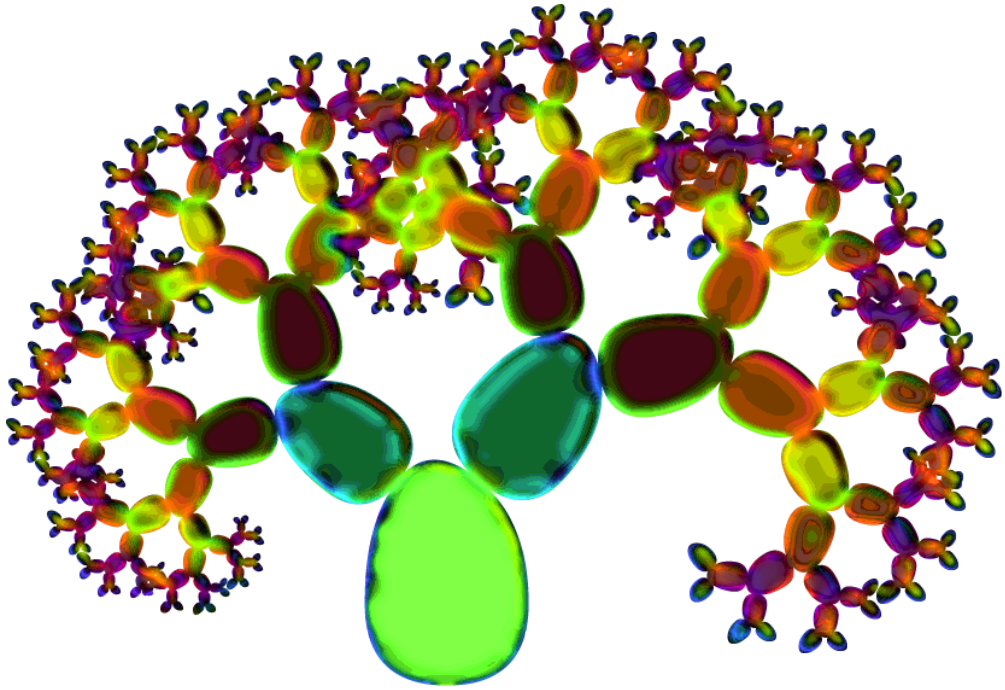


Übersetzerbau

Algebraic Compiler Construction

Peter Padawitz, TU Dortmund, Germany

29. Juli 2023

(actual version: http://fldit-www.cs.uni-dortmund.de/∼peter/CbauFolien.pdf)

Inhalt

Zur Navigation auf Titel, nicht auf Seitenzahlen klicken!

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**. Bezüglich 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.

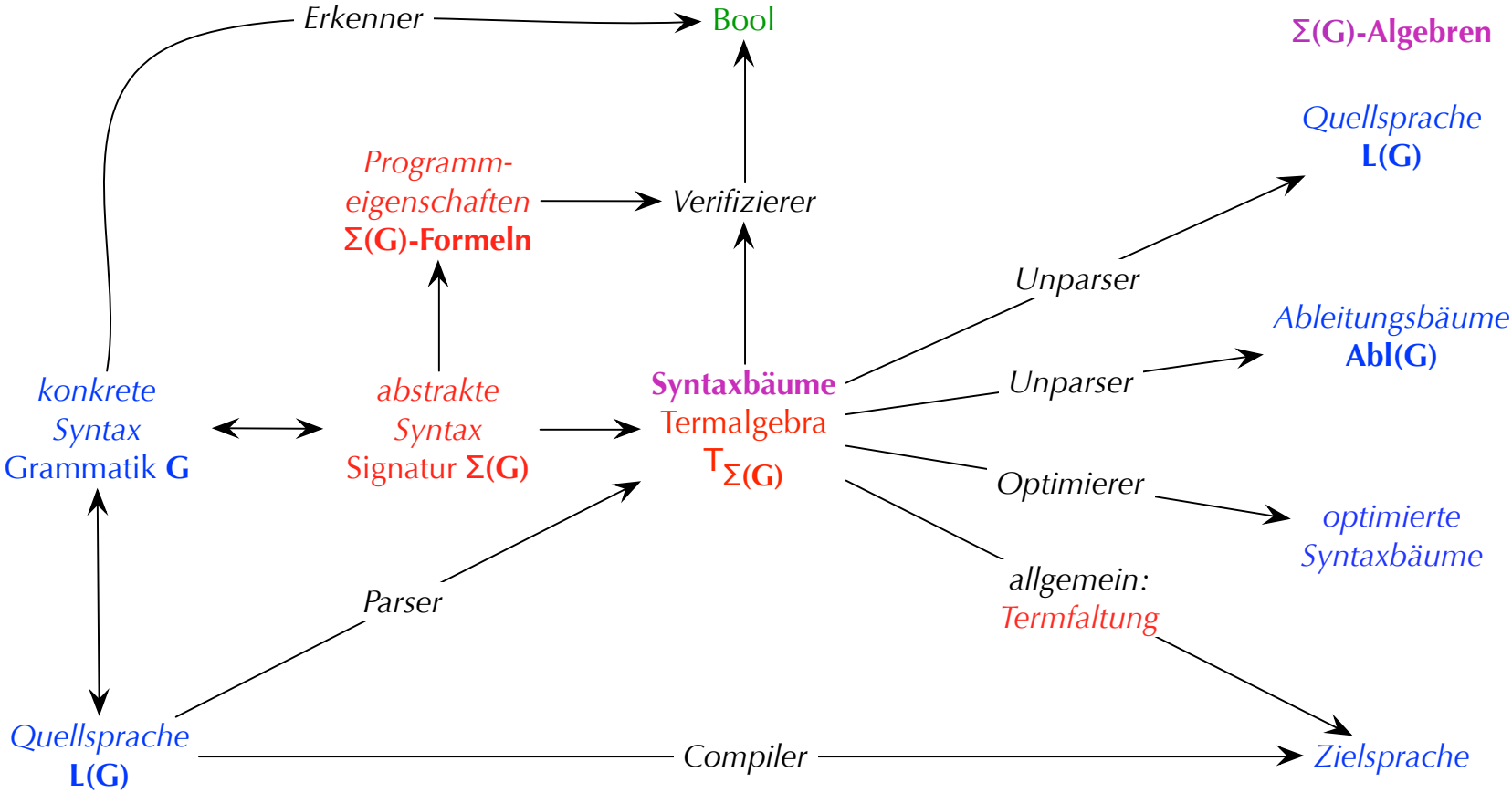
Die Entwicklung dieser Prinzipien und ihrer formalen Grundlagen wurzelt in der **mathematischen Logik** und dort vor allem in der **Theorie formaler Sprachen** und der **Universellen Algebra**, deren Anwendung im Compilerbau in dieser LV eine zentrale Rolle spielen wird. Die Aufgabe, Übersetzer zu schreiben, 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 Scannern, 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.

