehe 4.9) ähnelt.

```
typedef struct {char name[10]; int leng} ident;
typedef struct {char name[10]; int val} decl;
decl env[30];
int i,j,c;
}

%start prog
%union {int num; ident id;}
%token <id> ID
%token <num> NUMBER
%type <num> exp
%left '+'
%left '*'
%%
```

```

prog      :  statements exp '.' {printf("The final result
                                is %d\n", $2);}

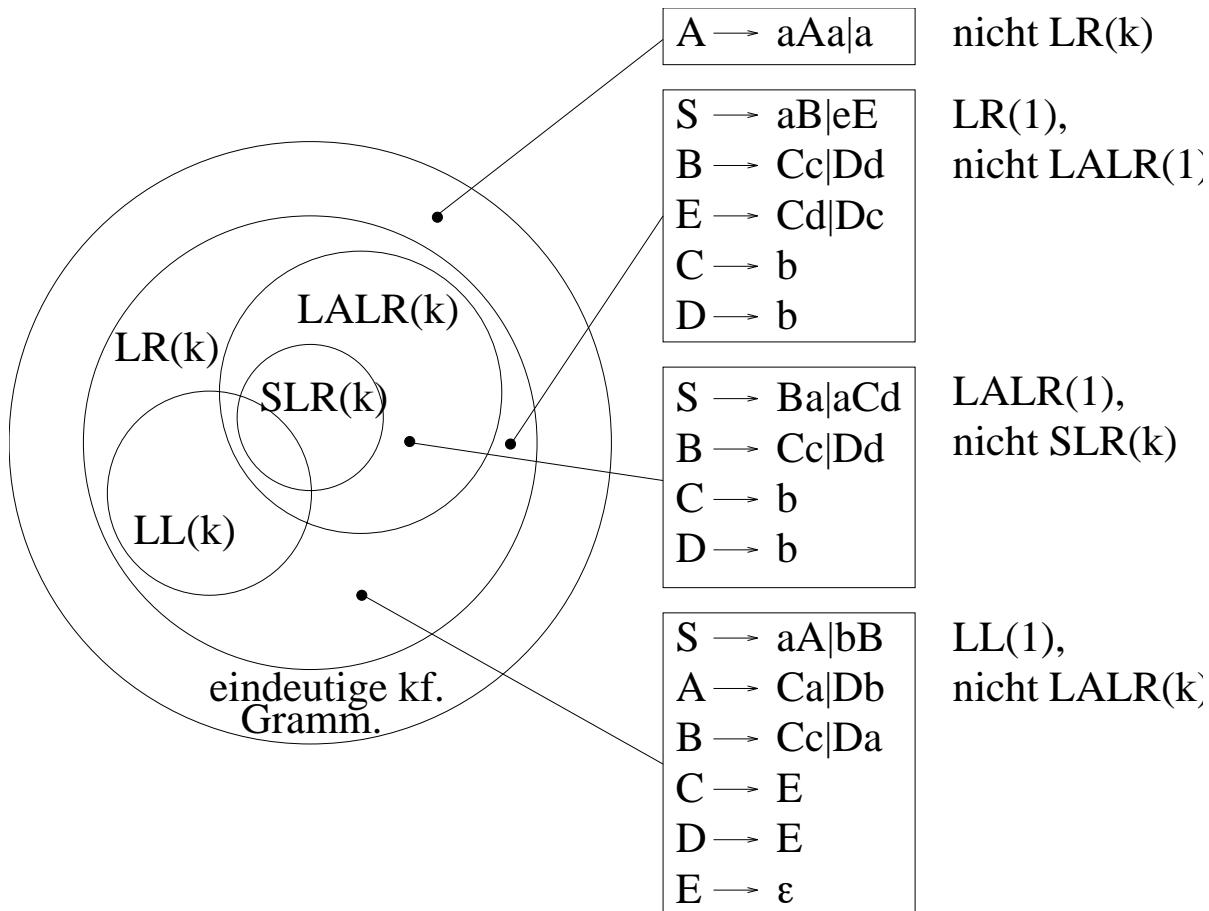
statements :  statements assign ';' {
|           {}
;
assign    :  ID ':' '=' exp  {i = 0;
                            while (i < c)
                            {for (j = 0; j <= $1.leng; j++)
                             if ($1.name[j] != env[i].name[j]
                                 break;
                             if (j > $1.leng) break;
                             i++;}
                            for (j = 0; j <= $1.leng; j++)
                            env[i].name[j] = $1.name[j];
                            env[i].val = $4;
                            if (i >= c) c++;}

exp       :  exp '+' exp    {$$ = $1 + $3;}
|  exp '*' exp    {$$ = $1 * $3;}
|  '(' exp ')'   {$$ = $2;}
|  NUMBER          {$$ = $1;}
|  ID               {i = 0;
                     while (i < c)
                     {for (j = 0; j <= $1.leng; j++)
                      if ($1.name[j] != env[i].name[j]
                          break;
                      if (j > $1.leng) break;
                      i++;}
                     if (i >= c)
                     printf("%s is not defined\n",
                            $1.name);
                     else $$ = env[i].val;}}

;

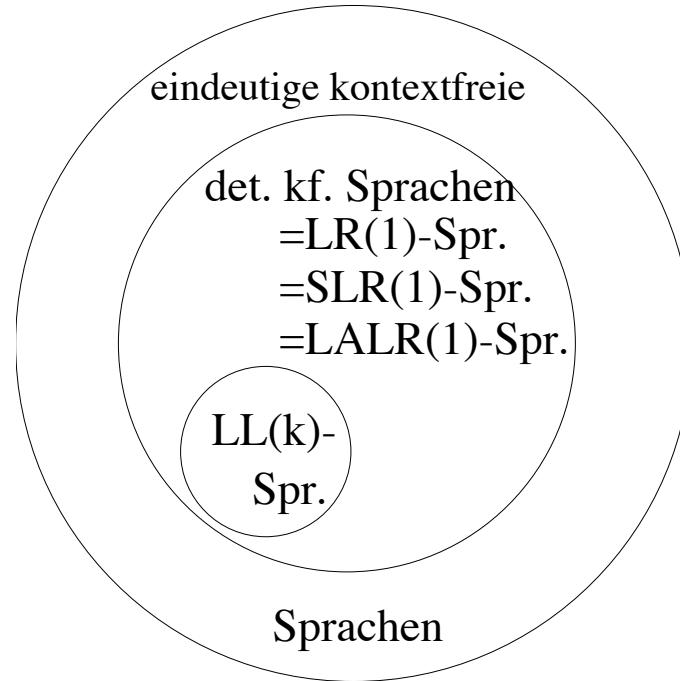
```





Hierarchie der CFG-Klassen

Zur Definition von LL(k)-Grammatiken verweisen wir auf die einschlägige Literatur. Kurz gesagt, sind das diejenigen nicht-linksrekursiven CFGs, deren LL-Parser ohne Backtracking auskommen.



Hierarchie der entsprechenden Sprachklassen

Für jeden Grammatiktyp T bedeutet die Formulierung

L ist eine T -Sprache

lediglich, dass eine T -Grammatik *existiert*, die L erzeugt.

Während der **LL-Compiler** von Kapitel 6 – nach Beseitigung von Linksrekursion – jede kontextfreie Grammatik verarbeitet, selbst dann, wenn sie mehrdeutig ist, zeigt die obige Grafik, dass die Forderung, dabei ohne Backtracking auszukommen, die Klasse der kompilierbaren Sprachen erheblich einschränkt: Unter dieser Bedingung ist die bottom-up-Übersetzung offenbar mächtiger als die top-down-Compilation.

Umgekehrt wäre es den Versuch wert (z.B. in Form einer Bachelorarbeit), in Anlehnung an den obigen Compiler für LR(1)-Grammatiken einen bottom-up-Compiler *mit* Backtracking zu entwickeln. Da die Determinismusforderung wegfiel, bräuchten wir keinen Lookahead beim Verarbeiten der Eingabe, womit die Zustände generell nur aus Tripeln bestünden – wie im Beispiel 7.4.

8 Haskell: Typen und Funktionen

Die Menge 1 wird in Haskell **unit-Typ** genannt und mit $()$ bezeichnet. $()$ steht auch für das einzige Element ϵ von 1.

Die Menge 2 entspricht dem Haskell-Typ *Bool*. Ihre beiden Elemente 0 und 1 werden mit *False* bzw. *True* bezeichnet.

Alle Typen von Haskell sind aus Standardtypen wie *Bool*, *Int* oder *Float*, **Typvariablen** sowie **Typkonstruktoren** wie \times (Produkt), $+$ (Summe oder disjunkte Vereinigung) oder \rightarrow (Funktionsmenge) aufgebaut.

Jeder Typ bezeichnet eine Menge, jeder Typkonstruktor eine Funktion auf Mengen von Mengen.

Typnamen beginnen stets mit einem Großbuchstaben, Typvariablen mit einem Kleinbuchstaben. Typvariablen stehen für Mengen, **Individuenvariablen** für Elemente einer Menge. Beide müssen mit einem Kleinbuchstaben beginnen.

Das (kartesische) Produkt $A_1 \times \dots \times A_n$ von Mengen A_1, \dots, A_n wird in Haskell wie seine Elemente, die n -Tupel (a_1, \dots, a_n) , mit runden Klammern und Kommas notiert:

$$(A_1, \dots, A_n).$$

Die Summe oder disjunkte Vereinigung *SumAn* von Mengen A_1, \dots, A_n wird durch einen (Haskell-)Datentyp implementiert (siehe Kapitel 10).

Funktionen werden benannt und durch rekursive Gleichungen definiert (s.u.) oder als λ -Abstraktion $\lambda p.e$ (Haskell-Notation: `\p -> e`) dargestellt, wobei p ein **Muster** für die möglichen Argumente (Parameter) von $\lambda p.e$ ist, z.B. $p = (x, (y, z))$. Muster bestehen aus Individuenvariablen und (Individuen-)Konstruktoren. Jede Variable kommt in einem Muster höchstens einmal vor. (Die λ -Abstraktionen von Kapitel 2 haben nur Variablen als Muster.) Der “Rumpf” e der λ -Abstraktion $\lambda p.e$ ist ein aus beliebigen Funktionen und Variablen zusammengesetzter Ausdruck.

Ein Ausdruck der Form $f(e)$ heißt **Funktionsapplikation** (auch: Funktionsanwendung oder -aufruf). Ist f eine λ -Abstraktion, dann nennt man $f(e)$ eine **λ -Applikation**. Die λ -Applikation $(\lambda p.e)(e')$ ist auswertbar, wenn der Ausdruck e' das Muster p **matcht**. Beim Matching werden die Variablen von p in e durch Teilausdrücke von e' ersetzt.

Die Auswertung einer Applikation von e wie z.B. $(\lambda(x, y).x * y + 5 + x)(7, 8)$, besteht in der Bildung der **Instanz** $7 * 8 + 5 + 7$ der λ -Abstraktion $\lambda(x, y).x * y + 5 + x$ und deren anschließender Auswertung:

$$(\lambda(x, y).x * y + 5 + x)(7, 8) \rightsquigarrow 7 * 8 + 5 + 7 \rightsquigarrow 56 + 5 + 7 \rightsquigarrow 68$$

Die ghci-Befehle

:type \(\lambda(x,y).x * y + 5 + x\) bzw. :type $(\lambda(x,y).x * y + 5 + x)(7,8)$

liefern die Typen

Num a => (a,a) -> a bzw. Num a => a

der λ -Abstraktion $\lambda(x,y).x * y + 5 + x$ bzw. λ -Applikation $(\lambda(x,y).x * y + 5 + x)(7,8)$.

Backslash (\) und Pfeil (->) sind offenbar die Haskell-Notationen des Symbols λ bzw. des Punktes einer λ -Abstraktion. Num ist die Typklasse für numerische Typen. Allgemein werden Typklassen in Kapitel 5 behandelt.

[Link zur schrittweisen Auswertung der \$\lambda\$ -Applikation \$\(\lambda\(x,y\).x * y + 5 + x\)\(7,8\)\$](#)

Der Redex, d.i. der Teilausdruck, der im jeweils nächsten Schritt ersetzt wird, ist rot gefärbt.
Das Redukt, d.i. der Teilausdruck, durch den der Redex ersetzt wird, ist grün gefärbt.

[Link zur schrittweisen Auswertung der \$\lambda\$ -Applikation](#)

$(\lambda x.x(true,4,77) + x(false,4,77))(\lambda(x,y,z).if\ x\ then\ y + 5\ else\ z * 6)$

Eine geschachelte λ -Abstraktion $\lambda p_1.\lambda p_2.\dots.\lambda p_n.e$ kann durch $\lambda p_1\ p_2\ \dots\ p_n.e$ abgekürzt werden. Sie hat einen Typ der Form $t_1 \rightarrow (t_2 \rightarrow \dots (t_n \rightarrow t) \dots)$, wobei auf diese Klammerung verzichtet werden kann, weil Haskell Funktionstypen automatisch rechtsassoziativ klammert.

Anstelle der Angabe eines λ -Ausdrucks kann eine Funktion benannt und dann mit f mit Hilfe von Gleichungen definiert werden:

$f = \lambda p \rightarrow e$ *ist äquivalent zu* $f\ p = e$ (*applikative Definition*)

Funktionen, die andere Funktionen als Argumente oder Werte haben, heißen **Funktionen höherer Ordnung**. Der Typkonstruktor \rightarrow ist **rechtsassoziativ**. Also ist die Deklaration

$(+) :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$ *ist äquivalent zu* $(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

Die Applikation einer Funktion ist **linksassoziativ**. Also ist

$((+) 5) 6$ *ist äquivalent zu* $(+) 5 6$

$(+)$ ist zwar als Präfixfunktion deklariert, kann aber auch infix verwendet werden. Dann entfallen die runden Klammern um den Infixoperator:

$5 + 6$ *ist äquivalent zu* $(+) 5 6$

Das Gleiche gilt für jede Funktion f eines Typs der Form $A \rightarrow B \rightarrow C$. Besteht f aus Sonderzeichen, dann wird f bei der Präfixverwendung in runde Klammern gesetzt, bei der Infixverwendung nicht. Beginnt f mit einem Kleinbuchstaben und enthält f keine Sonderzeichen, dann wird f bei der Infixverwendung in Akzentzeichen gesetzt, bei der Präfixverwendung nicht:

```
mod :: Int -> Int -> Int  
mod 9 5    ist äquivalent zu    9 `mod` 5
```

Die Infixnotation wird auch verwendet, um die in f enthaltenen **Sektionen** (Teilfunktionen) des Typs $A \rightarrow C$ bzw. $B \rightarrow C$ zu benennen. Z.B. sind die folgenden Sektionen Funktionen des Typs `Int -> Int`, während `(+)` und `mod` den Typ `Int -> Int -> Int` haben.

$(+5)$	ist äquivalent zu	$(+) 5$	ist äquivalent zu	$\lambda x \rightarrow x + 5$
$(9 `mod`)$	ist äquivalent zu	$mod 9$	ist äquivalent zu	$\lambda x \rightarrow 9 `mod` x$

Eine Sektion wird stets in runde Klammern eingeschlossen. Die Klammern gehören zum Namen der jeweiligen Funktion.

Der **Applikationsoperator**

$$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f \$ a = f a$$

führt die Anwendung einer gegebenen Funktion auf ein gegebenes Argument durch. Sie ist rechtsassoziativ und hat unter allen Operationen die **niedrigste Priorität**. Daher kann durch Benutzung von \$ manch schließende Klammer vermieden werden:

$$f1 \$ f2 \$ \dots \$ fn a \rightsquigarrow f1 (f2 (\dots (fn a) \dots))$$

Demgegenüber ist der **Kompositionsooperator**

$$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$$
$$(g . f) a = g (f a)$$

zwar auch rechtsassoziativ, hat aber – nach den Präfixoperationen – die **höchste Priorität**.

$$(f1 . f2 . \dots . fn) a \rightsquigarrow f1 (f2 (\dots (fn a) \dots))$$

U.a. benutzt man den Kompositionsoperator, um in einer applikativen Definition Argumentvariablen einzusparen. So sind die folgenden drei Definitionen einer Funktion $f : a \rightarrow b \rightarrow c$ äquivalent zueinander:

$$f \ a \ b = g \ (h \ a) \ b$$

$$f \ a = g \ (h \ a)$$

$$f = g \ . \ h$$

Welchen Typ hat hier g bzw. h ?

Auch die folgenden drei Definitionen einer Funktion $f : a \rightarrow b$ sind äquivalent zueinander:

$$f \ a = g \ . \ h \ a$$

$$f \ a = (g.) \ (h \ a)$$

$$f = (g.) \ . \ h$$

Welchen Typ hat hier g bzw. h ?

Monomorphe und polymorphe Typen

Ein Typ ohne Typvariablen heißt **monomorph**.

Ein Typ mit Typvariablen wie z.B. `a -> Int -> b` heißt **polymorph**.

Eine Funktion heißt mono- bzw. polymorph, wenn ihr Typ mono- bzw. polymorph ist.

Ein Typ u heißt **Instanz eines Typs t** , wenn u durch Ersetzung der Typvariablen von t aus t entsteht.

Typvariablen stehen für Mengen, **Individuenvariablen** stehen für *Elemente* einer Menge. Sie müssen ebenfalls mit einem Kleinbuchstaben beginnen.

Eine besondere Individuenvariable ist der Unterstrich `_` (auch **Wildcard** genannt). Er darf nur auf der linken Seite einer Funktionsdefinition vorkommen, was zur Folge hat, dass er für einen Teil eines Argumentmusters der gerade definierten Funktion steht, der ihre Werte an Stellen, die das Muster matchen, nicht beeinflusst (siehe Beispiele im nächsten Kapitel).

Die oben definierten Funktionen `($)` und `(.)` sind demnach polymorph.

Weitere polymorphe Funktionen höherer Ordnung

`id :: a -> a`

Identität

`id a = a`

`id = \a -> a`

äquivalente Definition

`const :: a -> b -> a`

konstante Funktion

`const a b = a`

`const a = \b -> a`

äquivalente Definitionen

`const = \a -> \b -> a`

`const = \a b -> a`

`update :: Eq a => (a -> b) -> a -> b -> a -> b`

Funktionsupdate

`update f a b a' = if a == a' then b else f a'`

(nicht im Prelude)

`update f :: a -> b -> a -> b`

äquivalente Schreibweisen
`update(f)`

`update f a :: b -> a -> b`

`update(f)(a)`
`(update f) a`

<code>update f a b :: a -> b</code>	<code>update(f)(a)(b)</code> $((\text{update } f) \text{ a}) \text{ b}$ $\text{-- } f[b/a]$
<code>update f a b a' :: b</code>	<code>update(f)(a)(b)(a')</code> $((\text{update } f) \text{ a}) \text{ b} \text{ a}'$
<code>flip :: (a -> b -> c) -> b -> a -> c</code> <code>flip f b a = f a b</code>	<i>Vertauschung der Argumente</i>
<code>flip mod 11</code> ist äquivalent zu <code>(`mod` 11)</code> (s.o.)	
<code>curry :: ((a,b) -> c) -> a -> b -> c</code> <code>curry f a b = f (a,b)</code>	<i>Kaskadierung (Currying)</i>
<code>uncurry :: (a -> b -> c) -> (a,b) -> c</code> <code>uncurry f (a,b) = f a b</code>	<i>Dekaskadierung</i>

9 Haskell: Listen

Sei A eine Menge. Die Menge A^* (siehe Kapitel 2) wird in Haskell mit eckigen Klammern notiert, also durch $[A]$ — ebenso wie ihre Elemente: Für $(a_1, \dots, a_n) \in A^*$ schreibt man $[a_1, \dots, a_n]$.

Eine n -elementige Liste kann extensional oder als funktionaler Ausdruck dargestellt werden:

$$[a_1, \dots, a_n] \quad \text{ist äquivalent zu} \quad a_1 : (a_2 : (\dots (a_n : [])) \dots))$$

Die Konstante $[]$ (leere Liste) vom Typ $[A]$ und die Funktion $(:)$ (Anfügen eines Elementes von A ans linke Ende einer Liste) vom Typ $A \rightarrow [A] \rightarrow [A]$ heißen **Konstruktoren**, weil sie nicht wie andere Funktionen Werte berechnen, sondern dazu dienen, die Elemente eines Typs aufzubauen. Auch werden sie benötigt, um die Zugehörigkeit eines Elementes eines Summentyps zu einem bestimmten Summanden wiederzugeben (siehe Kapitel 4).

Die Klammern in $a_1 : (a_2 : (\dots (a_n : []) \dots))$ können weggelassen werden, weil der Typ von $(:)$ keine andere Klammerung zulässt.

Die durch mehrere Gleichungen ausgedrückten Fallunterscheidungen bei den folgenden Definitionen von Funktionen auf Listen ergeben sich aus verschiedenen **Mustern** der Funktionsargumente bzw. Bedingungen an die Argumente (Boolesche Ausdrücke hinter $|$).

Seien x, y, s Individuenvariablen. s ist ein Muster für alle Listen, $[]$ das Muster für die leere Liste, $[x]$ ein Muster für alle einelementigen Listen, $x : s$ ein Muster für alle nichtleeren Listen, $x : y : s$ ein Muster für alle mindestens zweielementigen Listen, usw.

```
length :: [a] -> Int  
length (_ : s) = length s + 1  
length _ = 0
```

length [3, 44, -5, 222, 29] \rightsquigarrow 5

Link zur schrittweisen Auswertung von length [3, 44, -5, 222, 29]

```
head :: [a] -> a  
head (a : _) = a
```

head [3, 2, 8, 4] \rightsquigarrow 3

```
tail :: [a] -> [a]  
tail (_ : s) = s
```

tail [3, 2, 8, 4] \rightsquigarrow [2, 8, 4]

```
(++) :: [a] -> [a] -> [a]  
(a : s) ++ s' = a : (s ++ s')  
_ ++ s = s
```

[3, 2, 4] ++ [8, 4, 5] \rightsquigarrow [3, 2, 4, 8, 4, 5]

```
(!!) :: [a] -> Int -> a           [3,2,4] !! 1 ~> 2
(a:_)! !0                      = a
(_ : s) !! n | n > 0 = s !! (n-1)
```

```
init :: [a] -> [a]                  init [3,2,8,4] ~> [3,2,8]
init []    = []
init (a:s) = a: init s
```

```
last :: [a] -> a                   last [3,2,8,4] ~> 4
last [a]   = a
last (_ : s) = last s
```

```
take :: Int -> [a] -> [a]        take 3 [3,2,4,8,4,5] ~> [3,2,4]
take 0 _                     = []
take n (a:s) | n > 0 = a: take (n-1) s
take _ []                    = []
```

```
drop :: Int -> [a] -> [a]        drop 4 [3,2,4,8,4,5] ~> [4,5]
drop 0 s                      = s
drop n (_ : s) | n > 0 = drop (n-1) s
drop _ []                      = []
```

Funktionslifting auf Listen

```
map :: (a -> b) -> [a] -> [b]
```

```
map f (a:s) = f a : map f s
```

```
map _ _ _ = []
```

```
map (+1) [3,2,8,4] ~> [4,3,9,5]
```

```
map ($ 7) [(+1),(+2),(*5)] ~> [8,9,35]
```

```
map ($ a) [f1,f2,...,fn] ~> [f1 a,f2 a,...,fn a]
```

Link zur schrittweisen Auswertung von `map(+3) [2..9]`

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
zipWith f (a:s) (b:s') = f a b : zipWith f s s'
```

```
zipWith _ _ _ = []
```

```
zipWith (+) [3,2,8,4] [8,9,35] ~> [11,11,43]
```

```
zip :: [a] -> [b] -> [(a,b)]
```

```
zip = zipWith (,)
```

```
zip [3,2,8,4] [8,9,35] ~> [(3,8),(2,9),(8,35)]
```

Strings sind Listen von Zeichen

Strings werden als Listen von Zeichen betrachtet, d.h. die Typen `String` und `[Char]` sind identisch.

Z.B. haben die folgenden Booleschen Ausdrücke den Wert *True*:

```
"" == []
"H" == ['H']
"Hello" == ['H', 'a', 'l', 'l', 'o']
```

Also sind alle Listenfunktionen auf Strings anwendbar.

`words :: String -> [String]` und `unwords :: [String] -> String` zerlegen bzw. konkatenieren Strings, wobei Leerzeichen, Zeilenumbrüche ('`\n`') und Tabulatoren ('`\t`') als Trennsymbole fungieren.

`unwords` fügt Leerzeichen zwischen die zu konkatenierenden Strings.

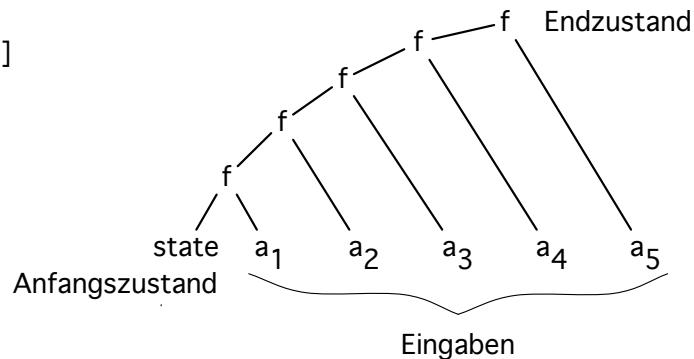
`lines :: String -> [String]` und `unlines :: [String] -> String` zerlegen bzw. konkatenieren Strings, wobei nur Zeilenumbrüche als Trennsymbole fungieren.

`unlines` fügt '`\n`' zwischen die zu konkatenierenden Strings.

Listenfaltung

Faltung einer Liste von links her

`foldl f state [a1,a2,a3,a4,a5]`



`foldl :: (a -> b -> a) -> a -> [b] -> a`

`foldl f a (b:s) = foldl f (f a b) s`

`foldl _ a _ = a`

a ist Zustand, b ist Eingabe

f ist Zustandsüberführung

Link zur schrittweisen Auswertung von `foldl(+)(0)[1..5]`

`foldl1 :: (a -> a -> a) -> [a] -> a`

`foldl1 f (a:s) = foldl f a s`

`sum = foldl (+) 0`

`product = foldl (*) 1`

`and = foldl (&&) True`

`or = foldl (||) False`

`minimum = foldl1 min`

`maximum = foldl1 max`

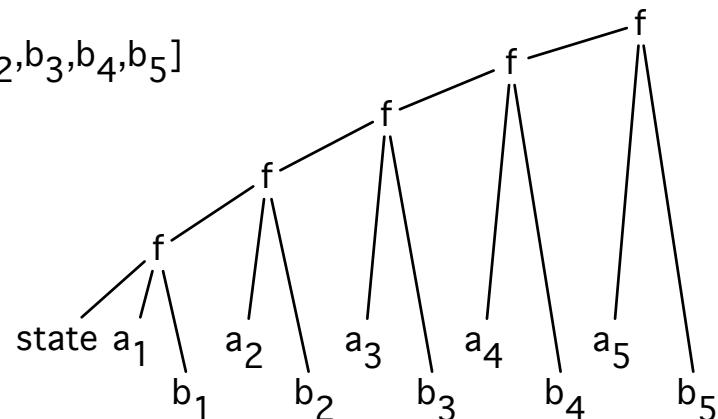
```
concat  = foldl (++) []
```

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

```
concatMap f = concat . map f
```

Parallele Faltung zweier Listen von links her

```
fold2 f state [a1,a2,a3,a4,a5] [b1,b2,b3,b4,b5]
```



```
fold2 :: (a -> b -> c -> a) -> a -> [b] -> [c] -> a
```

```
fold2 f a (b:s) (c:s') = fold2 f (f a b c) s s'
```

```
fold2 _ a _ _ = a
```

```

listsToFun :: Eq a => b -> [a] -> [b] -> a -> b
listsToFun = fold2 update . const

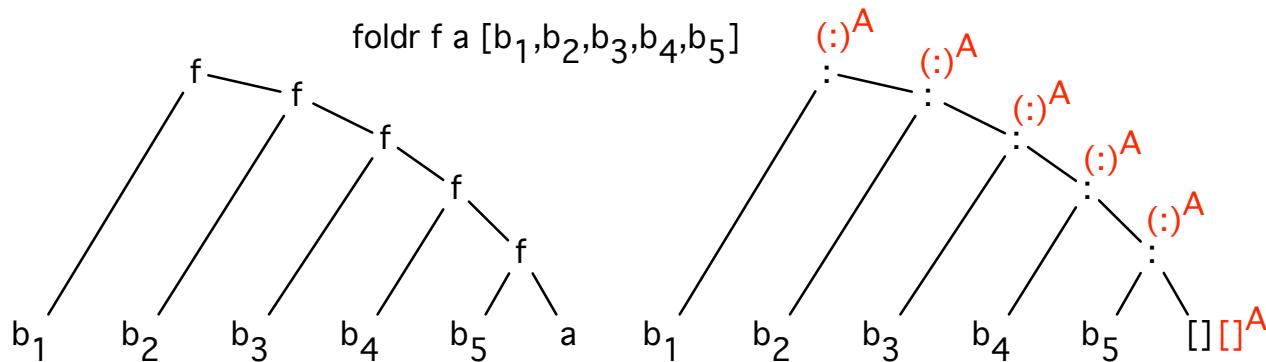
```

Beginnend mit `const b`, erzeugt `listsToFun b` schrittweise aus einer Argumentliste `as` und einer Werteliste `bs` die entsprechende Funktion:

$$\text{listsToFun } b \text{ as } bs \text{ a} = \begin{cases} bs!!i & \text{falls } i = \max\{k \mid as!!k = a, k < \text{length}(bs)\}, \\ b & \text{sonst.} \end{cases}$$

Faltung einer Liste von rechts her

entspricht der Auswertung ihrer Konstruktordarstellung in einer Algebra A , die die Konstruktoren `(:) :: b -> [b] -> [b]` und `[] :: [b]` als Funktion $(:)^A$ bzw. Konstante $[]^A$ interpretiert (siehe Kapitel 2).



```
foldr :: (b -> a -> a) -> a -> [b] -> a  
foldr f a (b:s) = f b $ foldr f a s  
foldr _ a _      = a
```

Der Applikationsoperator \$ als Parameter von Listenfunktionen

foldr (\$) a [f1,f2,f3,f4]	~>	f1 \$ f2 \$ f3 \$ f4 a
foldl (flip (\$)) a [f4,f3,f2,f1]	~>	f1 \$ f2 \$ f3 \$ f4 a
map f [a1,a2,a3,a4]	~>	[f a1,f a2,f a3,f a4]
map (\$a) [f1,f2,f3,f4]	~>	[f1 a,f2 a,f3 a,f4 a]
zipWith (\$) [f1,f2,f3,f4] [a1,a2,a3,a4]	~>	[f1 a1,f2 a2,f3 a3,f4 a4]

Listenlogik

```
any :: (a -> Bool) -> [a] -> Bool      any (>4) [3,2,8,4] ~> True
any f = or . map f

all :: (a -> Bool) -> [a] -> Bool      all (>2) [3,2,8,4] ~> False
all f = and . map f

elem :: Eq a => a -> [a] -> Bool      elem 2 [3,2,8,4] ~> True
elem a = any (a ==)

notElem :: Eq a => a -> [a] -> Bool    notElem 9 [3,2,8,4] ~> True
notElem a = all (a /=)

filter :: (a -> Bool) -> [a] -> [a]   filter (<8) [3,2,8,4] ~> [3,2,4]
filter f (a:s) = if f a then a:filter f s else filter f s
filter f _     = []
```

Jeder Aufruf von `map`, `zipWith`, `filter` oder einer Komposition dieser Funktionen entspricht einer **Listenkomprehension**:

$$\begin{aligned}\text{map } f \ s &= [f \ a \mid a \leftarrow s] \\ \text{zipWith } f \ s \ s' &= [f \ a \ b \mid (a,b) \leftarrow \text{zip } s \ s'] \\ \text{filter } f \ s &= [a \mid a \leftarrow s, \ f \ a]\end{aligned}$$

`zip(s)(s')` ist nicht das kartesische Produkt von `s` und `s'`. Dieses entspricht der Komprehension `[(a,b) | a <- s, b <- s']`.

Allgemeines Schema von Listenkomprehensionen:

$$[e(x_1, \dots, x_n) \mid x_1 \leftarrow s_1, \dots, x_n \leftarrow s_n, \text{be}(x_1, \dots, x_n)] :: [a]$$

- x_1, \dots, x_n sind Variablen,
- s_1, \dots, s_n sind Listen,
- $e(x_1, \dots, x_n)$ ist ein Ausdruck des Typs a ,
- $x_i \leftarrow s_i$ heißt **Generator** und steht für $x_i \in s_i$,
- $\text{be}(x_1, \dots, x_n)$ heißt **Guard** und ist ein Boolescher Ausdruck.

Jede endlichstellige Relation lässt sich als Listenkomprehension implementieren, z.B. die Menge aller Tripel $(a, b, c) \in A_1 \times A_2 \times A_3$, die ein Prädikat $p : A_1 \times A_2 \times A_3 \rightarrow \text{Bool}$ erfüllen, durch `[(a,b,c) | a <- a1, b <- a2, c <- a3, p(a,b,c)]`.

10 Haskell: Datentypen und Typklassen

Zunächst das allgemeine Schema einer Datentypdefinition:

```
data DT x_1 ... x_m = C_1 typ_11 ... typ_1n_1 | ... |
                      C_k typ_k1 ... typ_kn_k
```

$typ_{11}, \dots, typ_{kn_k}$ sind beliebige Typen, die außer x_1, \dots, x_m keine Typvariablen enthalten.
 DT heißt **rekursiv**, wenn DT in mindestens einem dieser Typen vorkommt.

Die durch DT implementierte Menge besteht aus allen Ausdrücken der Form

$C_i e_1 \dots e_{n_i},$

wobei $1 \leq i \leq n$ und für alle $1 \leq j \leq n_i$ e_j ein Element des Typs typ_{ij} ist. Als Funktion hat C_i den Typ

$typ_{i1} \rightarrow \dots \rightarrow typ_{in_i} \rightarrow DT a_1 \dots a_m.$

Alle mit einem Großbuchstaben beginnenden Funktionssymbole und alle mit einem Doppelpunkt beginnenden Folgen von Sonderzeichen werden vom Haskell-Compiler als Konstruktoren eines Datentyps aufgefasst und müssen deshalb irgendwo im Programm in einer Datentypdefinition vorkommen.

Die Unterschiede in der Schreibweise von Konstruktoren bei Infix- bzw. Präfixverwendung sind dieselben wie bei anderen Funktionen.

Beispiel 10.1 Der Haskell-Standardtyp für Listen

```
data [a] = [] | a : [a]
```

Für jede Menge A besteht $[A]$ aus

- allen endlichen Ausdrücken $a_1 : \dots : a_n : []$ mit $a_1, \dots, a_n \in A$ und
- allen unendlichen Ausdrücken $a_1 : a_2 : a_3 : \dots$ mit $\{a_i \mid i \in \mathbb{N}\} \subseteq A$.

$[A]$ ist die größte Lösung der Gleichung

$$M = \{[]\} \cup \{a : s \mid a \in A, s \in \textcolor{red}{M}\} \quad (1)$$

in der Mengenvariablen M und damit $[A]$ eine Haskell-Implementierung der Menge $\mathcal{CT}_{List(A)}$ aller $List(A)$ -Grundterme.

Als Elemente von $[Int]$ lauten die $List(\mathbb{Z})$ -Grundterme $blink$ und $blink'$ wie folgt:

```
blink,blink' :: [Int]
blink = 0:blink'
blink' = 1:blink
```

Die endlichen Ausdrücke von $[A]$ bilden die kleinste Lösung von (1) und damit eine Haskell-Implementierung der Menge $T_{List(A)}$ der $List(A)$ -Grundterme endlicher Tiefe (siehe 2.8). \square

Beispiel 10.2 (siehe 2.8) $T_{Reg(BL)}$ ist isomorph zur kleinsten und $CT_{Reg(BL)}$ zur größten Lösung der Gleichung

$$\begin{aligned} \textcolor{red}{M} = & \{par(t, u) \mid t, u \in \textcolor{red}{M}\} \cup \{seq(t, u) \mid t, u \in \textcolor{red}{M}\} \cup \{iter(t) \mid t \in \textcolor{red}{M}\} \cup \\ & \{base(B) \mid B \in BL\} \end{aligned} \quad (2)$$

in der Mengenvariablen $\textcolor{red}{M}$. Beide Mengen werden deshalb durch folgenden Datentyp implementiert:

```
data RegT bs = Par (RegT bs) (RegT bs) | Seq (RegT bs) (RegT bs) |
  Iter (RegT bs) | Base bs
```

\square

Summentypen

Die Summe oder disjunkte Vereinigung

$$A_1 + \cdots + A_n =_{def} \{(a, 1) \mid a \in A_1\} \cup \cdots \cup \{(a, n) \mid a \in A_n\}$$

von $n > 1$ Mengen A_1, \dots, A_n (siehe Kapitel 2) wird durch einen **nicht-rekursiven** Datentyp mit n Konstruktoren C_1, \dots, C_n implementiert, welche die Summanden voneinander

trennen, m.a.W.: es gibt eine Bijektion zwischen $A_1 + \cdots + A_n$ und der Menge

$$\{C_1(a) \mid a \in A_1\} \cup \cdots \cup \{C_n(a) \mid a \in A_n\}.$$

Z.B. implementiert der Standardtyp

```
data Maybe a = Nothing | Just a
```

die binäre Summe $1 + A$, die A um ein Element zur Darstellung “undefinierter” Werte erweitert. Damit lassen sich partielle Funktionen totalisieren wie die Funktion *lookup*, mit der der Graph einer Funktion von A nach B modifiziert wird (siehe Kapitel 2):

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup a ((a',b):r) = if a == a' then Just b else lookup a r
lookup _ _           = Nothing
```

Der Standardtyp *Either* implementiert beliebige binäre Summen:

```
data Either a b = Left a | Right b
```

Die Menge aller ganzen Zahlen kann als dreistellige Summe implementiert werden:

```
data INT = Zero | Plus PosNat | Minus PosNat
data PosNat = One | Succ PosNat
```

PosNat implementiert die Menge aller positiven natürlichen Zahlen (und ist rekursiv).