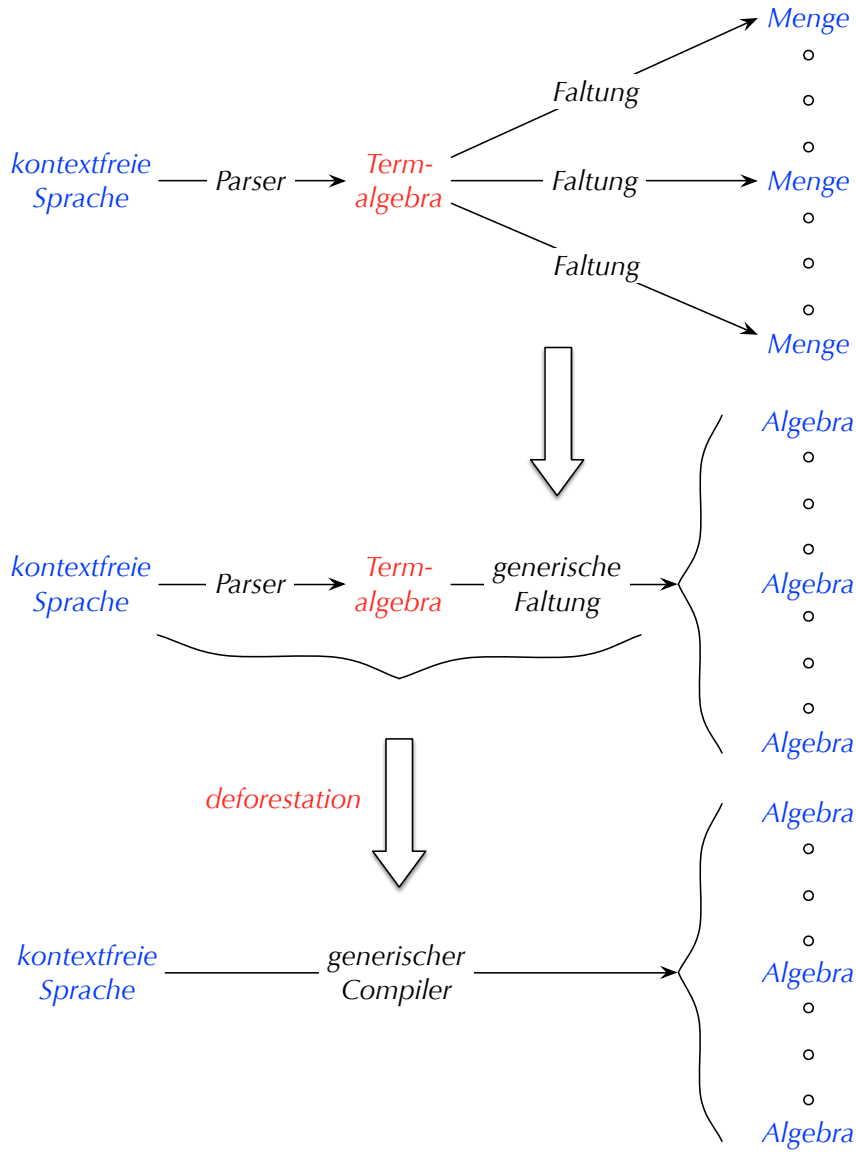
Der algebraische Ansatz klärt nicht nur die Bezüge zwischen den oben genannten Programmen, sondern auch eine Reihe weiterer in der Compilerbauliteratur meist sehr speziell definierter und daher im Kern vager Begriffe wie *attributierter Grammatiken* und *mehrpässiger Compilation*.

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 instanziiert 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* instanziiert 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



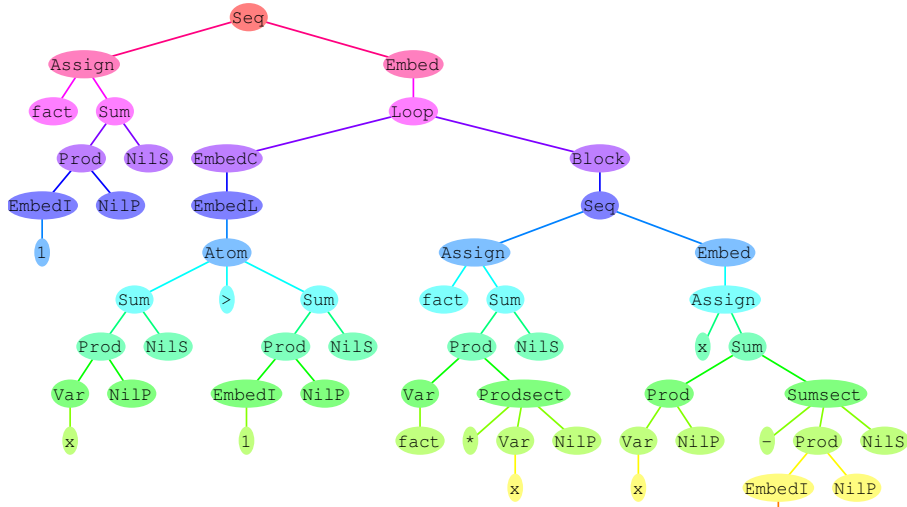
Der Weg zum generischen Compiler

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

Eingabewort (Element der *JavaLight*-Algebra *javaWord*)

parse

fold



Syntaxbaum
(Element der *JavaLight*-Algebra *javaTerm)*

fold

fold

$$trans : (String \to \mathbb{Z}) \to (String \to \mathbb{Z})$$
$$store \mapsto \lambda str.if\ str = x\ then\ 0$$
$$else\ if\ str = fact\ then\ store(x)!\ else\ store(str)$$

Zustandstransitionsfunktion
(Element der *JavaLight*-Algebra *javaState*)

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

Assemblerprogramm
(Element der *JavaLight*-Algebra *javaStack*)

2 Algebraische Modellierung

2.1 Mengen und Funktionen

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

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 $id_A : A \to A$ die durch $id_A(a) = a$ für alle $a \in A$ definierte **Identität auf** A.

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

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

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

Δ_A^n	$=_{def}$	$\{(a_1, \ldots, a_n) \in A^n \mid \forall\, 1 \leq i < n : a_i = a_{i+1}\}$	**Diagonale** von A
$f(C)$	$=_{def}$	$\{f(a) \mid a \in C\}$	**Bild** von C unter f
		$= \mathcal{P}(f)(C)$ (siehe Mengenfunktor)	
$img(f)$	$=_{def}$	$f(A)$	
$f^{-1}(D)$	$=_{def}$	$\{a \in A \mid f(a) \in D\}$	**Urbild** von D unter f
$ker(f)$	$=_{def}$	$\{(a, a') \in A \times A \mid f(a) = f(a')\}$	**Kern** von f
$graph(f)$	$=_{def}$	$\{(a, f(a)) \in A \times B \mid a \in A\}$	**Graph** von f
$f\|_C : C$	$\to$	B	**Einschränkung** von f
c	$\mapsto$	$f(c)$	auf C
$f[b/a] : A$	$\to$	B	**Update** von f
c	$\mapsto$	$if\ c = a\ then\ b\ else\ f(c)$	
$\chi : \mathcal{P}(A)$	$\to$	2^A	**charakteristische**
C	$\mapsto$	$\lambda a.if\ a \in C\ then\ 1\ else\ 0$	**Funktion** von C

$$
\begin{aligned}
f \text{ ist surjektiv} &\Leftrightarrow_{def} img(f) = B \\
f \text{ ist injektiv} &\Leftrightarrow_{def} ker(f) = \Delta_A^2 \\
f \text{ ist bijektiv} &\Leftrightarrow_{def} \exists\, g : B \to A : g \circ f = id_A \wedge f \circ g = id_B
\end{aligned}
$$

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:

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

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

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

$$\bigtimes_{i \in I} A_i \;=\; \{f : I \to \bigcup_{i \in I} A_i \mid \forall\, i \in I : f(i) \in A_i\}.$$

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

$f \in \bigtimes_{i \in I} A_i$ wird auch als I-Tupel $(f(i))_{i \in I}$ geschrieben.

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

Im Fall $I = [n]$ für ein $n > 1$ schreibt man auch $A_1 \times \cdots \times A_n$ anstelle von $\bigtimes_{i \in I} A_i$ 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 $A_1 + \cdots + A_n$ 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:

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

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

$$\begin{aligned} \epsilon \cdot w &= w, \\ v \cdot \epsilon &= v, \\ v \cdot w &= (a_1, \ldots, a_m, b_1, \ldots, b_n), \\ B \cdot C &= \{v \cdot w \mid v \in B,\ w \in C\}. \end{aligned}$$

Für alle $w \in A^+$ bezeichnet $head(w)$ das erste Element von w und $tail(w)$ die Restliste, d.h. es gilt $w = head(w) \cdot tail(w)$. $|w|$ 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 $def(t)$ 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 $w \in def(t)$. 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. In Anlehnung an Haskell-Datentypen mit Attributen (*field names*) notieren wir einen markierten Baum $t : A^* \rightharpoonup B$ auch wie folgt:

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

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

2.2 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 disjunkten Vereinigung von A hat, nennt man *Produkt* bzw. *Summe von* A:

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

Das Paar (P, π) heißt **Produkt von** A, wenn es für alle Tupel $(f_i : B \to A_i)_{i\in I}$ genau eine Funktion $f : B \to P$ 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 f **Produktextension** oder **Range-Tuplung von** $(f_i)_{i\in I}$.

Da f durch $(f_i)_{i\in I}$ bestimmt ist, wird f mit $\langle f_i\rangle_{i\in I}$ und im Fall $I = [n]$, $n > 1$, auch mit $\langle f_1, \ldots, f_n\rangle$ bezeichnet.

Demnach gilt:

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

Mit $f = \lambda x.a : 1 \to P$ und $g = \lambda x.b : 1 \to P$ 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.

$\forall\, i \in I : \pi_i(a) = \pi_i(b) \;\Rightarrow\; \forall\, i \in I : \pi_i \circ f = \pi_i \circ g \overset{(2)}{\Rightarrow} f = g \;\Rightarrow\; a = f(\epsilon) = g(\epsilon) = b.$ ❑

$\mathsf{X}_{i\in I} A_i$ ist ein Produkt von A.

Die Projektionen und Produktextensionen für $\mathsf{X}_{i\in I} A_i$ sind wie folgt definiert:

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

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

$$\pi_i \circ h' \circ h = \pi_i \circ \langle \pi'_i \rangle_{i \in I} \circ \langle \pi_i \rangle'_{i \in I} \overset{(1)}{=} \pi'_i \circ \langle \pi_i \rangle'_{i \in I} \overset{(1)}{=} \pi_i,$$
$$\pi'_i \circ h \circ h' = \pi'_i \circ \langle \pi_i \rangle'_{i \in I} \circ \langle \pi'_i \rangle_{i \in I} \overset{(1)}{=} \pi_i \circ \langle \pi'_i \rangle_{i \in I} \overset{(1)}{=} \pi'_i$$

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

Beweis von (ii). Siehe [35], Kapitel 2. ❏

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

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

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

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

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

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

Die Funktion

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

heißt **Produkt von** f.

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

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

Folgerungen:

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

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

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

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

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

$$f \circ \iota_i = f_i \tag{3}$$

für alle $i \in I$.

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

Demnach gilt:

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

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 $f = \lambda x.0 : S \to 2$ und

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

(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 $\iota_i(a) = \iota_j(b)$:

$$(a, i) = f_i(a) \overset{(3)}{=} [f_i]_{i\in I}(\iota_i(a)) = [f_i]_{i\in I}(\iota_j(b)) \overset{(3)}{=} f_j(b) = (b, j). \qquad \text{❑}$$

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

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 $h : S \to S'$ 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 $f : B \to \coprod_{i\in I} B_i$ wird wie folgt zur Funktion

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

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

$$lift_A(f)((g_i)_{i\in I}) \;=_{def}\; [g_i]_{i\in I} \circ f.$$

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

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

Die Funktion

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

heißt **Summe von** f.

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

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

Folgerungen:

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

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

2.3 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:

- $S \subseteq \mathcal{T}_p(S)$.
- $1 \in \mathcal{T}_p(S)$. (**unit type**)
- Für alle Mengen I mit $|I| > 1$ und $(e_i)_{i\in I} \in \mathcal{T}_p(S)^I$, $I, \prod_{i\in I} e_i, \coprod_{i\in I} e_i \in \mathcal{T}_p(S)$.

1 und alle Mengen mit mindestens zwei Elementen heißen **Basistypen**. Ein Typ der Form $\coprod_{i\in I} e_i$ oder $\prod_{i\in I} e_i$ heißt **Summen-** bzw. **Produkttyp** mit **Summanden** bzw. **Faktoren** e_i, $i \in I$. Wir setzen voraus, dass kein Summand (Faktor) eines Summentyps (Produkttyps) selbst ein Summentyp bzw. Produkttyp ist.

Für alle $n > 1$, $e_1, \ldots, e_n, e \in \mathcal{T}_p(S)$ und Mengen I mit $|I| > 1$,

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

$$
\begin{aligned}
e^* &=_{def} \coprod_{n\in\mathbb{N}} e^n,\\
\prod_{i\in 1} e_i &=_{def} e_i,\\
\coprod_{i\in 1} e_i &=_{def} e_i.
\end{aligned}
$$

Eine S-**sortige** oder S-**indizierte Menge** ist ein S-Tupel $A = (A_s)_{s\in S}$ von Mengen. Manchmal steht A auch für die Vereinigung der Mengen A_s über alle $s \in S$. Auch schreibt man A für A_s, falls S einelementig ist.

Seien $A = (A_s)_{s\in S}$ und $B = (B_s)_{s\in S}$ S-sortige Mengen. Eine S-**sortige Funktion** $f : A \to B$ 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.

Eine S-sortige Menge A wird wie folgt zur $\mathcal{T}_p(S)$-sortigen Menge erweitert: Für alle nichtleeren Mengen I, $A_I = I$, und für alle Mengen I mit $|I| > 1$ und $(e_i)_{i\in I} \in \mathcal{T}_p(S)^I$,

$$
\begin{aligned}
A_{\prod_{i\in I} e_i} &= \bigtimes_{i\in I} A_{e_i} \text{ (oder eine dazu isomorphe Menge)},\\
A_{\coprod_{i\in I} e_i} &= \biguplus_{i\in I} A_{e_i} \text{ (oder eine dazu isomorphe Menge)}.
\end{aligned}
$$

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

Eine S-sortige Funktion $h : A \to B$ wird wie folgt zur $\mathcal{T}_p(S)$-sortigen Menge erweitert:

Für alle nichtleeren Mengen I, $h_I = id_I$, und für alle Mengen I mit $|I| > 1$ und $(e_i)_{i\in I} \in \mathcal{T}_p(S)^I$,

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

Eine **Signatur** $\Sigma = (S, F)$ besteht aus einer Menge S von Sorten und einer $\mathcal{T}_p(S)^2$-sortierten Menge F, deren Elemente wir **Operationen** nennen.

Man schreibt $f : e \to e' \in F$ anstelle von $f \in F_{e\times e'}$.

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

$f \in F$ heißt **Konstruktor**, falls $dom(f)$ kein Summentyp und $ran(f)$ eine Sorte ist.

$f \in F$ heißt **Destruktor**, falls $dom(f)$ eine Sorte und $ran(f)$ kein Produkttyp 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 eines (initialen bzw. finalen) Standardmodells der jeweiligen Signatur (siehe 2.6 und 2.12).

Gelegentlich tauchen in der Sortenmenge S aus Produkten oder Summen zusammengesetzte Typen auf. Diese sind dann als zusätzliche Sorten mit eigenen Konstruktoren (i.d.R. Injektionen) bzw. Destruktoren (i.d.R. Projektionen) zu verstehen. Damit können wir die Domains der Konstruktoren anderer Sorten auf **flache** Produkttypen und die Ranges der Destruktoren anderer Sorten auf **flache** Summentypen beschränken, d.h. alle Faktoren bzw. Summanden sind Sorten oder beobachtbare Typen.

Seien $X, Y, Act, X_1, \ldots, X_n, Y_1, \ldots, Y_n, E_1, \ldots, E_n$ beliebige Mengen,

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

und BL eine endliche Menge – **Basissprachen** (*base languages*) genannter – Mengen, die $\emptyset$ und $\{\epsilon\}$ enthält.

2.4 Konstruktive Signaturen

- Mon ↬ Unmarkierte binäre Bäume (Monoide sind Mon-Algebren; siehe 2.6.)

$$S = \{mon\}, \quad F = \{\ one : 1 \to mon, \quad mul : mon \times mon \to mon\ \}.$$

- Nat ↬ $\mathbb{N}$

$$S = \{nat\}, \quad F = \{\ zero : 1 \to nat, \quad succ : nat \to nat\ \}.$$

- $Dyn(X, Y)$ ↬ $X^* \times Y$ (dynamics)

$$S = \{list\}, \quad F = \{\ nil : Y \to list, \quad cons : X \times list \to list\ \}.$$

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

- $Bintree(X) \rightsquigarrow$ binäre Bäume endlicher Tiefe mit Knotenmarkierungen aus X

$$S = \{btree\}, \quad F = \{\ empty : 1 \to btree,\ bjoin : btree \times X \times btree \to btree\ \}.$$

- $Tree(X) \rightsquigarrow$ Bäume endlicher Tiefe und endlichen Knotenausgrads mit Knotenmarkierungen aus X

$$S = \{tree, trees\}, \quad F = \{\ join : X \times trees \to tree,\ nil : 1 \to trees,\ cons : tree \times trees \to trees\ \}.$$

- $Reg(BL) \rightsquigarrow$ reguläre Ausdrücke über BL

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

- $CCS(Act)$ ↬ Calculus of Communicating Systems (kommunizierende Prozesse)

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

2.5 Destruktive Signaturen

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

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

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

$$S = \{list\}, \quad F = \{\ head : list \to X, \\ tail : list \to list\ \}.$$

- $coDyn(X, Y) \rightsquigarrow X^* \times Y + X^{\mathbb{N}}$ (codynamics)

$$S = \{list\}, \quad F = \{split : list \to X \times list + Y\}.$$

- $coList(X) = coDyn(X, 1) \rightsquigarrow X^* + X^{\mathbb{N}}$

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

$$S = \{btree\},$$
$$F = \{split : btree \to 1 + btree \times X \times btree\}.$$

- $Infbintree(X)$ ⇝ binäre Bäume unendlicher Tiefe mit Knotenmarkierungen aus X

$$S = \{btree\}, \quad F = \{ \; root : btree \to X, \; left, right : btree \to btree \; \}.$$

- $Med(X)$ (Medvedev-Automaten)

$$\begin{array}{lcll} S & = & \{state\}, & \text{(Zustandsmenge)} \\ F & = & \{\delta : state \to state^X\}, & \text{(Transitionsfunktion)} \end{array}$$

- $DAut(X, Y)$ ⇝ Y^{X^*} (deterministische Moore-Automaten mit Eingabemenge X und Ausgabemenge Y = Medvedev-Automaten mit Ausgabefunktion $\beta : state \to Y$.