Datentypen mit Destruktoren

Um auf die Argumente eines Konstruktors zugreifen zu können, ordnet man ihnen Namen zu, die **field labels** genannt werden und Destruktoren im Sinne von Kapitel 2 wiedergeben. In obiger Definition von DT wird

$C_i \text{ typ_i1 } \dots \text{ typ_in_i}$

erweitert zu:

$C_i \{d_{i1} :: \text{typ_i1}, \dots, d_{in_i} :: \text{typ_in_i}\}$

Wie C_i , so ist auch d_{ij} eine Funktion. Als solche hat sie den Typ

$\text{DT a1 } \dots \text{ am } \rightarrow \text{typ_ij}$

Destruktoren sind invers zu Konstruktoren. Z.B. hat der folgende Ausdruck den Wert e_j :

$d_{ij} (C_i e_1 \dots e_n)$

Destruktoren nennt man in OO-Sprachen **Attribute**, wenn ihre Wertebereiche aus Standardtypen zusammengesetzt sind, bzw. **Methoden** (Zustandstransformationen), wenn der

Wertebereich mindestens einen Datentyp mit Destruktoren enthält. Außerdem schreibt man dort in der Regel **x.d_ij** anstelle von **d_ij(x)**.

Mit Destruktoren lautet das allgemeine Schema einer Datentypdefinition also wie folgt:

```
data DT a_1 ... a_m = C_1 {d_11 :: typ_11, ..., d_1n_1 :: typ_1n_1} |
                      ...
                      |
                      C_k {d_k1 :: typ_k1, ..., d_kn_k :: typ_kn_k}
```

Elemente von *DT* können mit oder ohne Destruktoren definiert werden:

obj = *C_i e_i1 ... e_in_i* *ist äquivalent zu*
obj = *C_i {d_i1 = e_i1, ..., d_in_i = e_in_i}*

Die Werte einzelner Destruktoren von *obj* können wie folgt verändert werden:

obj' = *obj {d_ij_1 = e_1', ..., d_ij_m = e_m'}*

obj' unterscheidet sich von *obj* dadurch, dass den Destruktoren *d_ij_1, ..., d_ij_m* neue Werte, nämlich *e_1', ..., e_m'* zugewiesen wurden.

Destruktoren dürfen nicht rekursiv definiert werden. Folglich deutet der Haskell-Compiler jedes Vorkommen von $attr_{ij}$ auf der rechten Seite einer Definitionsgleichung als eine vom gleichnamigen Destruktor verschiedene Funktion und sucht nach deren Definition.

Dies kann man nutzen, um d_{ij} doch rekursiv zu definieren, indem in der zweiten Definition von **obj** (s.o.) die Gleichung $d_ij = ej$ durch $d_ij = d_ij$ ersetzt und d_ij lokal definiert wird:

```
obj = C_i {d_ij1 = e_1, ..., d_ij = d_ij, ..., d_in_i = en_i}  
where d_ij ... = ...
```

Ein Konstruktor darf nicht zu mehreren Datentypen gehören.

Ein Destruktor darf nicht zu mehreren Konstruktoren unterschiedlicher Datentypen gehören.

Beispiel Listen mit Destruktoren

könnten in Haskell durch folgenden Datentyp definiert werden:

```
data ListD a = Nil | Cons {hd :: a, tl :: ListD a}
```

Man beachte, dass die Destrukturen

```
hd :: ListD a -> a und tl :: ListD a -> ListD a,
```

die den Kopf bzw. Rest einer nichtleeren Liste liefern, *partielle* Funktionen sind, weil sie nur auf mit dem Konstruktor $Cons$ gebildete Elemente von $ListD(A)$ anwendbar sind. \square

Die Destruktoren eines Datentyps DT sind immer dann totale Funktionen, wenn DT genau einen Konstruktor hat. Wie das folgende Beispiel nahelegt, ist das aus semantischer Sicht keine Einschränkung: Jeder Datentyp ohne Destruktoren ist isomorph zu einem Datentyp mit genau einem Konstruktor und einem Destruktor.

Beispiel 10.3 Listen mit totalen Destruktoren (siehe 2.8)

$DT_{coList(A)}$ ist isomorph zur größten Lösung der Gleichung

$$M = \{coListC(Nothing)\} \cup \{coListC(Just(a, s)) \mid a \in A, s \in M\} \quad (3)$$

in der Mengenvariablen M und wird deshalb durch folgenden Datentyp implementiert:

```
data ColistC a = ColistC {split :: Maybe (a, ColistC a)}
```

Wie man leicht sieht, ist die größte Lösung von (3) isomorph zur größten Lösung von (1), also zu $CT_{List(A)}$.

Die Einschränkung von $CT_{List(A)}$ auf unendliche Listen ist isomorph zu $DT_{Stream(A)}$ und wird daher durch folgenden Datentyp implementiert:

```
data StreamC a = (:<) {hd :: a, tl :: StreamC a}
```

Als Elemente von $StreamC(\mathbb{Z})$ lauten die $Stream(\mathbb{Z})$ -Coterme $blink$ und $blink'$ wie folgt:

```
blink,blink' :: StreamC Int
blink  = 0:<blink'
blink' = 1:<blink
```

□

Beispiel 10.4 (siehe 2.8) $DT_{DAut(X,Y)}$ ist isomorph zur größten Lösung der Gleichung

$$M = \{DA(f)(y) \mid f : X \rightarrow M, y \in Y\}$$

in der Mengenvariablen M und wird deshalb durch folgenden Datentyp implementiert:

```
data DAutC x y = DA {deltaC :: x -> DAutC x y, betaC :: y}
```

Als Elemente von $DAutC(\mathbb{Z})$ lauten die $Acc(\mathbb{Z})$ -Coterme $esum$ und $osum$ wie folgt:

```
esum,osum :: DAutC Int Bool
```

```
esum = DA {deltaC = \x -> if even x then esum else osum, betaC = True}
osum = DA {deltaC = \x -> if odd x then esum else osum, betaC = False}
```

□

Typklassen

stellen Bedingungen an die Instanzen einer Typvariablen. Die Bedingungen bestehen in der Existenz bestimmter Funktionen, z.B.

```
class Eq a where (==), (/=) :: a -> a -> Bool  
    (/=) = (not .) . (==)
```

Eine Instanz einer Typklasse besteht aus den Instanzen ihrer Typvariablen sowie Definitionen der in ihr deklarierten Funktionen.

```
instance Eq (Int,Bool) where (x,b) == (y,c) = x == y && b == c  
  
instance Eq a => Eq [a]  
    where s == s' = length s == length s' && and $ zipWith (==) s s'
```

Auch `(/=)` könnte hier definiert werden. Die Definition von `(/=)` in der Klasse `Eq` als Negation von `(==)` ist nur ein Default!

Der Typ jeder Funktion f einer Typklasse muss deren Typvariable a mindestens einmal enthalten. Sonst wäre f von (der Instanziierung von) a unabhängig!

10.5 Mengenoperationen auf Listen

```
insert :: Eq a => a -> [a] -> [a]
insert a s@(b:s') = if a == b then s else b:insert a s'
insert a _         = [a]
```

```
union :: Eq a => [a] -> [a] -> [a]                      Mengenvereinigung
union = foldl $ flip insert
```

```
unionMap :: Eq a => (a -> [b]) -> [a] -> [b]      concatMap für Mengen
unionMap f = foldl union [] . map f
```

```
subset :: Eq a => [a] -> [a] -> Bool                  Mengeninklusion
s `subset` s' = all (`elem` s') s
```

Unterklassen

Typklassen können wie Objektklassen andere Typklassen erben. Die jeweiligen Oberklassen werden vor dem Erben vor dem Pfeil \Rightarrow aufgelistet.

```
class Eq a => Ord a where (≤), (<), (≥), (>) :: a -> a -> Bool
```

```
max, min :: a -> a -> a
a < b    = a <= b && a /= b
a >= b   = b <= a
a > b    = b < a
max x y = if x >= y then x else y
min x y = if x <= y then x else y
```

Ausgeben

Vor der Ausgabe von Daten eines Typs T wird automatisch die T-Instanz der Funktion `show` aufgerufen, die zur Typklasse `Show a` gehört.

```
class Show a where
  show :: a -> String
  show x = shows x ""

  shows :: a -> String -> String
  shows = showsPrec 0

  showsPrec :: Int -> a -> String -> String
```

Das String-Argument von `showsPrec` wird an die Ausgabe des Argumentes vom Typ `a` angefügt.

Steht `deriving Show` am Ende der Definition eines Datentyps, dann werden dessen Elemente in der Darstellung ausgegeben, in der sie im Programmen vorkommen.

Für andere Ausgabeformate müssen entsprechende Instanzen von `show` oder `showsPrec` definiert werden.

Einlesen

Bei der Eingabe von Daten eines Typs `T` wird automatisch die `T`-Instanz der Funktion `read` aufgerufen, die zur Typklasse `Read a` gehört:

```
class Read a where
    readsPrec :: Int -> String -> [(a, String)]
    reads :: String -> [(a, String)]
    reads = readsPrec 0

    read :: String -> a
    read s = case [x | (x, t) <- reads s, ("", "") <- lex t] of
        [x] -> x
```

```
[]  -> error "PreludeText.read: no parse"  
_  -> error "PreludeText.read: ambiguous parse"
```

`reads s` liefert eine Liste von Paaren, bestehend aus dem als Element vom Typ `a` erkannten Präfix von `s` und der jeweiligen Resteingabe (= Suffix von `s`).

`lex :: String -> [(a, String)]` ist eine Standardfunktion, die ein evtl. aus mehreren Zeichen bestehendes Symbol erkennt, vom Eingabestring abspaltet und sowohl das Symbol als auch die Resteingabe ausgibt.

Der Generator `("","",") <- lex t` in der obigen Definition von `read s` bewirkt, dass nur die Paare `(x,t)` von `reads s` berücksichtigt werden, bei denen die Resteingabe `t` aus Leerzeichen, Zeilenumbrüchen und Tabulatoren besteht.

Steht `deriving Read` am Ende der Definition eines Datentyps, dann werden dessen Elemente in der Darstellung erkannt, in der sie in Programmen vorkommen.

Für andere Eingabeformate müssen entsprechende Instanzen von `readsPrec` definiert werden. Jede Instanz von `readsPrec` ist – im Sinne von Kapitel 5 – ein Parser in die Listenmonade. Da die Implementierung von Parsern und Compilern in Haskell erst in späteren Kapiteln behandelt wird, geben wir an dieser Stelle keine Beispiele für `readsPrec` an.

11 Algebren in Haskell

Sei $\Sigma = (S, F)$ eine Signatur, $obs(\Sigma) = \{x_1, \dots, x_k\}$, $S = \{s_1, \dots, s_m\}$ und $F = \{f_1 : e_1 \rightarrow e'_1, \dots, f_n : e_n \rightarrow e'_n\}$.

Jede Σ -Algebra entspricht einem Element des folgenden polymorphen Datentyps:

```
data Sigma x1 ... xk s1 ... sm = Sigma {f1 :: e1 -> e1', ...,
                                             fn :: en -> en'}
```

Die Sorten und Operationen von Σ werden durch Typvariablen bzw. Destruktoren wiedergegeben und durch die Trägermengen bzw. (kaskadierten) Funktionen der jeweiligen Algebra instanziert.

Um eine Signatur Σ in Haskell zu implementieren, genügt es daher, den Datentyp ihrer Algebren nach obigem Schema zu formulieren.

Der Datentyp $\textcolor{blue}{Sigma}(x_1) \dots (x_k)$ repräsentiert im Gegensatz zu den Datentypen der Beispiele 10.1-10.4, die Trägermengen einzelner Algebren implementieren, die Klasse aller Σ -Algebren.

11.1 Erweiterung von Term- und anderen Trägermengen zu Algebren (siehe 2.8 und 2.9)

```
data Nat nat = Nat {zero :: nat, succ :: nat -> nat}
```

natT implementiert T_{Nat} , *listT* implementiert $T_{List(X)}$:

```
natT :: Nat Int
```

```
natT = Nat {zero = 0, succ = (+1)}
```

```
data List x list = List {nil :: list, cons :: x -> list -> list}
```

listT implementiert $T_{List(X)}$ (siehe 10.1):

```
listT :: List x [x]
```

```
listT = List {nil = [], cons = (:)}
```

Die folgende Funktion implementiert die Faltung $fold^{alg} : T_{List(X)} \rightarrow alg$ von $List(X)$ -Grundterminen endlicher Tiefe in einer beliebigen $List(X)$ -Algebra alg :

```
foldList :: List x list -> [x] -> list
```

```
foldList alg [] = nil alg
```

```
foldList alg (x:s) = cons alg x $ foldList alg s
```

```
data Reg bs reg = Reg {par,seq :: reg -> reg -> reg,
                      iter :: reg -> reg,
                      base :: bs -> reg,}
```

regT implementiert $T_{Reg(BL)}$, *regWord* implementiert $Regword(BL)$ (siehe 2.9):

```
regT :: bs -> Reg bs (RegT bs)                                (siehe 10.2)
regT = Reg {par = Par, seq = Seq, iter = Iter, base = Base}
```

```
regWord :: Show bs => bs -> Reg bs (Int -> String)
regWord bs = Reg {par = \f g n -> enclose (n > 0) $ f 0 ++ '+':g 0,
                  seq = \f g n -> enclose (n > 1) $ f 1 ++ '.':g 1,
                  iter = \f n -> enclose (n > 2) $ f 2 ++ "*",
                  base = \b -> show . const b}
where enclose b w = if b then '(:w++)' else w
```

foldReg implementiert die Faltung $fold^{alg} : T_{Reg(BL)} \rightarrow alg$ von regulären Ausdrücken (= $Reg(BL)$ -Grundtermen) in einer beliebigen $Reg(BL)$ -Algebra alg :

```
foldReg :: Reg bs reg -> RegT bs -> reg
foldReg alg = \case Par t u -> par alg (f t) $ f u
                     Seq t u -> seq alg (f t) $ f u
```

```
Iter t -> iter alg $ f t  
Base b -> base alg b  
where f = foldReg alg
```

\case ist eine Abkürzung für \x -> case x of – falls das *LANGUAGE-Pragma* des verwendenden Moduls *LambdaCase* enthält.

```
instance Show bs => Show (RegT bs) where  
    showsPrec = flip $ foldReg regWord
```

```
data Stream x list = Stream {head :: list -> x, tail :: list -> list}
```

```
streamC :: Stream x (StreamC x)                                (siehe 10.3)  
streamC = Stream {head = hd, tail = tl}
```

Die *Stream(\mathbb{Z})*-Algebra *zo* (siehe 2.6) kann wie folgt implementiert werden:

```
data Z0 = Blink | Blink'  
zo :: Stream Int Z0  
zo = Stream {head = \case Blink -> 0; Blink' -> 1,  
            tail = \case Blink -> Blink'; Blink' -> Blink}
```

Die finale $Stream(X)$ -Algebra kann auch wie folgt implementiert werden:

```
streamFun :: Stream x (Int -> x)
streamFun = Stream {head = ($0), tail = \s -> s . (+1)}
```

Die folgenden zwei Funktionen implementieren die Entfaltungen von Elementen einer beliebigen $Stream(X)$ -Algebra alg zu Elementen von $streamC$ bzw. $streamFun$:

```
unfoldStream :: Stream x list -> list -> StreamC x
unfoldStream alg s = head alg s :< unfoldStream alg $ tail alg s

unfoldStreamF :: Stream x list -> list -> Int -> x
unfoldStreamF alg s = \case 0 -> head_ alg s
                           n -> unfoldStreamF alg (tail_ alg s) $ n-1
```

Datentyp der $DAut(X, Y)$ -Algebren und Implementierung der Cotermalgebra $DAutC(X)(Y)$ von Beispiel 10.4:

```
data DAut x y state = DAut {delta :: state -> x -> state,
                                beta :: state -> y}

dAutC :: DAut x y (DAutC x y)
dAutC = DAut {delta = deltaC, beta = betaC}
```

Die $Acc(\mathbb{Z})$ -Algebra `eo` (siehe 2.6) kann wie folgt implementiert werden:

```
data EO = Esum | Osum deriving Eq
eo :: DAut Int Bool EO
eo = DAut {delta = \case Esum -> f . even; Osum -> f . odd,
           beta = (== Esum)}
where f b = if b then Esum else Osum
```

Die finale $DAut(X, Y)$ -Algebra $Beh(X, Y)$ (siehe 2.10) kann wie folgt implementiert werden:

```
behFun :: DAut x y ([x] -> y)
behFun = DAut {delta = \f x -> f . (x:), beta = ($ [])}
```

Die folgenden Funktionen implementieren die Entfaltungen von Elementen einer beliebigen $DAut(X, Y)$ -Algebra alg zu Elementen von $Beh(X, Y)$ bzw. $DT_{DAut(X, Y)}$:

```
unfoldDAutF :: DAut x y state -> state -> [x] -> y
unfoldDAutF alg s = \case [] -> beta alg s
                           x:w -> unfoldDAutF alg (delta alg s x) w
```

```
unfoldDAut :: DAut x y state -> state -> StateC x y
unfoldDAut alg s = DA {deltaC = unfoldDAut alg . delta alg s,
```

```
betaC = beta alg s}
```

Nach Beispiel 2.19 realisieren die initialen Automaten $(eo, Esum)$ und $(eo, Osum)$ die Verhaltensfunktion $even \circ sum : \mathbb{Z}^* \rightarrow 2$ bzw. $odd \circ sum : \mathbb{Z}^* \rightarrow 2$.

Da eo $Acc(\mathbb{Z})$ -isomorph zur Unteralgebra von $A = DAutC(\mathbb{Z})(Bool)$ mit der Trägermenge $\{esum, osum\}$ ist, werden die beiden Funktionen auch von den initialen Automaten $(A, esum)$ und $(A, osum)$ realisiert. Es gelten also folgende Gleichungen:

```
unfoldDAutF eo Esum = even . sum = unfoldDAutF dAutC esum  
unfoldDAutF eo Osum = odd . sum = unfoldDAutF dAutC osum
```

□

11.2 Datentyp der JavaLight-Algebren

(siehe Beispiel 4.2 und [Java.hs](#))

```
data JavaLight commands command sum prod factor disjunct conjunct literal =  
    JavaLight {seq_          :: command -> commands -> commands,  
               embed         :: command -> commands,  
               block         :: commands -> command,  
               assign        :: String -> sum -> command,  
               cond          :: disjunct -> command -> command -> command,  
               cond1,loop   :: disjunct -> command -> command,  
               sum_          :: prod -> sum,  
               plus,minus   :: sum -> prod -> sum,
```

```

prod      :: factor -> prod,
times,div_ :: prod -> factor -> prod,
embedI    :: Int -> factor,
var       :: String -> factor,
encloseS  :: sum -> factor,
disjunct   :: conjunct -> disjunct -> disjunct,
embedC    :: conjunct -> disjunct,
conjunct   :: literal -> conjunct -> conjunct,
embedL    :: literal -> conjunct,
not_       :: literal -> literal,
atom       :: String -> sum -> sum -> literal,
embedB    :: Bool -> literal,
encloseD  :: disjunct -> literal}

```

Der folgende Datentyp implementiert eine Teilsignatur der abstrakten Syntax von *derec*(JavaLight).

```

data SumProd sum sumsect prod prodsect factor =
  SumProd {sum'           :: prod -> sumsect -> sum,
            plus',minus' :: prod -> sumsect -> sumsect,
            nilS         :: sumsect,
            prod'        :: factor -> prodsect -> prod,
            times',div'  :: factor -> prodsect -> prodsect,
            nilP         :: prodsect}

```

Die folgende Funktion implementiert die in Kapitel 4 definierte Erweiterung von JavaLight- zu *derec*(JavaLight)-Algebren:

```
derec :: JavaLight s1 s2 sum prod factor s3 s4 s5
        -> SumProd sum (sum -> sum) prod (prod -> prod) factor
derec alg = SumProd {sum' = \a g -> g $ sum_ alg a,
                      plus' = \a g x -> g $ plus alg x a,
                      minus' = \a g x -> g $ minus alg x a,
                      nilS = id,
                      prod' = \a g -> g $ prod alg a,
                      times' = \a g x -> g $ times alg x a,
                      div' = \a g x -> g $ div_ alg x a,
                      nilP = id}
```

11.3 Die Termalgebra von JavaLight

Zunächst wird die Menge der JavaLight-Terme analog zu den $T_{List(X)}$ und $T_{Reg(BL)}$ (siehe 10.1 bzw. 10.2) durch Datentypen implementiert und zwar jeweils einen für jede Sorte von JavaLight:

```
data Commands  = Seq (Command,Commands) | Embed Command deriving Show
data Command   = Block Commands | Assign (String,Sum) |
                Cond (Disjunct,Command,Command) | Cond1 (Disjunct,Command) |
```

```

Loop (Disjunct,Command) deriving Show
data Sum      = SUM Prod | PLUS (Sum,Prod) | MINUS (Sum,Prod) deriving Show
data Prod     = PROD Factor | TIMES (Prod,Factor) | DIV (Prod,Factor) deriving Show
data Factor   = EmbedI Int | Var String | EncloseS Sum deriving Show
data Disjunct  = Disjunct (Conjunct,Disjunct) | EmbedC Conjunct deriving Show
data Conjunct  = Conjunct (Literal,Conjunct) | EmbedL Literal deriving Show
data Literal   = Not Literal | Atom (String,Sum,Sum) | EmbedB Bool |
                EncloseD Disjunct deriving Show

javaTerm :: JavaLight Commands Command Sum Prod Factor Disjunct Conjunct Literal
javaTerm = JavaLight {seq_ = curry Seq, embed = Embed, block = Block,
                     assign = curry Assign, cond = curry3 Cond, cond1 = curry Cond1,
                     loop = curry Loop, sum_ = SUM, plus = curry PLUS,
                     minus = curry MINUS, prod = PROD, times = curry TIMES,
                     div_ = curry DIV, embedI = EmbedI, var = Var,
                     encloseS = EncloseS, disjunct = curry Disjunct, embedC = EmbedC,
                     conjunct = curry Conjunct, embedL = EmbedL, not_ = Not,
                     atom = curry3 Atom, embedB = EmbedB, enclosedD = EncloseD}

```

11.4 Die Wortalgebra von JavaLight

```

javaWord :: JavaLight String String String String String String String String
javaWord = JavaLight {seq_        = (++),
                     embed       = id,
                     block       = \cs -> " {" ++ cs ++ " }",
                     assign      = \x e -> x ++ " = " ++ e ++ "; ",

```

```

cond      = \e c c' -> "if "++e++c++" else"++c',
cond1     = \e c -> "if "    ++e++c,
loop      = \e c -> "while "++e++c,
sum_      = id,
plus      = \e e' -> e++'+':e',
minus     = \e e' -> e++'-':e,
prod      = id,
times     = \e e' -> e++'*':e',
div_      = \e e' -> e++'/':e',
embedI    = show,
var       = id,
encloseS  = \e -> '(:e++)",
disjunct   = \e e' -> e++" || "++e',
embedC    = id,
conjunct   = \e e' -> e++" && "++e',
embedL    = id,
not_      = \be -> '!':be,
atom      = \rel e e' -> e++rel++e',
embedB    = show,
encloseD  = \e -> '(:e++)"

```

11.5 Das Zustandsmodell von JavaLight (siehe 4.9)

```
type St a = Store -> a
```

```

rel :: String -> Int -> Int -> Bool
rel = \case "<" -> (<)
           ">" -> (>)
           "<=" -> (<=)
           ">=" -> (>=)
           "==" -> (==)
           "!=" -> (/=)

javaState :: JavaLight (St Store) (St Store) (St Int) (St Int) (St Int) (St Bool)
                           (St Bool) (St Bool)
javaState = JavaLight {seq_          = flip (.) ,
                      embed         = id,
                      block         = id,
                      assign        = \x e st -> update st x $ e st,
                      cond          = cond,
                      cond1         = \p f -> cond p f id,
                      loop          = loop,
                      sum_          = id,
                      plus          = liftM2 (+),
                      minus         = liftM2 (-),
                      prod          = id,
                      times         = liftM2 (*),
                      div_          = liftM2 div,
                      embedI        = const,
                      var           = flip ($),

```

```

encloseS   = id,
disjunct   = liftM2 (||),
embedC     = id,
conjunct   = liftM2 (&&),
embedL     = id,
not_        = (not .),
atom        = liftM2 . rel,
embedB     = const,
encloseD   = id}

```

where $\text{cond} :: \text{St Bool} \rightarrow \text{St Store} \rightarrow \text{St Store} \rightarrow \text{St Store}$
 $\text{cond } p \ f \ g \ \text{st} = \text{if } p \ \text{st} \text{ then } f \ \text{st} \text{ else } g \ \text{st}$

$\text{loop} :: \text{St Bool} \rightarrow \text{St Store} \rightarrow \text{St Store}$
 $\text{loop } p \ f \ \text{st} = \text{if } p \ \text{st} \text{ then } \text{loop } f \ \$ \ f \ \text{st} \text{ else } \text{st}$

Z.B. hat das JavaLight-Programm

prog = fact = 1; while x > 1 {fact = fact*x; x = x-1;}

die folgende Interpretation in $A = \text{javaState}$:

$\text{compile}_{\text{JavaLight}}^A(\text{prog}) : \text{Store} \rightarrow \text{Store}$

$\text{store} \mapsto \lambda z. \text{if } z = x \text{ then } 0$

$\text{else if } z = \text{fact} \text{ then } \text{store}(x)! \text{ else } \text{store}(z)$

11.6 Die Ableitungsbaumalgebra von JavaLight

```
type TS = Tree String
```

```
javaDeri :: JavaLight TS TS TS TS TS TS TS TS TS  
javaDeri = JavaLight {seq_          = \c c' -> F "Commands" [c,c'] ,  
                      embed      = \c -> F "Commands" [c] ,  
                      block      = \c -> command [c] ,  
                      assign     = \x e -> command [leaf x,leaf "=" ,e,leaf ";" ] ,  
                      cond       = \e c c' -> command [leaf "if" ,e,c,leaf "else" ,c'] ,  
                      cond1      = \e c -> command [leaf "if" ,e,c] ,  
                      loop       = \e c -> command [leaf "while" ,e,c] ,  
                      sum_       = \e -> F "Sum" [e] ,  
                      plus       = \e e' -> F "Sum" [e,e'] ,  
                      minus      = \e e' -> F "Sum" [e,e'] ,  
                      prod       = \e -> F "Prod" [e] ,  
                      times      = \e e' -> F "Prod" [e,e'] ,  
                      div_       = \e e' -> F "Prod" [e,e'] ,  
                      embedI     = \i -> factor [leaf $ show i] ,  
                      var        = \x -> factor [leaf x] ,  
                      encloseS   = \e -> factor [leaf "(" ,e,leaf ")" ] ,  
                      disjunct   = \e e' -> F "Disjunct" [e,leaf "||" ,e'] ,  
                      embedC     = \e -> F "Disjunct" [e] ,  
                      conjunct   = \e e' -> F "Conjunct" [e,leaf "&&" ,e'] ,  
                      embedL     = \e -> F "Conjunct" [e] ,
```

```
not_      = \be -> literal [leaf "!",be] ,
atom      = \rel e e' -> literal [e,leaf rel,e'] ,
embedB    = \b -> literal [leaf $ show b] ,
encloseD  = \e -> literal [leaf "(" ,e,leaf ")" ] }
```

where command = F "Command"

factor = F "Factor"

literal = F "Literal"

leaf = flip F []

11.7 Datentyp der XMLstore-Algebren (siehe Beispiel 4.3 und Compiler.hs)

```
data XMLstore store orders person emails email items stock suppliers
    id =
    XMLstore {store      :: stock -> store,
              store0     :: orders -> stock -> store,
              orders     :: person -> items -> orders -> orders,
              embed0     :: person -> items -> orders,
              person     :: String -> person,
              personE    :: String -> emails -> person,
              emails     :: email -> emails -> emails,
              none       :: emails,
              email      :: String -> email,
              items      :: id -> String -> items -> items,
              embedI    :: id -> String -> items,
              stock      :: id -> Int -> suppliers -> stock -> stock,
              embedS    :: id -> Int -> suppliers -> stock,
              supplier   :: person -> suppliers,
              parts      :: stock -> suppliers,
              id_        :: String -> id}
```

12 Attributierte Übersetzung

Um die Operationen der Zielalgebra einer Übersetzung induktiv definieren zu können, müssen Trägermengen oft parametrisiert werden oder die Wertebereiche bereits parametrierter Trägermengen um zusätzliche Komponenten erweitert werden, die zur Berechnung von Parametern rekursiver Aufrufe des Übersetzers benötigt werden. Die Zielalgebra A hat dann die Form

$$A_{v_1} \times \dots \times A_{v_m} \rightarrow A_{a_1} \times \dots \times A_{a_n}. \quad (1)$$

Die Indizes v_1, \dots, v_m und a_1, \dots, a_n heißen **vererbte Attribute** (*inherited attributes*) bzw. **abgeleitete Attribute** (*derived, synthesized attributes*) von s . Für alle $1 \leq i \leq m$ und $1 \leq j \leq n$ ist A_{v_i} bzw. A_{a_j} die Menge der möglichen Werte des Attributs v_i bzw. a_j .

Vererbte Attribut(wert)e sind z.B. Positionen, Adressen, Schachtelungstiefen, etc. Abgeleitete Attribut(wert)e sind das eigentliche Zielobjekt wie auch z.B. dessen Typ, Größe oder Platzbedarf, das selbst einen Wert eines abgeleiteten Attributs bildet.

Gleichzeitig vererbte und abgeleitete Attribute heißen **transient**. Deren Werte bilden den Zustandsraum, der einen Compiler zur Transitionsfunktion eines Automaten macht, dessen Ein- und Ausgaben Quell- bzw. Zielprogramme sind.

Kurz gesagt, ergänzen Attribute einen Syntaxbaum um Zusatzinformation, die erforderlich ist, um ein bestimmtes Übersetzungsproblem zu lösen. In unserem generischen Ansatz sind sie aber nicht – wie beim klassischen Begriff einer **Attributgrammatik** – Dekorationen von Syntaxbäumen, sondern Komponenten der jeweiligen Zielalgebra.

Nach der Übersetzung in eine Zielfunktion vom Typ (1) werden alle vererbten Attribute mit bestimmten Anfangswerten initialisiert. Dann werden die Zielfunktion auf die Anfangswerte angewendet und am Schluss das Ergebnistupel abgeleiteter Attributwerte mit π_1 auf die erste Komponente, die das eigentliche Zielobjekt darstellt, projiziert.

Beispiel 12.1

Binärdarstellung rationaler Zahlen (siehe auch [Compiler.hs](#))

konkrete Syntax G

$rat \rightarrow nat. \mid rat0 \mid rat1$

$nat \rightarrow 0 \mid 1 \mid nat0 \mid nat1$

abstrakte Syntax $\Sigma(G)$

$mkRat : nat \rightarrow rat$

$app0, app1 : rat \rightarrow rat$

$0, 1 : 1 \rightarrow nat$

$app0, app1 : nat \rightarrow nat$

Die Zielalgebra $\mathcal{A} = (A, Op)$

A_{rat} enthält neben dem eigentlichen Zielobjekt ein weiteres abgeleitetes Attribut mit Wertebereich \mathbb{Q} , welches das **Inkrement** liefert, um das sich der Dezimalwert einer rationalen Zahl erhöht, wenn an die Mantisse ihrer Binärdarstellung eine 1 angefügt wird.

$$A_{nat} = \mathbb{N}$$

$$A_{rat} = \mathbb{Q} \times \mathbb{Q}$$

$$0^A : A_{nat}$$

$$1^A : A_{nat}$$

$$0^A = 0$$

$$1^A = 1$$

$$app0^{\mathcal{A}} : A_{nat} \rightarrow A_{nat}$$

$$app1^{\mathcal{A}} : A_{nat} \rightarrow A_{nat}$$

$$app0^{\mathcal{A}}(n) = n * 2$$

$$app1^{\mathcal{A}}(n) = n * 2 + 1$$

$$mkRat^{\mathcal{A}} : A_{nat} \rightarrow A_{rat}$$

$$mkRat^{\mathcal{A}}(val) = (val, 1)$$

$$app0^{\mathcal{A}} : A_{rat} \rightarrow A_{rat}$$

$$app1^{\mathcal{A}} : A_{rat} \rightarrow A_{rat}$$

$$app0^{\mathcal{A}}(val, inc) = (val, inc/2)$$

$$app1^{\mathcal{A}}(val, inc) = (val + inc/2, inc/2)$$

Beispiel 12.2

Strings mit Hoch- und Tiefstellungen

konkrete Syntax G

$string \rightarrow string\ box \mid$

$string \uparrow box \mid$

$string \downarrow box \mid$

box

$box \rightarrow (string) \mid$

$Char$

abstrakte Syntax $\Sigma(G)$

$app : string \times box \rightarrow string$

$up : string \times box \rightarrow string$

$down : string \times box \rightarrow string$

$mkString : box \rightarrow string$

$mkBox : string \rightarrow box$

$embed : Char \rightarrow box$

Die Zielalgebra $\mathcal{A} = (A, Op)$

A_{string} und A_{box} enthalten

- ein vererbtes Attribut mit Wertebereich \mathbb{N}^2 , das die **Koordinaten der linken unteren Ecke des Rechtecks** liefert, in das der eingelesene String geschrieben wird,
- neben dem eigentlichen Zielobjekt ein weiteres abgeleitetes Attribut mit Wertebereich \mathbb{N}^3 , das die Länge sowie - auf eine feste Grundlinie bezogen - Höhe und Tiefe des Rechtecks liefert.

$$A_{string} = A_{box} = \text{Strings mit Hoch- und Tiefstellungen} \times \mathbb{N}^2 \rightarrow \mathbb{N}^3$$

$$app^{\mathcal{A}} : A_{string} \times A_{box} \rightarrow A_{string}$$

$$app^{\mathcal{A}}(f, g)(x, y) = (str\ str', l + l', \max h\ h', \max t\ t')$$

$$\text{where } (str, l, h, t) = f(x, y)$$

$$(str', l', h', t') = g(x + l, y)$$

$$up^{\mathcal{A}} : A_{string} \times A_{box} \rightarrow A_{string}$$

$$up^{\mathcal{A}}(f, g)(x, y) = (str^{str'}, l + l', h + h' - 1, \max t\ (t' - h + 1))$$

$$\text{where } (str, l, h, t) = f(x, y)$$

$$(str', l', h', t') = g(x + l, y + h - 1)$$

$$down^{\mathcal{A}} : A_{string} \times A_{box} \rightarrow A_{string}$$

$$down^{\mathcal{A}}(f, g)(x, y) = (str_{str'}, l + l', \max h\ (h' - t - 1), t + t' - 1)$$

$$\text{where } (str, l, h, t) = f(x, y)$$

$$(str', l', h', t') = g(x + l, y - t + 1)$$

$$mkString^{\mathcal{A}} : A_{box} \rightarrow A_{string}$$

$$mkString^{\mathcal{A}}(f) = f$$

$$mkBox^{\mathcal{A}} : A_{string} \rightarrow A_{box}$$

$$mkBox^{\mathcal{A}}(f) = f$$

$$embed^{\mathcal{A}} : Char \rightarrow A_{box}$$

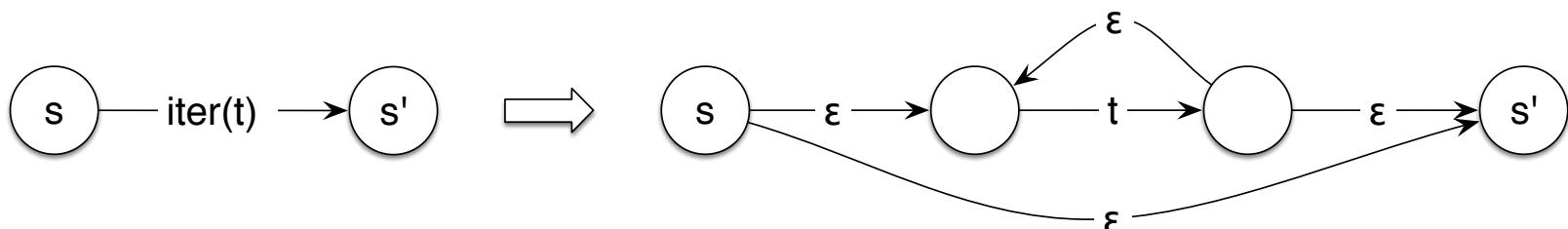
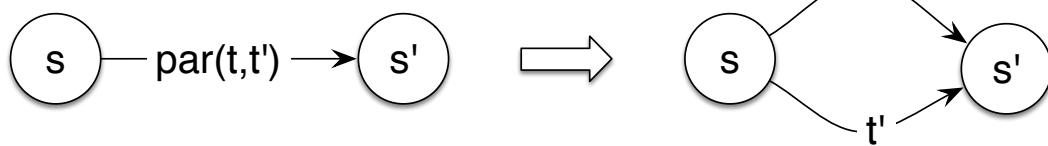
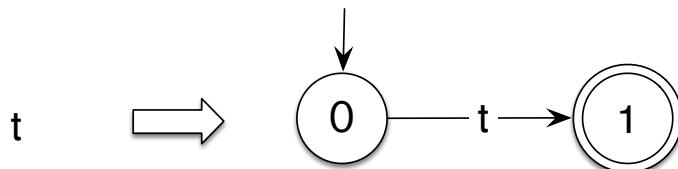
$$embed^{\mathcal{A}}(c)(x, y) = (\textcolor{red}{c}, 1, 2, 0)$$

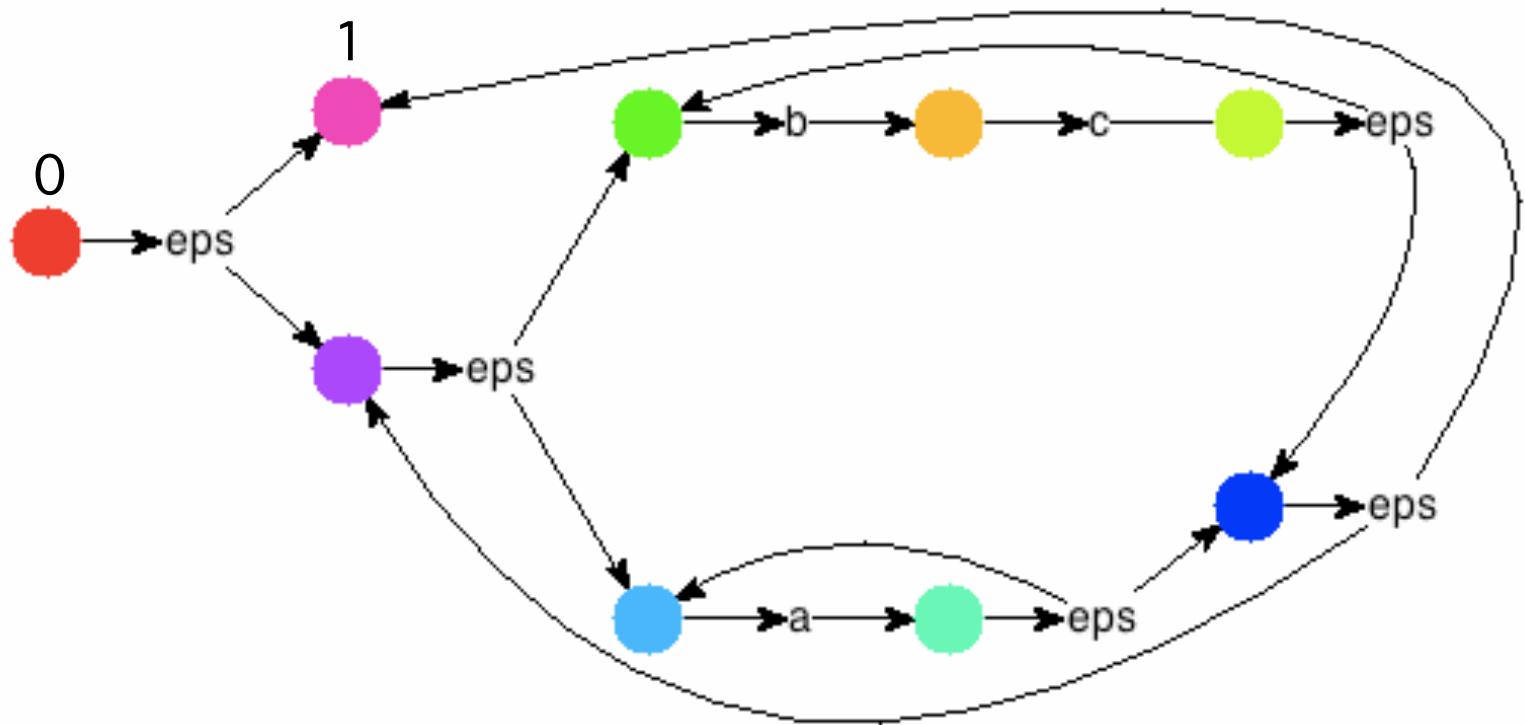
Beispiel 12.3

Übersetzung regulärer Ausdrücke in erkennende Automaten

Die folgenden Ersetzungsregeln beschreiben die schrittweise Übersetzung eines regulären Ausdrucks ($= Reg(BL)$ -Grundterms) t in einen nichtdeterministischen Automaten, der die Sprache von t erkennt (siehe 2.12). Die Regeln basieren auf [11], Abschnitt 3.2.3.

Zunächst wird die erste Regel auf t angewendet. Es entsteht ein Graph mit zwei Knoten und einer mit t markierten Kante. Dieser wird nun mit den anderen Regeln schrittweise verfeinert, bis an allen seinen Kanten nur noch Elemente von BL stehen. Dann stellt er den Transitionsgraphen eines Automaten dar, der t erkennt.





Mit obigen Regeln aus dem regulären Ausdruck $t = (aa^* + bc(bc)^*)^*$
in Wortdarstellung (siehe 2.9) konstruierter Automat,
der die Sprache von t erkennt

Die Zielalgebra $regNDA = (A, Op)$

Jede Regel außer der ersten entspricht der Interpretation einer Operation von $Reg(BL)$ in folgender $Reg(BL)$ -Algebra.

$$A_{reg} = (\textcolor{red}{NDA} \times \mathbb{N}^3 \rightarrow \textcolor{blue}{NDA} \times \mathbb{N})$$

Hierbei ist $\textcolor{blue}{NDA} = \mathbb{N} \rightarrow (BL \rightarrow \mathbb{N}^*)$ der Typ nichtdeterministischer erkennender Automaten mit ganzzahligen Zuständen und Eingaben aus BL .

$regNDA_{reg}$ enthält

- den Automaten als transientes Attribut, das mit der Funktion $\lambda n. \lambda s. \epsilon$ (Automat ohne Zustandsübergänge) initialisiert wird,
- zwei vererbte Attribute mit Wertebereich \mathbb{N} , die mit 0 bzw. 1 initialisiert werden und den Start- bzw. Endzustand des Automaten bezeichnen,
- ein transientes Attribut mit Wertebereich \mathbb{N} , das mit 2 initialisiert wird und die nächste ganze Zahl liefert, die als Zustandsname vergeben werden kann.

Die Operationen von $Reg(BL)$ werden von $regNDA$ wie folgt interpretiert:

$$\text{par}^{regNDA} : regNDA_{reg} \times regNDA_{reg} \rightarrow regNDA_{reg}$$

$$\text{par}^{regNDA}(f, g)(\delta, s, s', next) = g(\delta', s, s', next')$$

$$\text{where } (\delta', next') = f(\delta, s, s', next)$$

$seq^{regNDA}_- : regNDA_{reg} \times regNDA_{reg} \rightarrow regNDA_{reg}$

$seq^{regNDA}(f, g)(\delta, s, s', next) = g(\delta', next, s', next')$

where $(\delta', next') = f(\delta, s, next, next + 1)$

$iter^{regNDA} : regNDA_{reg} \rightarrow regNDA_{reg}$

$iter^{regNDA}(f)(\delta, s, s', next) = (addTo(\delta_4)(s)(\epsilon)(s'), next_3)$

where $next_1 = next + 1$

$next_2 = next_1 + 1$

$(\delta_1, next_3) = f(\delta, next, next_1, next_2)$

$\delta_2 = addTo(\delta_1)(s)(\epsilon)(next)$

$\delta_3 = addTo(\delta_2)(next_1)(\epsilon)(next)$

$\delta_4 = addTo(\delta_3)(next_1)(\epsilon)(s')$

$base^{regNDA} : BL \rightarrow regNDA_{reg}$

$base^{regNDA}(\emptyset)(\delta, s, s', next) = (\delta, next)$

$\forall B \in BL \setminus \{\emptyset\} : base^{regNDA}(B)(\delta, s, s', next) = (addTo(\delta)(s)(B)(s'), next)$

$addTo(\delta)(s)(x)(s')$ fügt die Transition $s \xrightarrow{x} s'$ zur Transitionsfunktion δ hinzu.

Der durch Auswertung eines regulären Ausdrucks ($= Reg(BL)$ -Grundterms) in $regNDA$ erzeugte Automat $nda \in NDA$ ist nichtdeterministisch und hat ϵ -Übergänge. Er lässt sich wie üblich in einen deterministischen **Potenzautomaten** ohne ϵ -Übergänge transformieren.

Da 0 der Anfangszustand, 1 der Endzustand und auch alle anderen Zustände von nda ganze Zahlen sind, lassen sich alle Zustände des Potenzautomaten $pow(nda)$ als Listen ganzer Zahlen darstellen. $pow(nda)$ ist, zusammen mit seinem Anfangszustand, der ϵ -Hülle des Anfangszustandes 0 von nda , eine $DAut(BL, 2)$ -Algebra mit Zustandsmenge \mathbb{Z}^* (siehe 2.8):

```
accNDA :: NDA -> (DAut BL Bool [Int], [Int])
accNDA nda = (DAut {delta = (epsHull .) . deltaP, beta = (1 `elem`)},
                  epsHull [0])
  where deltaP :: [Int] -> BL -> [Int]
        deltaP qs x = unionMap (flip nda x) qs
        epsHull :: [Int] -> [Int]
        epsHull qs = if qs' `subset` qs
                      then qs else epsHull $ qs `union` qs'
                           where qs' = deltaP qs ε
```

$regNDA$ und $accNDA$ sind im Haskell-Modul **Compiler.hs** implementiert.

12.4 Darstellung von Termen als hierarchische Listen

Mit folgender JavaLight-Algebra *javaList* wird z.B. der **Syntaxbaum** des Programms

```
fact = 1; while x > 1 {fact = fact*x; x = x-1;}
```

in die Form einer hierarchischen Liste gebracht:

```
seq_ assign fact
    sum prod embedI 1
        nilP
        nilS
    embed loop embedC embedL atom sum prod var x
        nilP
        nilS
        >
        sum prod embedI 1
            nilP
            nilS
    block seq_ assign fact
        sum prod var fact
            prodsect *
                var x
                nilP
        nilS
    embed assign x
        sum prod var x
            nilP
            sumsect -
                prod embedI 1
                    nilP
                    nilS
```

Die Trägermengen von *javaList* enthalten zwei vererbte Attribute vom Typ 2 bzw. \mathbb{Z} .

Der Boolesche Wert gibt an, ob der jeweilige Teilstring *str* hinter oder unter den vorangehenden zu schreiben ist. Im zweiten Fall wird *str* um den Wert des zweiten Attributs eingerückt.

```
type BIS = Bool -> Int -> String
```

```
javaList :: JavaLight BIS BIS BIS BIS BIS BIS BIS BIS
javaList = JavaLight {seq_      = indent2 "commands",
                      embed     = indent1 "embed",
                      block     = indent1 "block",
                      assign    = indent2 "assign" . indent0,
                      cond      = indent3 "cond",
                      cond1    = indent2 "cond1",
                      loop      = indent2 "loop",
                      sum_     = indent1 "sum",
                      plus     = indent2 "plus",
                      minus    = indent2 "minus",
                      prod     = indent1 "prod",
                      times   = indent2 "times",
                      div_     = indent2 "div",
                      embedI   = indent1 "embedI" . indent0 . show,
                      var      = indent1 "var" . indent0,
                      encloseS = indent1 "encloseS",
```

```

disjunct = indent2 "disjunct",
embedC   = indent1 "embedC",
conjunct = indent2 "conjunct",
embedL   = indent1 "embedL",
not_     = indent1 "not",
atom     = indent3 "atom" . indent0,
embedB   = indent1 "embedB" . indent0 . show,
encloseD = indent1 "encloseD"}}

where indent0 x      = blanks x []
      indent1 x f    = blanks x [f]
      indent2 x f g  = blanks x [f,g]
      indent3 x f g h = blanks x [f,g,h]
      blanks :: String -> [BIS] -> BIS
      blanks x fs b n = if b then str else '\n':replicate n ' '++str
                           where str = case fs of f:fs -> x++' ':g True f++
                                             concatMap (g False) fs
                                               _ -> x
      g b f = f b \$ n+length x+1

```

12.5 Eine kellerbasierte Zielsprache für JavaLight

Der folgende Datentyp liefert die Befehle einer Assemblersprache, die auf einem Keller vom Typ \mathbb{Z} und einem Speicher vom Typ

$$Store = String \rightarrow \mathbb{Z}$$

operiert. Letzterer ist natürlich nur die Abstraktion eines realen Speichers, auf dessen Inhalt nicht über Strings, sondern über Adressen, z.B. Kellerpositionen, zugegriffen wird. Eine solche – realistischere – Assemblersprache wird erst im nächsten Kapitel behandelt.

```
data StackCom = Push Int | Pop | Load String | Save String | Add |
               Sub | Mul | Div | Or_ | And_ | Inv | Cmp String |
               Jump Int | JumpF Int
```

Diese Befehle bilden die Eingaben eines endlichen Automaten, dessen Zustände jeweils aus einem Kellerinhalt, einer Variablenbelegung und der Position des Befehls, den ein Interpreter der Befehle als nächsten ausführt, bestehen:

```
type State = ([Int], Store, Int)
```

Die Bedeutung der einzelnen Befehle ergibt sich aus ihrer Verarbeitung durch folgenden Interpreter:

```

executeCom :: StackCom -> State -> State
executeCom com (stack,store,n) =
  case com of Push a      -> (a:stack,store,n+1)
              Pop       -> (tail stack,store,n+1)
              Load x    -> (store x:stack,store,n+1)
              Save x    -> (stack,update store x $ head stack,n+1)
              Add       -> (a+b:s,store,n+1) where a:b:s = stack
              Sub       -> (b-a:s,store,n+1) where a:b:s = stack
              Mul       -> (a*b:s,store,n+1) where a:b:s = stack
              Div       -> (b`div`a:s,store,n+1) where a:b:s = stack
              Or_       -> (max a b:s,store,n+1) where a:b:s = stack
              And_      -> (a*b:s,store,n+1) where a:b:s = stack
              Inv       -> ((a+1)`mod`2:s,store,n+1) where a:s = stack
              Cmp str   -> (c:s,store,n+1)
                            where a:b:s = stack
                            c = if rel str a b then 1 else 0
                            -- siehe 11.5
              Jump k    -> (stack,store,k)
              JumpF k   -> (stack,store,if a == 0 then k else n+1)
                            where a:_ = stack

```

Offenbar wird die dritte Zustandskomponente zur Verarbeitung der Sprungbefehle benötigt.

```
execute :: [StackCom] -> State -> State
execute cs state@(_,_,n) = if n >= length cs then state
                           else execute cs $ executeCom (cs!!n) state
```

`execute` führt Befehlsfolgen aus.

Diese erzeugt ein JavaLight-Compiler durch Interpretation des (abstrakten) Quellprogramms in der folgenden JavaLight-Algebra *javaStack*. Deren Trägermengen haben neben dem jeweiligen Zielcode *code* ein (vererbtes) Attribut, das die Nummer des ersten Befehls von *code* wiedergibt. Dementsprechend interpretiert *javaStack* alle Sorten von JavaLight durch den Funktionstyp

`LCom = Int -> [StackCom]`

```
javaStack :: JavaLight LCom LCom LCom LCom LCom LCom LCom LCom
javaStack = JavaLight {seq_          = seq_,
                      embed         = id,
                      block         = id,
                      assign        = `x e lab -> e lab++[Save x,Pop],
                      cond          = `e c c' lab
                                     -> let (code,exit) = fork e c 1 lab
                                         code' = c' exit
```

```

in code++Jump (exit+length code'):code',
cond1      = \e c -> fst . fork e c 0,
loop        = \e c lab -> fst (fork e c 1 lab)++[Jump lab],
sum_        = id,
plus        = apply2 Add,
minus        = apply2 Sub,
prod        = id,
times        = apply2 Mul,
div_         = apply2 Div,
embedI       = \i -> const [Push i],
var          = \x -> const [Load x],
encloseS     = id,
disjunct     = apply2 Or_,
embedC       = id,
conjunct     = apply2 And_,
embedL       = id,
not_         = apply1 Inv,
atom          = apply2 . Cmp,
embedB       = \b -> const [Push $ if b then 1 else 0],
encloseD     = id}

```

```

where apply1 :: StackCom -> LCom -> LCom
apply1 op e lab = e lab++[op]

seq_ :: LCom -> LCom -> LCom
seq_ e e' lab = code++e' (lab+length code)
    where code = e lab

apply2 :: StackCom -> LCom -> LCom -> LCom
apply2 op e e' lab = code++e' (lab+length code)++[op]
    where code = e lab

fork :: LCom -> LCom -> Int -> Int -> ([StackCom],Int)
fork e c n lab = (code++JumpF exit:code',exit)
    where code = e lab
        lab' = lab+length code+1
        code' = c lab'
        exit = lab'+length code'+n

```

Beispiel 12.6 (Fakultätsfunktion) Steht das JavaLight-Programm

```
fact = 1; while x > 1 {fact = fact*x; x = x-1;}
```

in der Datei `prog`, dann übersetzt es `javaToAlg "prog" 6` (siehe `Java.hs`) in ein Zielprogramm vom Typ `[StackCom]` und schreibt dieses in die Datei `javatarget` ab. Es lautet wie folgt:

0: Push 1	8: Load "x"	16: Pop
1: Save "fact"	9: Mul	17: Jump 3
2: Pop	10: Save "fact"	
3: Load "x"	11: Pop	
4: Push 1	12: Load "x"	
5: Cmp ">"	13: Push 1	
6: JumpF 18	14: Sub	
7: Load "fact"	15: Save "x"	

13 JavaLight+ = JavaLight + I/O + Deklarationen + Prozeduren

(siehe [Java2.hs](#))

13.1 Assemblersprache mit I/O und Kelleradressierung

Die Variablenbelegung $store : String \rightarrow \mathbb{Z}$ im Zustandsmodell von Abschnitt [Assemblerprogramme als JavaLight-Zielalgebra](#) wird ersetzt durch den Keller $stack \in \mathbb{Z}^*$, der jetzt nicht nur der schrittweisen Auswertung von Ausdrücken dient, sondern auch der Ablage von Variablenwerten unter vom Compiler berechneten Adressen. Weitere Zustandskomponenten sind:

- der Inhalt des Registers **BA** für die jeweils aktuelle Basisadresse (s.u.),
- der Inhalt des Registers **STP** für die Basisadresse des statischen Vorgängers des jeweils zu übersetzenen Blocks bzw. Funktionsaufrufs (s.u.),
- der schon in Abschnitt 12.5 benutzte **Befehlszähler pc** (*program counter*),
- der Ein/Ausgabestrom **io**, auf den Lese- bzw. Schreibbefehle zugreifen.

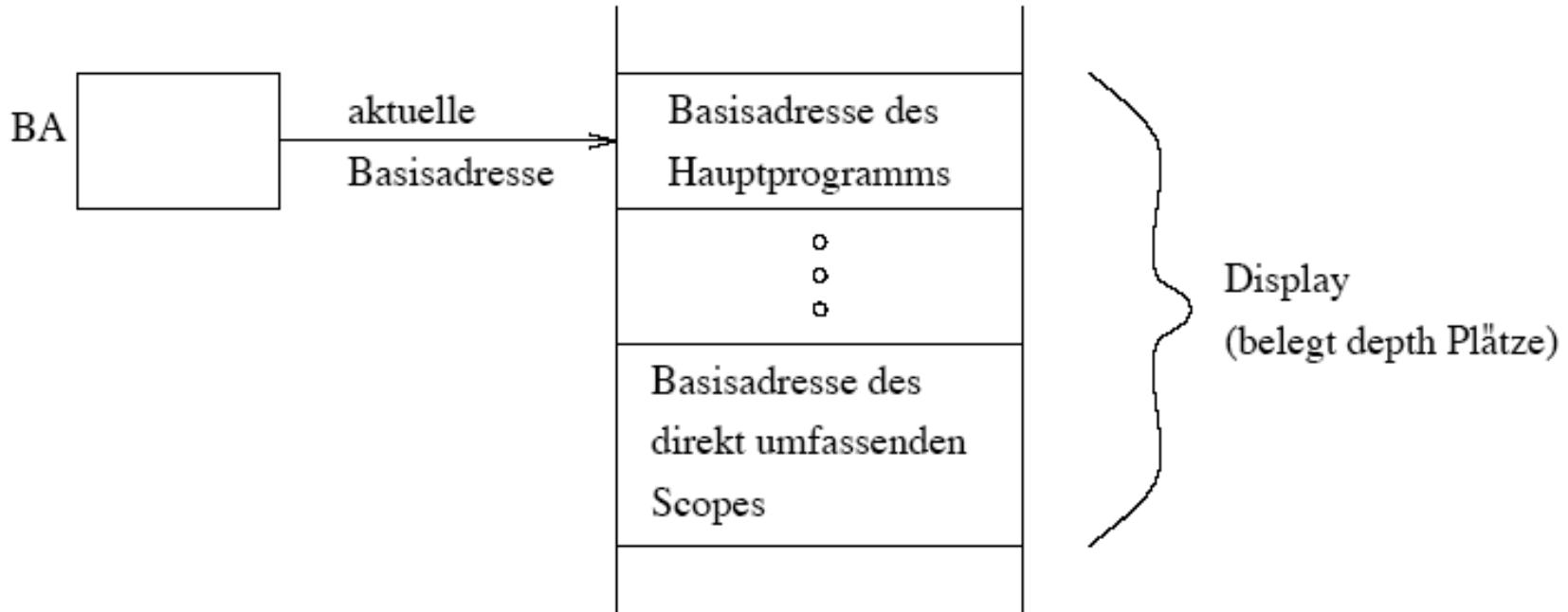
Der entsprechende Datentyp lautet daher wie folgt:

```
data State = State {stack,io :: [Int], ba,stp,pc :: Int}
```

Bei der Übersetzung eines Blocks b oder Prozeduraufrufs $f(es)$ reserviert der Compiler Speicherplatz für die Werte aller lokalen Variablen des Blocks b bzw. Rumpfs (der Deklaration) von f . Dieser Speicherplatz ist Teil eines b bzw. $f(es)$ zugeordneten Kellerabschnitts (*stack frame, activation record*). Dessen Anfangsadresse ist die **Basisadresse** ba der lokalen Variablen. Die **absolute Adresse** einer lokalen Variablen x ist die Kellerposition, an welcher der jeweils aktuelle Wert von x steht. Sie ist immer die Summe von ba und dem – **Relativadresse** von x genannten und vom Compiler in der **Symboltabelle** (s.u.) gespeicherten – Abstand zum Beginn des Kellerabschnitts.

Wird zur Laufzeit der Block b bzw. – als Teil der Ausführung von $f(es)$ – der Rumpf von f betreten, dann speichert ein vom Compiler erzeugter Befehl die nächste freie Kellerposition als aktuelle Basisadresse im Register **BA**.

Der **statische Vorgänger** von b bzw. $f(es)$ ist der innerste Block bzw. Prozedurrumpf, in dem b bzw. die Deklaration von f steht. Der b bzw. $f(es)$ zugeordnete Kellerabschnitt beginnt stets mit dem **Display**, das aus den – nach aufsteigender Schachtelungstiefe geordneten – Basisadressen der Kellerabschnitte aller umfassenden Blöcke und Prozedurrümpfe besteht.



Der Compiler erzeugt und verwendet keine ganzzahligen, sondern nur **symbolische Adressen**, d.h. Registernamen (BA, STP, TOP) oder mit einer ganzen Zahl indizierte (inDexed) Adressen der Form Dex(addr)(i):

```
data SymAdr = BA | STP | TOP | Dex SymAdr Int | Con Int
```

Der Inhalt von TOP ist immer die Adresse der ersten freien Kellerposition.

Die folgende Funktion **baseAddr** berechnet die jeweils aktuelle (symbolische) Basisadresse:

```

baseAddr :: Int -> Int -> SymAddr
baseAddr declDep dep = if declDep == dep then BA else Dex BA declDep

```

Der Compiler ruft `baseAddr` bei der Übersetzung jeder Verwendung einer Variable x auf. `declDep` und `dep` bezeichnen die Deklarations- bzw. Verwendungstiefe von x . Stimmen beide Werte überein, dann ist x eine lokale Variable und die Basisadresse von x steht (zur Laufzeit) im Register `BA`.

Andernfalls ist x eine globale Variable, d.h. x wurde in einem Block bzw. Prozedurrumpf deklariert, der denjenigen, in dem x verwendet wird, umfasst (`declDep < dep`). Die Basisadresse von x ist in diesem Fall nicht im Register `BA`, sondern unter dem Inhalt der `declDep`-ten Position des dem Block bzw. Prozedurrumpf, in dem x verwendet wird, zugeordneten Kellerabschnitt zu finden (s.o.).

Die folgenden Funktionen berechnen aus symbolischen Adressen absolute Adressen bzw. Kellerinhalte:

```

absAddr, contents :: State -> SymAddr -> Int
absAddr _ (Con i)          = i
absAddr state BA           = ba state
absAddr state STP          = stp state
absAddr state TOP          = length $ stack state
absAddr state (Dex BA i)   = ba state+i

```

<code>absAdr state (Dex STP i)</code>	$= \text{stp state} + i$
<code>absAdr state (Dex TOP i)</code>	$= \text{length}(\text{stack state}) + i$
<code>absAdr state (Dex adr i)</code>	$= \text{contents state}[\text{adr}] + i$
<code>contents state (Dex adr i)</code>	$= s[k-i]$ where $(s, k) = \text{stackPos state}[\text{adr}]$
<code>contents state adr</code>	$= \text{absAdr state}[\text{adr}]$

`stackPos :: State -> SymAddr -> ([Int], Int)`

`stackPos state adr` = $(s, \text{length } s - 1 - \text{contents state}[\text{adr}])$
where $s = \text{stack state}$

`updState :: State -> SymAddr -> Int -> State`

<code>updState state BA x</code>	$= \text{state}\{\text{ba} = x\}$
<code>updState state STP x</code>	$= \text{state}\{\text{stp} = x\}$
<code>updState state (Dex adr i) x</code>	$= \text{state}\{\text{stack} = \text{updList } s(k-i) x\}$ where $(s, k) = \text{stackPos state}[\text{adr}]$

Mit der Anwendung von `absAdr` oder `contents` auf eine – evtl. geschachtelte – indizierte Adresse wird eine Folge von Kelleradressen und -inhalten durchlaufen.

`absAdr` liefert die letzte Adresse der Folge, `contents` den Kellerinhalt an der durch diese Adresse beschriebenen Position.

Die Zielprogramme von JavaLight+ setzen sich aus folgenden Assemblerbefehlen mit symbolischen Adressen zusammen:

```
data StackCom = PushA SymAddr | Push SymAddr | Pop | Save SymAddr |
               Move SymAddr SymAddr | Add | Sub | Mul | Div | Or_ |
               And_ | Inv | Cmp String | Jump SymAddr | JumpF Int |
               Read SymAddr | Write
```

Analog zu Abschnitt 12.5 ist die Bedeutung der Befehle durch eine Transitionsfunktion `executeCom` gegeben – die jetzt auch die Sprungbefehle verarbeitet:

```
executeCom :: StackCom -> State -> State
executeCom com state =
  case com of
    PushA adr      -> state' {stack = absAdr state adr:stack state}
    Push adr       -> state' {stack = contents state adr:
                                stack state}
    Pop            -> state' {stack = tail $ stack state}
    Save adr       -> updState state' adr $ head $ stack state
    Move adr adr' -> updState state' adr' $ contents state adr
```

```

Add          -> applyOp state' (+)
Sub          -> applyOp state' (-)
Mul          -> applyOp state' (*)
Div          -> applyOp state' div
Or_          -> applyOp state' max
And_         -> applyOp state' (*)
Inv          -> state' {stack = (a+1)`mod`2:s}
                  where a:s = stack state
Cmp rel     -> state' {stack = mkInt (evalRel rel b a):s}
                  where a:b:s = stack state
Jump adr    -> state {pc = contents state adr}
JumpF lab   -> state {pc = if a == 0 then lab
                      else pc state+1,
                      stack = s} where a:s = stack state
Read adr    -> if null s then state'
                  else (updState state' adr $ head s)
                  {io = tail s} where s = io state
Write        -> if null s then state'
                  else state' {io = io state++[head s]}
                  where s = stack state
where state' = state {pc = pc state+1}

```

```
applyOp :: State -> (Int -> Int -> Int) -> State
applyOp state op = state {stack = op b a:s}
                     where a:b:s = stack state
```

execute wird ebenfalls an das neue Zustandsmodell angepasst:

```
execute :: [StackCom] -> State -> State
execute cs state = if curr >= length cs then state
                   else execute cs $ executeCom (cs!!curr) state
                     where curr = pc state
```

13.2 Grammatik und abstrakte Syntax von JavaLight+

JavaLight+ enthält neben den Sorten von JavaLight die Sorten *Formals* und *Actuals* für Listen formaler bzw. aktueller Parameter von Prozeduren. Auch die Basismengen von JavaLight werden übernommen. Hinzu kommt eine für formale Parameter. Sie besteht aus mit zwei Konstruktoren aus dem jeweiligen Parameternamen und einem **Typdeskriptor** gebildeten Ausdruck:

```
data TypeDesc = INT | BOOL | UNIT | Fun TypeDesc Int | ForFun TypeDesc
data Formal   = Par String TypeDesc | FunPar String [Formal] TypeDesc
```

`FunPar(x)(t)` bezeichnet einen funktionalen formalen Parameter, also eine Prozedurvariable. Sie hat den Typ `ForFun(t)`. `t` ist hier der Typ der Prozedurergebnisse. Demgegenüber bezeichnet `Fun(t,lab)` den Typ einer Prozedurkonstanten mit Ergebnistyp `t` und Codeadresse `lab`.

Dementsprechend enthält JavaLight+ auch die Regeln von [JavaLight](#). Hinzu kommen die folgenden Regeln für Ein/Ausgabebefehle, Deklarationen, Prozeduraufrufe und Parameterlisten. Außerdem sind jetzt auch Boolesche Variablen und Zuweisungen an diese zugelassen.

<i>Command</i>	\rightarrow	<i>read String</i> ; <i>write ExpSemi</i> { <i>Commands</i> }
		<i>TypeDesc String Formals</i> { <i>Commands</i> } <i>TypeDesc String</i> ;
		<i>String = ExpSemi</i> <i>String Formals</i> { <i>Commands</i> }
		<i>String Actuals</i>
<i>Formals</i>	\rightarrow	() (<i>Formals</i>)'
<i>Formals</i> '	\rightarrow	<i>Formal</i>) <i>Formal</i> , <i>Formals</i> '
<i>ExpSemi</i>	\rightarrow	<i>Sum</i> ; <i>Disjunct</i> ;
<i>ExpBrac</i>	\rightarrow	<i>Sum</i>) <i>Disjunct</i>)
<i>ExpComm</i>	\rightarrow	<i>Sum</i> , <i>Disjunct</i> ,
<i>Factor</i>	\rightarrow	<i>String Actuals</i>
<i>Literal</i>	\rightarrow	<i>String</i> <i>String Actuals</i>
<i>Actuals</i>	\rightarrow	() (<i>Actuals</i>)'
<i>Actuals</i> '	\rightarrow	<i>ExpBrac</i> <i>ExpComm Actuals</i> '

In Beispiel 6.2 wurde begründet, warum drei verschiedene Sorten für Ausdrücke (*ExpSemi*, *ExpBrac* und *ExpComm*) benötigt werden. Beim Übergang zur abstrakten Syntax können wir die Unterscheidung zwischen den drei Sorten wieder aufheben.

Erlaubt man nur JavaLight+-Algebren A , die *Formals* durch $Formal^*$ und *Actuals* durch $(A_{Sum} + A_{Disjunct})^*$ interpretieren, dann lautet der Datentyp der JavaLight+-Algebren wie folgt (siehe [Java2.hs](#)):

```
data JavaLightP commands command exp sum_ prod factor disjunct conjunct literal
    formals actuals =
  JavaLightP {seq_      :: command -> commands -> commands,
              embed     :: command -> commands,
              block     :: commands -> command,
              assign    :: String -> exp -> command,
              applyProc :: String -> actuals -> command,
              cond      :: disjunct -> command -> command -> command,
              cond1,loop :: disjunct -> command -> command,
              read_     :: String -> command,
              write_    :: exp -> command,
              vardecl   :: String -> TypeDesc -> command,
              fundecl   :: String -> formals -> TypeDesc -> commands -> command,
              formals   :: [Formal] -> formals,
              embedS    :: sum_ -> exp,
              sum_      :: prod -> sum,
              plus,minus :: sum -> prod -> sum,
```

```
prod      :: factor -> prod,
times,div_ :: prod -> factor -> prod,
embedI     :: Int -> factor,
varInt     :: String -> factor,
applyInt   :: String -> actuals -> factor,
encloseS   :: sum_ -> factor,
embedD     :: disjunct -> exp,
disjunct   :: conjunct -> disjunct -> disjunct,
embedC     :: conjunct -> disjunct,
conjunct   :: literal -> conjunct -> conjunct,
embedL     :: literal -> conjunct,
not_       :: literal -> literal,
atom       :: String -> sum_ -> sum_ -> literal,
embedB     :: Bool -> literal,
varBool    :: String -> literal,
applyBool  :: String -> actuals -> literal,
encloseD   :: disjunct -> literal,
actuals    :: [exp] -> actuals}
```

13.3 JavaLight+-Algebra *javaStackP* (siehe *Java2.hs*)

javaStackP hat wie *javaStack* nur eine Trägermenge (**ComStack**) für alle Kommandosorten sowie eine Trägermenge (**ExpStack**) für alle Ausdruckssorten (außer denen für Sektionen):

```
type ComStack = Int -> Symtab -> Int -> Int -> ([StackCom],Symtab,Int)
type ExpStack = Int -> Symtab -> Int -> ([StackCom],TypeDesc)
type Symtab = String -> (TypeDesc,Int,Int)
```

Die **Symboltabelle** (vom Typ *Symtab*) ordnet jeder Variablen drei Werte zu: Typ, Schachtelungstiefe ihrer Deklaration, also die Zahl der die Deklaration umfassenden Blöcke und Prozedurrümpfe, sowie eine Relativadresse (s.o.).

Transiente Attribute der Kommandosorten sind die Symboltabelle und die nächste freie Relativadresse (**adr**; s.u.).

Weitere vererbte Attribute der Kommando- und Ausdruckssorten sind die nächste freie Befehlsnummer (**lab**; s.u.) und die Schachtelungstiefe (**depth**; s.u.) des jeweiligen Kommandos bzw. Ausdrucks.

Weitere abgeleitete Attribute der Ausdruckssorten sind der Zielcode und der Typdeskriptor des jeweiligen Ausdrucks. Die Symboltabelle ist bei Ausdruckssorten nur vererbt.

Listen formaler bzw. aktueller Parameter werden von *javaStackP* als Elemente der folgenden Trägermengen interpretiert:

```
type FormsStack = Symtab -> Int -> Int -> ([ComStack],Symtab,Int)
type ActsStack = Int -> Symtab -> Int -> ([StackCom],Int)
```

Formale Parameter sind Teile von Funktionsdeklarationen. Deshalb haben Listen formaler Parameter Attribute von Kommandosorten: Symboltabelle, Schachtelungstiefe, nächste freie Relativadresse und sogar zusätzlichen Quellcode (vom Typ **[ComStack]**), der aus jedem funktionalen Parameter eine lokale Funktionsdeklaration erzeugt (siehe Übersetzung formaler Parameter).