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

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

$$S = \{tree\}, \quad F = \{\delta : tree \to (Act \times tree)^*\}.$$

- $Class(BS)$ ⇝ Verhalten einer Objektklasse mit n Methoden [20]

$$\begin{array}{lcl} S & = & \{state\}, \\ F & = & \{m_i : state \to ((Y_i \times state) + E_i)^{X_i} \mid 1 \leq i \leq n\}. \end{array}$$

2.6 Algebren

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

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

von Funktionen, den **Operationen** von $\mathcal{A}$. In manchen Kontexten können eine Algebra und ihre Trägermenge gleich benannt sein.

Für alle $s \in S$ heißt A_s **Trägermenge** (*carrier set*) oder **Interpretation von s in $\mathcal{A}$**. Für alle $f : e \to e' \in F$ heißt $f^{\mathcal{A}} : A_e \to A_{e'}$ **Interpretation von f in A**.

Alg_Σ bezeichnet die Klasse aller Σ-Algebren. Algebren destruktiver Signaturen werden auch **Coalgebren** genannt.

Sei Σ' eine Teilsignatur von Σ und $\mathcal{A}$ eine Σ-Algebra. Die Σ'-Algebra, die alle Sorten und Operationen von Σ' wie $\mathcal{A}$ interpretiert, heißt **Σ'-Redukt von $\mathcal{A}$** und wird mit $\mathcal{A}|_{\Sigma'}$ bezeichnet.

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

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

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 $f : e \to e' \in F$ 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

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

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

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

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

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

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

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

$X^* \times Y$ ist die Trägermenge der $Dyn(X,Y)$-algebra $Seq(X,Y)$, deren Operationen

$$nil^{Seq(X,Y)} : 1 \to X^* \times Y,\ cons^{Seq(X,Y)} : X \times (X^* \times Y) \to (X^* \times Y)$$

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

$$\begin{aligned} nil^{Seq(X,Y)}(y) &= (\epsilon, y), \\ cons^{Seq(X,Y)}(x,(w,y)) &= (xw, y) \end{aligned}$$

X^* ist Trägermenge der gleichnamigen $List(X)$-Algebra X^*, deren Operationen

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

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

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

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

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

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

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

Die Menge X^X der Funktionen von X nach X ist Trägermenge der *Mon*-Algebra *Endo(X)*, deren Operationen

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

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

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

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

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

$$head^{InfSeq(X)} : X^{\mathbb{N}} \to X,\ tail^{InfSeq(X)} : X^{\mathbb{N}} \to X^{\mathbb{N}}$$

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

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

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

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

Aufgabe Zeigen Sie, dass die Funktion

$$\begin{aligned} h : zo_{list} &\to Int^{\mathbb{N}} \\ Blink &\mapsto \lambda n.\, if\ even(n)\ then\ 0\ else\ 1 \\ Blink' &\mapsto \lambda n.\, if\ even(n)\ then\ 1\ else\ 0 \end{aligned}$$

$Stream(\mathbb{Z})$-homomorph ist. ❑

$X^* \times Y + X^{\mathbb{N}}$ ist die Trägermenge der $coDyn(X, Y)$-Algebra $coSeq(X, Y)$, deren Operationen

$$split^{coSeq(X,Y)} : X^* \times Y + X^{\mathbb{N}} \to X \times (X^* \times Y + X^{\mathbb{N}}) + Y$$

wie folgt definiert sind: Für alle $y \in Y$ und $f \in X^+ \times Y + X^{\mathbb{N}}$,

$$\begin{aligned} split^{coSeq(X,Y)}(y) &= y, \\ split^{coSeq(X,Y)}(f) &= (f(0), \lambda n.f(n+1)). \end{aligned}$$

Die Elemente von $X^+ \times Y + X^{\mathbb{N}}$ werden hier als partielle Funktionen von $\mathbb{N}$ nach $X \cup Y$ aufgefasst.

$X^* + X^{\mathbb{N}}$ ist die Trägermenge der gleichnamigen $coList(X)$-Algebra, deren Operationen

$$split^{coList(X)} : X^* + X^{\mathbb{N}} \to X \times (X^* + X^{\mathbb{N}}) + 1$$

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

$$\begin{aligned} split^{coList(X)}(\epsilon) &= *, \\ split^{coList(X)}(f) &= (f(0), \lambda n.f(n+1)). \end{aligned}$$

Die Elemente von $X^+ + X^{\mathbb{N}}$ werden hier als partielle Funktionen von $\mathbb{N}$ nach X betrachtet.

Die folgende $Acc(\mathbb{Z})$-Algebra eo dient der Erkennung der Parität der Summe von Zahlenfolgen (siehe Abschnitt 2.16) und ist wie folgt definiert:

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

2.7 Terme und Coterme

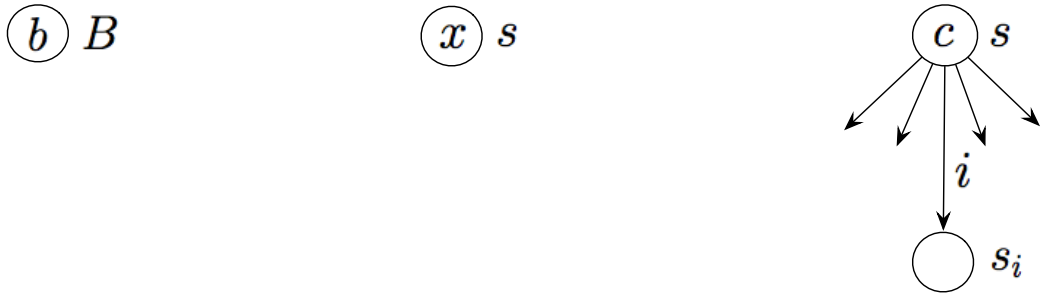
Im Folgenden verwenden wir die im Abschnitt Mengen und Typen eingeführte Notation für markierte Bäume.

Sei BT die Menge aller Basistypen, die in Σ vorkommen, und $X = \bigcup BT$.

Sei $\Sigma = (S, C)$ eine konstruktive Signatur, ST die Menge der Summentypen von $\mathcal{T}_p(S)$, C' die Vereinigung von C und der Menge aller Injektionen in Summentypen von $\mathcal{T}_p(S)$ und V eine $\mathcal{T}_p(S)$-sortige Menge von "Variablen" mit $V_e = \emptyset$ für alle Produkttypen $e \in \mathcal{T}_p(S)$.