Aktuelle Parameter sind Ausdrücke und haben deshalb (fast) die gleichen Attribute wie Ausdruckssorten. Anstelle eines Typdeskriptors wird die Länge der jeweiligen Parameter berechnet. Den Platz von **TypeDesc** nimmt daher **Int** ein.

Aktuelle Parameter können im Gegensatz zu anderen Vorkommen von Ausdrücken Prozedurvariablen sein. Der Einfachheit halber überprüft unser Compiler nicht, ob diese nur an Parameterpositionen auftreten, so wie er auch Anwendungen arithmetischer Operationen auf Boolesche Variablen oder Boolescher Operationen auf Variablen vom Typ \mathbb{Z} ignoriert.

Die Typverträglichkeit von Deklarationen mit den Verwendungen der deklarierten Variablen könnte aber mit Hilfe einer Variante von *javaStackP* überprüft werden, deren Trägermengen um Fehlermeldungen angereichert sind.

Bis auf die Interpretation von Blöcken und die Einbindung der o.g. Attribute gleicht die Interpretation der Programmkonstrukte von JavaLight in *javaStackP* derjenigen in *javaStack*.

An die Stelle der Lade- und Speicherbefehle, die *javaStack* erzeugt und die zur Laufzeit Daten von einer Variablenbelegung *store* : *String* → \mathbb{Z} zum Keller bzw. vom Keller nach *store* transportieren, treten die oben definierten Lade- und Speicherbefehle, die Daten oder Kelleradressen zwischen Kellerplätzen hin- und herschieben.

```
javaStackP ::: JavaAlgP ComStack ComStack ExpStack ExpStack ExpStack  
ExpStack ExpStack ExpStack FormsStack ActsStack  
javaStackP = JavaLightP {seq_          = seq_,  
                        embed        = id,  
                        block        = block,  
                        assign       = assign,  
                        applyProc    = applyProc,  
                        cond         = cond,  
                        cond1        = cond1,  
                        loop         = loop,  
                        read_        = read_,  
                        write_       = write_,  
                        vardecl     = vardecl,
```

```
fundecl      = fundecl,  
formals      = formals,  
embedS       = id,  
sum_          = id,  
plus          = apply2 Add,  
minus          = apply2 Sub,  
prod          = id,  
times          = apply2 Mul,  
div_          = apply2 Div,  
embedI        = \i _ _ _ -> ([Push $ Con i],INT),  
varInt        = var,  
applyInt      = applyFun,  
encloseS      = id,  
embedD        = id,  
disjunct      = apply2 Or_,  
embedC        = id,  
conjunct      = apply2 And_,  
embedL        = id,  
not_          = apply1 Inv,  
atom          = apply2 . Cmp,  
embedB        = \b _ _ _ -> ([Push $ Con $ mkInt b],BOOL),  
varBool        = var,  
applyBool      = applyFun,  
encloseD      = id,  
actuals        = actuals}
```

```

where apply1 :: String -> ExpStack -> ExpStack
apply1 op e lab st dep = (fst (e lab st dep)++[op],INT)

seq_ :: ComStack -> ComStack -> ComStack
seq_ c c' lab st dep adr = (code++code',st2,adr2)
    where (code,st1,adr1) = c lab st dep adr
          (code',st2,adr2) = c' (lab+length code) st1 dep adr1

apply2 :: String -> ExpStack -> ExpStack -> ExpStack
apply2 op e e' lab st dep = (code++code'++[op],td)
    where (code,td) = e lab st dep
          code' = fst $ e' (lab+length code) st dep

```

Übersetzung eines Blocks

```

block :: ComStack -> ComStack
block c lab st dep adr = (code',st,adr)
    where bodylab = lab+dep+3; dep' = dep+1
          (code,_,local) = c bodylab st dep' dep'
          code' = Move TOP STP:pushDisplay BA dep++Move STP BA:
bodylab:   code++replicate (local-dep') Pop++Save BA:
            replicate dep' Pop

```

```

pushDisplay :: Int -> SymAdr -> [StackCom]
pushDisplay dep reg = foldr push [Push reg] [0..dep-1]
    where push i code = Push (Dex reg i):code

```

javaStackP umschließt im Gegensatz zu *javaStack* bei der Zusammenfassung einer Kommandofolge cs zu einem Block den Code von cs mit zusätzlichen Zielcode:

Move TOP STP	<i>Speichern des Stacktops im Register STP</i>
pushDisplay dep BA	<i>Kellern des Displays des umfassenden Blocks und dessen Adresse</i>
Move STP BA	<i>Der Inhalt von STP wird zur neuen Basisadresse</i>
code	<i>Zielcode für die Kommandofolge des Blocks</i>
replicate (local-dep') Pop	<i>Entkellern der lokalen Variablen des Blocks</i>
Save BA	<i>Speichern der alten Basisadresse in BA</i>
replicate dep' Pop	<i>Entkellern des Displays</i>

Übersetzung einer Zuweisung

```
assign :: String -> ExpStack -> ComStack
assign x e lab st dep adr = (fst (e lab st dep)++[Save $ Dex ba adrx,Pop] ,
                               st,adr)
                           where (_,declDep,adrx) = st x
                                 ba = baseAddr declDep dep
```

Zielcodeerläuterung:

code	<i>Zielcode für den Ausdruck e, der mit einem Befehl zur Kellerung des aktuellen Wertes von e schließt</i>
Save \$ Dex ba adrx	<i>Speichern des aktuellen Wertes von e unter der Kelleradresse Dex ba adrx von x, die sich aus der Basisadresse ba von x und der in der Symboltabelle</i>

Pop

gespeicherten Relativadresse `adrx` von `x` ergibt
Entkellern des aktuellen Wertes von `e`

Übersetzung von Konditionalen und Schleifen

```
cond :: ExpStack -> ComStack -> ComStack -> ComStack
cond e c c' lab st dep adr = (code++Jump (Con $ exit+length code')):code',
                                st2,adr2)
                                where ((code,st1,adr1),exit) = fork e c 1 lab st dep adr
                                      (code',st2,adr2) = c' exit st1 dep adr1

cond1 :: ExpStack -> ComStack -> ComStack
cond1 e c lab st dep = fst . fork e c 0 lab st dep

loop :: ExpStack -> ComStack -> ComStack
loop e c lab st dep adr = (code++[Jump $ Con lab],st',adr')
                            where (code,st',adr') = fst $ fork e c 1 lab st dep adr

fork :: ExpStack -> ComStack -> Int -> Int -> Symtab -> Int -> Int
                            -> (([StackCom],Int,Symtab,Int),Int)
fork e c n lab st dep adr = ((code++JumpF exit:code',st',adr'),exit)
                            where code = fst $ e lab st dep
                                  lab' = lab+length code+1
                                  (code',st',adr') = c lab' st dep adr
                                  exit = lab'+length code'+n
```

Übersetzung eines Lesebefehls

```
read_ :: String -> ComStack
read_ x _ st dep adr = ([Read $ Dex ba adrx],st,adr)
    where (_ ,declDep,adrx) = st x
          ba = baseAddr declDep dep
```

Zielcodeerläuterung:

Read \$ Dex ba adrx Speichern des ersten Elementes *c* des Ein/Ausgabestroms *io* unter der Adresse **Dex ba adrx**, die sich aus der Basisadresse **ba** des Kellerbereichs, in dem **x** deklariert wurde, und der in der Symboltabelle gespeicherten Relativadresse **adrx** von **x** ergibt. *c* wird aus *io* entfernt.

Übersetzung eines Schreibbefehls

```
write_ :: ExpStack -> ComStack
write_ e lab st dep adr = (fst (e lab st dep)++[Write,Pop],st,adr)
```

Zielcodeerläuterung:

code Zielcode für den Ausdruck *e*, der mit einem Befehl zur Kellerung des aktuellen Wertes von *e* schließt

Write Anhängen des Wertes von *e* an den Ein/Ausgabestrom *io*

Pop Entkellern des aktuellen Wertes von *e*

Übersetzung der Deklaration einer nichtfunktionalen Variable

```
vardecl :: String -> TypeDesc -> ComStack
vardecl x td _ st dep adr = ([Push $ Con 0], update st x (td, dep, adr), adr+1)
                                Reservierung eines Kellerplatzes für Werte von x
```

Außerdem trägt vardecl den zum Typ von x gehörigen Typdeskriptor sowie dep (aktuelle Schachtelungstiefe) und adr (nächste freie Relativadresse) unter x in die Symboltabelle ein.

Übersetzung einer Funktions- oder Prozedurdeklaration

```
fundecl :: String -> FormsStack -> TypeDesc -> ComStack -> ComStack
fundecl f pars td body lab st dep adr = (code', st1, adr+1)
    where codelab = lab+2
          st1 = update st f (Fun td codelab, dep, adr)
          dep' = dep+1
          (parcode, st2, _) = pars st1 dep' $ -2
          coms = foldl1 seq_ $ parcode++[body]
          bodylab = codelab+dep+2
          (code, _, local) = coms bodylab st2 dep' dep'
          retlab = Dex TOP $ -1
          exit = bodylab+length code+local+1
          code' = Push (Con 0):Jump (Con exit):
codelab: Move TOP BA:pushDisplay dep STP++
bodylab: code++replicate local Pop++[Jump retlab]
exit:
```

Zielcodeerläuterung:

Push \$ Con 0	<i>Reservierung eines Kellerplatzes für den Wert eines Aufrufs von f</i>
Jump \$ Con exit	<i>Sprung hinter den Zielcode</i>

Der Code zwischen *codelab* und *exit* wird erst bei der Ausführung des Zielcodes eines Aufrufs von *f* abgearbeitet (siehe *applyFun*):

Move TOP BA	<i>Die nächste freie Kellerposition wird zur neuen Basisadresse. Dieser Befehl hat die von fundecl berechnete Nummer codelab und wird angesprungen, wenn der Befehl Jump codelab des Zielcodes eines Aufrufs von <i>f</i> ausgeführt wird (siehe applyProc).</i>
pushDisplay dep STP	<i>Kellern des Displays des umfassenden Prozedurrumpfs und dessen Adresse</i>
parcode++[body]	<i>erweiterter Prozedurrumpf (siehe formals)</i>
replicate local Pop	<i>Entkellern der lokalen Variablen und des Displays des Aufrufs von <i>f</i></i>
Jump retlab	<i>Sprung zur Rücksprungadresse, die bei der Ausführung des Zielcodes des Aufruf von <i>f</i> vor dem Sprung zur Adresse des Codes von <i>f</i> gekellert wurde, so dass sie am Ende von dessen Ausführung wieder oben im Keller steht</i>

Übersetzung formaler Parameter

```
formals :: [Formal] -> FormsStack
formals pars st dep adr = foldl f ([] ,st,adr) pars
    where f (cs,st,adr) (Par x td) = (cs,update st x (td,dep,adr'),adr')
                                                where adr' = adr-1
```

```

f (cs,st,adr) (FunPar x@('0':g) pars td) = (cs++[c],st',adr')      (2)
    where adr' = adr-3
          st' = update st x (ForFun td,dep,adr')
          c = fundecl g (formals pars) td $ assign g $
                applyFun x $ actuals $ map act pars
          act (Par x _)           = var x
          act (FunPar (_:g) _ _) = var g

```

Formale Parameter einer Prozedur g sind Variablen mit *negativen* Relativadressen, weil ihre aktuellen Werte beim Aufruf von g direkt *vor* dem für den Aufruf reservierten Kellerabschnitt abgelegt werden.

Eine Variable vom Typ INT (Fall 1) benötigt einen Kellerplatz für ihre aktuelle Basisadresse, eine Prozedurvariable (Fall 2) hingegen drei:

- einen für die aktuelle Basisadresse, die hier die Anfangsadresse des dem aktuellen Prozeduraufruf zugeordneten Kellerabschnitt ist,
- einen für die aktuelle Codeadresse und
- einen für die aktuelle **Resultatadresse**, das ist die Position des Kellerplatzes mit dem Wert des aktuellen Prozeduraufrufs.

Der Compiler **formal** (siehe [Java2.hs](#)) erkennt einen formalen Funktions- oder Prozedurparameter g an dessen Parameterliste $pars$ und speichert diesen unter dem Namen $@g$. **formals** erzeugt den attribuierten Code **c** der Funktionsdeklaration $g(pars)\{g = @g(pars)\}$ und übergibt ihn als Teil von **parcode** an die Deklaration des statischen Vorgängers von g (s.u.).

Damit werden g feste Basis-, Code- und Resultatadressen zugeordnet, so dass nicht nur die Aufrufe von g , sondern auch die Zugriffe auf g als Variable korrekt übersetzt werden können.

Übersetzung von Variablenzugriffen

```
var :: String -> ExpStack
var x _ st dep = case td of Fun _ codelab -> ([Push ba,Push $ Con codelab,      (1)
                                                PushA $ Dex ba adr],td)
                                -> ([Push $ Dex ba adr],td)          (2)
where (td,declDep,adr) = st x
      ba = baseAddr declDep dep
```

Zielcodeerläuterung:

Im Fall (1) ist x ein aktueller Parameter eines Aufrufs $e = g(e_1, \dots, e_n)$ einer Prozedur g , d.h. es gibt $1 \leq i \leq n$ mit $e_i = x$, und x ist selbst der Name einer Prozedur! Im Quellprogramm geht e eine Deklaration von g voran, deren i -ter formaler Parameter eine Prozedurvariable f ist. f wird bei der Ausführung des Codes für e durch x aktualisiert, indem die drei von (siehe **formals**) für f reservierten Kellerplätze mit den o.g. drei Wertkomponenten von x belegt werden (siehe parcode in **applyFun**). Beim Aufruf von x werden die drei Komponenten gekellert:

Push ba	<i>Kellern der Basisadresse ba von x</i>
Push \$ Con codelab	<i>Kellern der Codeadresse codelab von x</i>
PushA \$ Dex ba adr	<i>Kellern der Resultatadresse Dex ba adr von x, die sich aus der Basisadresse ba von x und der von fundecl bzw. formals in die Symboltabelle eingetragenen Relativadresse adr für das Resultat von e ergibt</i>

Im Fall (2) genügt ein Befehl:

Push \$ Dex ba adr *Kellern des Wertes von x, der unter der Kelleradresse Dex ba adr steht, die sich aus der Basisadresse ba von x und der von vardecl bzw. formals in die Symboltabelle eingetragenen Relativadresse adr von x ergibt*

Übersetzung von Prozeduraufrufen

```
applyFun :: String -> ActsStack -> ExpStack
applyFun f acts lab st dep =
    case td of Fun td codelab -> (code ba (Con codelab) $ Dex ba adr,      (1)
                                         td)
               ForFun td      -> (code (Dex ba adr) (Dex ba $ adr+1)      (2)
                                         $ Dex (Dex ba $ adr+2) 0,
                                         td)
    where (td,declDep,adr) = st f
          (parcode,parLg) = acts lab st dep
          retlab = lab+length parcode+4
          ba = baseAddr declDep dep
          code ba codelab result = parcode++Push BA:Move ba STP:
                                         Push (Con retlab):Jump codelab:
          retlab: Pop:Save BA:Pop:replicate parLg Pop++
                                         [Push result]
```

Zielcodeerläuterung:

parcode	<i>Zielcode für die Aufrufparameter</i>
Push BA	<i>Kellern der aktuellen Basisadresse</i>
Move ba STP	<i>Speichern der Basisadresse von f im Register STP</i>
Push \$ Con retlab	<i>Kellern der Rücksprungadresse <code>retlab</code></i>
Jump codelab	<i>Sprung zur Codeadresse <code>codelab</code> von f, die von <code>fundecl</code> berechnet wurde</i>

An dieser Stelle wird bei der Ausführung des Zielcodes zunächst $\text{code}(f)$ abgearbeitet (siehe `fundecl`). Dann geht es weiter mit:

Pop	<i>Entkellern der Rücksprungadresse <code>retlab</code></i>
Save BA	<i>Speichern der alten Basisadresse in <code>BA</code></i>
Pop	<i>Entkellern der alten Basisadresse</i>
replicate parLg Pop	<i>Entkellern der Aufrufparameter</i>
Push result	<i>Kellern des Aufrufwertes</i>

Der Zielcode der Parameterliste $acts$ setzt sich aus dem Zielcode der einzelnen Ausdrücke von $acts$ zusammen, wobei $acts$ von hinten nach vorn abgearbeitet wird, weil in dieser Reihenfolge Speicherplatz für die entsprechenden formalen Parameter reserviert wurde (siehe `formals`).

Im Fall (1) geht dem Aufruf von f im Quellprogramm eine Deklaration von f voran, deren Übersetzung die Symboltabelle um Einträge für f erweitert hat, aus denen `applyFun` die Adressen bzw. Befehlsnummern `ba`, `codelab` und `result` berechnet und in obigen Zielcode einsetzt.

Im Fall (2) ist f eine Prozedurvariable, d.h. im Quellprogramm kommt der Aufruf von f im Rumpf der Deklaration einer Prozedur g vor, die f als formalen Parameter enthält.

Der Zielcode wird hier erst bei einem Aufruf von g ausgeführt, d.h. nachdem f durch eine Prozedur h aktualisiert und die von **formals** für f reservierten Kellerplätze belegt wurden. Der Zielcode des Aufrufs von f ist also in Wirklichkeit Zielcode für einen Aufruf von h . Dementsprechend sind ba die Anfangsadresse des dem Aufruf zugeordneten Kellerabschnitts und damit

Dex ba adr, Dex ba \$ adr+1 und Dex ba \$ adr+2

die Positionen der Kellerplätze mit der Basisadresse, der Codeadresse bzw. der Resultatadresse von h .

Folglich kommt **applyFun** durch Zugriff auf diese Plätze an die aktuellen Werte von ba , $codelab$ und $result$ heran und kann sie wie im Fall (1) in den Zielcode des Aufrufs einsetzen.

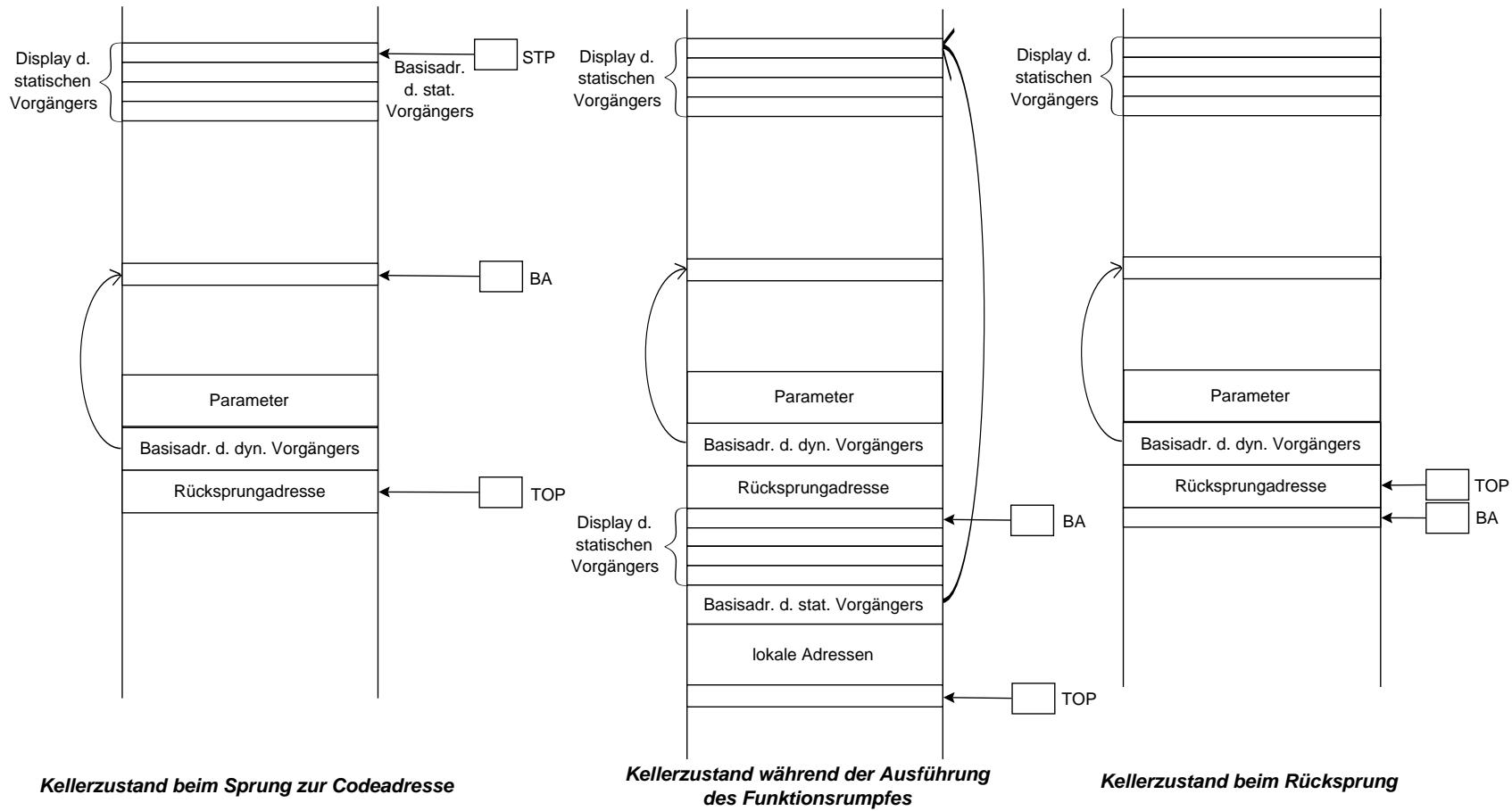
Damit **Push result** analog zum Fall (1) nicht die Resultatadresse von h , sondern den dort abgelegten Wert kellert, muss sie vorher derefenziert werden. Deshalb wird **result** im Fall (2) auf

Dex (Dex ba \$ adr+2) 0

gesetzt.

Den Aufruf einer Prozedur p ohne Rückgabewert (genauer gesagt: mit Rückgabewert vom Typ UNIT) betten wir in eine Zuweisung an p ein:

```
applyProc :: String -> ActsStack -> ComStack
applyProc f p = assign p . applyFun p
```



Kellerzustand beim Sprung zur Codeadresse

**Kellerzustand während der Ausführung
des Funktionsrumpfes**

Kellerzustand beim Rücksprung

Übersetzung aktueller Parameter

```
actuals :: [ExpStack] -> ActsStack
actuals pars lab st dep = foldr f ([] ,0) pars
    where f e (code,parLg) = (code++code',parLg+ case td of Fun _ _ -> 3
                                _ -> 1)
          where (code',td) = e (lab+length code) st dep
```

In [29], Kapitel 5, wird auch die Übersetzung von Feldern und Records behandelt.

Grundlagen der Kompilation funktionaler Sprachen liefert [29], Kapitel 7.

Die Übersetzung objektorientierter Sprachen ist Thema von [46], Kapitel 5.

Beispiel 13.4 (Vier JavaLight+-Programme für die Fakultätsfunktion aus [Java2.hs](#))

```
Int x; read x; Int fact; fact=1;  
while x>1 {fact=x*fact; x=x-1;} write fact;
```

```
Int f(Int x) {if x<2 f=1; else f=x*f(x-1);}  
Int x; read x; write f(x);
```

```
f(Int x,Int fact) {if x<2 write fact; else f(x-1,fact*x)}  
Int x; read x; f(x,1)
```

```
Int f(Int x,Int g(Int x,Int y)) {if x<2 f=1; else f=g(x,f(x-1,g));}  
Int g(Int x,Int y) {g=x*y;}  
Int x; read x; write f(x,g);
```

`javaToStack "prog" [n]` übersetzt jedes der obigen Programme in eine Befehlsfolge vom Typ `[StackCom]`, legt diese in der Datei `javacode` ab und transformiert die Eingabeliste `[n]` in die Ausgabeliste `[n!]`.

Der Zielcode des vierten Programms lautet wie folgt:

```
0: Push (Con 0)  
1: Jump (Con 68)
```

```
2: Move TOP BA           begin f
3: Push STP
4: Push (Con 0)
5: Jump (Con 26)
6: Move TOP BA           begin g
7: Push (Dex STP 0)
8: Push STP
9: Push (Dex BA (-4))    y
10: Push (Dex BA (-3))   x
11: Push BA
12: Move (Dex (Dex BA 1) (-6)) STP
13: Push (Con 15)
14: Jump (Dex (Dex BA 1) (-5))
15: Pop
16: Save BA
17: Pop
18: Pop
19: Pop
20: Push (Dex (Dex (Dex BA 1) (-4)) 0)
21: Save (Dex (Dex BA 1) 1)   g=@g(x,y)
22: Pop
23: Pop
24: Pop
25: Jump (Dex TOP (-1))      end g
26: Push (Dex BA (-3))      x
```

27: Push (Con 2)
28: Cmp "<" $x < \mathcal{Z}$
29: JumpF 34
30: Push (Con 1)
31: Save (Dex (Dex BA 0) 0) $f=1$
32: Pop
33: Jump (Con 65)
34: Push BA g
35: Push (Con 6) g
36: PushA (Dex BA 1) g
37: Push (Dex BA (-3)) x
38: Push (Con 1)
39: Sub $x-1$
40: Push BA
41: Move (Dex BA 0) STP
42: Push (Con 44)
43: Jump (Con 2) *jump to f*
44: Pop
45: Save BA
46: Pop
47: Pop
48: Pop
49: Pop
50: Pop
51: Push (Dex (Dex BA 0) 0) $f(x-1, g)$

52: Push (Dex BA (-3))	x
53: Push BA	
54: Move BA STP	
55: Push (Con 57)	
56: Jump (Con 6)	<i>jump to g</i>
57: Pop	
58: Save BA	
59: Pop	
60: Pop	
61: Pop	
62: Push (Dex BA 1)	$g(x, f(x-1, g))$
63: Save (Dex (Dex BA 0) 0)	$f = g(x, f(x-1, g))$
64: Pop	
65: Pop	
66: Pop	
67: Jump (Dex TOP (-1))	<i>end f</i>
68: Push (Con 0)	
69: Jump (Con 79)	
70: Move TOP BA	<i>begin g</i>
71: Push STP	
72: Push (Dex BA (-3))	x
73: Push (Dex BA (-4))	y
74: Mul	x^*y
75: Save (Dex (Dex BA 0) 1)	$g = x^*y$
76: Pop	

77: Pop
78: Jump (Dex TOP (-1)) *end g*
79: Push (Con 0)
80: Read (Dex BA 2) *read x*
81: Push BA *g*
82: Push (Con 70) *g*
83: PushA (Dex BA 1) *g*
84: Push (Dex BA 2) *x*
85: Push BA
86: Move BA STP
87: Push (Con 89)
88: Jump (Con 2) *jump to f*
89: Pop
90: Save BA
91: Pop
92: Pop
93: Pop
94: Pop
95: Pop
96: Push (Dex BA 0) *f(x,g)*
97: Write *write f(x,g)*
98: Pop

14 Mehrpässige Compiler

Sei $G = (S, BS, R)$ eine CFG und $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$ -Algebra.

Wir kommen zurück auf die in Kapitel 12 behandelte Form

$$A_{v_1} \times \dots \times A_{v_m} \rightarrow A_{a_1} \times \dots \times A_{a_n}$$

der Trägermengen von A und sehen uns das Schema der Definition einer Operation von \mathcal{A} mal etwas genauer an. O.B.d.A. setzen wir für die allgemeine Betrachtung voraus, dass alle Trägermengen von A miteinander übereinstimmen.

Sei $c : s_1 \times \dots \times s_k \rightarrow s$ ein Konstruktor von $\Sigma(G)$. Das Schema einer Interpretation $c^A : A^k \rightarrow A$ von c in A offenbar wie folgt:

$$\begin{aligned} c^A(f_1, \dots, f_k)(x_1, \dots, x_m) &= (t_1, \dots, t_n) \quad \text{where} \quad (x_{11}, \dots, x_{1n}) = f_1(t_{11}, \dots, t_{1m}) \\ &\quad \vdots \\ &\quad (x_{k1}, \dots, x_{kn}) = f_k(t_{k1}, \dots, t_{km}) \end{aligned} \tag{1}$$

Hier sind $f_1, \dots, f_k, x_1, \dots, x_m, x_{i1}, \dots, x_{1n}, \dots, x_{k1}, \dots, x_{kn}$ paarweise verschiedene Variablen und $e_1, \dots, e_n, x_{i1}, \dots, t_{1m}, \dots, t_{k1}, \dots, t_{km}$ $\Sigma(G)$ -Terme.

Ist $s \rightarrow w_0 s_1 w_1 \dots s_k w_k$ die Grammatikregel, aus der c hervorgeht, dann wird (1) oft als Liste von Zuweisungen an die Attribute der Sorten s, s_1, \dots, s_k geschrieben:

$$\begin{array}{ll} s.a_1 := t_1 & \dots \\ s.a_n := t_n & s_1.v_1 := t_{11} \dots s_1.v_m := t_{1m} \\ & \dots \\ & s_k.v_1 := t_{k1} \dots s_k.v_m := t_{km} \end{array}$$

c^A ist genau dann wohldefiniert, wenn für alle $f_1, \dots, f_k, x_1, \dots, x_m$ (1) eine Lösung in A hat, d.h., wenn es eine Belegung g der Variablen $x_{i1}, \dots, x_{in}, \dots, x_{k1}, \dots, x_{kn}$ in A gibt mit

$$(g(x_{i1}), \dots, g(x_{in})) = f_1(g^*(t_{i1}), \dots, g^*(t_{im}))$$

für alle $1 \leq i \leq k$. Hinreichend für die Lösbarkeit ist die Zyklusfreiheit der Benutzt-Relation zwischen den Termen und Variablen von (1). Enthält sie einen Zyklus, dann versucht man, die parallele Berechnung und Verwendung von Attributwerten in r hintereinander ausgeführte Schritte (“Pässe”) zu zerlegen, so dass in jedem Schritt zwar nur einige Attribute berechnet bzw. benutzt werden, die Komposition der Schritte jedoch eine äquivalente Definition von c^A liefert.

Notwendig wird die Zerlegung der Übersetzung zum Beispiel dann, wenn die Quellsprache Deklarationen von Variablen verlangt, diese aber im Text des Quellprogramms bereits vor ihrer Deklaration benutzt werden dürfen.

Zerlegt werden müssen die Menge $At = \{v_1, \dots, v_m, a_1, \dots, a_n\}$ aller Attribute in r Teilmengen At_1, \dots, At_r sowie jedes (funktionale) Argument f_i von c^A , $1 \leq i \leq k$, in r Teilfunktionen f_i^1, \dots, f_i^r derart, dass die *Benutzt-Relation* in jedem Pass azyklisch ist.

Um die Benutzt-Relation zu bestimmen, erweitern wir die Indizierung der äußeren Variablen bzw. Terme von (1) wie folgt:

$$\begin{aligned}
 c^A(f_1, \dots, f_k)(x_{01}, \dots, x_{0m}) &= (t_{(k+1)1}, \dots, t_{(k+1)n}) \\
 \text{where } (x_{11}, \dots, x_{1n}) &= f_1(t_{11}, \dots, t_{1m}) \\
 &\vdots \\
 (x_{k1}, \dots, x_{kn}) &= f_k(t_{k1}, \dots, t_{km})
 \end{aligned} \tag{2}$$

Für jeden Konstruktor c und jedes Attributpaar (at, at') ist der **Abhängigkeitsgraph**

$$depgraph(c)(at, at') \subseteq \{0, \dots, k\} \times \{1, \dots, k+1\}$$

für c und (at, at') wie folgt definiert:

$$\begin{aligned}
& \exists 1 \leq i \leq m, 1 \leq j \leq n : at = v_i \wedge at' = a_j && (\text{at vererbt, } at' \text{ abgeleitet}) \\
& \Rightarrow depgraph(c)(at, at') = \begin{cases} \{(0, k+1)\} & \text{falls } x_{0i} \text{ in } t_{(k+1)j} \text{ vorkommt,} \\ \emptyset & \text{sonst,} \end{cases} \\
& \exists 1 \leq i, j \leq m : at = v_i \wedge at' = v_j && (\text{at und } at' \text{ vererbt}) \\
& \Rightarrow depgraph(c)(at, at') = \{(0, s) \mid 0 \leq s \leq k, x_{0i} \text{ kommt in } t_{sj} \text{ vor}\}, \\
& \exists 1 \leq i \leq n, 1 \leq j \leq m : at = a_i \wedge at' = v_j && (\text{at abgeleitet, } at' \text{ vererbt}) \\
& \Rightarrow depgraph(c)(at, at') = \{(r, s) \mid 0 \leq r, s \leq k, x_{ri} \text{ kommt in } t_{sj} \text{ vor}\}, \\
& \exists 1 \leq i, j \leq n : at = a_i \wedge at' = a_j && (\text{at und } at' \text{ abgeleitet}) \\
& \Rightarrow depgraph(c)(at, at') = \{(r, k+1) \mid 0 \leq r \leq k, x_{ri} \text{ kommt in } t_{(k+1)j} \text{ vor}\}.
\end{aligned}$$

(2) hat genau dann eine Lösung in \mathcal{A} (die durch Auswertung der Terme von (2) berechnet werden kann), wenn

- für alle Konstruktoren c von $\Sigma(G)$, $at, at' \in At$ und $(i, j) \in depgraph(c)(at, at')$ $i < j$ gilt.

Ist diese Bedingung verletzt, dann wird At so in Teilmengen At_1, \dots, At_r zerlegt, dass für alle $at, at' \in At$ entweder at in einem früheren Pass als at' berechnet wird oder beide Attribute in demselben Pass berechnet werden und die Bedingung erfüllen.

Für alle $1 \leq p, q \leq r$, $at \in At_p$ und $at' \in At_q$ muss also Folgendes gelten:

$$p < q \vee (p = q \wedge \forall (i, j) \in depgraph(c)(at, at') : i < j). \quad (3)$$

Für alle $1 \leq p \leq r$ und $1 \leq i \leq k$ sei

$$\{v_{i_{p1}}, \dots, v_{i_{pm_p}}\} = \{v_1, \dots, v_m\} \cap At_p, \quad \{a_{j_{p1}}, \dots, a_{j_{pn_p}}\} = \{a_1, \dots, a_n\} \cap At_p,$$

$$\begin{aligned} \pi_p : A_{v_1} \times \dots \times A_{v_m} &\rightarrow A_{v_{i_{p1}}} \times \dots \times A_{v_{i_{pm_p}}} \\ (x_1, \dots, x_m) &\mapsto (x_{i_{p1}}, \dots, x_{i_{pm_p}}), \\ \pi'_p : A_{a_1} \times \dots \times A_{a_n} &\rightarrow A_{a_{j_{p1}}} \times \dots \times A_{a_{j_{pn_p}}} \\ (x_1, \dots, x_n) &\mapsto (x_{j_{p1}}, \dots, x_{j_{pn_p}}) \end{aligned}$$

und $f_i^p : A_{v_{i_1}} \times \dots \times A_{v_{i_{mp}}} \rightarrow A_{a_{j_1}} \times \dots \times A_{a_{j_{np}}}$ die eindeutige Funktion, die das folgende Diagramm kommutativ macht:

$$\begin{array}{ccc} A_{v_1} \times \dots \times A_{v_m} & \xrightarrow{\pi_p} & A_{v_{i_{p1}}} \times \dots \times A_{v_{i_{pm_p}}} \\ f_i \downarrow & & \downarrow f_i^p \\ A_{a_1} \times \dots \times A_{a_n} & \xrightarrow{\pi'_p} & A_{a_{j_{p1}}} \times \dots \times A_{a_{j_{pn_p}}} \end{array}$$

Die zu (2) äquivalente Definition von c^A mit r Pässen lautet wie folgt:

$$\begin{aligned}
 c^A(f_1, \dots, f_k)(x_{01}, \dots, x_{0m}) &= (t_{(k+1)1}, \dots, t_{(k+1)n}) \\
 \text{where } (x_{1j_{11}}, \dots, x_{1j_{1n_1}}) &= f_1^1(t_{1i_{11}}, \dots, t_{1i_{1m_1}}) \\
 &\vdots && \text{Pass 1} \\
 (x_{kj_{11}}, \dots, x_{kj_{1n_1}}) &= f_k^1(t_{ki_{11}}, \dots, t_{ki_{1m_1}}) \\
 &\vdots \\
 &\vdots \\
 (x_{1j_{r1}}, \dots, x_{1j_{rn_r}}) &= f_1^r(t_{1i_{r1}}, \dots, t_{1i_{rm_r}}) \\
 &\vdots && \text{Pass } r \\
 (x_{kj_{r1}}, \dots, x_{kj_{rn_r}}) &= f_k^r(t_{ki_{r1}}, \dots, t_{ki_{rm_r}})
 \end{aligned}$$

Der **LAG-Algorithmus** (*Left-to-right-Attributed-Grammar*) berechnet die kleinste Zerlegung von At , die (3) erfüllt, sofern eine solche existiert.

Sei $ats = At$ und $constrs$ die Menge aller Konstruktoren von $\Sigma(G)$. Ausgehend von der ein-elementigen Zerlegung $[ats]$ verändert *check_partition* die letzten beiden Elemente (*curr* und *next*) der jeweils aktuellen Zerlegung, bis entweder *next* leer und damit eine Zerlegung gefunden ist, die (3) erfüllt, oder *curr* leer ist, was bedeutet, dass keine solche Zerlegung existiert.

```
least_partition ats = reverse . check_partition [] [ats]
```

```
check_partition next (curr:partition) =
```

```
if changed
```

```
then check_partition next' (curr':partition)
```

```
else case (next',curr') of
```

```
([],_) -> curr':partition      Zerlegung von ats, die (3) erfüllt
```

```
(_,[]) -> []                    Es gibt keine Zerlegung, die (3) erfüllt.
```

```
_ -> check_partition [] (next':curr':partition)
```

Die aktuelle Zerlegung wird um das Element nextfl erweitert.

```
where (next',curr',changed) = foldl check_constr (next,curr,False)  
                                  constrs
```

```
check_constr state c = foldl (check_atpair deps) state
```

```
[(at,at') | at <- ats, at' <- ats,
```

```
                              not $ null $ deps (at,at')]
```

```
where deps = depgraph c
```

```
check_atpair deps state atpair = foldl (check_dep atpair) state $
```

```
                              deps atpair
```

```
check_dep (at,at') state@(next,curr,changed) (i,j) =  
  if at` `elem` curr && ((at ` `elem` curr && i>=j) || at ` `elem` next)  
    Die aktuelle Zerlegung next:curr:... verletzt (3).  
  then (at':next,curr`minus`[at'],True)  
    atfl wird vom vorletzten Zerlegungselement (curr) zum letzten (next) verschoben.  
  else state
```

15 Funktoren und Monaden in Haskell

Typen und Typvariablen höherer Ordnung

Während bisher nur Typvariablen erster Ordnung vorkamen, sind Funktoren und Monaden Instanzen der Typklasse **Functor** bzw. **Monad**, die eine Typvariable zweiter Ordnung enthält. Typvariablen erster Ordnung werden durch Typen erster Ordnung instanziert wie z.B. **Int** oder **Bool**. Typvariablen zweiter Ordnung werden durch Typen zweiter Ordnung instanziert wie z.B. durch den folgenden:

```
data Tree a = F a [Tree a] | V a
```

Demnach ist ein Typ erster Ordnung eine Menge und ein Typ zweiter Ordnung eine Funktion von einer Menge von Mengen in eine – i.d.R. andere – Menge von Mengen: **Tree** bildet jede Menge A auf eine Menge von Bäumen ab, deren Knoteneinträge Elemente von A sind.

Funktoren

Die meisten der in Abschnitt 5.1 definierten Funktoren sind in Haskell standardmäßig als Instanzen der Typklasse **Functor** implementiert:

```
class Functor f where fmap :: (a -> b) -> f a -> f b
```

```
newtype Id a = Id {run :: a}
```

Identitätsfunktor

```
instance Functor Id where fmap h (Id a) = Id $ h a
```

```
instance Functor [] where fmap = map
```

Listenfunktor

```
data Maybe a = Just a | Nothing
```

Ausnahmefunktoren

```
instance Functor Maybe where
```

```
    fmap f (Just a) = Just $ f a  
    fmap _ _        = Nothing
```

```
data Either e a = Left e | Right a
```

```
instance Functor Either e where
```

```
    fmap f (Right a) = Right $ f a  
    fmap _ e        = e
```

```
instance Functor ((->) state) where  
    fmap f h = f . h
```

Leserfunktor

```
instance Functor ((,) state) where  
    fmap f (st,a) = (st,f a)
```

Schreiberfunktor

Transitionsfunktoren

```
newtype Trans state a = T {runT :: state -> (a,state)}
```

```
instance Functor (Trans state) where  
    fmap f (T h) = T $ (\(a,st) -> (f a,st)) . h
```

```
newtype TransM state m a = TM {runTM :: state -> m (a,state)}
```

```
instance Monad m => Functor (TransM state m) where  
    fmap f (TM h) = TM $ (>>= \(a,st) -> return (f a,st)) . h
```

Anforderungen an die Instanzen der Typklasse Functor:

Für alle Mengen $a, b, c, f : a \rightarrow b$ und $g : b \rightarrow c$,

$$\begin{aligned} \text{fmap id} &= \text{id} \\ \text{fmap } (f . g) &= \text{fmap } f . \text{fmap } g \end{aligned}$$

Monaden

Monad ist eine Unterklasse von Functor:

```
class Functor m => Monad m where
    return :: a -> m a
    (=>)   :: m a -> (a -> m b) -> m b
    (++)    :: m a -> m b -> m b
    fail    :: String -> m a
    m >> m' = m >=> const m'
```

```
instance Monad Id where return = Id
                    Id a >=> f = f a
```

```
instance Monad [] where return a = [a]
                    (=>) = flip concatMap
                    fail _ = []
```

```
instance Monad Maybe where return = Just
                    Just a >=> f = f a
                    e >=> _      = e
                    fail _ = Nothing
```

Einheit η

bind-Operatoren

Wert im Fall eines Matchfehlers

Identitätsmonade

Listenmonade

Ausnahmemonaden

```
instance Monad (Either e) where return = Right
                    Right a >>= f = f a
                    e >>= _      = e
```

```
instance Monad ((->) state) where return = const           Lesermonade
                                         (h >>= f) st = f (h st) st
```

```
class Monoid a where                                Schreibermonade
    mempty :: a; mappend :: a -> a -> a
```

Anforderungen an die Instanzen von Monoid:

$$\begin{aligned} (a \text{ `mappend` } b) \text{ `mappend` } c &= a \text{ `mappend` } (b \text{ `mappend` } c) \\ \text{mempty} \text{ `mappend` } a &= a \\ a \text{ `mappend` } \text{mempty} &= a \end{aligned}$$

```
instance Monoid state => Monad ((,) state) where
    return a = (mempty, a)
    (st, a) >>= f = (st `mappend` st', b) where (st', b) = f a
```

```

instance Monad (Trans state) where
    return a = T $ \st -> (a,st)
    T h >>= f = T $ (\(a,st) -> runT (f a) st) . h

instance Monad m => Monad (TransM state m) where
    return a = TM $ \st -> return (a,st)
    TM h >>= f = TM $ (>>= \(a,st) -> runTM (f a) st) . h

```

Hier komponiert der bind-Operator $\gg=$ zwei Zustandstransformationen (h und dann f) sequentiell. Dabei liefert die von der ersten erzeugte Ausgabe die Eingabe der zweiten.

$Trans(state)$ und $TransM(state)$ werden auch **Zustandsmonade** bzw. (Zustands-)**Monadentransformer** genannt. Wir bevorzugen den Begriff *Transitionsmonade*, weil Zustände auch zu Leser- und Schreibermonaden gehören (s.o.), aber nur *Transitionsmonaden* aus *Zustandstransformationen* bestehen.

Weitere Bemerkungen zu Monaden in Haskell finden sich in Abschnitt 7.3 (*Monaden und Plusmonaden*) von [30].

Monaden-Kombinatoren

```
sequence :: Monad m => [m a] -> m [a]
sequence (m:ms) = do a <- m; as <- sequence ms; return $ a:as
sequence _      = return []
```

sequence(ms) führt die Prozeduren der Liste *ms* hintereinander aus. Wie bei *some(m)* und *many(m)* werden die dabei erzeugten Ausgaben aufgesammelt.

Die *do*-Notation für monadische Ausdrücke wurde in Kapitel 6 eingeführt.

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (">>>") $ return ()
```

sequence_(ms) arbeitet wie *sequence(ms)*, vergisst aber die erzeugten Ausgaben.

Die folgenden Funktionen führen die Elemente mit **map** bzw. **zipWith** erzeugter Prozedurlisten hintereinander aus:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f = sequence . map f
```

```
mapM_ :: Monad m => (a -> m b) -> [a] -> m ()
mapM_ f = sequence_ . map f
```

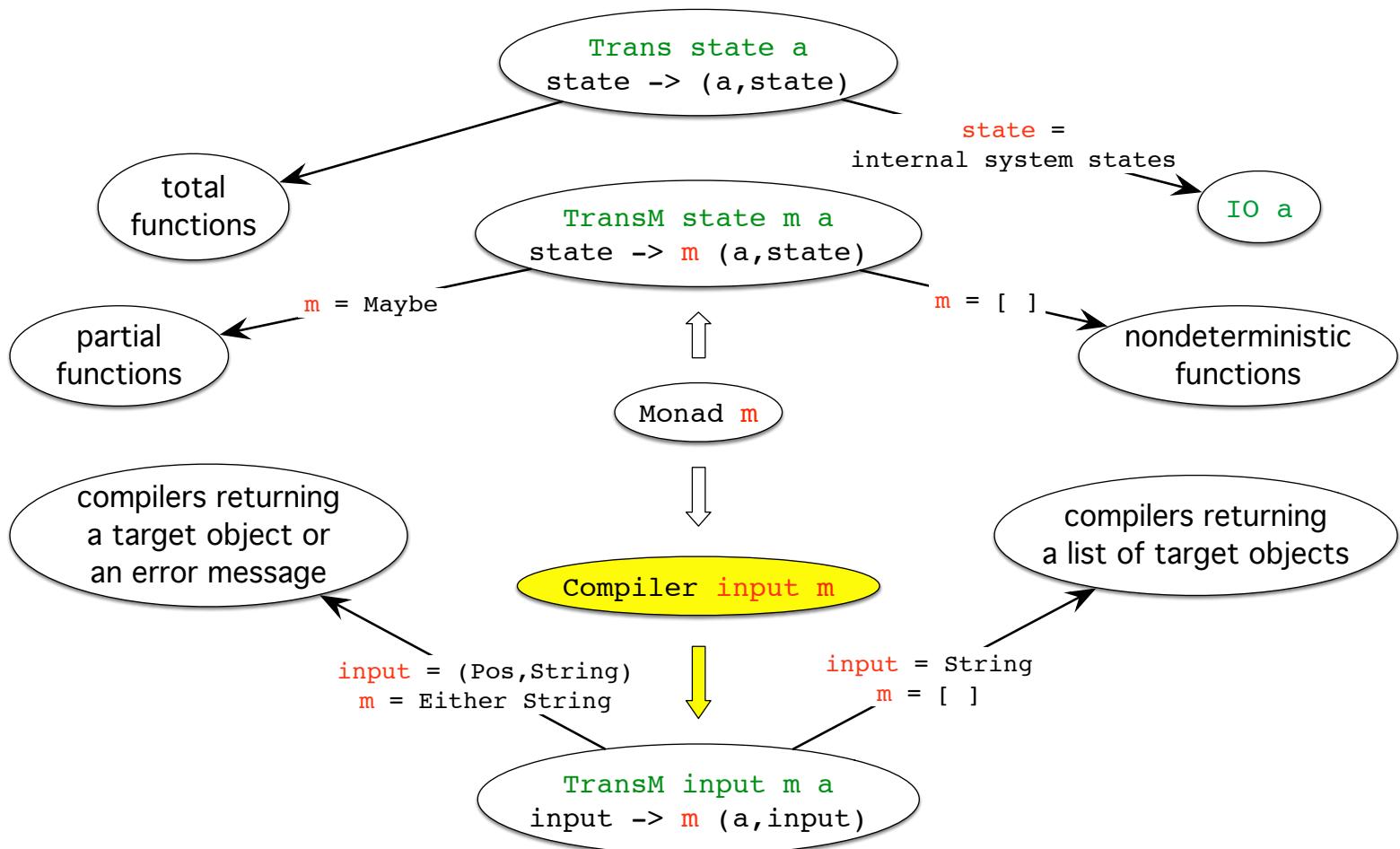
liftM2 liftet eine zweistellige Funktion $f : A \rightarrow (B \rightarrow C)$ zu einer Funktion des Typs $M(A) \rightarrow (M(B) \rightarrow M(C))$:

```
liftM2 :: Monad m => (a -> b -> c) -> m a -> m b -> m c  
liftM2 f ma mb = do a <- ma; b <- mb; return $ f a b
```

liftM2 gehört zum ghc-Modul **Control.Monad**.

Wir werden im folgenden Kapitel eine auf die Implementierung von Compilern zugeschnittene Unterklasse von *Monad* einführen, die Bedingungen nicht nur an die Menge *state* der Zustände (die dort den möglichen Compiler-Eingaben entsprechen), sondern auch an die eingebettete Monade *m* stellt.

16 Monadische Compiler



Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG, $X = \bigcup BS$ und M eine Compilermonade (siehe Kapitel 5).

Ein generischer Compiler

$$\text{compile}_G = (\text{compile}_G^A : X^* \rightarrow M(A))_{A \in \text{Alg}_{\Sigma}(G)},$$

genauer gesagt: die von compile_G verwendeten Transitionsfunktionen

$$\text{trans}_s^A : X^* \rightarrow M(A_s \times X^*), \quad s \in S \cup BS, A \in \text{Alg}_{\Sigma}(G),$$

ließen sich als Transitionsmonaden vom Typ $\text{TransM}(\text{String})(M)$ implementieren.

Oft erfordern die Übersetzung oder auch die Ausgabe differenzierter Fehlermeldungen zusätzliche Informationen über die Eingabe wie z.B. Zeilen- und Spaltenpositionen von Zeichen des Eingabetextes, um Syntaxfehler den Textstellen, an denen sie auftreten können, zuzuordnen. Den Eingabetypr auf String zu beschränken ist dann nicht mehr adäquat.

Deshalb ersetzen wir String durch die Typvariable input und fassen die erforderlichen, input bzw. die Compilermonade M betreffenden Grundfunktionen in folgender Unterklasse von Monad zusammen. Die Compiler selbst werden dann als Objekte vom Typ $\text{TransM}(\text{input})(M)$ implementiert.

```
class Monad m => Compiler input m | input -> m, m -> input where
    errmsg :: input -> m a
```

```

empty   :: input -> Bool
ht      :: input -> m (Char, input)
                    (erstes Eingabezeichen, Resteingabe)
plus    :: m a -> m a -> m a

```

errmsg behandelt fehlerhafte Eingaben. *empty* prüft, ob die Eingabe leer ist. Wenn nicht, dann liefert *ht* das erste Zeichen der Eingabe und die Resteingabe.

Kommen im Typ einer Funktion einer Typklasse nicht alle Typvariablen der Klasse vor, dann müssen die fehlenden von den vorkommenden abhängig gemacht werden. So sind z.B. in der Typklasse *Compiler* die Abhängigkeiten $input \rightarrow m$ und $m \rightarrow input$ der Funktion *empty* bzw. *plus* geschuldet.

Die Menge der im Programm definierten Instanzen einer Typklasse muss deren funktionale Abhängigkeiten tatsächlich erfüllen. Zwei Instanzen von *Compiler* mit derselben *input*-Instanz müssen also auch dieselbe *m*-Instanz haben.

Zwei Instanzen der Typklasse *Compiler*

Für mehrere korrekte Ausgaben, aber nur eine mögliche Fehlermeldung:

```

instance Compiler String [ ] where
  errmsg _ = []

```

```

empty = null
ht (c:str) = [(c,str)]
plus = (++)

```

Für höchstens eine korrekte Ausgabe, aber mehrere mögliche Fehlermeldungen:

```
type Pos = (Int,Int)
```

```

instance Compiler (Pos,String) (Either String) where
    errmsg (pos,_) = Left $ "error at position "++show pos
    empty = null . snd
    ht ((i,j),c:str) = Right (c,(pos,str)) where
        pos = if c == '\n' then (i+1,1)
              else (i,j+1)
    Left _ `plus` m = m
    m `plus` _       = m

```

Hier sind die Eingaben vom Typ *(Pos, String)*. Am Argument *((i,j), c:str)* von *ht* erkennt man, dass, bezogen auf die gesamte Eingabe, *c* in der *i*-ten Zeile und *j*-ten Spalte steht. Folglich ist *pos* im Wert *Right (c, (pos, str))* von *ht((i,j), c:str)* die Anfangsposition des Reststrings *str*.

Scheitert ein Compiler dieses Typs, dann ruft er *errmsg* mit der Eingabeposition auf, an der er den Fehler erkannt hat, der ihn scheitern ließ.

16.1 Compilerkombinatoren

```
runC :: Compiler input m => TransM input m a -> input -> m a
runC comp input = do (a,input) <- runTM comp input
                     if empty input then return a else errmsg input
```

runC(comp)(input) führt den Compiler *comp* auf der Eingabe *input* aus und scheitert, falls *comp* eine nichtleere Resteingabe zurücklässt.

```
cplus :: Compiler input m => TransM input m a -> TransM input m a
                                               -> TransM input m a
TM f `cplus` TM g = TM $ liftM2 plus f g
```

```
csum :: Compiler input m => [TransM input m a] -> TransM input m a
csum = foldr1 cplus
```

```
some, many :: Compiler input m => TransM input m a
                           -> TransM input m [a]
```

```

some comp = do a <- comp; as <- many comp; return $ a:as
many comp = csum [some comp, return []]

```

some(comp) und *many(comp)* wenden den Compiler *comp* auf die Eingabe an. Akzeptiert *comp* ein Präfix der Eingabe, dann wird *comp* auf die Resteingabe angewendet und dieser Vorgang wiederholt, bis *comp* scheitert. Die Ausgabe beider Compiler ist die Liste der Ausgaben der einzelnen Iterationen von *comp*.

some(comp) scheitert, wenn bereits die erste Iteration von *comp* scheitert. *many(comp)* scheitert in diesem Fall nicht, sondern liefert die leere Liste von Ausgaben.

```

cguard :: Compiler input m => Bool -> TransM input m ()
cguard b = if b then return () else TM errmsg

```

Sind (1), (4) und (5) erfüllt, dann gelten die folgenden semantischen Äquivalenzen:

$\text{do } \text{cguard True; } m_1; \dots; m_n$	<i>ist äquivalent zu</i>	$\text{do } m_1; \dots; m_n$
$\text{do } \text{cguard False; } m_1; \dots; m_n$	<i>ist äquivalent zu</i>	m_{zero}

Lifting von $(:) : a \rightarrow [a] \rightarrow [a]$ auf Compilerebene

```

append :: Compiler input m => TransM input m a
                                -> TransM input m [a] -> TransM input m [a]
append = liftM2 (:)

```

16.2 Monadische Scanner

Scanner sind Compiler, die einzelne Symbole erkennen. Der folgende Scanner $\text{sat}(f)$ erwartet, dass das Zeichen am Anfang des Eingabestrings die Bedingung f erfüllt:

```
sat :: Compiler input m => (Char -> Bool) -> TransM input m Char
sat f = TM $ \input -> do p@(c,_) <- if empty input then errmsg input
                           else ht input
                           if f c then return p else errmsg input
```

```
char :: Compiler input m => Char -> TransM input m Char
char chr = sat (== chr)
```

```
nchar :: Compiler input m => String -> TransM input m Char
nchar chrs = sat (`notElem` chrs)
```

Darauf aufbauend, erwarten die folgenden Scanner eine Ziffer, einen Buchstaben bzw. einen Begrenzer am Anfang des Eingabestrings:

```
digit,letter,delim :: Compiler input m => TransM input m Char
digit  = csum $ map char ['0'..'9']
letter = csum $ map char $ ['a'..'z']++['A'..'Z']
delim  = csum $ map char " \n\t"
```

Der folgende Scanner `string(str)` erwartet den String `str` am Anfang des Eingabestrings:

```
string :: Compiler input m => TransM input m String
string = mapM char
```

Die folgenden Scanner erkennen Elemente von Standardtypen und übersetzen sie in entsprechende Haskell-Typen:

```
bool :: Compiler input m => TransM input m Bool
bool = csum [do string "True"; return True,
             do string "False"; return False]
```

```
nat,int :: Compiler input m => TransM input m Int
nat = do ds <- some digit; return $ read ds
int = csum [nat, do char '-'; n <- nat; return $ -n]
```

```
identifier :: Compiler input m => TransM input m String
identifier = append letter $ many $ nchar "(){}<>!=!&|+-*/^.,,:; \n\t"
```

```
relation :: Compiler input m => TransM input m String
relation = csum $ map string $ words "<= >= < > == !="
```

Die Kommas trennen die Elemente der Argumentliste von *csum*.

token(comp) erlaubt vor und hinter dem von *comp* erkannten String Leerzeichen, Zeilenumbrüche oder Tabulatoren:

```
token :: Compiler a -> Compiler a
token comp = do many delim; a <- comp; many delim; return a

tchar      = token . char
tstring    = token . string
tbool      = token bool
tint       = token int
tidentifier = token identifier
trelation   = token relation
```

16.3 Monadische LL-Compiler

Sei $G = (S, BS, R)$ eine LL-kompilierbare CFG, $X = \bigcup BS$, $G' = (S', BS, R')$ die daraus gebildete nicht-linksrekursive CFG, M eine Compilermonade, $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$ -Algebra und $s \in S \cup BS$.

Wir implementieren

$$\text{compile}_{G,s}^A : X^* \rightarrow M(A_s)$$

(siehe Kapitel 6) wie folgt: Sei $S = \{s_1, \dots, s_k\}$.

```
compile_G :: Compiler input m => Alg s1...sk -> input -> m s
compile_G = runC . trans_s
```

Für alle $B \in BS$ sei

```
trans_B :: Compiler input m => TransM input m B
```

gegeben.

Seien $s \in S'$ und r_1, \dots, r_m die Regeln von R' mit linker Seite s .

```
trans_s :: Compiler input m => TransM input m s
trans_s alg = csum [try_r1, ..., try_rm] where
    ...
    try_ri = do a1 <- trans_e1; ...; an <- trans_en
              return $ f_ri (deref alg) a_i1 ... a_ik
    ...

```

wobei $r_i = (s \rightarrow e_1 \dots e_n)$ und $\{i_1, \dots, i_k\} = \{1 \leq i \leq n \mid e_i \in S' \cup BS \setminus Z(G)\}$.

Beispiel

Aus der abstrakten Syntax der Grammatik **SAB** von Beispiel 4.5 ergibt sich die folgende Haskell-Implementierung der Klasse aller $\Sigma(\text{SAB})$ -Algebren:

```
data SAB s a b = SAB {f_1 :: b -> s, f_2 :: a -> s, f_3 :: s,
                      f_4 :: s -> a, f_5 :: a -> a -> a,
                      f_6 :: s -> b, f_7 :: b -> b -> b,}
```

Nach obigem Schema liefern die Regeln

$$\begin{aligned} r_1 &= S \rightarrow aB, & r_2 &= S \rightarrow bA, & r_3 &= S \rightarrow \epsilon, \\ r_4 &= A \rightarrow aS & r_5 &= A \rightarrow bAA, & r_6 &= B \rightarrow bS, & r_7 &= B \rightarrow aBB \end{aligned}$$

von SAB die folgende Haskell-Version des generischen SAB-Compilers von Beispiel 6.1:

```
compS :: Compiler input m => SAB s a b -> TransM input m s
compS alg = csum [do char 'a'; c <- compB alg; return $ f_1 alg c,
                  do char 'b'; c <- compA alg; return $ f_2 alg c,
                  return $ f_3 alg]

compA :: Compiler input m => SAB s a b -> TransM input m a
compA alg = csum [do char 'a'; c <- compS alg; return $ f_4 alg c,
                  do char 'b'; c <- compA alg; d <- compA alg; return $ f_5 alg c d]
```

```

compB :: Compiler input m => SAB s a b -> TransM input m b
compB alg = csum [do char 'b'; c <- compS alg; return $ f_6 alg c,
                  do char 'a'; c <- compB alg; d <- compB alg; return $ f_7 alg c d]

```

In **Compiler.hs** steht dieser Compiler zusammen mit der Zielalgebra *SABcount* von Beispiel 4.5. Für $m = \text{Maybe}$ und alle $w \in \{a, b\}^*$ und $i, j \in \mathbb{N}$ gilt

$$\text{compile}_{SAB}(SABcount)(w) = \text{Just}(i, j)$$

genau dann, wenn i und j die Anzahl der Vorkommen von a bzw. b in w ist. □

16.4 Generischer JavaLight-Compiler (siehe **Java.hs**)

```

compJava :: Compiler input m => JavaLight s1 s2 s3 s4 s5 s6 s7 s8
                           -> TransM input m s1
compJava alg = commands where
    alg' = derec alg                                -- siehe 11.2
    commands = do c <- command
                 csum [do cs <- commands; return $ seq_ alg c cs,
                           return $ embed alg c]
    command = csum [do tstring "if"; e <- disjunctC; c <- command
                    csum [do tstring "else"; c' <- command
                           return $ cond alg e c c',
                           return $ cond1 alg e c],
                    do tstring "while"; e <- disjunctC; c <- command

```

```

        return $ loop alg e c,
do tchar '{'; cs <- commandsC; tchar '}'
        return $ block alg cs,
do x <- tidentifier; tchar '='; e <- sumC
        tchar ';'; return $ assign alg x e]
sumC = do e <- prodC; f <- sumsect; return $ sum' alg' e f
sumsect = csum [do op <- csum $ map tchar "+-"
                e <- prodC; f <- sumsect
                return $ if op == '+' then plus' alg' e f
                           else minus' alg' e f,
                return $ nilS alg']
prodC = do e <- factor; f <- prodsect; return $ prod' alg' e f
prodsect = csum [do op <- csum $ map tchar "*/"
                e <- factor; f <- prodsect
                return $ if op == '*' then times' alg' e f
                           else div' alg' e f,
                return $ nilP alg']
factor = csum [do i <- tint; return $ embedI alg i,
                do x <- tidentifier; return $ var alg x,
                do tchar '('; e <- sumC; tchar ')'
                return $ encloseS alg e]
disjunctC = do e <- conjunctC
            csum [do tstring "||"; e' <- disjunctC
                    return $ disjunct alg e e',
                    return $ embedC alg e]

```

```

conjunctC = do e <- literal
              csum [do tstring "&&"; e' <- conjunctC
                     return $ conjunct alg e e',
                     return $ embedL alg e]
literal = csum [do b <- tbool; return $ embedB alg b,
                 do tchar '!'; e <- literal; return $ not_ alg e,
                 do e <- sumC; rel <- trelation; e' <- sumC
                     return $ atom alg rel e e',
                 do tchar '('; e <- disjunctC; tchar ')'
                     return $ encloseD alg e]

```

Der entsprechende Compiler für JavaLight+ (siehe Kapitel 13) steht [hier](#).

16.5 Korrektheit des JavaLight-Compilers

Sei $\text{Store} = \text{String} \rightarrow \mathbb{Z}$,

$$\begin{aligned}
\text{comS} &= \{\text{Commands}, \text{Command}\}, \\
\text{expS} &= \{\text{Sum}, \text{Prod}, \text{Factor}\}, \\
\text{bexpS} &= \{\text{Disjunct}, \text{Conjunct}, \text{Literal}\},
\end{aligned}$$

$\text{Sem} = \mathcal{A} = (A, Op)$ wie in Abschnitt 4.9 und $\text{Ziel} = \text{javaStack} = (B, Op')$ wie in Abschnitt 12.5 definiert. Dann gilt für alle Sorten s (der nicht-linksrekursiven Version) von JavaLight:

$$A_s = \begin{cases} \text{Store} \multimap \text{Store} & \text{falls } s \in \text{comS}, \\ \text{Store} \rightarrow \mathbb{Z} & \text{falls } s \in \text{expS}, \\ \text{Store} \rightarrow 2 & \text{falls } s \in \text{bexpS}, \end{cases}$$

$$B_s = \mathbb{Z} \rightarrow \text{StackCom}^*.$$

Sei $\text{Mach}_s = (\mathbb{Z}^* \times \text{Store}) \multimap (\mathbb{Z}^* \times \text{Store})$. Die Funktionen $\text{encode} : \text{Sem} \rightarrow \text{Mach}$ und $\text{evaluate} : \text{Ziel} \rightarrow \text{Mach}$ des Interpreterdiagramms (1) am Anfang von Kapitel 5 sind im Fall von JavaLight wie folgt definiert:

Für alle $f \in A_s$, $g \in B_s$, $\text{stack} \in \mathbb{Z}^*$ und $\text{store} \in \text{Store}$,

$$\text{encode}_s(f)(\text{stack}, \text{store}) = \begin{cases} (\text{stack}, f(\text{store})) & \text{falls } s \in \text{comS} \\ & \text{und } f(\text{store}) \text{ definiert ist,} \\ (f(\text{store}) : \text{stack}, \text{store}) & \text{falls } s \in \text{expS} \cup \text{bexpS} \\ & \text{und } f(\text{store}) \text{ definiert ist,} \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

$$\text{evaluate}_s(g)(\text{stack}, \text{store}) = \langle \pi_1, \pi_2 \rangle(\text{execute}(g(0))(\text{stack}, \text{store}, 0))$$

(siehe Abschnitt 12.5).

Sei $(S, BS, R) = \text{JavaLight}$, $X = \bigcup BS$ und M eine Compilermonade (siehe Kapitel 5).

Da $compJava$ ein generischer Compiler für JavaLight ist, kommutiert (1) am Anfang von Kapitel 5 genau dann, wenn das folgende Diagramm kommutiert:

$$\begin{array}{ccc}
 X^* & \xrightarrow{\quad compJava(Ziel) \quad} & M(Ziel) \\
 \downarrow compJava(Sem) & (2) & \downarrow M(evaluate) \\
 M(Sem) & \xrightarrow{\quad M(encode) \quad} & M(Mach)
 \end{array}$$

(siehe Diagramm (14) in Kapitel 5). (2) beschreibt vor allem folgende Zusammenhänge zwischen $compJava(Ziel)$ und dem Interpreter $evaluate$ der Zielsprache $Ziel$:

- Sei com ein JavaLight-Programm einer Sorte von $comS$. Wird com von $compJava(Ziel)$ in die Befehlsfolge cs übersetzt und beginnt die Ausführung von cs im Zustand $state$, dann endet sie in einem Zustand, dessen Speicherkomponente mit der Speicherkomponente von $encode(compJava(Sem)(com))(state)$ übereinstimmt.
- Sei exp ein JavaLight-Programm einer Sorte von $expS \cup bexpS$. Wird exp von $compJava(Ziel)$ in die Befehlsliste cs übersetzt und beginnt die Ausführung von cs im Zustand $state$, dann endet sie in einem Zustand, dessen oberstes Kellerelement mit dem obersten Kellerelement von $encode(compJava(Sem)(exp))(state)$ übereinstimmt.

Wie in Kapitel 5 erwähnt wurde, kommutiert (1) am Anfang von Kapitel 5 und damit auch das obige Diagramm (2), wenn *Mach* eine JavaLight-Algebra ist und die mehrsortigen Funktionen *encode* und *evaluate* JavaLight-homomorph sind.

Eine **Testumgebung** für Übersetzungen des JavaLight-Compilers in die in den Kapiteln 11 und 12 definierten JavaLight-Algebren liefert die Funktion *javaToAlg(file)* von *Java.hs*, die ein Quellprogramm vom Typ *commands* aus der Datei *file* und in die jeweilige Zielalgebra überführt.

javaToAlg(file)(5) und *javaToAlg(file)(6)* starten nach jeder Übersetzung eine Schleife, die in jeder Iteration eine Variablenbelegung einliest und die sich aus dem jeweiligen Zielcode entsprechenden Zustandstransformation darauf anwendet.

Eine **Testumgebung** für Übersetzungen des JavaLight+-Compilers in die im Abschnitt 13.3 definierte JavaLight+-Algebra *javaStackP* liefert die Funktion *javaToStack(file)* von *Java2.hs*, die ein Quellprogramm vom Typ *commands* aus der Datei *file* und nach *javaStackP* überführt.

16.6 Generischer XMLstore-Compiler (siehe Compiler.hs)

```
compXML :: Compiler input m => XMLstore s1 s2 s3 s4 s5 s6 s7 s8 s9
          -> TransM input m s1

compXML alg = storeC where
    storeC      = do tstring "<store>"
                    csum [do stck <- stock'; return $ store alg stck,
                           do ords <- ordersC; stck <- stock'
                           return $ store0 alg ords stck]
    stock'      = do tstring "<stock>"; stck <- stockC
                     tstring "</stock>"; tstring "</store>"; return stck
    ordersC     = do (p,is) <- order
                     csum [do os <- ordersC
                            return $ orders alg p is os,
                            return $ embed0 alg p is]
    order       = do tstring "<order>"; tstring "<customer>"
                     p <- personC; tstring "</customer>"
                     is <- itemsC; tstring "</order>"; return (p,is)
    personC     = do tstring "<name>"; name <- text; tstring "</name>"
                     csum [do ems <- emailsC
                            return $ personE alg name ems,
                            return $ person alg name]
    emailsC     = csum [do em <- emailC; ems <- emailsC
                            return $ emails alg em ems,
                            return $ none alg]
```

```

emailC      = do tstring "<email>"; em <- text; tstring "</email>"
                return $ email alg em
itemsC      = do (id,price) <- item
                csum [do is <- itemsC; return $ items alg id price is,
                      return $ embedI alg id price]
item         = do tstring "<item>"; id <- idC; tstring "<price>"
                price <- text; tstring "</price>"
                tstring "</item>"; return (id,price)
stockC      = do (id,qty,supps) <- iqs
                csum [do is <- stockC
                      return $ stock alg id qty supps is,
                      return $ embedS alg id qty supps]
iqs          = do tstring "<item>"; id <- idC; tstring "<quantity>"
                qty <- tint; tstring "</quantity>"
                supps <- suppliers; tstring "</item>"
                return (id,qty,supps)
suppliers   = csum [do tstring "<supplier>"; p <- personC
                      tstring "</supplier>"; return $ supplier alg p,
                      do stck <- stockC; return $ parts alg stck]
idC          = do tstring "<id>"; t <- text; tstring "</id>"
                return $ id_ alg t

```

text :: Compiler input m => TransM input m String

```

text = do strs <- some $ token $ some $ nchar "< \n\t"
        return $ unwords strs

```

17 Induktion, Coinduktion und rekursive Gleichungen

Induktion

Sei $C\Sigma = (S, C)$ eine konstruktive Signatur. Die zentrale Methode zum Beweis von Eigenschaften der initialen $C\Sigma$ -Algebra – unabhängig von deren konkreter Repräsentation – ist die Induktion. Ihre Korrektheit (*soundness*) folgt aus [Satz 3.2](#) (3):

Sei A eine initiale $C\Sigma$ -Algebra.

$$A \text{ ist die einzige } C\Sigma\text{-Unteralgebra von } A. \quad (1)$$

Sei φ eine prädikatenlogische Formel, die eine Eigenschaft der Elemente von A beschreibt, und $B = \{a \in A \mid a \text{ erfüllt } \varphi\}$. Die Frage, ob φ für alle Elemente von A gilt, reduziert sich wegen (1) auf die Frage, ob B eine Σ -Unteralgebra von A ist, ob also

$$\text{für alle } c : e \rightarrow s \in C \quad c^A(B_e) \subseteq B_s \text{ gilt.} \quad (2)$$

Induktion heißt demnach: die Gültigkeit von $\forall x : \varphi(x)$ in $T_{C\Sigma}$ aus einem Beweis von (2) schließen.

Induktion erlaubt es u.a., die Korrektheit als Gleichungen formulierter rekursiver Programme zu beweisen, indem man zeigt, dass deren gewünschte Semantik die Gleichungen löst:

Rekursive Ψ -Gleichungssysteme

Sei $C\Sigma = (S, C)$ eine konstruktive und $D\Sigma = (S, D)$ eine destruktive Signatur. Dann nennen wir $\Psi = (C\Sigma, D\Sigma)$ eine **Bisignatur**.

Eine Menge

$$E = \{dc(x_1, \dots, x_{n_c}) = t_{d,c} \mid c : e_1 \times \dots \times e_{n_c} \rightarrow s \in C, d : s \rightarrow e \in D\}$$

von Σ -Gleichungen (siehe Kapitel 3) mit folgenden Eigenschaften heißt **rekursives Ψ -Gleichungssystem**:

- Für alle $d \in D$ und $c \in C$, $\text{free}(t_{d,c}) \subseteq \{x_1, \dots, x_{n_c}\}$.
- C ist die Vereinigung disjunkter Mengen C_1 und C_2 .
- Für alle $d \in D$, $c \in C_1$ und Teilterme du von $t_{d,c}$ ist u eine Variable und $t_{d,c}$ ein Term ohne Elemente von C_2 .
- Für alle $d \in D$, $c \in C_2$, Teilterme du und Pfade p (der Baumdarstellung) von $t_{d,c}$ besteht u aus Destruktoren und einer Variable und kommt auf p höchstens einmal ein Element von C_2 vor.