Die Menge $CT_\Sigma(V)$ der **Σ-Terme über** V ist die größte $\mathcal{T}_p(S)$-sortige Menge M markierter Bäume über $(X, C' \cup X \cup V)$ mit folgenden Eigenschaften:

- Für alle $B \in BT$, $M_B = B \cup V_B$. (1)
- Für alle Produkttypen $e \in \mathcal{T}_p(S)$, $M_e = \emptyset$. (2)
- Für alle $s \in S \cup ST$ und $t \in M_s$ ist $t \in V_s$ oder gibt es $c : \prod_{i \in I} s_i \to s \in C'$ und $(t_i)_{i \in I} \in \bigtimes_{i \in I} M_{s_i}$ mit $t = c\{i \to t_i \mid i \in I\}$. (3/4)



(1)

$b \in B$

$B \in BT$

(3/5)

$s \in S \cup BT \cup ST$

$x \in V_s$

(4/6)

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

$T_\Sigma(V)$ bezeichnet die kleinste $\mathcal{T}_p(S)$-sortige Menge M wohlfundierter markierter Bäume über $(X, C \cup X \cup V)$ mit (1), (2) und folgenden Eigenschaften:

- Für alle $s \in S \cup ST$, $V_s \subseteq M_s$. (5)
- Für alle $c : \prod_{i \in I} s_i \to s \in C'$ und $(t_i)_{i \in I} \in \mathsf{X}_{i \in I} M_{s_i}$, $c\{i \to t_i \mid i \in I\} \in M_s$. (6)

Für alle $n > 0$ und $c : s_1 \times \cdots \times s_n \to s \in C$ schreibt man üblicherweise

$$c(t_1, \ldots, t_n) \quad \text{anstelle von} \quad c\{1 \to t_1, \ldots, n \to t_n\}.$$

Die Elemente von $CT_\Sigma =_{def} CT_\Sigma(\lambda s.\emptyset)$ und $T_\Sigma =_{def} T_\Sigma(\lambda s.\emptyset)$ heißen **Σ-Grundterme**. $\lambda s.\emptyset$ steht für das S-Tupel $(\emptyset)_{s \in S}$ leerer Mengen.

$CT_\Sigma(V)$ und $T_\Sigma(V)$ sind die Trägermengen von Σ-Algebren (s.u.). Darüberhinaus sind wohlfundierte Σ-Terme Bestandteile von Formeln wie z.B. iterativen Gleichungen (siehe Kapitel 16).

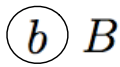
Sei $\Sigma = (S, D)$ eine destruktive Signatur, PT die Menge der Produkttypen von $\mathcal{T}_p(S)$, D' die Vereinigung von D und der Menge aller Projektionen in Produkttypen von $\mathcal{T}_p(S)$ und V eine $\mathcal{T}_p(S)$-sortige Menge von "Farben" mit $V_e = \emptyset$ für alle Basis- oder Summentypen $e \in \mathcal{T}_p(S)$.

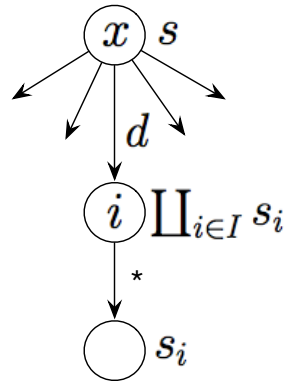
Die Menge $DT_\Sigma(V)$ der **Σ-Coterme über** V ist die größte $\mathcal{T}_p(S)$-sortige Menge M markierter Bäume über $(D' \cup 1, X \cup V)$ mit (1) und folgenden Eigenschaften:

- Für alle Summentypen $e \in \mathcal{T}_p(S)$, $M_e = \emptyset$. (7)
- Für alle $s \in S \cup PT$, $t \in M_s$ und $d : s \to \coprod_{i \in I} s_i \in D'$ gibt es $x \in V_s$, $i_d \in I$ und $t_d \in M_{s_i}$ mit $t = x\{d \to i\{* \to t_d\} \mid d : s \to e \in D\}$. (8)

$coT_\Sigma(V)$ bezeichnet die kleinste $\mathcal{T}_p(S)$-sortige Menge M wohlfundierter markierter Bäume über $(D' \cup 1, X \cup V)$ mit (1), (7) und folgender Eigenschaft:

- Für alle $d : s \to \coprod_{i \in I} s_i \in D'$, $x \in V_s$, $i_d \in I$ and $t_d \in M_{s_i}$, $x\{d \to i_d\{* \to t_d\} \mid d : s \to e \in D\} \in M_s$. (9)





(1)

$$b \in B$$
$$B \in BT$$

(8/9)

$$x \in V_s$$
$$d : s \to \coprod_{i\in I} s_i \in D'$$

Anstelle von $i\{* \to t_d\}$ schreibt man kurz $i(t_d)$.

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

Die Elemente von $DT_\Sigma =_{def} DT_\Sigma(\lambda s.1)$ und $coT_\Sigma =_{def} coT_\Sigma(\lambda s.1)$ heißen **Σ-Grundcoterme**. $\lambda s.1$ steht für das S-Tupel $(1)_{s\in S}$ einelementiger Mengen.

$DT_\Sigma(V)$ und $coT_\Sigma(V)$ sind die Trägermengen von Σ-Algebren (s.u.).

2.8 Beispiele

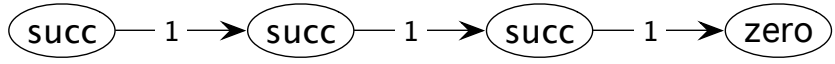
Sei $\Sigma = \mathit{Nat}$ (siehe 2.4).

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

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

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

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



Nat-Term, der die Zahl 3 darstellt

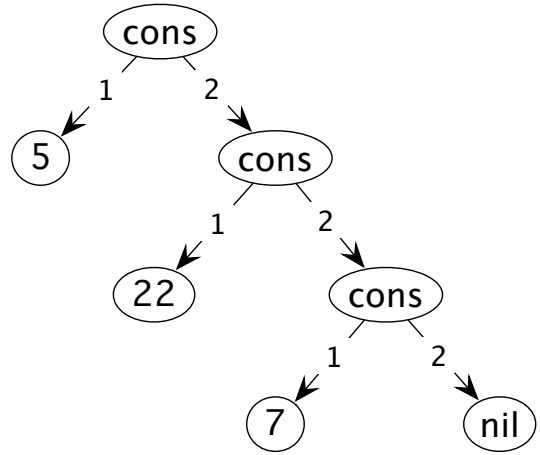
Sei $\Sigma = \mathit{List}(X)$ (siehe 2.4).

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

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

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

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



List($\mathbb{N}$)*-Term, der die Liste [5,22,7] darstellt*

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 16):

$$blink = cons(0, blink'),\ blink' = cons(1, blink). \tag{1}$$

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.

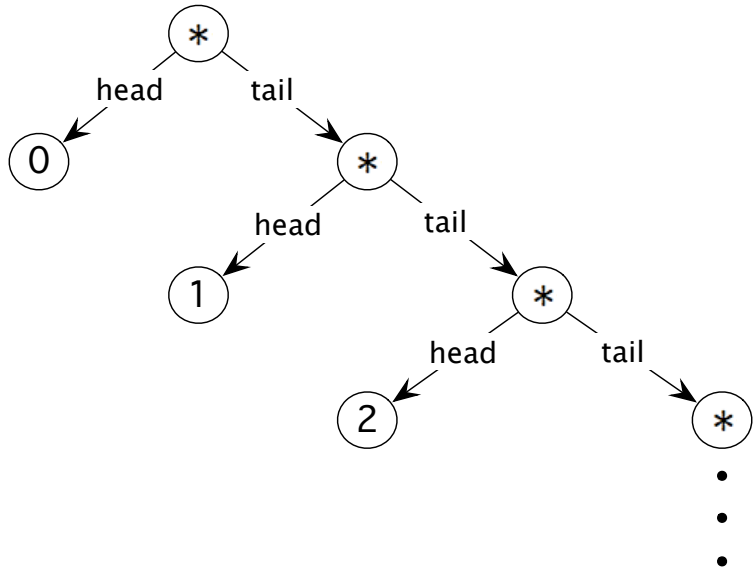
Sei $\Sigma = Reg(BL)$ (siehe 2.4).

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

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

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

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



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. $blink'$:

$$blink = *\{head \to 0, tail \to blink'\},\ blink' = *\{head \to 1, tail \to blink\} \qquad (2)$$

(siehe Kapitel 16). Deshalb definiert (2) $blink$ und $blink'$ als zwei Elemente von $DT_{Stream(\mathbb{Z})}$.