Induktive Lösungen

Sei Σ' eine Teilsignatur einer Signatur Σ und \mathcal{A} eine Σ -Algebra. Die in \mathcal{A} enthaltene Σ' -Algebra heißt Σ' -**Redukt von \mathcal{A}** und wird mit $\mathcal{A}|_{\Sigma'}$ bezeichnet.

Sei E ein rekursives Ψ -Gleichungssystem und A eine $C\Sigma$ -Algebra.

Eine **induktive Lösung von E in A** ist eine Σ -Algebra, deren $C\Sigma$ -Redukt mit A übereinstimmt und die E erfüllt.

Lambeks Lemma Sei $s \in S$.

(1) Sei $\{c_1 : e_1 \rightarrow s, \dots, c_n : e_n \rightarrow s\} = \{c : e \rightarrow s' \in C \mid s' = s\}$. Die Summenextension $[c_1^A, \dots, c_n^A]$ ist bijektiv. Es gibt also eine Funktion

$$d_s^A : A_s \rightarrow A_{e_1} + \dots + A_{e_n}$$

mit $[c_1^A, \dots, c_n^A] \circ d_s^A = id_{A_s}$ und $d_s^A \circ [c_1^A, \dots, c_n^A] = id_{A_{e_1} + \dots + A_{e_n}}$.

(2) Sei $\{d_1 : s_1 \rightarrow e_1, \dots, d_n : s_n \rightarrow e_n\} = \{d : s' \rightarrow e \in D \mid s' = s\}$. Die Produktextension $\langle d_1^A, \dots, d_n^A \rangle$ ist bijektiv. Es gibt also eine Funktion

$$c_s^A : A_{e_1} \times \dots \times A_{e_n} \rightarrow A_s$$

mit $c_s^A \circ \langle d_1^A, \dots, d_n^A \rangle = id_{A_s}$ und $\langle d_1^A, \dots, d_n^A \rangle \circ c_s^A = id_{A_{e_1} \times \dots \times A_{e_n}}$. □

Satz 17.1 Sei C_2 leer und A eine initiale $C\Sigma$ -Algebra. Dann hat E genau eine induktive Lösung in A .

Beweis. Siehe [34], Theorem INDSOL. □

Beispiel 17.4 ($C\Sigma = \text{Reg}(BL)$; siehe 2.4 und 2.5) Sei $X = \bigcup BL$,

$$D\Sigma = (\{\text{reg}\}, \{2, X\}, \{\max, * : 2 \times 2 \rightarrow 2\} \cup \{_- \in B : X \rightarrow 2 \mid B \in BL\}, \\ \{\delta : \text{reg} \rightarrow \text{reg}^X, \beta : \text{reg} \rightarrow 2\})$$

und $\Psi = (\text{Reg}(BL), D\Sigma)$. Dann bildet die Menge BRE der Brzozowski-Gleichungen von Abschnitt 3.11 ein rekursives Ψ -Gleichungssystem, das in der initialen $\text{Reg}(BL)$ -Algebra $T_{\text{Reg}(BL)}$ die in Abschnitt 2.10 durch $\text{Bro}(BL)$ definierte induktive Lösung A hat.

Nach Satz 17.1 ist A die einzige induktive Lösung von BRE in $T_{\text{Reg}(BL)}$. □

Weitere Induktionsverfahren, mit denen Eigenschaften kleinstter Relationen bewiesen werden und die daher nicht nur in initialen Modellen anwendbar sind, werden in meinen LVs über **Logisch-Algebraischen Systementwurf** behandelt.

Coinduktion

Sei $D\Sigma = (S, D)$ eine destruktive Signatur. Die zentrale Methode zum Beweis von Eigenschaften der finalen $D\Sigma$ -Algebra - unabhängig von deren konkreter Repräsentation – ist die Coinduktion. Ihre Korrektheit folgt aus [Satz 3.4](#) (3):

Sei A eine finale $D\Sigma$ -Algebra.

Die Diagonale von A ist die einzige $D\Sigma$ -Kongruenz auf A . (1)

Sei E eine Menge von Σ -Gleichungen und A zu einer Σ -Algebra erweiterbar.

Wegen (1) gilt E in A , wenn

$$R_E = \{(g^*(t), g^*(u)) \in A^2 \mid t = u \in E, g : V \rightarrow A\}$$

in einer $D\Sigma$ -Kongruenz R enthalten ist (siehe [Termäquivalenz und Normalformen](#)).

Coinduktion heißt demnach: die Gültigkeit von E in A aus der Existenz einer $D\Sigma$ -Kongruenz schließen, die R_E enthält.

Beispiel 17.5

Die finale $Stream(X)$ -Algebra $\mathcal{A} = Seq(X)$ (siehe 2.6) interpretiert die Konstruktoren $evens : list \rightarrow list$ und $zip : list \times list \rightarrow list$ wie folgt:

Für alle $f : \mathbb{N} \rightarrow X$ und $n \in \mathbb{N}$,

$$\begin{aligned} even^{\mathcal{A}}(f)(n) &= \begin{cases} f(2n) & \text{falls } n \text{ gerade ist,} \\ f(2n+1) & \text{sonst,} \end{cases} \\ zip^{\mathcal{A}}(f, g)(n) &= \begin{cases} f(n/2) & \text{falls } n \text{ gerade ist,} \\ g(n/2+1) & \text{sonst.} \end{cases} \end{aligned}$$

Fin erfüllt die Gleichungen

$$head(evens(s)) = head(s) \tag{1}$$

$$tail(evens(s)) = even(tail(tail(s))) \tag{2}$$

$$head(zip(s, s')) = head(s) \tag{3}$$

$$tail(zip(s, s')) = zip(s', tail(s)) \tag{4}$$

mit $s, s' \in V_{list}$. Die Gültigkeit der Gleichung

$$evens(zip(s, s')) = s \tag{5}$$

in Fin soll allein unter Verwendung von (1)-(4) gezeigt werden.

Gemäß Coinduktion gilt (5) in Fin , falls

$$R_E = \{(evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(f, g)), f) \mid f, g : \mathbb{N} \rightarrow X\}$$

eine $\text{Stream}(X)$ -Kongruenz ist, d.h. falls für alle $(f, g) \in R_E$ Folgendes gilt:

$$\text{head}^{\mathcal{A}}(f) = \text{head}^{\mathcal{A}}(g) \wedge (\text{tail}^{\mathcal{A}}(f), \text{tail}^{\mathcal{A}}(g)) \in R. \quad (6)$$

Beweis von (6). Sei $(f, g) \in R_E$. Dann gibt es $h : \mathbb{N} \rightarrow X$ mit $f = evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(g, h))$. Daraus folgt

$$\begin{aligned} \text{head}^{\mathcal{A}}(f) &= \text{head}^{\mathcal{A}}(evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(g, h))) \stackrel{(1)}{=} \text{head}^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(g, h)) \stackrel{(3)}{=} \text{head}^{\mathcal{A}}(g), \\ \text{tail}^{\mathcal{A}}(f) &= \text{tail}^{\mathcal{A}}(evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(g, h))) \stackrel{(2)}{=} evens^{\mathcal{A}}(\text{tail}^{\mathcal{A}}(\text{tail}^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(g, h)))) \\ &\stackrel{(4)}{=} evens^{\mathcal{A}}(\text{tail}^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(h, \text{tail}(g)))) \stackrel{(4)}{=} evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(\text{tail}^{\mathcal{A}}(g), \text{tail}^{\mathcal{A}}(h))). \end{aligned}$$

Daraus folgt $(\text{tail}^{\mathcal{A}}(f), \text{tail}^{\mathcal{A}}(g)) = (evens^{\mathcal{A}}(\text{zip}^{\mathcal{A}}(\text{tail}^{\mathcal{A}}(g), \text{tail}^{\mathcal{A}}(h))), \text{tail}^{\mathcal{A}}(g)) \in R_E$. \square

Sei A eine Σ -Algebra und $R \subseteq A^2$. Der **C -Abschluss von R** ist die kleinste Äquivalenzrelation auf A , die R enthält und für alle $c : e \rightarrow e' \in C$ und $a, b \in A_e$ folgende Bedingung erfüllt:

$$(a, b) \in R_e^C \Rightarrow (c^A(a), c^A(b)) \in R_{e'}^C.$$

R ist eine **$D\Sigma$ -Kongruenz modulo C** , wenn für alle $d : s \rightarrow e \in D$ und $a, b \in A_e$ gilt:

$$(a, b) \in R_s \Rightarrow (d^A(a), d^A(b)) \in R_e^C.$$

Ist A final und gibt es ein rekursives Ψ -Gleichungssystem E , dann ist der C -Abschluss einer $D\Sigma$ -Kongruenz modulo C auf A nach Satz 17.7 eine $D\Sigma$ -Kongruenz. Deshalb folgt die Gültigkeit von E in A bereits aus der Existenz einer $D\Sigma$ -Kongruenz modulo C , die R_E enthält. **Coinduktion modulo C** heißt demnach: die Gültigkeit von E in A aus der Existenz einer $D\Sigma$ -Kongruenz modulo C schließen, die R_E enthält.

Beispiel 17.6

Sei $X = \bigcup BL$ und A die $(Reg(BL) \cup Acc(X))$ -Algebra mit $A|_{Reg(BL)} = Lang(X)$ und $A|_{Acc(X)} = \mathcal{P}(X)$ (siehe 2.10). A erfüllt die Gleichungen

$$\delta(par(t, u)) = \lambda x.par(\delta(t)(x), \delta(u)(x)) \quad (1)$$

$$\delta(seq(t, u)) = \lambda x.par(seq(\delta(t)(x), u), ite(\beta(t), \delta(u)(x), mt)) \quad (2)$$

$$\beta(par(t, u)) = max\{\beta(t), \beta(u)\} \quad (3)$$

$$\beta(seq(t, u)) = \beta(t) * \beta(u) \quad (4)$$

$$par(par(t_1, u_1), par(t_2, u_2)) = par(par(t_1, t_2), par(u_1, u_2)) \quad (5)$$

$$par(t, mt) = t \quad (6)$$

mit $t, u, v \in V_{reg}$ und $x \in V_X$. Die Gültigkeit des Distributivgesetzes

$$seq(t, par(u, v)) = par(seq(t, u), seq(t, v)) \quad (7)$$

in A soll allein unter Verwendung von (1)-(6) gezeigt werden.

Gemäß Coinduktion modulo $C = \{par\}$ gilt (7) in A , falls

$$R_E = \{(seq^A(f, par^A(g, h)), par^A(seq^A(f, g), seq^A(f, h))) \mid f, g, h \in Lang(X)\}$$

eine $Acc(X)$ -Kongruenz modulo C ist, d.h. falls für alle $(f, g) \in R_E$ Folgendes gilt:

$$\beta^{\mathcal{A}}(f) = \beta^{\mathcal{A}}(g) \wedge (\delta^{\mathcal{A}}(f), \delta^{\mathcal{A}}(g)) \in R_E^C. \quad (8)$$

Beweis von (8). Sei $(f, g) \in R_E$ und $x \in X$. Dann gibt es $f', g', h \in A$ mit

$$f = seq^A(f', par^A(g', h)) \quad \text{und} \quad g = par^A(seq^A(f', g'), seq^A(f', h))).$$

Daraus folgt

$$\begin{aligned} \beta^{\mathcal{A}}(f) &= \beta^{\mathcal{A}}(seq^A(f', par^A(g', h))) \stackrel{(4)}{=} \beta^{\mathcal{A}}(f') * \beta^{\mathcal{A}}(par^A(g', h)) \\ &\stackrel{(3)}{=} \beta^{\mathcal{A}}(f') * max\{\beta^{\mathcal{A}}(g'), \beta^{\mathcal{A}}(h)\} = max\{\beta^{\mathcal{A}}(f') * \beta^{\mathcal{A}}(g'), \beta^{\mathcal{A}}(f') * \beta^{\mathcal{A}}(h)\} \\ &\stackrel{(4)}{=} max\{\beta^{\mathcal{A}}(seq^A(f', g')), \beta^{\mathcal{A}}(seq^A(f', h))\} \\ &\stackrel{(3)}{=} \beta^{\mathcal{A}}(par^A(seq^A(f', g'), seq^A(f', h))) = \beta^{\mathcal{A}}(g), \end{aligned}$$

$$\delta^{\mathcal{A}}(f)(x) = \delta^{\mathcal{A}}(seq^A(f', par^A(g', h)))$$

$$\begin{aligned} &\stackrel{(2)}{=} par^A(seq^A(\delta^{\mathcal{A}}(f')(x), par^A(g', h)), \text{ if } \beta^{\mathcal{A}}(f') = 1 \text{ then } \delta^{\mathcal{A}}(par^A(g', h))(x) \text{ else } mt^A) \\ &\stackrel{(1)}{=} par^A(seq^A(\delta^{\mathcal{A}}(f')(x), par^A(g', h)), \\ &\quad \text{if } \beta^{\mathcal{A}}(f') = 1 \text{ then } par^A(\delta^{\mathcal{A}}(g')(x) \text{ else } \delta^{\mathcal{A}}(h)(x), mt^A)), \end{aligned}$$

$$\stackrel{(6)}{=} \begin{cases} \textcolor{blue}{par^A(\textcolor{red}{seq^A(\delta^{\mathcal{A}}(f')(x), par^A(g', h))}, \\ \textcolor{blue}{par^A(\delta^{\mathcal{A}}(g')(x), \delta^{\mathcal{A}}(h)(x)))}} & \text{falls } \beta^{\mathcal{A}}(f') = 1 \\ \textcolor{red}{seq^A(\delta^{\mathcal{A}}(f')(x), par^A(g', h))} & \text{sonst,} \end{cases}$$

$$\begin{aligned} \delta^{\mathcal{A}}(g)(x) &= \delta^{\mathcal{A}}(par^A(seq^A(f', g'), seq^A(f', h))))(x) \\ &\stackrel{(1)}{=} par^A(\delta^{\mathcal{A}}(seq^A(f', g'))(x), \delta^{\mathcal{A}}(seq^A(f', h))(x)) \\ &\stackrel{(2)}{=} par^A(par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), \text{ if } \beta^{\mathcal{A}}(f') = 1 \text{ then } \delta^{\mathcal{A}}(g')(x) \text{ else } mt^A), \\ &\quad par^A(seq^A(\delta^{\mathcal{A}}(f')(x), h), \text{ if } \beta^{\mathcal{A}}(f') = 1 \text{ then } \delta^{\mathcal{A}}(h)(x) \text{ else } mt^A)) \\ &= \begin{cases} par^A(par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), \delta^{\mathcal{A}}(g')(x)), \\ \quad par^A(seq^A(\delta^{\mathcal{A}}(f')(x), h), \delta^{\mathcal{A}}(h)(x))) & \text{falls } \beta^{\mathcal{A}}(f') = 1 \\ par^A(par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), mt^A), \\ \quad par^A(seq^A(\delta^{\mathcal{A}}(f')(x), h), mt^A)) & \text{sonst} \end{cases} \\ &\stackrel{(5),(6)}{=} \begin{cases} par^A(par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), seq^A(\delta^{\mathcal{A}}(f')(x), h)), \\ \quad par^A(\delta^{\mathcal{A}}(g')(x), \delta^{\mathcal{A}}(h)(x))) & \text{falls } \beta^{\mathcal{A}}(f') = 1 \\ par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), seq^A(\delta^{\mathcal{A}}(f')(x), h)) & \text{sonst.} \end{cases} \end{aligned}$$

Sei

$$\begin{aligned} f_1 &= seq^A(\delta^{\mathcal{A}}(f')(x), par^A(g', h)), \\ f_2 &= par^A(seq^A(\delta^{\mathcal{A}}(f')(x), g'), seq^A(\delta^{\mathcal{A}}(f')(x), h)), \\ f_3 &= par^A(\delta^{\mathcal{A}}(g')(x), \delta^{\mathcal{A}}(h)(x)). \end{aligned}$$

Wegen $(f_1, f_2) \in R_E$ gilt also

$$(\delta^{\mathcal{A}}(f)(x), \delta^{\mathcal{A}}(g)(x)) = (\textcolor{blue}{par}^A(f_1, f_3), \textcolor{blue}{par}^A(f_2, f_3)) \in R_E^C$$

im Fall $\beta^{\mathcal{A}}(f') = 1$ und

$$(\delta^{\mathcal{A}}(f)(x), \delta^{\mathcal{A}}(g)(x)) = (\textcolor{red}{f}_1, \textcolor{red}{f}_2) \in R_E^C$$

im Fall $\beta^{\mathcal{A}}(f') = 0$.

□

Weitere Coinduktionsverfahren, mit denen Eigenschaften größter Relationen bewiesen werden und die daher nicht nur in finalen Modellen anwendbar sind, werden in meinen LVs über **Logisch-Algebraischen Systementwurf** behandelt.

Coinduktive Lösungen

Sei E ein rekursives Ψ -Gleichungssystem und A eine $D\Sigma$ -Algebra.

Eine **coinduktive Lösung von E in A** ist eine Σ -Algebra, deren $D\Sigma$ -Redukt mit A übereinstimmt und die E erfüllt.

Satz 17.7 Sei A eine finale $D\Sigma$ -Algebra und \sim eine $D\Sigma$ -Kongruenz modulo C . Dann hat E genau eine coinduktive Lösung in A , die zur Σ -Algebra erweiterte Termalgebra $T_{C\Sigma}$ erfüllt E , \sim_C ist eine $D\Sigma$ -Kongruenz und es gilt

$$\text{unfold}^{T_{C\Sigma}} = \text{fold}^A.$$

Beweis. Siehe [34], Theorem COINDSOL und Lemma MOD. □

Beispiel 17.10

Sei $\Psi = (\text{Reg}(BL), D\Sigma)$ die Bisignatur von Beispiel 17.4, $\Sigma = \text{Reg}(BL) \cup D\Sigma$ und A die Σ -Algebra mit $A|_{\text{Reg}(BL)} = \text{Lang}(X)$ und $A|_{\text{Acc}(X)} = \mathcal{P}(X)$.

A erfüllt das rekursive Ψ -Gleichungssystem BRE von Abschnitt 3.11 und ist daher nach Satz 17.7 die einzige coinduktive Lösung von BRE in $\mathcal{P}(X)$.

Da $\chi : \mathcal{P}(X^*) \rightarrow 2^{X^*}$ (siehe 2.10) Σ -homomorph ist, wird BRE auch von der Bildalgebra $\chi(A)$ erfüllt. Nach Satz 17.7 ist $\chi(A)$ die einzige coinduktive Lösung von BRE in $\chi(\mathcal{P}(X)) = \text{Beh}(X, 2)$ und gelten die Gleichungen

$$\begin{aligned} \text{fold}^{\text{Lang}(X)} &= \text{unfold}^{\text{Bro}(BL)} : \text{Bro}(BL) \rightarrow \mathcal{P}(X), \\ \text{fold}^{\text{reg}B} &= \text{unfold}^{\text{Bro}(BL)} : \text{Bro}(BL) \rightarrow \text{Beh}(X, 2). \end{aligned}$$

Sei $t \in T_{Reg(BL)}$ und $v = fold^{Regword(BL)}(t)$. Im Haskell-Modul `Compiler.hs` entspricht der Erkenner $fold^{regB}(t)$ bzw. $unfold^{Bro(BL)}(t)$ dem Aufruf `regToAlg "" v n` mit $n = 1$ bzw. $n = 3$. Er startet eine Schleife, die nach der Eingabe von Wörtern w fragt, auf die der Erkenner angewendet werden soll, um zu prüfen, ob w zur Sprache von t gehört oder nicht. \square

Entscheidende Argumente im Beweis von Satz 17.7 entstammen dem Beweis von [39], Thm. 3.1, und [40], Thm. A.1, wo rekursive $Stream(X)$ -Gleichungen für Stromkonstruktoren untersucht werden (siehe auch [9], Anhänge A.5 and A.6).

Analoge Ergebnisse erhält man für weitere destruktive Signaturen wie z.B. unendliche Binärbäume ([41], Thm. 2), nichtdeterministische Systeme [2], Mealy-Automaten [8], nebenläufige Prozesse [12, 37, 13] und formale Potenzreihen ([39], Kapitel 9).

Hierbei wird oft von der traditionellen Darstellung rekursiver Gleichungssysteme als **strukturell-operationelle Semantikregeln (SOS)** ausgegangen, wobei “strukturell” und “operationell” für die jeweiligen Konstruktoren bzw. Destruktoren steht. Z.B. lauten die Gleichungen von BRE als SOS-Regeln wie folgt:

$$\overline{base(B) \xrightarrow{\delta} \lambda x.ite(x \in B, base(1), base(\emptyset))}$$

$$\begin{array}{c}
\frac{}{par(t, u) \xrightarrow{\delta} \lambda x. par(t'(x), u'(x))} \quad \frac{}{seq(t, u) \xrightarrow{\delta} \lambda x. par(seq(t'(x), u), ite(\beta(t), u'(x), mt))} \\
\\
\frac{}{iter(t) \xrightarrow{\delta} \lambda x. seq(t'(x), iter(t))} \quad \frac{}{base(B) \xrightarrow{\beta} ite(B = 1, 1, 0)} \\
\\
\frac{t \xrightarrow{\beta} m, \ u \xrightarrow{\beta} n}{par(t, u) \xrightarrow{\beta} max\{m, n\}} \quad \frac{t \xrightarrow{\beta} m, \ u \xrightarrow{\beta} n}{seq(t, u) \xrightarrow{\beta} m * n} \quad \frac{}{iter(t) \xrightarrow{\beta} 0}
\end{array}$$

Im Gegensatz zur operationellen Semantik besteht eine *denotationelle Semantik* nicht aus Regeln, sondern aus der Interpretation von Konstruktoren und Destruktoren in einer Algebra.

In der **Kategorientheorie** werden rekursive Gleichungssysteme und die Beziehungen zwischen ihren Komponenten mit Hilfe von **distributive laws** und **Bialgebren** verallgemeinert (siehe z.B. [42, 15, 10, 13]).

17.11 Cotermbasierte Erkenner regulärer Sprachen

Unter Verwendung der Haskell-Darstellung von Cotermen als Elemente des rekursiven Datentyps $\text{StateC}(x)(y)$ (siehe Beispiel 10.4) machen wir $DT_{Acc(X)}$ zur $Reg(BL)$ -Algebra $regA$ und zwar so, dass $regA$ mit $\xi(\chi(Lang(X))$ übereinstimmt, wobei $\xi : Beh(X, Y) \rightarrow DT_\Sigma$ den $DAut(X, Y)$ -Isomorphismus von $Beh(X, Y)$ nach DT_Σ bezeichnet (siehe Beispiel 2.15).

Für alle $B \in BL$, $f, g : X \rightarrow DT_{Acc(X)}$, $b, c \in 2$ und $t \in DT_{Acc(X)}$,

$$\begin{aligned} base^{regA}(B) &= StateC(\lambda x. if\ x \in B\ then\ base^{regA}(1)\ else\ base^{regA}(\emptyset)) \\ &\quad (if\ B = 1\ then\ 1\ else\ 0), \end{aligned}$$

$$par^{regA}(StateC(f)(b), StateC(g)(c)) = StateC(\lambda x. par^{regA}(f(x), g(x)))(max\{b, c\}),$$

$$\begin{aligned} seq^{regA}(StateC(f)(1), StateC(g)(c)) &= StateC(\lambda x. par^{regA}(seq^{regA}(f(x), StateC(g)(c)), \\ &\quad g(x)))(c), \end{aligned}$$

$$seq^{regA}(StateC(f)(0), t) = StateC(\lambda x. seq^{regA}(f(x), t))(0),$$

$$iter^{regA}(StateC(f)(b)) = StateC(\lambda x. seq^{regA}(f(x), iter^{regA}(StateC(f)(b))))(1).$$

Sei $\Psi = (Reg(BL), D\Sigma)$ die Bisignatur von Beispiel 17.4, $\Sigma = Reg(BL) \cup D\Sigma$ und A die Σ -Algebra mit $A|_{Reg(BL)} = Lang(X)$ und $A|_{Acc(X)} = DT_{Acc(X)}$.

Da das rekursive Ψ -Gleichungssystem BRE (siehe 3.11) von A erfüllt wird und $\chi : \mathcal{P}(X^*) \rightarrow 2^{X^*}$ (siehe 2.10) und ξ Σ -homomorph sind, wird das rekursive Ψ -Gleichungssystem BRE von Abschnitt 3.11 BRE auch von der Bildalgebra $\xi(\chi(A))$ erfüllt.

Nach Satz 17.7 ist $\xi(\chi(A))$ die einzige coinduktive Lösung von *BRE* in

$$\xi(\chi(\mathcal{P}(X))) = \xi(Beh(X, 2)) = DT_{Acc(X)},$$

und gilt die Gleichung

$$fold^{regA} = unfold^{Bro(BL)} : Bro(BL) \rightarrow DT_{Acc(X)}.$$

Aus der Eindeutigkeit von $unfold^{Bro(BL)}$, der $Acc(X)$ -Homomorphie von χ und ξ (s.o.) und (1) in Abschnitt 2.6 erhalten wir

$$unfold^{Bro(BL)} = \xi \circ \chi \circ unfold^{Bro(BL)} = \xi \circ \chi \circ fold^{Lang(X)}. \quad (3)$$

Sei $t \in T_{Reg(BL)}$. Wegen (3) realisiert der initiale Automat $(Bro(BL), t)$ den $Acc(X)$ -Coterm $\xi(\chi(fold^{Lang(X)}(t)))$.

Sei $t \in T_{Reg(BL)}$ und $v = fold^{Regword(BL)}(t)$. Im Haskell-Modul **Compiler.hs** entspricht der Erkenner $unfold^{DT_{Acc(X)}}(fold^{regA}(t))$ dem Aufruf `regToAlg "" v 2`. Dieser startet eine Schleife, die nach der Eingabe von Wörtern w fragt, auf die der Erkenner angewendet werden soll, um zu prüfen, ob w zur Sprache von t gehört oder nicht.

18 Iterative Gleichungen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur und V eine endliche S -sortige Menge von “Variablen”. Eine S -sortige Funktion

$$E : V \rightarrow T_\Sigma(V)$$

heißt **iteratives Σ -Gleichungssystem**, falls das Bild von E keine Variablen enthält.

Sei $\mathcal{A} = (A, Op)$ eine Σ -Algebra und A^V die Menge der S -sortigen Funktionen von V nach A .

$g : V \rightarrow A$ löst E in \mathcal{A} , wenn $g^* \circ E = g$ gilt.

Lemma 18.1

Sei $reduce : T_\Sigma(V) \rightarrow T_\Sigma(V)$ eine Funktion mit

$$g^* \circ reduce = g^* \tag{1}$$

für alle $g : V \rightarrow A$. Dann gilt

$$g^* \circ (reduce \circ E^*)^n = g^* \tag{2}$$

für alle Lösungen $g : V \rightarrow A$ von E in A und $n \in \mathbb{N}$.

Beweis durch Induktion über n .

$$g^* \circ (\text{reduce} \circ E^*)^0 = g^* \circ id_{T_\Sigma(V)} = g^*.$$

Sei $n > 0$. Dann ist

$$\begin{aligned} g^* \circ (\text{reduce} \circ E^*)^n &= g^* \circ \text{reduce} \circ E^* \circ (\text{reduce} \circ E^*)^{n-1} \\ &\stackrel{(1)}{=} g^* \circ E^* \circ (\text{reduce} \circ E^*)^{n-1} \stackrel{3.7(1)}{=} (g^* \circ E)^* \circ (\text{reduce} \circ E^*)^{n-1} \\ &\stackrel{g \text{ l\"ost } E \text{ in } A}{=} g^* \circ (\text{reduce} \circ E^*)^{n-1} \stackrel{\text{ind. hyp.}}{=} g^*. \quad \square \end{aligned}$$

18.2 Das iterative Gleichungssystem einer CFG

Sei $G = (S, BS, R)$ eine CFG und $X = \bigcup BS$.

Im Folgenden erlauben wir den Konstruktoren *par* und *seq* von $Reg(BL)$ mehr als zwei Argumente und schreiben daher

- $\text{par}(t_1, \dots, t_n)$ anstelle von $\text{par}(t_1, \text{par}(t_2, \dots, \text{par}(t_{n-1}, t_n) \dots))$ und
- $\text{seq}(t_1, \dots, t_n)$ anstelle von $\text{seq}(t_1, \text{seq}(t_2, \dots, \text{seq}(t_{n-1}, t_n) \dots))$.

par(t) und *seq(t)* stehen für t .

G induziert ein iteratives $Reg(BL)$ -Gleichungssystem:

$$\begin{aligned} E_G : S &\rightarrow T_{Reg(BL)}(S) \\ s &\mapsto par(\overline{w_1}, \dots, \overline{w_k}), \end{aligned}$$

wobei $\{w_1, \dots, w_k\} = \{w \in (S \cup BS)^* \mid s \rightarrow w \in R\}$

und für alle $n > 1$, $e_1, \dots, e_n \in S \cup BS$ und $s \in S$,

$$\begin{aligned} \overline{e_1 \dots e_n} &= seq(\overline{e_1}, \dots, \overline{e_n}), \\ \overline{s} &= s. \end{aligned}$$

E_G heißt **Gleichungssystem von G** .

Satz 18.3 (Fixpunktsatz für CFGs)

- (i) Die Funktion $sol_G : S \rightarrow Lang(X)$ mit $sol_G(s) = L(G)_s$ für alle $s \in S$ löst E_G in $Lang(X)$.

Sei $g : S \rightarrow \mathcal{P}(X^*)$ eine Lösung von E_G in $Lang(X)$.

- (ii) Die S -sortige Menge Sol mit $Sol_s = g(s)$ für alle $s \in S$ ist Trägermenge einer $\Sigma(G)$ -Unteralgebra von $Word(G)$.
- (iii) Für alle $s \in S$, $sol_G(s) \subseteq g(s)$, m.a.W.: die Sprache von G ist die kleinste Lösung von E_G in $Lang(X)$.

Beweis.

Sei $g : V \rightarrow \text{Lang}(X)$, $s \in S$, $\{r_1, \dots, r_k\}$ die Menge aller Regeln von G mit linker Seite s . Sei $1 \leq i \leq k$,

$$r_i = (s \rightarrow w_i) \quad \text{und} \quad \text{dom}(f_{r_i}) = e_{i1} \times \dots \times e_{in_i}.$$

Dann gibt es $s_1, \dots, s_n \in S \cup BS$ mit $e_1 \dots e_n = w_i$ und

$$\begin{aligned} g^*(\overline{w_i}) &= g^*(\overline{e_1 \dots e_n}) = g^*(\text{seq}(\overline{e_1}, \dots, \overline{e_n})) = \text{seq}^{\text{Lang}(X)}(g^*(\overline{e_1}), \dots, g^*(\overline{e_n})) \\ &= g^*(\overline{e_1}) \cdot \dots \cdot g^*(\overline{e_n}) = f_{r_i}^{\text{Word}(G)}(g^*(\overline{e_{i1}}) \cdot \dots \cdot g^*(\overline{e_{in_i}})). \end{aligned} \tag{1}$$

Beweis von (i).

$$\begin{aligned}
sol_G(s) &= L(G)_s = fold_s^{Word(G)}(T_{\Sigma(G),s}) \\
&= \bigcup_{i=1}^k \{fold_s^{Word(G)}(f_{r_i}(t)) \mid t \in T_{\Sigma(G),e_{i1} \times \dots \times e_{in_i}}\} \\
&= \bigcup_{i=1}^k \{f_{r_i}^{Word(G)}(fold_{e_{i1} \times \dots \times e_{in_i}}^{Word(G)}(t)) \mid t \in T_{\Sigma(G),e_{i1} \times \dots \times e_{in_i}}\} \\
&= \bigcup_{i=1}^k f_{r_i}^{Word(G)}(fold_{e_{i1} \times \dots \times e_{in_i}}^{Word(G)}(T_{\Sigma(G),e_{i1} \times \dots \times e_{in_i}})) = \bigcup_{i=1}^k f_{r_i}^{Word(G)}(L(G)_{e_{i1} \times \dots \times e_{in_i}}) \\
&= \bigcup_{i=1}^k f_{r_i}^{Word(G)}(L(G)_{e_{i1}} \cdot \dots \cdot L(G)_{e_{in_i}}) \\
&= \bigcup_{i=1}^k f_{r_i}^{Word(G)}(sol_G^*(\overline{e_{i1}}) \cdot \dots \cdot sol_G^*(\overline{e_{in_i}})) \\
&\stackrel{(1)}{=} \bigcup_{i=1}^k sol_G^*(\overline{w_i}) = par^{Lang(X)}(sol_G^*(\overline{w_1}), \dots, sol_G^*(\overline{w_k})) \\
&= sol_G^*(par(\overline{w_1}, \dots, \overline{w_k})) = \textcolor{blue}{sol_G^*(E_G(s))}.
\end{aligned}$$

Also ist $sol_G = sol_G^* \circ E_G$ und damit eine Lösung von E_G in $Lang(X)$.

Beweis von (ii). Zu zeigen: Für alle $1 \leq i \leq k$,

$$f_{r_i}^{Word(G)}(Sol_{e_{i1}} \cdot \dots \cdot Sol_{e_{in_i}}) \subseteq Sol_s. \quad (2)$$

Nach Definition von Sol gilt $Sol_{e_{ij}} = g^*(\overline{e_{ij}})$ für alle $1 \leq j \leq n_i$.

Beweis von (2).

$$\begin{aligned}
 f_{r_i}^{Word(G)}(Sol_{e_{i1}} \cdot \dots \cdot Sol_{e_{in_i}}) &= f_{r_i}^{Word(G)}(g^*(\overline{e_{i1}}) \cdot \dots \cdot g^*(\overline{e_{in_i}})) \\
 \stackrel{(1)}{=} g^*(\overline{w_i}) &\subseteq \bigcup_{i=1}^k g^*(\overline{w_i}) = par^{Lang(X)}(g^*(\overline{w_1}), \dots, g^*(\overline{w_k})) = g^*(par(\overline{w_1}, \dots, \overline{w_k})) \\
 \stackrel{Def.}{=} E_G g^*(E_G(s)) &= g(s) = Sol_s.
 \end{aligned}$$

Beweis von (iii). Nach Satz 3.2 (3) ist $L(G) = fold^{Word(G)}(T_{\Sigma(G)})$ die kleinste $\Sigma(G)$ -Unteralgebra von $Word(G)$. Wegen (ii) gilt demnach $L(G)_s \subseteq Sol_s$ für alle $s \in S$, also

$$sol_G(s) = L(G)_s \subseteq Sol_s = g(s). \quad \square$$

18.4 Beispiele

1. Sei $X = \{a, b\}$ und $G = (\{A, B\}, \emptyset, X, \{A \rightarrow BA, A \rightarrow a, B \rightarrow b\})$. E_G ist wie folgt definiert:

$$\begin{aligned}
 E_G(A) &= par(seq(A, B), \overline{a}), \\
 E_G(B) &= \overline{b}.
 \end{aligned}$$

Die einzige Lösung g von E_G in $Lang(X)$ lautet:

$$g(A) = \{b\}^* \cdot \{a\}, \quad g(B) = \{b\}.$$

2. Sei $G = \text{SAB}$ (siehe Beispiel 4.5). E_G ist wie folgt definiert:

$$\begin{aligned} E_G(S) &= \text{par}(\text{seq}(\bar{a}, B), \text{seq}(\bar{b}, A), \text{eps}) \\ E_G(A) &= \text{par}(\text{seq}(\bar{a}, S), \text{seq}(\bar{b}, A, A)), \\ E_G(B) &= \text{par}(\text{seq}(\bar{b}, S), \text{seq}(\bar{a}, B, B)). \end{aligned}$$

Die einzige Lösung g von E_G in $\text{Lang}(X)$ lautet:

$$\begin{aligned} g(S) &= \{w \in \{a, b\}^* \mid \#a(w) = \#b(w)\}, \\ g(A) &= \{w \in \{a, b\}^* \mid \#a(w) = \#b(w) + 1\}, \\ g(B) &= \{w \in \{a, b\}^* \mid \#a(w) = \#b(w) - 1\}. \end{aligned}$$

Andererseits ist g mit $g(S) = g(A) = g(B) = \{a, b\}^+$ keine Lösung von E_G in $\text{Lang}(X)$, weil a einerseits zu $g(S)$ gehört, aber nicht zu

$$g^*(E_G(S)) = g^*(\text{par}(\text{seq}(\bar{a}, B), \text{seq}(\bar{b}, A))) = (\{a\} \cdot g(B)) \cup (\{b\} \cdot g(A)).$$

Beide Grammatiken sind nicht linksrekursiv. Deshalb haben ihre Gleichungssysteme nach Satz 18.8 (s.u.) jeweils genau eine Lösung in $\text{Lang}(X)$. □

18.5 Von Erkennern regulärer Sprachen zu Erkennern kontextfreier Sprachen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur und V eine S -sortige Menge.

$$\Sigma_V =_{def} (S, F \cup \{in_s : V_s \rightarrow s \mid s \in S\}).$$

Lemma 18.6

$\sigma_V : V \rightarrow T_{\Sigma_V}$ bezeichnet die Substitution mit $\sigma_V(x) = in_s x$ für alle $x \in V_s$ und $s \in S$.

Für alle Σ_V -Algebren A gilt

$$(in^A)^* = fold^A \circ \sigma_V^* : T_\Sigma(V) \rightarrow A,$$

wobei $in^A = (in_s^A : V_s \rightarrow A_s)_{s \in S}$.

Beweis. Wir zeigen zunächst

$$fold^A \circ \sigma_V = in^A. \tag{3}$$

Beweis von (3). Sei $s \in S$ und $x \in X_s$. Da $fold^A : T_{\Sigma_V} \rightarrow A$ mit in_s verträglich ist, gilt

$$fold^A(\sigma_V(x)) = fold^A(in_s x) = in_s^A(x).$$

Aus (3) folgt $(in^A)^* = (fold^A \circ \sigma_V)^* \stackrel{Satz 3.7(1)}{=} fold^A \circ \sigma_V^*$. □

Sei $G = (S, BS, R)$ eine nicht-linksrekursive CFG und *reduce* die in Beispiel 3.10 beschriebene Reduktionsfunktion für reguläre Ausdrücke.

Dann gibt es für alle $s \in S$ $k_s, n_s > 0$, $B_{s,1}, \dots, B_{s,n_s} \in BS$ und $Reg(BL)$ -Terme $t_{s,1}, \dots, t_{s,n_s}$ über S mit

$$(reduce \circ E_G^*)^{k_s}(s) = par(seq(\overline{B_{s,1}}, t_{s,1}), \dots, seq(\overline{B_{s,n_s}}, t_{s,n_s})) \quad (4)$$

oder

$$(reduce \circ E_G^*)^{k_s}(s) = par(seq(\overline{B_{s,1}}, t_{s,1}), \dots, seq(\overline{B_{s,n_s}}, t_{s,n_s}), eps). \quad (5)$$

S_{eps} bezeichne die Menge aller Sorten von S , die (5) erfüllen.

Sei $Reg(BL)'$ die Erweiterung von $Reg(BL)$ um die Menge S der Sorten von G als weitere Basismenge und den Konstruktor $\text{in} =_{def} in_{reg} : S \rightarrow reg$ als weiteres Operationssymbol.

Sei $D\Sigma$ wie in Beispiel 17.4 definiert, $\Psi_S = (Reg(BL)', D\Sigma)$ und $\Sigma = Reg(BL)' \cup D\Sigma$.

Mit den Notationen von (4) und (5) erhalten wir das folgende rekursive Ψ_S -Gleichungssystem:

$$\begin{aligned} rec(E_G) = & \{ \delta(in(s)) = \lambda x. \sigma_S^*(par(ite(x \in B_{s,1}, t_{s,1}, mt), \dots, \\ & ite(x \in B_{s,n_s}, t_{s,n_s}, mt))) \mid s \in S \} \cup \\ & \{ \beta(in(s)) = 1 \mid s \in S_{eps} \} \cup \\ & \{ \beta(in(s)) = 0 \mid s \in S \setminus S_{eps} \}. \end{aligned}$$

Satz 18.7

Sei $X = \bigcup BS$, $g : S \rightarrow Lang(X)$ eine Lösung von E_G in $Lang(X)$ und A_g die Σ -Algebra mit $A_g|_{Reg(BL)} = Lang(X)$, $A_g|_{Acc(X)} = \mathcal{P}(X)$ und $in_s^{A_g} = g_s$ für alle $s \in S$.

A_g erfüllt das rekursive Ψ_S -Gleichungssystem $rec(E_G)$.

Beweis.

Zu zeigen ist, dass für alle $t = t' \in rec(E_G)$ und $h : V \rightarrow A_g$ $h^*(t) = h^*(t')$ gilt.

Sei $h : V \rightarrow A_g$. Für alle $s \in S$,

$$h^*(in(s)) = in^{A_g}(s) = g(s) = g^*(s) \stackrel{\text{Lemma 18.1}}{=} g^*((reduce \circ E_G^*)^{k_s}(s)) \quad (6)$$

Lemma 18.6 liefert

$$g^* = (in^{A_g})^* = fold^{A_g} \circ \sigma_S^* : T_{Reg(BL)}(S) \rightarrow A. \quad (7)$$

Gehört s nicht zu S_{eps} , dann gilt

$$\begin{aligned} g^*((reduce \circ E_G^*)^{k_s}(s)) &= g^*(par(seq(\overline{B_{s,1}}, t_{s,1}), \dots, seq(\overline{B_{s,n_s}}, t_{s,n_s}))) \\ &= par^{A_g}(seq^{A_g}(\overline{B_{s,1}}^{A_g}, g^*(t_{s,1})), \dots, seq^{A_g}(\overline{B_{s,n_s}}^{A_g}, g^*(t_{s,n_s}))) \\ &= \bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})), \end{aligned} \quad (8)$$

also

$$\begin{aligned}
h^*(\delta(\text{in}(s))) &= \delta^{A_g}(h^*(\text{in}(s))) \stackrel{(6)}{=} \delta^{A_g}(g^*((\text{reduce} \circ E_G^*)^{k_s}(s))) \\
&\stackrel{(8)}{=} \delta^{A_g}(\bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i}))) = \lambda x. \delta^{A_g}(\bigcup_{i=1}^n (B_{s,i} \cdot g^*(t_{s,i}))(x)) \\
&\stackrel{\text{Def. } \delta^{A_g}}{=} \lambda x. \{w \in X^* \mid xw \in \bigcup_{i=1}^n (B_{s,i} \cdot g^*(t_{s,i}))\} \\
&= \lambda x. \{w \in X^* \mid x \in B_{s,i}, w \in g^*(t_{s,i}), 1 \leq i \leq n\} \\
&= \lambda x. \bigcup_{i=1}^{n_s} \{w \in X^* \mid x \in B_{s,i}, w \in g^*(t_{s,i})\} \\
&= \lambda x. \bigcup_{i=1}^{n_s} \{w \in X^* \mid \text{if } x \in B_{s,i} \text{ then } w \in g^*(t_{s,i}) \text{ else } w \in \emptyset\} \\
&= \lambda x. \bigcup_{i=1}^{n_s} (\text{if } x \in B_{s,i} \text{ then } g^*(t_{s,i}) \text{ else } \emptyset) \\
&= \lambda x. \bigcup_{i=1}^{n_s} (\text{if } x \in B_{s,i} \text{ then } g^*(t_{s,i}) \text{ else } \text{mt}^A) \\
&= \lambda x. \bigcup_{i=1}^{n_s} (\text{if } x \in B_{s,i} \text{ then } g^*(t_{s,i}) \text{ else } g^*(\text{mt})) \\
&= \lambda x. \bigcup_{i=1}^{n_s} g^*(\text{ite}(x \in B_{s,i}, t_{s,i}, \text{mt})) \\
&= \lambda x. g^*(\text{par}(\text{ite}(x \in B_{s,1}, t_{s,1}, \text{mt}), \dots, \text{ite}(x \in B_{s,n_s}, t_{s,n_s}, \text{mt}))) \\
&= g^*(\lambda x. \text{par}(\text{ite}(x \in B_{s,1}, t_{s,1}, \text{mt}), \dots, \text{ite}(x \in B_{s,n_s}, t_{s,n_s}, \text{mt}))) \\
&\stackrel{(7)}{=} \text{fold}^{A_g}(\sigma_S^*(\lambda x. \text{par}(\text{ite}(x \in B_{s,1}, t_{s,1}, \text{mt}), \dots, \text{ite}(x \in B_{s,n_s}, t_{s,n_s}, \text{mt})))) \\
&= h^*(\sigma_S^*(\lambda x. \text{par}(\text{ite}(x \in B_{s,1}, t_{s,1}, \text{mt}), \dots, \text{ite}(x \in B_{s,n_s}, t_{s,n_s}, \text{mt}))))
\end{aligned}$$

und

$$\begin{aligned} h^*(\beta(in(s))) &= \beta^{A_g}(h^*(in(s))) \stackrel{(6)}{=} \beta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s))) \\ &\stackrel{(8)}{=} \beta^{A_g}(\bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i}))) \stackrel{\text{Def. } \beta^{A_g}}{=} 0 = h^*(0). \end{aligned}$$

Gehört s zu S_{eps} , dann gilt

$$\begin{aligned} g^*((reduce \circ E_G^*)^{k_s}(s)) &= g^*(par(seq(\overline{B_{s,1}}, t_{s,1}), \dots, seq(\overline{B_{s,n_s}}, t_{s,n_s}), eps)) \\ &= par^{A_g}(seq^{A_g}(\overline{B_{s,1}}^{A_g}, g^*(t_{s,1})), \dots, seq^{A_g}(\overline{B_{s,n_s}}^{A_g}, g^*(t_{s,n_s}), eps^{A_g})) \\ &= \bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}, \end{aligned} \tag{9}$$

also

$$\begin{aligned} h^*(\delta(in(s))) &= \delta^{A_g}(h^*(in(s))) \stackrel{(6)}{=} \delta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s))) \\ &\stackrel{(9)}{=} \delta^{A_g}(\bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}) = \lambda x. \delta^{A_g}(\bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\})(x) \\ &\stackrel{\text{Def. } \delta^{A_g}}{=} \lambda x. \{w \in X^* \mid xw \in \bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}\} \\ &= \lambda x. \{w \in X^* \mid xw \in \bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i}))\} \\ &\stackrel{\text{wie oben}}{=} h^*(\sigma_S^*(\lambda x. par(ite(x \in B_{s,1}, t_{s,1}, mt), \dots, ite(x \in B_{s,n_s}, t_{s,n_s}, mt)))) \end{aligned}$$

und

$$\begin{aligned} h^*(\beta(in(s))) &= \beta^{A_g} A(h^*(in(s))) \stackrel{(6)}{=} \beta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s))) \\ &\stackrel{(9)}{=} \beta^{A_g}(\bigcup_{i=1}^{n_s} (B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}) \stackrel{\text{Def. } \beta^{A_g}}{=} 1 = h^*(1). \quad \square \end{aligned}$$

Satz 18.8 A_{sol_G} ist die einzige coinduktive Lösung von $E = BRE \cup rec(E_G)$ in $\mathcal{P}(X)$.

Proof. Nach Satz 18.3 (i) ist sol_G eine Lösung von E_G in $Lang(X)$. Also wird $rec(E_G)$ nach Satz 18.7 von A_{sol_G} erfüllt. Nach Beispiel 17.10 wird auch BRE von A_{sol_G} erfüllt. Folglich ist A_{sol_G} nach Satz 17.7 die einzige Lösung von E in $\mathcal{P}(X)$. \square

Da $\chi : \mathcal{P}(X^*) \rightarrow 2^{X^*}$ (siehe 2.10) und die Bijektion $\xi : 2^{X^*} \rightarrow DT_{Acc(X)}$ von Beispiel 2.15 Σ -homomorph sind, wird E auch von den Bildalgebren $\chi(A_{sol_G})$ und $\xi(\chi(A_{sol_G}))$ erfüllt.

Folglich ist $\chi(A_{sol_G})$ bzw. $\xi(\chi(A_{sol_G}))$ nach Satz 17.7 die einzige coinduktive Lösung von E in $\chi(\mathcal{P}(X)) = Beh(X, 2)$ bzw. $\xi(\chi(\mathcal{P}(X))) = \xi(Beh(X, 2)) = DT_{Acc(X)}$.

Satz 18.9 sol_G ist die einzige Lösung von E_G in $Lang(X)$.

Beweis. Sei sol eine weitere Lösung von E_G in $Lang(X)$. Dann sind nach Satz 18.7 sind A_{sol_G} und A_{sol} coinduktive Lösungen von E in $\mathcal{P}(X)$. Nach Satz 18.8 stimmen sie miteinander überein. Also sind auch sol_G und sol gleich. \square

$rec(E_G)$ legt folgende Erweiterung des Brzozowski-Automaten $Bro(BL)$ zur $Reg(BL)'$ -Algebra $Bro(BL)'$ nahe:

Für alle $s \in S$,

$$\delta^{Bro(BL)'}(in(s)) = \lambda x. \sigma_S^*(par(ite(x \in B_{s,1}, t_{s,1}, mt), \dots, ite(x \in B_{s,n_s}, t_{s,n_s}, mt))),$$

$$\beta^{Bro(BL)'}(in(s)) = \text{if } s \in S_{eps} \text{ then } 1 \text{ else } 0.$$

Sei $\text{Lang}(X)' = A_{sol_G}|_{Reg(BL)'}$, $\text{regB}' = \chi(A_{sol_G})|_{Reg(BL)'}$, $\text{regA}' = \xi(\chi(A_{sol_G}))|_{Reg(BL)'}$ und $\Sigma = Reg(BL)' \cup D\Sigma$.

$Bro(BL)'$ stimmt mit der zur Σ -Algebra erweiterten Termalgebra $T_{Reg(BL)'}$ überein (siehe Satz 17.7). Demnach gelten die Gleichungen

$$fold^{\text{Lang}(X)'} = unfold^{Bro(BL)'} : Bro(BL)' \rightarrow \mathcal{P}(X), \quad (10)$$

$$fold^{\text{regB}'} = unfold^{Bro(BL)'} : Bro(BL)' \rightarrow Beh(X, 2), \quad (11)$$

$$fold^{\text{regA}'} = unfold^{Bro(BL)'} : Bro(BL)' \rightarrow DT_{Acc(X)}, \quad (12)$$

was die $Acc(X)$ -Homomorphie der drei Faltungen impliziert.

$Bro(BL)'$, regB' und regA' sind in `Compiler.hs` durch **accT rules**, **regB rules** bzw. **regA rules** implementiert. Hierbei ist **rules** eine Liste der Regeln von G . Alle drei Algebren-Implementierungen verwenden die Funktion **eqs rules**, die $\sigma_S^* \circ E_G$ berechnet.

Sei $s \in S$. Die obigen Definitionen von $\delta^{Bro(BL)'}(in(s))$ bzw. $\beta^{Bro(BL)'}(in(s))$ sind in der Implementierung von **accT rules** von $Bro(BL)'$ durch die Gleichungen

$$\begin{aligned}\delta^{Bro(BL)'}(in(s)) &= \delta^{Bro(BL)'}(\sigma_S^*(E_G(s))), \\ \beta^{Bro(BL)'}(in(s)) &= \beta^{Bro(BL)'}(\sigma_S^*(E_G(s)))\end{aligned}\tag{13}$$

realisiert. Wegen der Nicht-Linksrekursivität von G garantiert Haskells *lazy evaluation*, dass für jeden $Reg(BL)'$ -Grundterm t die Reduktionen von $\delta^{Bro(BL)'}(t)$ und $\beta^{Bro(BL)'}(t)$ mit Hilfe von *BRE* und (13) terminieren und die gewünschten Ergebnisse liefern.

Wie lässt sich die zugrundeliegende Eindeutigkeit der Lösung von (13) in den “Variablen” $\delta^{Bro(BL)'} \circ in$ und $\beta^{Bro(BL)'} \circ in$ nachweisen?

Aus (10), der Eindeutigkeit von $unfold^{Bro(BL)'}$ und $unfold^{Bro(BL)'}$ und der $Acc(X)$ -Homomorphie von χ und ξ folgt

$$unfold^{Bro(BL)'} = \chi \circ unfold^{Bro(BL)'} = \chi \circ fold^{Lang(X)'},$$

also auch

$$unfold^{Bro(BL)'} = \xi \circ \chi \circ unfold^{Bro(BL)'} = \xi \circ \chi \circ fold^{Lang(X)'}. \quad \text{384}$$

Also erkennt für alle $Reg(BL)'$ -Grundterme t der initiale Automat $(Bro(BL)', t)$ die Sprache $fold^{Lang(X)'}(t)$ von t (im Sinne von Abschnitt 2.6) und realisiert den $Acc(X)$ -Coterm $\xi(\chi(fold^{Lang(X)'}(t)))$ (im Sinne von Kapitel 20). Für $Reg(BL)$ -Grundterme t haben wir das bereits in Beispiel 17.10 bzw. Abschnitt 17.11 gezeigt.

Zusätzlich erhalten wir für alle $s \in S$:

$$unfold^{Bro(BL)'}(in(s)) \stackrel{(10)}{=} fold^{Lang(X)'}(in(s)) = in^{Lang(X)'}(s) = sol_G(s) = L(G)_s. \quad (14)$$

Also erkennt der initiale Automat $(Bro(BL)', in(s))$ die Sprache von G , realisiert also deren charakteristische Funktion:

$$unfold^{Bro(BL)'}(in(s)) = \chi(unfold^{Bro(BL)'}(in(s))) \stackrel{(14)}{=} \chi(L(G)_s), \quad (15)$$

und den entsprechenden $Acc(X)$ -Coterm:

$$unfold^{Bro(BL)'}(in(s)) = \xi(\chi(unfold^{Bro(BL)'}(in(s)))) \stackrel{(14)}{=} \xi(\chi(L(G)_s)). \quad (16)$$

Daraus folgt

$$in^{regB'}(s) = fold^{regB'}(in(s)) \stackrel{(11)}{=} unfold^{Bro(BL)'}(in(s)) \stackrel{(15)}{=} \chi(L(G)_s), \quad (17)$$

$$in^{regA'}(s) = fold^{regA'}(in(s)) \stackrel{(12)}{=} unfold^{Bro(BL)'}(in(s)) \stackrel{(16)}{=} \xi(\chi(L(G)_s)). \quad (18)$$

Die durch (17) und (18) gegebenen Definitionen von $in^{regB'}$ bzw. $in^{regA'}$ sind in den Implementierungen **regB rules** von $regB'$ und **regA rules** von $regA'$ durch die Gleichungen

$$\begin{aligned} in^{regB'} &= fold^{regB'} \circ \sigma_S^* \circ E_G, \\ in^{regA'} &= fold^{regA'} \circ \sigma_S^* \circ E_G \end{aligned} \tag{19}$$

realisiert.

Wieder garantiert Haskells *lazy evaluation* wegen der Nicht-Linksrekursivität von G , dass für jeden $Reg(BL)'$ -Grundterm t die Reduktionen von $fold^{regB'}(t)$ und $fold^{regA'}(t)$ mit Hilfe der induktiven Definition von $fold^{regB'}$ bzw. $fold^{regA'}$ und (19) terminieren und die gewünschten Ergebnisse liefern.

Wie lässt sich die zugrundeliegende Eindeutigkeit der Lösung von (19) in den “Variablen” $in^{regB'}$ und $in^{regA'}$ nachweisen?

Sei $t \in T_{Reg(BL)'}$ und $v = fold^{Regword(BL)}(t)$. Im Haskell-Modul **Compiler.hs** entspricht der Erkenner $fold^{regB'}(t)$, $unfold^{DT_{Acc}(X)}(fold^{regA'}(t))$ bzw. $unfold^{Bro(BL)'}(t)$ dem Aufruf

```
regToAlg file v n
```

mit $n = 1$, $n = 2$ bzw. $n = 3$, wobei die Datei **file** die Regeln von G enthält. Der Aufruf startet eine Schleife, die nach der Eingabe von Wörtern w fragt, auf die er angewendet werden soll, um zu prüfen, ob w zu $L(G)_t$ gehört oder nicht.

19 Interpretation in stetigen Algebren

Eine Menge A ist **halbgeordnet**, wenn es eine **Halbordnung**, also eine reflexive, transitive und antisymmetrische binäre Relation auf A gibt. Eine Teilmenge $\{a_i \mid i \in \mathbb{N}\}$ von A heißt **ω -Kette**, wenn für alle $i \in \mathbb{N}$ $a_i \leq a_{i+1}$ gilt.

Eine halbgeordnete Menge A heißt **ω -CPO** (ω -complete partially ordered set), wenn A ein bzgl. \leq kleinstes Element \perp und Suprema $\bigsqcup_{i \in \mathbb{N}} a_i$ aller ω -Ketten $\{a_i \mid i \in \mathbb{N}\}$ von A enthält.

Für jede Menge A liefert die **flache Erweiterung**, $A_\perp =_{\text{def}} A + \{\perp\}$, einen ω -CPO: Die zugrundeliegende Halbordnung \leq ist

$$\{(a, b) \in A^2 \mid a = \perp \vee a = b\}.$$

Der ω -CPO der partiellen Funktionen von A nach B

Für alle $f, g : A \multimap B$,

$$f \leq g \Leftrightarrow_{\text{def}} \text{def}(f) \subseteq \text{def}(g) \wedge \forall a \in \text{def}(f) : f(a) = g(a).$$

Die **nirgends definierte Funktion** $\Omega : A \multimap B$ mit $\text{def}(\Omega) =_{\text{def}} \emptyset$ ist das kleinste Element von $A \multimap B$ bzgl. \leq .

Jede ω -Kette $f_0 \leq f_1 \leq f_2 \leq \dots$ von $A \rightarrow B$ hat das folgende Supremum: Für alle $a \in A$,

$$(\bigsqcup_{i \in \mathbb{N}} f_i)(a) =_{\text{def}} \begin{cases} f_i(a) & \text{falls } \exists i \in \mathbb{N} : a \in \text{def}(f_i), \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Ein Produkt $A_1 \times \dots \times A_n$ von n ω -CPOs wie auch die Menge A^B aller Funktionen von einer Menge B in einen ω -CPO A bilden selbst ω -CPOs: Die Halbordnungen auf A_1, \dots, A_n bzw. A werden – wie im Abschnitt **Kongruenzen und Quotienten** beschrieben – zur Halbordnung auf $A_1 \times \dots \times A_n$ bzw. A^B geliftet.

Das kleinste Element von $A_1 \times \dots \times A_n$ ist das n -Tupel der kleinsten Elemente von A_1, \dots, A_n . $\lambda x. \perp$ ist das kleinste Element von A^B , wobei \perp das kleinste Element von B ist.

Suprema sind komponenten- bzw. argumentweise definiert: Für alle ω -Ketten $a_0 = (a_{0,1}, \dots, a_{0,n}) \leq a_1 = (a_{1,1}, \dots, a_{1,n}) \leq a_2 \leq \dots$ von $A_1 \times \dots \times A_n$,

$$\bigsqcup_{i \in \mathbb{N}} a_i =_{\text{def}} (\bigsqcup_{i \in \mathbb{N}} a_{i,0}, \dots, \bigsqcup_{i \in \mathbb{N}} a_{i,n}). \quad (2)$$

Für alle ω -Ketten $f_0 \leq f_1 \leq f_2 \leq \dots$ von B^A und $a \in A$,

$$(\bigsqcup_{i \in \mathbb{N}} f_i)(a) =_{\text{def}} \bigsqcup_{i \in \mathbb{N}} f_i(a). \quad (3)$$

Seien A, B halbgeordnete Mengen. Eine Funktion $f : A \rightarrow B$ ist **monoton**, wenn für alle $a, b \in A$ gilt:

$$a \leq b \Rightarrow f(a) \leq f(b).$$

f ist **strikt**, wenn f das kleinste Element \perp_A von A auf das kleinste Element \perp_B von B abbildet. Ist A ein Produkt, dann bildet eine strikte Funktion f nicht nur \perp_A , sondern jedes Tupel von A , das ein kleinstes Element enthält, auf \perp_B ab.

Seien A, B ω -CPOs. f ist **ω -stetig** (ω -continuous), wenn $f(\bigsqcup_{i \in \mathbb{N}} a_i) = \bigsqcup_{i \in \mathbb{N}} f(a_i)$ für alle ω -Ketten $\{a_i \mid i \in \mathbb{N}\}$ von A gilt.

ω -stetige Funktionen sind monoton.

Die Menge $A \rightarrow_c B$ aller ω -stetigen Funktionen von A nach B ist ein ω -CPO, weil Ω und die Suprema ω -Ketten ω -stetiger Funktionen (siehe (2)) selbst ω -stetig sind.

Sei $\Sigma = (S, F)$ eine konstruktive Signatur und $\mathcal{A} = (A, Op)$ eine Σ -Algebra.

\mathcal{A} ist **monoton**, wenn es für alle $s \in S$ eine Halbordnung $\leq_{A,s} \subseteq A_s^2$ und ein bzgl. $\leq_{A,s}$ kleinstes Element $\perp_{A,s}$ gibt und für alle $f \in F$ $f^{\mathcal{A}}$ monoton ist.

\mathcal{A} ist **ω -stetig**, wenn \mathcal{A} monoton ist, für alle $s \in S$ A_s ein ω -CPO ist und für alle $f \in F$ $f^{\mathcal{A}}$ ω -stetig ist.

Fixpunktsatz von Kleene

Sei A ein ω -CPO und $f : A \rightarrow A$ ω -stetig. Da f monoton ist, erhalten wir die ω -Kette

$$\perp \leq f(\perp) \leq f^2(\perp) \leq \dots$$

Deren Supremum

$$lfp(f) =_{def} \bigsqcup_{n \in \mathbb{N}} f^n(\perp)$$

ist der (bzgl. \leq) kleinste Fixpunkt von f (siehe [34]). □

Anwendung auf iterative Gleichungssysteme

Sei $E : V \rightarrow T_\Sigma(V)$ ein iteratives Σ -Gleichungssystem (siehe Kapitel 18) und $\mathcal{A} = (A, Op)$ eine ω -stetige Σ -Algebra. Dann können die Halbordnungen, kleinsten Elemente und Suprema von A , wie in Kapitel 2 beschrieben, nach A^V geliftet werden, d.h. A^V ist ein ω -CPO.

Die Funktion $E_{\mathcal{A}} : A^V \rightarrow A^V$ mit

$$E_{\mathcal{A}}(g) = g^* \circ E$$

für alle $g \in A^V$ nennen wir **Schrittfunktion von E** . Offenbar wird E von g genau dann in A gelöst, wenn g ein Fixpunkt von $E_{\mathcal{A}}$ ist.

Nach [7], Proposition 4.13, ist $E_{\mathcal{A}}$ ω -stetig. Also folgt aus dem Fixpunktsatz von Kleene, dass

$$lfp(E_{\mathcal{A}}) = \bigsqcup_{n \in \mathbb{N}} E_{\mathcal{A}}^n(\lambda x. \perp_A) \quad (1)$$

der kleinste Fixpunkt von $E_{\mathcal{A}}$ in A ist.

Für alle strikten und ω -stetigen Σ -Homomorphismen $h : \mathcal{A} \rightarrow \mathcal{B}$ gilt:

$$h \circ lfp(E_{\mathcal{A}}) = lfp(E_{\mathcal{B}}(h)). \quad (2)$$

Beweis. Für alle $g \in A^V$,

$$h \circ E_{\mathcal{A}}(g) \stackrel{\text{Def. } E_{\mathcal{A}}}{=} h \circ g^* \circ E \stackrel{3.7(1)}{=} (h \circ g)^* \circ E = E_{\mathcal{B}}(h \circ g). \quad (3)$$

Wir nehmen an, dass für alle $n \in \mathbb{N}$ Folgendes gilt:

$$h \circ E_{\mathcal{A}}^n(\lambda x. \perp_A) = E_{\mathcal{B}}(h)^n(\lambda x. \perp_B). \quad (4)$$

Aus (4) folgt (2):

$$\begin{aligned} h \circ lfp(E_{\mathcal{A}}) &= h \circ \bigsqcup_{n \in \mathbb{N}} E_{\mathcal{A}}^n(\lambda x. \perp_A) \stackrel{h \text{ } \omega\text{-stetig}}{=} \bigsqcup_{n \in \mathbb{N}} (h \circ E_{\mathcal{A}}^n(\lambda x. \perp_A)) \\ &\stackrel{(4)}{=} \bigsqcup_{n \in \mathbb{N}} E_{\mathcal{B}}(h)^n(\lambda x. \perp_B) = lfp(E_{\mathcal{B}}(h)). \end{aligned}$$

Zu zeigen bleibt (4). Sei $n > 0$.

$$\begin{aligned} h \circ E_{\mathcal{A}}^0(\lambda x. \perp_A) &= h \circ \lambda x. \perp_A \stackrel{h \text{ strikt}}{=} \lambda x. \perp_B = E_{\mathcal{B}}(h)^0(\lambda x. \perp_B), \\ h \circ E_{\mathcal{A}}^n(\lambda x. \perp_A) &= h \circ E_{\mathcal{A}}(E_{\mathcal{A}}^{n-1}(\lambda x. \perp_A)) \stackrel{(3)}{=} E_{\mathcal{B}}(h)(h \circ E_{\mathcal{A}}^{n-1}(\lambda x. \perp_A)) \\ &\stackrel{\text{ind. hyp.}}{=} E_{\mathcal{B}}(h)(E_{\mathcal{B}}(h)^{n-1}(\lambda x. \perp_B)) = E_{\mathcal{B}}(h)^n(\lambda x. \perp_B). \end{aligned}$$
□

Iterative Gleichungssysteme dienen u.a. der Spezifikation partieller Funktionen. Sind z.B. zwei partielle Funktionen f, g gegeben, dann ist die Gleichung $\textcolor{blue}{h = g \circ h \circ f}$ zunächst nur eine *Bedingung* an eine dritte Funktion h . Die aus der Gleichung gebildete Schrittfunction

$$\lambda h. (\textcolor{blue}{g \circ h \circ f}) : (A \multimap B) \rightarrow (A \multimap B)$$

ist ω -stetig und hat deshalb nach dem Fixpunktsatz von Kleene einen kleinsten Fixpunkt, der als **denotationelle Semantik** von h bezeichnet wird.

19.1 Schleifensemantik

Sei Σ die abstrakte Syntax von JavaLight (siehe Beispiel 4.6) zusammen mit zwei Konstanten $e : 1 \rightarrow \textit{Disjunct}$ und $c : 1 \rightarrow \textit{Command}$. Sei \textit{Sem} das Zustandsmodell von JavaLight (siehe 4.9 und 11.5) zusammen mit festen Interpretationen $p \in \textit{javaState}_{\textit{Disjunct}}$ und $f \in \textit{javaState}_{\textit{Command}}$ von e bzw. c .

Sem ist ω -stetig.

Sei $V = \{x : Command\}$, E das iterative Σ -Gleichungssystem

$$\begin{aligned} E : V &\rightarrow T_\Sigma(V) \\ x &\mapsto cond(e, block(seq(c, x)), id) \end{aligned}$$

und E_{Sem} die Schrittfunktion von E (s.o.). Der kleinste Fixpunkt von E_{Sem} liefert uns die Interpretation von $loop(e, c)$ in Sem :

$$loop^{Sem}(p, f) =_{def} lfp(E_{Sem})(x) : Store \multimap Store.$$

Die Haskell-Funktion

```
loop :: St Bool -> St Store -> St Store
loop p f = cond p (loop p f . f) id
```

von Abschnitt 11.5 implementiert $loop^{Sem}$. Für alle $st \in Store$ terminiert $loop(p)(f)(st)$ genau dann, wenn $loop^{Sem}(p, f)(st)$ definiert ist. □

19.2 Semantik der Assemblersprache $StackCom^*$

Auch jede Befehlsfolge $cs \in StackCom^*$ lässt sich als iteratives Gleichungssystem $eqs(cs)$ darstellen. Im Gegensatz zum vorigen Abschnitt liegt hier wie in Abschnitt 18.2 der einfache Fall $I = \emptyset$ vor.

Die konstruktive Signatur $\Sigma(StackCom)$, aus deren Operationen die Terme von $eqs(cs)$ gebildet sind, lautet wie folgt:

$$\begin{aligned}\Sigma(StackCom) &= (\{com\}, \{\mathbb{Z}, String\}, F), \\ F &= \{ push : \mathbb{Z} \rightarrow com, \\ &\quad load, save, cmp : String \rightarrow com, \\ &\quad pop, add, sub, mul, div, or, and, inv, nix : 1 \rightarrow com, \\ &\quad cons, fork : com \times com \rightarrow com \}.\end{aligned}$$

$push, load, save, cmp, pop, add, sub, mul, div, or, and, inv$ entsprechen den Konstruktoren von $StackCom$. $nix, cons, fork$ ersetzen die Sprungbefehle von $StackCom$ (s.u.).

Sei Eqs die Menge der iterativen $\Sigma(StackCom)$ -Gleichungssysteme mit Variablen aus \mathbb{N} . Die folgendermaßen definierte Funktion $eqs : StackCom^* \rightarrow Eqs$ übersetzt Assemblerprogramme in solche Gleichungssysteme:

Sei $cs \in StackCom^*$ und $V(cs)$ die Menge der Nummern der Sprungziele von cs , also

$$V(cs) = \{0\} \cup \{n < |cs| \mid Jump(n) \in cs \vee JumpF(n) \in cs\}.$$

$$\begin{aligned} eqs(cs) : V(cs) &\rightarrow T_{\Sigma(StackCom)}(V(cs)) \\ n &\mapsto mkTerm(drop(n)(cs)) \end{aligned}$$

$$mkTerm : StackCom^* \rightarrow T_{\Sigma(StackCom)}(V(cs))$$

$$\epsilon \mapsto nix$$

$$c : cs' \mapsto \begin{cases} n & \text{falls } c = Jump(n) \wedge n < |cs| \\ nix & \text{falls } c = Jump(n) \wedge n \geq |cs| \\ fork(n, mkTerm(cs')) & \text{falls } c = JumpF(n) \wedge n < |cs| \\ fork(nix, mkTerm(cs')) & \text{falls } c = JumpF(n) \wedge n \geq |cs| \\ cons(c, mkTerm(cs')) & \text{sonst} \end{cases}$$

Beispiel Das Assemblerprogramm von Abschnitt 12.5 zur Berechnung der Fakultätsfunktion lautet als iteratives $\Sigma(StackCom)$ -Gleichungssystem wie folgt:

$$E : \{0, 3\} \rightarrow T_{\Sigma(StackCom)}(\{0, 3\})$$

$$\begin{aligned} 0 &\mapsto cons(push(1), cons(save(fact), cons(pop, 3))) \\ 3 &\mapsto cons(load(x), cons(push(1), cons(cmp(>), fork(nix, \\ &\quad cons(load(fact), cons(load(x), cons(mul, cons(save(fact), \\ &\quad cons(pop, cons(load(x), cons(push(1), cons(sub, cons(save(x), \\ &\quad cons(pop, 3))))))))))))))) \end{aligned}$$

In Abschnitt 16.5 haben wir informell gezeigt, dass $compJava(javaStack)$ ein Compiler für JavaLight bzgl. $javaState$ ist, was laut Kapitel 5 impliziert, dass folgendes Diagramm kommutiert:

$$\begin{array}{ccc}
 T_{\Sigma(JavaLight)} & \xrightarrow{fold^{javaStack}} & \textcolor{red}{javaStack} \\
 \downarrow fold^{javaState} & (1) & \downarrow evaluate \\
 \textcolor{red}{javaState} & \xrightarrow[encode]{} & \textcolor{red}{Mach} = State \multimap State
 \end{array}$$

Hier sind

- javaStack die in Abschnitt 12.5 definierte, zur JavaLight-Algebra erweiterte Assemblersprache StackCom^* ,
- javaState das in Abschnitt 4.9 und 19.1 definierte Zustandsmodell von JavaLight,
- Mach der ω -CPO der partiellen Funktionen auf der Zustandsmenge

$$\text{State} = \mathbb{Z}^* \times \mathbb{Z}^{\text{String}},$$

deren Paare aus jeweils einem Kellerinhalt und einer Speicherbelegung bestehen (siehe 12.5),

- encode und evaluate wie in Abschnitt 16.5 definiert.