Sei $\Sigma = \mathit{Colist}(X)$ (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 $t = *\{\mathit{split} \to 1(\epsilon)\}$ oder gibt es $x \in X$ und $u \in M$ mit

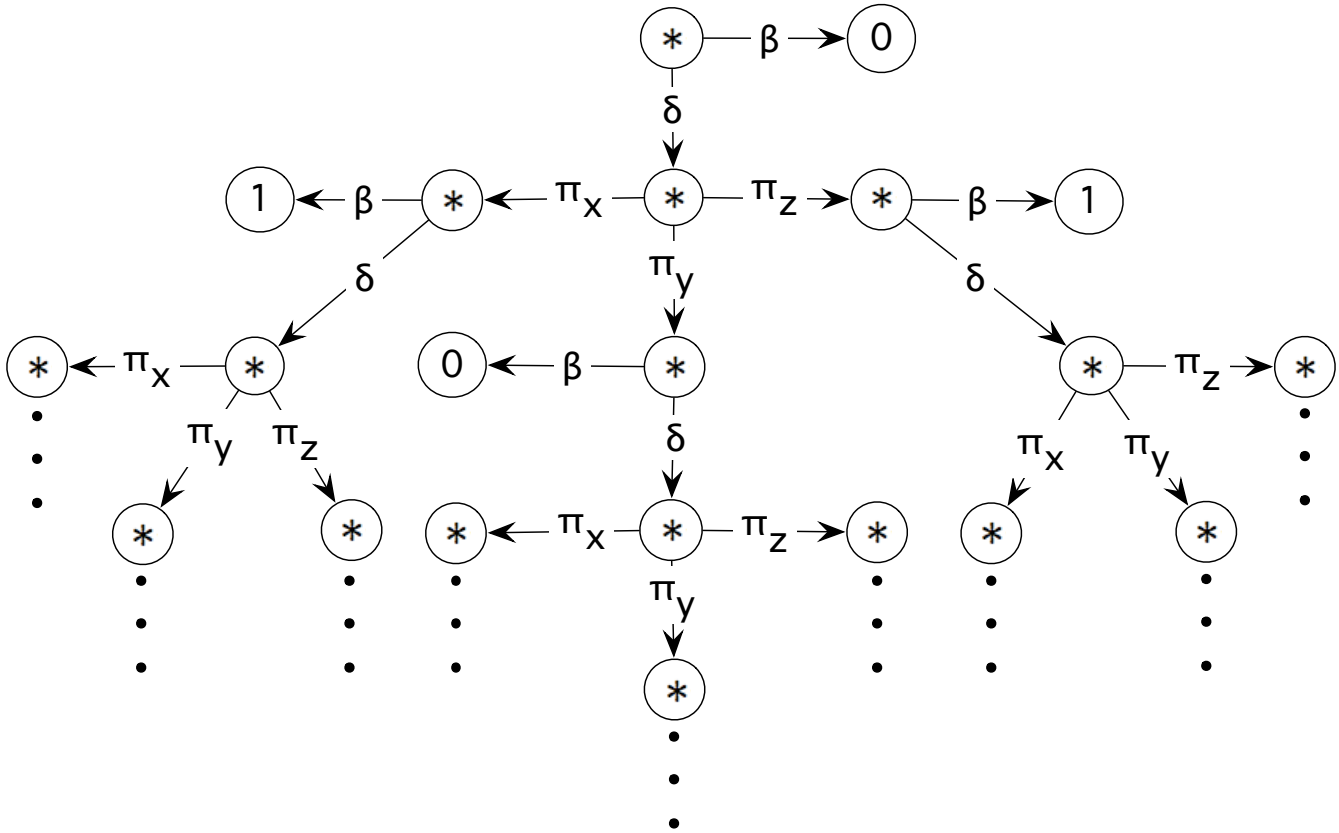
$$t = *\{\mathit{split} \to 2(*\{\pi_1 \to x, \pi_2 \to u\})\}.$$

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

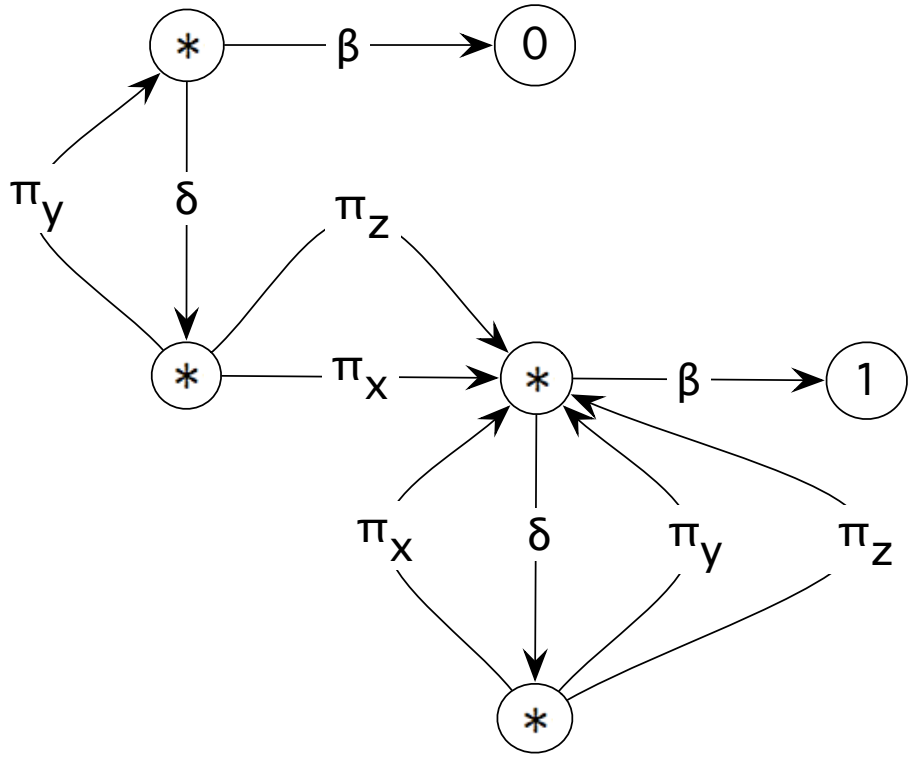
$DT_{\Sigma,state}$ ist die größte Teilmenge M von $ltr(D, Y \cup 1)$ mit folgender Eigenschaft:

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

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



Acc$(\{x, y, z\})$*-Coterm, der einen Akzeptor der Sprache* $\{w \in \{x, y, z\}^* \mid w$ *enthält* x *oder* $z\}$ *repräsentiert*



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

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

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

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

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

Für alle $s \in S$, $t \in DT_\Sigma(V)_s$, $d : s \to e \in D'$ und $w \in (D' \cup 1)^*$,

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

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 $d : s \to e$ 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 $B \in BL \setminus \{\epsilon\}$,

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

Die üblichen Wortdarstellungen regulärer Ausdrücke e wie z.B. $aab^* + c(a\overline{\mathbb{N}})^*$ 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:

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

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

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

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

Ist die Funktion $f : \mathbb{N} \to syms(BL)^*$ 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 $f : X^* \to Y$, $x \in X$ 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 $\chi : \mathcal{P}(X^*) \to 2^{X^*}$, 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. ❏

Sei BL wie in 2.4 und $X = \bigcup BL \setminus \{\epsilon\}$.

$\mathcal{P}(X^*)$ 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. ❏

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 [4, 18]), 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 $Reg(BL)$-Grundtermen definiert:

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

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

2.11 Die Erreichbarkeitsfunktion

Sei $\mathcal{A} = (A, Op)$ eine $Med(X)$-Algebra. Die **Erreichbarkeitsfunktion**

$$reach^{\mathcal{A}} : X^* \to A^A$$

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

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

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

$reach^{\mathcal{A}}$ ist ein *Mon*-Homomorphismus von *Word(X)* nach *Endo(A)*:

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

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

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

$= (reach^{\mathcal{A}}(w) \circ reach^{\mathcal{A}}(v)) \circ \lambda a.\delta^{\mathcal{A}}(a)(x) = reach^{\mathcal{A}}(w) \circ (reach^{\mathcal{A}}(v) \circ \lambda a.\delta^{\mathcal{A}}(a)(x))$
$= reach^{\mathcal{A}}(w) \circ reach^{\mathcal{A}}(xv) = mul^{Endo(A)}(reach^{\mathcal{A}}(xv), reach^{\mathcal{A}}(w)).$ ❏

2.12 Termfaltung und Zustandsentfaltung

Sei $\Sigma = (S, C)$ eine konstruktive Signatur, ST die Menge der Summentypen von $\mathcal{T}_p(S)$, C' die Vereinigung von C und der Menge aller Injektionen in Summentypen von $\mathcal{T}_p(S)$, V eine $\mathcal{T}_p(S)$-sortige Menge von "Variablen" mit $V_e = \emptyset$ für alle Produkttypen $e \in \mathcal{T}_p(S)$, $\mathcal{A} = (A, Op)$ eine Σ-Algebra und $g : V \to A$ eine – **Variablenbelegung** (*valuation*) genannte – $\mathcal{T}_p(S)$-sortige Funktion.

In Abhängigkeit von g wertet die wie folgt induktiv definierte $\mathcal{T}_p(S)$-sortige Funktion

$$g^* : T_\Sigma(V) \to A,$$

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

- Für alle $s \in S \cup BT \cup ST$ und $x \in V_s$, $g^*_s(x) = g_s(x)$. (1)
- Für alle $c : \prod_{i \in I} s_i \to s \in C'$ und $(t_i)_{i \in I} \in \bigtimes_{i \in I} T_\Sigma(V)_{s_i}$,
  $g^*_s(c\{i \to t_i \mid i \in I\}) = c^{\mathcal{A}}((g^*_{s_i}(t_i))_{i \in I})$. (2)

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

Satz 2.13 ([35], Theorem FREE)

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

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 ([35], Lemma SUBST)

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

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

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

Aus Satz 2.13 folgt sofort:

$fold^{\mathcal{A}}$ 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 $fold^{\mathcal{A}}$ 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 $T_{List(X)}$ auch X^* eine initiale $List(X)$-Algebra (siehe 2.6).

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

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

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

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

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

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

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

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

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

Sei $\Sigma = (S, D)$ eine destruktive Signatur, PT die Menge der Produkttypen von $\mathcal{T}_p(S)$, D' die Vereinigung von D und der Menge aller Projektionen in Produkttypen von $\mathcal{T}_p(S)$, V eine $\mathcal{T}_p(S)$-sortige Menge von "Farben" mit $V_e = \emptyset$ für alle Summentypen $e \in \mathcal{T}_p(S)$, $\mathcal{A} = (A, Op)$ eine Σ-Algebra und $g : A \to V$ eine – **Färbung** (*coloring*) genannte – $\mathcal{T}_p(S)$-sortige Funktion.

In Abhängigkeit von g entfaltet die wie folgt – auf $(D' \cup 1)^*$ induktiv – definierte $\mathcal{T}_p(S)$-sortige Funktion

$$g^{\#} : A \to DT_{\Sigma}(V),$$

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 $s \in S \cup BT \cup PT$ und $a \in A_s$, $g^{\#}_s(a)(\epsilon) = g_s(a)$.
- Für alle $s \in S \cup BT \cup PT$, $a \in A_s$, $d : s \to \coprod_{i \in I} s_i \in D$ und $w \in (D' \cup 1)^*$, $d^{\mathcal{A}}(a) = (b, i)$ impliziert $g^{\#}_s(a)(dw) = i(g^{\#}_{s_i}(b))(w)$.

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

Satz 2.14 ([35], 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. ❑

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 $g : A \to V$ und Σ-Homomorphismen $h : \mathcal{B} \to \mathcal{A}$,

$$(g \circ h)^{\#} = g^{\#} \circ h. \qquad ❑$$

Offenbar hängt die Einschränkung von $g^{\#}$ auf Grundcoterme nicht von g ab. Sie wird **Zustandsentfaltung** genannt und mit $unfold^{\mathcal{A}} : \mathcal{A} \to DT_\Sigma$ 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 $unfold^{\mathcal{A}}$ bezeichnet, egal welche isomorphe Darstellung von DT_Σ gerade verwendet wird.

Z.B. gibt es folgende bijektive Darstellungen für die Menge der $Stream(X)$-, $Beh(X,Y)$- bzw. $Acc(X)$-Grundcoterme:

$$\begin{aligned} DT_{DAut(X,Y)} &\cong Y^{X^*}, \\ DT_{Acc(X)} &\cong \mathcal{P}(X^*), \\ DT_{Stream(X)} &\cong X^{\mathbb{N}}. \end{aligned}$$

Die Bijektionen sind verträglich mit den Operationen der jeweiligen Signatur Σ. Sie sind also Σ-Isomorphismen zwischen der Σ-Algebra DT_Σ und einer der oben definierten Σ-Algebren:

$$\begin{aligned} DT_{DAut(X,Y)} &\cong Beh(X,Y), \\ DT_{Acc(X)} &\cong Pow(X), \\ DT_{Stream(X)} &\cong Seq(X). \end{aligned}$$

Auch die Menge $DT_\Sigma(V)$ aller Σ-Coterme, auch jener mit (mehr als einer) Variablen, hat für bestimmte Signaturen einfache Repräsentationen, z.B.:

$$DT_{Med(X)}(V) \;\cong\; DT_{DAut(X,V)} \;\cong\; V^{X^*}.$$

Ist V die Trägermenge (der Sorte *state*) einer $Med(X)$-Algebra $\mathcal{A}$, dann entspricht die Coextension $id_V^\# : V \to V^{X^*}$ der Erreichbarkeitsfunktion von $\mathcal{A}$:

$$id_V^\# \;=\; \mathit{flip}(reach^{\mathcal{A}}) : V \to V^{X^*}$$

(siehe Abschnitt 2.11).

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.:

$$DT_\Sigma \;\cong\; 1^{L^*} \times Y^{L^*\beta} \;\cong\; Y^{L^*\beta} \overset{L^*\beta \cong X^*}{\cong} Y^{X^*}. \qquad \text{❑}$$

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 $\chi : \mathcal{P}(X) \to 2^X$ 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 $unfold^{\mathcal{A}}$ bezeichnet.

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

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

$$\begin{array}{rcl} unfold^{\mathcal{A}} : A & \to & Y^{X^*} \\ a & \mapsto & \lambda w.\, if\ w = \epsilon\ then\ \beta^{\mathcal{A}}(a)\ else\ unfold^{\mathcal{A}}(\delta^{\mathcal{A}}(a)(head(w)))(tail(w)) \end{array}$$

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

$$\begin{array}{rcl} unfold^{\mathcal{A}} : A & \to & \mathcal{P}(X^*) \\ a & \mapsto & \{xw \mid w \in unfold^{\mathcal{A}}(\delta^{\mathcal{A}}(a)(x))\} \cup (if\ \beta^{\mathcal{A}}(a) = 1\ then\ \{\epsilon\}\ else\ \emptyset) \end{array}$$

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

$$\begin{aligned} unfold^{\mathcal{A}} : A &\to X^{\mathbb{N}} \\ a &\mapsto \lambda n.\, if\ n = 0\ then\ head^{\mathcal{A}}(a)\ else\ unfold^{\mathcal{A}}(tail^{\mathcal{A}}(a))(n-1) \end{aligned}$$

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

$$\begin{aligned} & fold^{Lang(X)} \text{ ist } Acc(X)\text{-homomorph} && (1) \\ \Leftrightarrow\ & fold^{Lang(X)} = unfold^{Bro(BL)} : T_{Reg(BL)} \to \mathcal{P}(X^*) && (2) \\ \Leftrightarrow\ & unfold^{Bro(BL)} \text{ ist } Reg(BL)\text{-homomorph.} && (3) \end{aligned}$$

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

(2) folgt auch aus der Form der Gleichungen, die $Bro(BL)$ definieren (siehe Beispiel 15.4).

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	
$\mathbb{N} \to syms(BL)^*$		$Regword(BL)$ `regWord`		
2		$Bool$		

2.16 Initiale Automaten

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

$$(\mathcal{A}, a) \textbf{ realisiert } t \in DT_{\Sigma,s} \quad \Leftrightarrow_{def} \quad unfold_s^{\mathcal{A}}(a) = t.$$

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

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

Beispiel

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

$(eo, Esum)$ erkennt $L = \{(x_1, \ldots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist gerade}\}$. (4)

$(eo, Osum)$ erkennt $L' = \{(x_1, \ldots, x_n) \in \mathbb{Z}^* \mid \sum_{i=1}^n x_i \text{ ist ungerade}\}$. (5)

Beweis. Sei $h : eo \to Pow(\mathbb{Z})$ wie folgt definiert: $h(Esum) = L$ and $h(Osum) = L'$.