$\text{Mach} : \text{State} \rightarrow \text{State}$ wird zunächst zu einer ω -stetigen $\Sigma(\text{StackCom})$ -Algebra erweitert:

Für alle $f, g : \text{State} \rightarrow \text{State}$, $\otimes \in \{\text{add}, \text{sub}, \text{mul}, \text{div}, \text{or}, \text{and}\}$,
 $(\text{stack}, \text{store}) \in \text{State}$, $a \in \mathbb{Z}$ und $x \in \text{String}$,

$$\text{push}^{\text{Mach}}(i)(\text{stack}, \text{store}) = (a : \text{stack}, \text{store}),$$

$$\text{load}^{\text{Mach}}(x)(\text{stack}, \text{store}) = (\text{store}(x) : \text{stack}, \text{store}),$$

$$\text{save}^{\text{Mach}}(x)(\text{stack}, \text{store}) = (\text{stack}, \text{update}(\text{store})(x)(a)),$$

$$\begin{aligned}
\textcolor{red}{cmp}^{\textcolor{brown}{Mach}}(x)(stack, store) &= \begin{cases} (1 : s, store) \text{ falls } \exists a, b, s : stack = a : b : s \\ \quad \wedge evalRel(x)(a)(b), \\ (0 : s, store) \text{ falls } \exists a, b, s : stack = a : b : s \\ \quad \wedge \neg evalRel(x)(a)(b), \\ \text{undefiniert} \quad \text{sonst,} \end{cases} \\
\textcolor{red}{pop}^{\textcolor{brown}{Mach}}(stack, store) &= \begin{cases} (s, store) \text{ falls } \exists a, s : stack = a : s, \\ \text{undefiniert sonst,} \end{cases} \\
\textcolor{red}{\otimes}^{\textcolor{brown}{Mach}}(stack, store) &= \begin{cases} (\textcolor{red}{op}(\otimes)(b, a) : s, store) \text{ falls } \exists a, b, s : stack = a : b : s, \\ \text{undefiniert sonst,} \end{cases} \\
\textcolor{red}{inv}^{\textcolor{brown}{Mach}}(stack, store) &= \begin{cases} ((a + 1) \bmod 2 : stack, store) \text{ falls } \exists a, s : stack = a : s, \\ \text{undefiniert sonst,} \end{cases} \\
\textcolor{red}{nix}^{\textcolor{brown}{Mach}}(stack, store) &= (stack, store), \\
\textcolor{red}{cons}^{\textcolor{brown}{Mach}}(f, g) &= g \circ f, \\
\textcolor{red}{fork}^{\textcolor{brown}{Mach}}(f, g)(stack, store) &= \begin{cases} f(s, store) \text{ falls } \exists s : stack = 0 : s, \\ g(s, store) \text{ falls } \exists a, s : stack = a : s \wedge a \neq 0, \\ \text{undefiniert sonst,} \end{cases}
\end{aligned}$$

wobei $op(add) = (+)$, $op(sub) = (-)$, $op(mul) = op(and) = (*)$, $op(div) = (/)$ und $op(or) = sign \circ (+)$.

Sei $cs \in StackCom^*$. Nach dem Fixpunktsatz von Kleene hat das iterative $\Sigma(StackCom)$ -Gleichungssystem $eqs(cs)$ (s.o.) eine kleinste Lösung in $Mach$. Sie liefert uns die Funktion **execute** von Abschnitt 12.5: Für alle $cs \in StackCom^*$ und $n \in V(cs)$,

$$\text{execute}(cs) = lfp(eqcs(cs)_{Mach}) : V(cs) \rightarrow Mach. \quad (4)$$

Tatsächlich entspricht $\text{execute}'$ der Haskell-Funktion **execute** in Abschnitt 12.5.

Im Folgenden wird $Mach$ so zu einer JavaLight-Algebra erweitert, dass execute und encode JavaLight-homomorph werden und aus der Initialität von $T_{\Sigma(\text{JavaLight})}$ die Kommutativität von (1) folgt (siehe Kapitel 5).

Für alle Sorten s von JavaLight, $Mach_s = State \multimap State$.

Für alle $op \in \{seq, sum, prod, disjunct, conjunct\}$ und $f, g : State \multimap State$,

$$op^{Mach}(f, g) = g \circ f.$$

Für alle $op \in \{embed, block, encloseS, embedC, embedL, encloseD\}$,

$$op^{Mach} = id_{State \multimap State}.$$

$$nilS^{Mach} = nilP^{Mach} = id_{State}.$$

Für alle $x \in String$ und $f : State \multimap State$,

$$\text{assign}^{Mach}(x, f) = \text{pop}^{Mach} \circ \text{save}^{Mach}(x) \circ f.$$

Für alle $f, g, h : State \multimap State$, $\text{cond}^{Mach}(f, g, h) = \text{fork}^{Mach}(h, g) \circ f$.

Für alle $f, g : State \multimap State$, $\text{cond1}^{Mach}(f, g) = \text{fork}^{Mach}(\text{id}_{Mach}, g) \circ f$.

Für alle $f, g : State \multimap State$, $\text{loop}^{Mach}(f, g) = \text{lfp}(\text{E}_{Mach})(x)$, wobei

$$\begin{aligned} E : \{x\} &\rightarrow T_{\Sigma(\text{StackCom})}(\{x, f, g\}) \\ x &\mapsto \text{cons}(f, \text{fork}(\text{nix}, \text{cons}(g, x))) \end{aligned}$$

Für alle $f, g : State \multimap State$,

$$\text{sumsect}^{Mach}(+, f, g) = g \circ \text{add}^{Mach} \circ f,$$

$$\text{sumsect}^{Mach}(-, f, g) = g \circ \text{sub}^{Mach} \circ f,$$

$$\text{prodsect}^{Mach}(*, f, g) = g \circ \text{mul}^{Mach} \circ f,$$

$$\text{prodsect}^{Mach}(/, f, g) = g \circ \text{div}^{Mach} \circ f.$$

$\text{embedI}^{Mach} = \text{push}^{Mach}$.

$\text{var}^{Mach} = \text{load}^{Mach}$.

Für alle $f : State \multimap State$, $\text{not}^{Mach}(f) = \text{inv}^{Mach} \circ f$.

Für alle $x \in String$ und $f, g : State \rightarrow State$,

$$\textcolor{brown}{atom}^{Mach}(f, x, g) = \text{cmp}^{Mach}(x) \circ g \circ f.$$

Für alle $b \in 2$, $\textcolor{brown}{embedB}^{Mach}(b) = \text{push}^{Mach}(b)$.

Nochmal iterative Gleichungen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur. Eine monotone bzw. ω -stetige Σ -Algebra \mathcal{A} ist **initial in der Klasse $PAlg_\Sigma$** bzw. **$CAlg_\Sigma$** aller monotonen bzw. ω -stetigen Σ -Algebren, wenn es zu jeder monotonen bzw. ω -stetigen Σ -Algebra \mathcal{B} genau einen strikten (!) und monotonen bzw. ω -stetigen Σ -Homomorphismus von \mathcal{A} nach \mathcal{B} gibt.

Alle initialen monotonen bzw. ω -stetigen Σ -Algebren sind durch strikte und monotone bzw. ω -stetige Σ -Isomorphismen miteinander verbunden.

**** Sei $\Sigma_\perp = (S, F \cup \{\perp_s : 1 \rightarrow s \mid s \in S\})$ und \leq die kleinste Σ -Kongruenz auf CT_{Σ_\perp} derart, dass für alle $s \in S$ und $t \in CT_{\Sigma_\perp, s}$, $\perp_s \leq t$, wobei \perp_s hier den Baum bezeichnet, der aus einem mit \perp_s markierten Blatt besteht. Er bildet offenbar das bzgl. \leq kleinste Element.

Satz 19.3 T_{Σ_\perp} ist initial in $PAlg_\Sigma$. CT_{Σ_\perp} ist initial in $CAlg_\Sigma$ (siehe [34]). □

Satz 19.4 Jedes iterative Σ -Gleichungssystem $E : V \rightarrow T_\Sigma(V)$ hat genau eine Lösung in CT_{Σ_\perp} .

Beweis. Sei $g : V \rightarrow CT_{\Sigma_\perp}$ eine Lösung von E in CT_{Σ_\perp} . Da $lfp(E_{CT_{\Sigma_\perp}})$ die kleinste Lösung von E in CT_{Σ_\perp} ist, gilt:

$$lfp(E_{CT_{\Sigma_\perp}}) \leq g. \quad (5)$$

Durch Induktion über n lässt sich zeigen, dass für alle $x \in V$ und $n \in \mathbb{N}$ gilt:

$$\text{def}(g(x)) \cap X_\Sigma^n \subseteq \text{def}(E_{CT_{\Sigma_\perp}}^{n+1}(\lambda x. \perp)(x)) \quad (6)$$

(siehe [27], Satz 17). Aus (6) folgt für alle $x \in V$:

$$\text{def}(g(x)) \subseteq \text{def}\left(\bigsqcup_{n \in \mathbb{N}} E_{CT_{\Sigma_\perp}}^n(\lambda x. \perp)(x)\right) \stackrel{\text{Def. } lfp(E_{CT_{\Sigma_\perp}})}{=} \text{def}(lfp(E_{CT_{\Sigma_\perp}})(x)),$$

also $g(x) = lfp(E_{CT_{\Sigma_\perp}})(x)$ wegen (5) und nach Definition von \leq . □

Beispiel *loop*

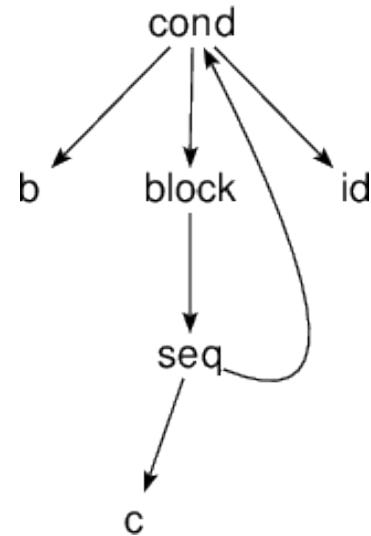
Sei $\Sigma = \Sigma(\text{JavaLight})$. Die eindeutige Lösung $sol : \{x\} \rightarrow CT_\Sigma(\{b, c\})$ des iterativen Σ -Gleichungssystems

$$E : \{x\} \rightarrow T_\Sigma(\{x, b, c\})$$

von Abschnitt 19.1 hat folgende Graphdarstellung:

Für alle $w \in \mathbb{N}^*$,

$$sol(x)(w) = \begin{cases} cond & \text{falls } w \in (212)^* \\ b & \text{falls } w \in (212)^*1 \\ block & \text{falls } w \in (212)^*2 \\ id & \text{falls } w \in (212)^*3 \\ seq & \text{falls } w \in (212)^*21 \\ c & \text{falls } w \in (212)^*211 \end{cases}$$

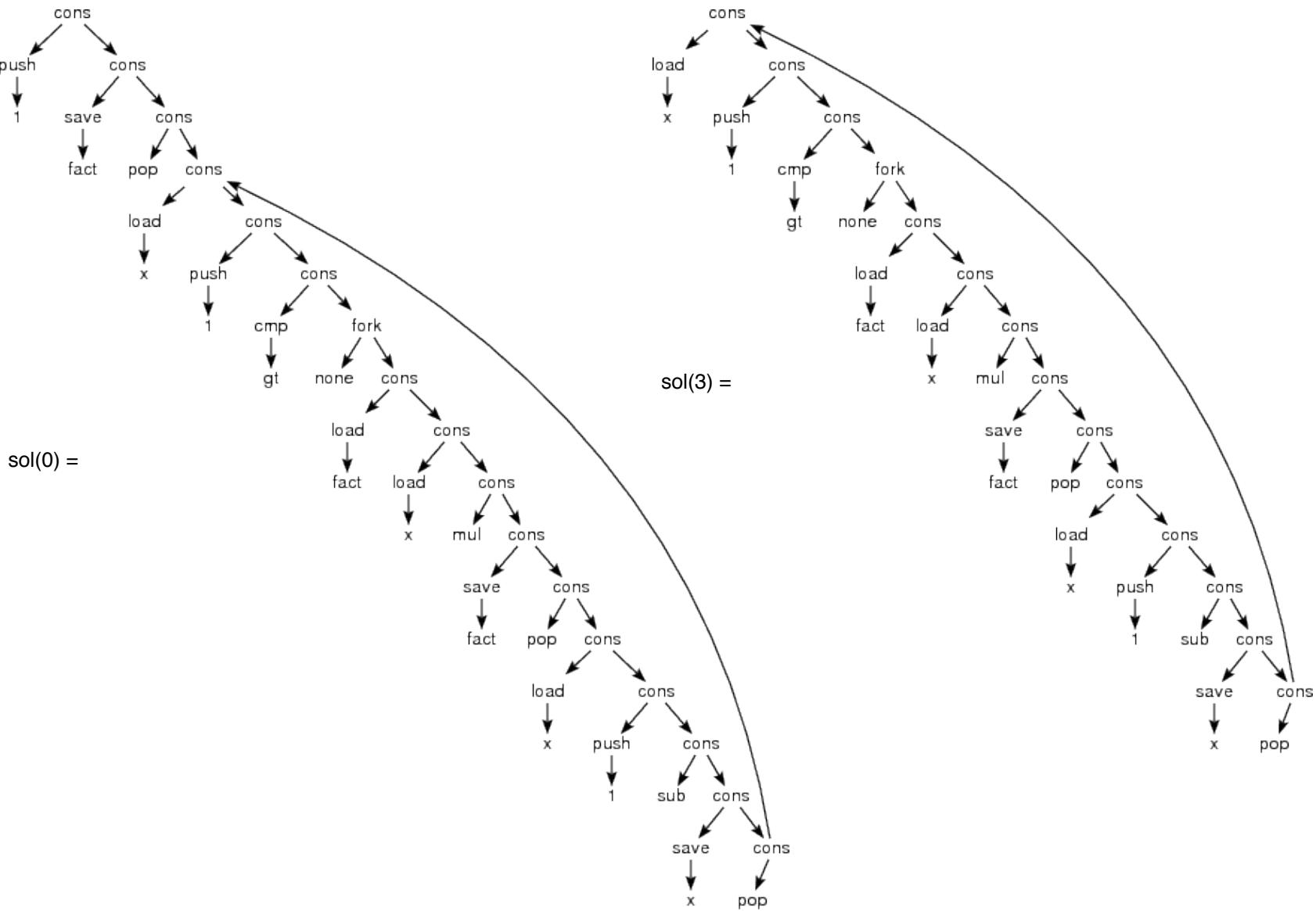


Beispiel *StackCom*

Sei $\Sigma = \Sigma(\text{StackCom})$ und cs das Assemblerprogramm von Beispiel 12.6. Die eindeutige Lösung $sol : \{0, 3\} \rightarrow CT_\Sigma$ des iterativen Σ -Gleichungssystems

$$eqs(cs) : \{0, 3\} \rightarrow T_\Sigma(\{0, 3\})$$

von Abschnitt 19.2 hat folgende Graphdarstellung:



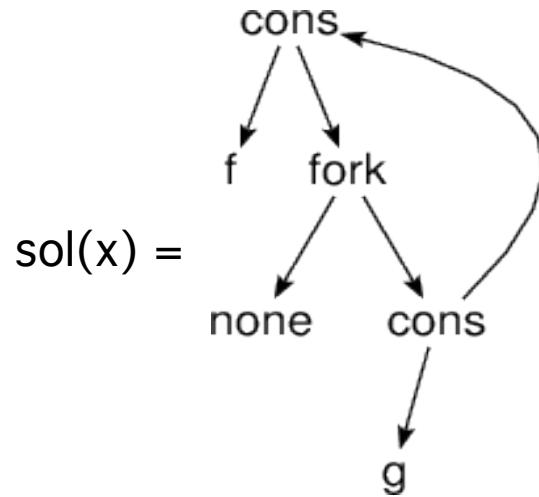
Offenbar repräsentiert jeder Pfad des Σ -Baums $sol(0)$ eine – möglicherweise unendliche – Kommandofolge, die einer Ausführungssequenz von cs entspricht. □

Beispiel

Sei $\Sigma = \Sigma(StackCom)$. Die eindeutige Lösung $sol : \{x\} \rightarrow CT_\Sigma(\{b, c\})$ des iterativen Σ -Gleichungssystems

$$E : \{x\} \rightarrow T_\Sigma(\{x, b, c\}),$$

dessen kleinste Lösung in *Mach* den Konstruktor *loop* von $\Sigma(StackCom)$ in *Mach* interpretiert (siehe 19.2), hat folgende Graphdarstellung:



Sei $cs \in StackCom^*$. Da $fold_{\omega}^{Mach}$ strikt, ω -stetig und Σ -homomorph ist, folgt

$$\textcolor{red}{execute}'(cs) = fold_{\omega}^{Mach} \circ lfp(eqs(cs)_{CT_{\Sigma}}) : V(cs) \rightarrow Mach \quad (7)$$

aus (2) und (4). Die Ausführung von cs im Zustand $state \in State$ liefert also dasselbe – möglicherweise undefinierte – Ergebnis wie die Faltung des Σ -Baums $lfp(eqs(cs)_{CT_{\Sigma}})(state)$ in $Mach$. \square

Die eindeutige Lösung eines iterativen Σ -Gleichungssystems E in CT_{Σ} lässt sich nicht nur als kleinsten Fixpunkt von $E_{CT_{\Sigma}}$ darstellen, sondern auch als Entfaltung einer Coalgebra. Dies folgt aus der Tatsache, dass CT_{Σ} eine finale $co\Sigma$ -Algebra ist, wobei $co\Sigma$ die folgendermaßen aus (der konstruktiven Signatur) $\Sigma = (S, F)$ gebildete destruktive Signatur ist:

$$co\Sigma = (S, \{d_s : s \rightarrow \coprod_{f:e \rightarrow s \in F} e \mid s \in S\}).$$

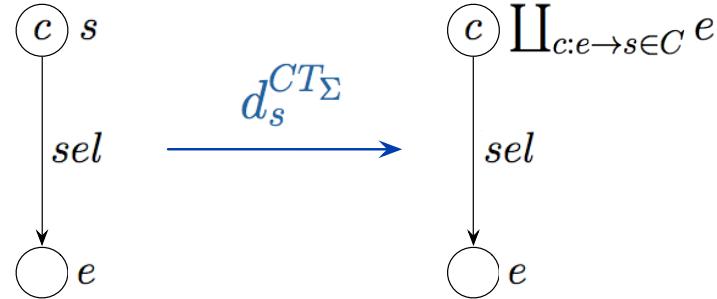
Satz 19.5 CT_{Σ} ist eine finale $co\Sigma$ -Algebra.

Beweis. CT_{Σ} ist eine $co\Sigma$ -Algebra: Sei $I \in \mathcal{I}$ und $\{e_i\} \subseteq \mathcal{T}_p(S)$.

- Für alle $s \in S, c : e \rightarrow s \in C, t \in CT_{\Sigma,e}$,

$$d_s^{CT_{\Sigma}}(c\{sel \rightarrow t\}) =_{def} c\{sel \rightarrow t\}.$$

Als Argument von $d_s^{CT_\Sigma}$ hat der Term $c\{sel \rightarrow t\}$ den Typ s , während er als Ergebnis den Typ $\coprod_{c:e \rightarrow s \in C} e$ hat. Aus diesem Grund haben wir vorausgesetzt, dass $\{c \in C \mid ran(c) = s\}$ zu \mathcal{I} gehört.



- Für alle $t_i \in CT_{\Sigma, e_i}$, $i \in I$, and $k \in I$, $\pi_k(tup\{i \rightarrow t_i \mid i \in I\}) =_{def} t_k$.
- Für alle $i \in I$ and $t \in CT_{\Sigma, e_i}$, $\iota_i(t) =_{def} i\{sel \rightarrow t\}$.

CT_Σ und $DT_{co\Sigma}$ sind $co\Sigma$ -isomorph:

Da CT_Σ und $DT_{co\Sigma}$ $co\Sigma$ -Algebren sind und $DT_{co\Sigma}$ darüberhinaus final ist, bleibt zu zeigen, dass

$$g =_{def} unfold^{CT_\Sigma} : CT_\Sigma \rightarrow DT_{co\Sigma}$$

bijektiv ist. Eine S -sortige Funktion $h : DT_{co\Sigma} \rightarrow CT_\Sigma$ ist wie folgt induktiv definiert: Sei $I \in \mathcal{I}$ und $\{e_i\} \subseteq \mathcal{T}_p(S)$.

- $h_I = id_I$.
- Für alle $c : e \rightarrow s \in C$ und $t \in CT_{\Sigma, e}$, $h_s(\epsilon\{d_s \rightarrow c\{sel \rightarrow t\}\}) = c\{sel \rightarrow h_e(t)\}$.

- Für alle $t_i \in CT_{\Sigma, e_i}$, $i \in I$,

$$h_{\prod_{i \in I} e_i}(tup\{i \rightarrow t_i \mid i \in I\}) = tup\{i \rightarrow h_{e_i}(t_i) \mid i \in I\}.$$

- Für alle $k \in I$ und $t \in CT_{\Sigma, e_k}$,

$$h_{\coprod_{i \in I} e_i}(k\{sel \rightarrow t\}) = k\{sel \rightarrow h_{e_k}(t)\}.$$

Dass g und h invers zueinander sind, wird in [34] gezeigt. □

Sei $E : V \rightarrow T_\Sigma(V)$ ein iteratives Σ -Gleichungssystem.

E induziert die folgende $co\Sigma$ -algebra $T(E)$: Sei $s \in S$, $I \in \mathcal{I}$ und $\{e_i\} \subseteq \mathcal{T}_p(S)$.

- $T(E)_s = T_\Sigma(V)_s$.
- Für alle $c : e \rightarrow s \in C$, $t \in T_\Sigma(V)_e$, $d_s^{T(E)}(c\{sel \rightarrow t\}) =_{def} c\{sel \rightarrow t\}$.
- Für alle $x \in V_s$, $d_s^{T(E)}(x) =_{def} d_s^{T(E)}(E(x))$.
- Für alle $t_i \in T_\Sigma(V)_{e_i}$, $i \in I$, und $k \in I$, $\pi_k(tup\{i \rightarrow t_i \mid i \in I\}) =_{def} t_k$.
- Für alle $i \in I$ und $t \in T_\Sigma(V)_{e_i}$, $\iota_i(t) =_{def} i\{sel \rightarrow t\}$.

(8) $V \xrightarrow{\text{inc}_V} T_\Sigma(V) \xrightarrow{\text{unfold}^{T(E)}} DT_{co\Sigma} \xrightarrow{h} CT_\Sigma$ löst E in CT_Σ (siehe [34]).

(9) $g : V \rightarrow CT_\Sigma$ löst E genau dann in CT_Σ , wenn $g^* : T(E) \rightarrow CT_\Sigma$ $co\Sigma$ -homomorph ist (siehe [34]).

Satz 19.6

(coalgebraische Version von Satz 19.4)

Jedes iterative Σ -Gleichungssystem $E : V \rightarrow T_\Sigma(V)$ hat genau eine Lösung in CT_Σ .

Beweis. Wegen (8) hat E eine Lösung in CT_Σ . Angenommen, $g, h : V \rightarrow CT_\Sigma$ lösen E in CT_Σ . Dann sind $g^*, h^* : T(E) \rightarrow CT_\Sigma$ wegen (9) $co\Sigma$ -homomorph. Da CT_Σ eine finale $co\Sigma$ -Algebra ist, gilt $g^* = h^*$, also $g = g^* \circ inc_V = h^* \circ inc_V = h$. \square

Sei $cs \in StackCom^*$. Aus (7), (8) und Satz 19.4 (oder 19.6) folgt, dass $execute'$ die Entfaltung von $V(cs)$ in CT_Σ mit der Faltung der entstandenen Σ -Terme in $Mach$ verknüpft:

$$\begin{array}{ccc}
 V(cs) & \xrightarrow{\textcolor{blue}{execute'}} & Mach \\
 \downarrow inc_V & = & \uparrow fold_\omega^{Mach} \\
 T(E) & \xrightarrow{\textcolor{blue}{h \circ unfold^{T(E)}}} & CT_\Sigma
 \end{array}$$

20 Literatur

- [1] L.S. Bobrow, M.A. Arbib, *Discrete Mathematics: Applied Algebra for Computer and information Science*, W.B. Saunders Company 1974
- [2] M. Bonsangue, J. Rutten, A. Silva, *An Algebra for Kripke Polynomial Coalgebras*, Proc. 24th LICS (2009) 49-58
- [3] J.A. Brzozowski, *Derivatives of regular expressions*, Journal ACM 11 (1964) 481–494
- [4] H. Comon et al., *Tree Automata: Techniques and Applications*, Inria 2008
- [5] F. Drewes, ed., *Tree Automata, Course notes*, Umeå University, Sweden 2009
- [6] J. Gibbons, R. Hinze, *Just do It: Simple Monadic Equational Reasoning*, Proc. 16th ICFP (2011) 2-14 Jeremy Gibbons and Ralf Hinze
- [7] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, Journal ACM 24 (1977) 68-95
- [8] H.H. Hansen, J. Rutten, *Symbolic Synthesis of Mealy Machines from Arithmetic Bitstream Functions*, Scientific Annals of Computer Science (2010) 97-130
- [9] R. Hinze, *Functional Pearl: Streams and Unique Fixed Points*, Proc. 13th ICFP (2008) 189-200

- [10] R. Hinze, D.W.H. James, *Proving the Unique-Fixed Point Principle Correct*, Proc. 16th ICFP (2011) 359-371
- [11] J.E. Hopcroft, R. Motwani, J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley 2001; deutsch: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*, Pearson Studium 2002
- [12] G. Hutton, *Fold and unfold for program semantics*, Proc. 3rd ICFP (1998) 280-288
- [13] B. Jacobs, *Introduction to Coalgebra*, draft, Radboud University Nijmegen 2012
- [14] B. Jacobs, *A Bialgebraic Review of Deterministic Automata, Regular Expressions and Languages*, in: K. Futatsugi et al. (eds.), Goguen Festschrift, Springer LNCS 4060 (2006) 375–404
- [15] B. Klin, *Bialgebras for structural operational semantics: An introduction*, Theoretical Computer Science 412 (2011) 5043-5069
- [16] D. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (1968) 127-145; Corrections: Math. Systems Theory 5 (1971) 95-96
- [17] D. Kozen, *Realization of Coinductive Types*, Proc. Math. Foundations of Prog. Lang. Semantics 27, Carnegie Mellon University, Pittsburgh 2011
- [18] C. Kupke, J. Rutten, *On the final coalgebra of automatic sequences*, in: Logic and Program Semantics, Springer LNCS 7230 (2012) 149-164

- [19] A. Kurz, *Specifying coalgebras with modal logic*, Theoretical Computer Science 260 (2001) 119–138
- [20] P.J. Landin, *The Mechanical Evaluation of Expressions*, Computer Journal 6 (1964) 308–320
- [21] W. Martens, F. Neven, Th. Schwentick, *Simple off the shelf abstractions for XML Schema*, SIGMOD Record 36,3 (2007) 15-22
- [22] K. Meinke, J.V. Tucker, *Universal Algebra*, in: Handbook of Logic in Computer Science 1 (1992) 189 - 368
- [23] E. Moggi, *Notions of Computation and Monads*, Information and Computation 93 (1991) 55-92
- [24] F.L. Morris, *Advice on Structuring Compilers and Proving Them Correct*, Proc. ACM POPL (1973) 144-152
- [25] T. Mossakowski, L. Schröder, S. Goncharov, *A Generic Complete Dynamic Logic for Reasoning About Purity and Effects*, Proc. FASE 2008, Springer LNCS 4961 (2008) 199-214
- [26] R.N. Moll, M.A. Arbib, A.J. Kfoury, *Introduction to Formal Language Theory*, Springer (1988)

- [27] P. Padawitz, *Church-Rosser-Eigenschaften von Graphgrammatiken und Anwendungen auf die Semantik von LISP*, Diplomarbeit, TU Berlin 1978
- [28] P. Padawitz, *Formale Methoden des Systementwurfs*, TU Dortmund 2015
- [29] P. Padawitz, *Übersetzerbau*, TU Dortmund 2015
- [30] P. Padawitz, *Modellieren und Implementieren in Haskell*, TU Dortmund 2015
- [31] P. Padawitz, *Logik für Informatiker*, TU Dortmund 2015
- [32] P. Padawitz, *Algebraic Model Checking*, in: F. Drewes, A. Habel, B. Hoffmann, D. Plump, eds., Manipulation of Graphs, Algebras and Pictures, *Electronic Communications of the EASST* Vol. 26 (2010)
- [33] P. Padawitz, *From Modal Logic to (Co)Algebraic Reasoning*, TU Dortmund 2013
- [34] P. Padawitz, *Fixpoints, Categories, and (Co)Algebraic Modeling*, TU Dortmund 2017
- [35] H. Reichel, *An Algebraic Approach to Regular Sets*, in: K. Futatsugi et al., Goguen Festschrift, Springer LNCS 4060 (2006) 449-458
- [36] W.C. Rounds, *Mappings and Grammars on Trees*, Mathematical Systems Theory 4 (1970) 256-287
- [37] J. Rutten, *Processes as terms: non-wellfounded models for bisimulation*, Math. Struct. in Comp. Science 15 (1992) 257-275

- [38] J. Rutten, *Automata and coinduction (an exercise in coalgebra)*, Proc. CONCUR '98, Springer LNCS 1466 (1998) 194–218
- [39] J. Rutten, *Behavioural differential equations: a coinductive calculus of streams, automata, and power series*, Theoretical Computer Science 308 (2003) 1-53
- [40] J. Rutten, *A coinductive calculus of streams*, Math. Struct. in Comp. Science 15 (2005) 93-147
- [41] A. Silva, J. Rutten, *A coinductive calculus of binary trees*, Information and Computation 208 (2010) 578–593
- [42] D. Turi, G. Plotkin, *Towards a Mathematical Operational Semantics*, Proc. LICS 1997, IEEE Computer Society Press (1997) 280–291
- [43] J.W. Thatcher, E.G. Wagner, J.B. Wright, *More on Advice on Structuring Compilers and Proving Them Correct*, Theoretical Computer Science 15 (1981) 223-249
- [44] J.W. Thatcher, J.B. Wright, *Generalized Finite Automata Theory with an Application to a Decision Problem of Second-Order Logic*, Theory of Computing Systems 2 (1968) 57-81
- [45] Ph. Wadler, *Monads for Functional Programming*, Proc. Advanced Functional Programming, Springer LNCS 925 (1995) 24-52
- [46] R. Wilhelm, H. Seidl, *Übersetzerbau - Virtuelle Maschinen*, Springer 2007

21 Index

- C -Abschluss, 360
 E -Normalform, 101
 E -Reduktionsrelation, 101
 E -Äquivalenz, 98
 $DAut(X, Y)$, 36
 Σ -Algebra, 38
 Σ -Gleichung, 90
 Σ -Homomorphismus, 38
 Σ -Isomorphismus, 38
 Σ -Kongruenz, 88
 $\Sigma(G)$, 120
 λ -Abstraktion, 216
 λ -Applikation, 216
 ω -CPO, 387
 ω -stetig, 389
 ω -stetige Funktion, 389
 $\langle a \rangle$, 77
 $\mathcal{T}_p(S)$, 28
 $state(\varphi)$, 193
 $(++)$, 226
- abgeleitetes Attribut, 266
 $Abl(G)$, 141
Ableitungsbaum, 141
Ableitungsrelation, 115
absolute Adresse, 287
abstrakte Syntax, 120
Aktion eines Monoids, 70
akzeptierte Sprache, 71
all, 234
any, 234
Applikationsoperator, 220
Attribut, 240
Ausnahmefunktor, 151
Basisadresse, 287
Baumautomat, 79
Baumsprache, 80
Befehlszähler, 286
Bild, 15
Bildalgebra, 38
bind-Operator, 156

- bottom-up-Compiler, 187
Brzozowski-Automat, 59
Brzozowski-Gleichungen, 103

Calculus of Communicating Systems, 34
CCS(Act), 34
CFG, 109
charakteristische Funktion, 15
Coalgebra, 38
Coextension, 64
cofreie Algebra, 65
Coinduktion, 358, 361
Coinduktionsprinzip, 90
Compiler, 8
Compilermonade, 160
const, 223
Copotenzfunktor, 152
Coprodukt, 25
Coterm, 45
curry, 224

denotationelle Semantik, 367, 392

destruktive Signatur, 31
Destruktor, 31, 240
deterministischer Baum, 19
Diagonale, 15
Diagonalfunktor, 150
Display, 287
Domain, 31
Domain-Tuplung, 25
drop, 227

eindeutig, 138
elem, 234
erkannte Sprache, 71
Erreichbarkeitsfunktion, 60
execute, 282, 293
executeCom, 281
Extension, 61

Färbung, 64
fail, 330
field label, 240
filter, 234

finale Algebra, 66
Fixpunktsatz für CFGs, 372
Fixpunktsatz für nicht-linksrekursive CFGs,
382
Fixpunktsatz von Kleene, 390
flache Erweiterung von A , 387
flip, 224
fold, 62
fold2, 231
foldl, 230
foldr, 233
freie Algebra, 62
Functor, 328
Funktion höherer Ordnung, 218
Funktionsapplikation, 216
Funktionseinschränkung, 15
Funktionslifting, 23, 26
Funktionsprodukt, 24
Funktionssumme, 27
Funktionsupdate, 15
Funktör, 149
generischer Compiler, 165
Gleichungssystem einer CFG, 372
goto-Tabelle, 193
Graph, 15
Grundcoterms, 46
Grundinstanz, 96
Grundterm, 44
halbgeordnete Menge, 387
Halbordnung, 387
head, 226
id, 223
Identität, 14
Identitätsfunktör, 150
Individuenvariable, 215, 222
Induktion, 354
Induktionsprinzip, 86
init, 227
initiale Algebra, 63, 401
initialer Automat, 71
Injektion, 25

Inklusion, 14
Instanz, 216
Instanz eines Terms, 96
Instanz eines Typs, 222
Instanzen von E , 98
Interpreter, 8
Invariante, 84
isomorph, 16, 38
iteratives Gleichungssystem, 370

JavaLight, 111
JavaLightP, 295
javaStackP, 299

Kategorie, 148
Kern, 15, 89
Kompositionsoperator, 220
Kongruenz modulo C , 360
Konkatenation, 19
konstanter Funktor, 150
konstruktive Signatur, 31
Konstruktor, 31

kontextfreie Grammatik, 109
Lösung eines iterativen Gleichungssystems, 370
LAG-Algorithmus, 324
Lambeks Lemma, 356
last, 227
leere Wort, 18
Leserfunktör, 152
lines, 229
linksrekursiv, 115
Liste, 18
Listenfunktör, 151
Listenkomprehension, 235
LL-Compiler, 343
lookup, 239
LR(k)-Grammatik, 188
LR-Automat für G , 193

map, 228
mapM, 333
Matching, 216
Medvedev-Automat, 68

Mengenfunktor, 151
Methode, 240
Monad, 330
Monade, 153
Monoid, 40
monomorph, 222
monotone Algebra, 389
monotone Funktion, 389
natürliche Abbildung, 89
Nerode-Relation, 82
notElem, 234
Operation, 31, 38
Parser, 8, 148
Paull-Unger-Verfahren, 92
Plusmonade, 158
polymorph, 222
Potenzfunktor, 152
präfixabgeschlossen, 19
Produkt, 21
Produktextension, 21
Produktfunktor, 150
Produktyp, 28
Projektion, 21
Quotientenalgebra, 88
Range, 31
Range-Tuplung, 21
rationaler Term, 48
Realisierung, 82
Realisierung einer Verhaltensfunktion, 71
Realisierung eines Coterms, 71
Rechtsreduktion, 187
Redukt, 356
Reduktionsfunktion, 101
Reg(BL), 34
Regel, 109
reguläre Sprache, 63
regulärer Ausdruck, 34
rekursives Ψ -Gleichungssystem, 355
Relativadresse, 287
Resultatadresse, 307

return, 330
SAB, 122
Scanner, 7
Schreiberfunktor, 152
Sektion, 219
sequence, 333
Signatur, 31
Sprache über A , 18
Sprache einer CFG, 138
Sprache eines regulären Ausdrucks, 63
StackCom, 280
State, 280
statischer Vorgänger, 287
strikt, 389
strukturell-operationelle Semantik, 366
Substitution, 95
Summe, 25
Summenextension, 25
Summentyp, 28
symbolische Adressen, 288
Symboltabelle, 297
syntaktische Kongruenz, 94
syntaktisches Monoid, 94
Syntaxbaum, 120, 137
tail, 226
take, 227
Term, 42
Termfaltung, 62
Terminal, 109
top-down-Parser, 167
Trägermenge, 38
transientes Attribut, 266
Transitionsfunktor, 152
Transitionsmonoid, 93
Typ, 29
typ, 28
Typdeskriptor, 293
Typklassen, 245
Typkonstruktur, 215
Typvariable, 215

uncurry, 224
unfold, 65
unit-Typ, 215
universelle Eigenschaft, 21
unlines, 229
Unteralgebra, 84
unwords, 229
update, 223
Urbild, 15

Variablenbelegung, 61
vererbtes Attribut, 266
Verhaltensfunktion, 57, 80
Verhaltenskongruenz, 90

Wildcard, 222
wohlfundierter Baum, 19
Word(G), 137
words, 229
Wort, 18
Wortalgebra, 137

zip, 228
zipWith, 228
Zustandsentfaltung, 65
Zustandsmonade, 332