Da $Pow(\mathbb{Z})$ eine finale $Acc(\mathbb{Z})$-Algebra ist, stimmen alle $Acc(\mathbb{Z})$-Homomorphismen von eo nach $Pow(\mathbb{Z})$ mit $unfold^{eo} : eo \to Pow(\mathbb{Z})$ überein. (4) und (5) 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:

$$\begin{aligned} h(\delta^{eo}(Esum)(x)) &= \delta^{Pow}(h(Esum))(x), \\ h(\delta^{eo}(Osum)(x)) &= \delta^{Pow}(h(Osum))(x), \\ \beta^{eo}(Esum) &= \beta^{Pow}(h(Esum)), \\ \beta^{eo}(Osum) &= \beta^{Pow}(h(Osum)). \end{aligned}$$

❑

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 10.3).

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 Erkenner $fold^{\chi(Lang(X))}(t)$ eine Funktion

$$parse_{REG} : syms(BL)^* \to M(T_{Reg(BL)})$$

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 M formulieren.

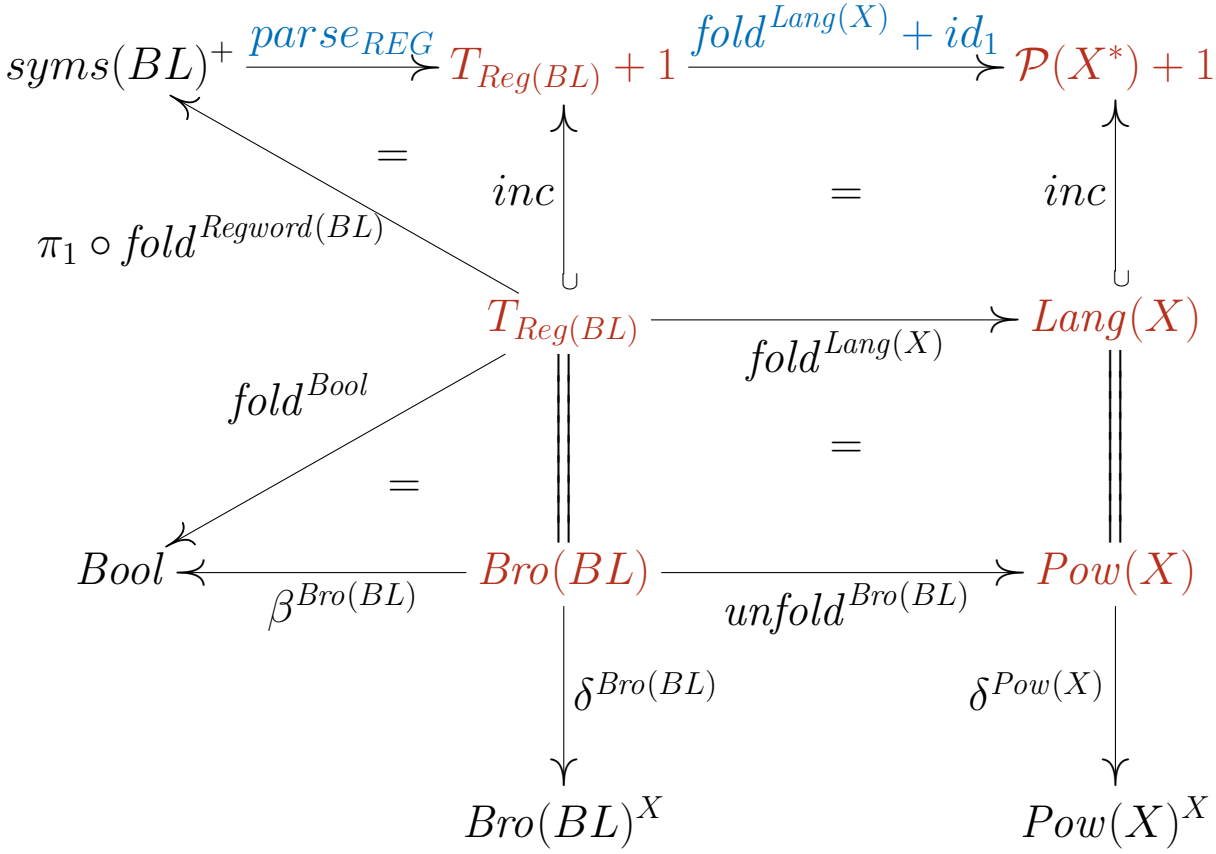
$parse_G$ ist diejenige Instanz von $compile_G$, die Wörter von L in Syntaxbäume (= Σ-Grundterme) übersetzt: $parse_G = compile_G^{T_\Sigma}$.

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

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

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 $Reg(BL)$- bzw. $Acc(X)$-Algebren:



2.18* Von Medvedev- zu Moore-Automaten

Streicht man die Ausgabefunktion $\beta : state \to Y$ 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

$$unfold^{\mathcal{A}} = (\beta^{\mathcal{A}})^{\#} : \mathcal{A} \to DT_{Med(X)}(Y). \tag{1}$$

Die Interpretation von β in $\mathcal{A}$ wird hier offenbar als Färbung der Zustände von $\mathcal{A}$ verwendet. Aus (1) und einer allgemeinen Beziehung zwischen Färbungen und ihren Coextensionen erhält man für initiale Automaten $(\mathcal{A}, a)$:

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

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 $id_A^{\#} : A \to DT_{Med(X)}(A) \cong Beh(X, A)$ (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 $w \in X^*$,

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

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

$$id_A^\#(a)(\epsilon) = root(id_A^\#(a)) = id_A(a) = a, \tag{2}$$

$$id_A^\#(a)(xw) = \delta^{Beh(X,A)}(id_A^\#(a))(x)(w) \overset{id_A^\# Med(X)-hom.}{=} id_A^\#(\delta^{\mathcal{A}}(a)(x))(w). \tag{3}$$

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

$$id_A^\#(a)(vw) \;=\; id_A^\#(id_A^\#(a)(v))(w). \tag{4}$$

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

$$\begin{aligned} q \cdot e &= q, \\ q \cdot (m * m') &= (q \cdot m) \cdot m'. \end{aligned}$$

Wegen (2) und (4) ist (die dekaskadierte Version von) $id_A^\#$ eine Aktion des Monoids X^*.

2.19* Minimale Automaten

Sei $(\mathcal{A}, a)$ ein initialer Automat. Die Elemente der Menge $\langle a \rangle =_{def} img(id_A^{\#}(a))$ 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 $h : \mathcal{A} \to B$ gilt $h(\langle a \rangle) = \langle h(a) \rangle$.

(iii) Für alle $f : X^* \to Y$ 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:

$$id_A^{\#}(a)(wx) = \delta^{\mathcal{A}}(id_A^{\#}(a)(w))(x). \tag{5}$$

Beweis von (5):

$$id_A^{\#}(a)(\epsilon x) = id_A^{\#}(a)(x\epsilon) \overset{(3)}{=} id_A^{\#}(\delta^{\mathcal{A}}(a)(x))(\epsilon) \overset{(2)}{=} \delta^{\mathcal{A}}(a)(x).$$

Für alle $y \in X$ und $w \in X^*$,

$$id_A^{\#}(a)(ywx) \overset{(3)}{=} id_A^{\#}(\delta^{\mathcal{A}}(a)(y))(wx) \overset{ind.\ hyp.}{=} \delta^{\mathcal{A}}(id_A^{\#}(\delta^{\mathcal{A}}(a)(y))(w))(x) \overset{(3)}{=} \delta^{\mathcal{A}}(id_A^{\#}(yw))(x).$$

Wegen (5) 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 $\mathcal{A}$.

Sei $\mathcal{B} = (B, Op')$ eine Unteralgebra von $\mathcal{A}$, die a enthält. Durch Induktion über $|w|$ erhält man aus (2) und (3), 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.). Daraus folgt $h(id_A^{\#}(a)(w)) = h^{(D \cup B)^*}(id_A^{\#}(a))(w) = id_B^{\#}(h(a))(w)$ für alle $w \in X^*$ und damit sowohl $h(\langle a \rangle) \subseteq \langle h(a) \rangle$ als auch $\langle h(a) \rangle \subseteq h(\langle a \rangle)$. h^{X^*} ist das Bild von h unter dem Leserfunktor $\_^{X^*}$ (siehe Abschnitt 5.1).

(iii):

Da $Beh(X, Y)$ final ist, stimmt $unfold^{Beh(X,Y)}$ (s.o.) mit der Identität auf Y^{X^*} überein. Also gilt $f = unfold^{Beh(X,Y)}(f) = unfold^{Beh(X,Y)}(inc_{\langle f \rangle}(f)) = unfold^{\langle f \rangle}(f)$, d.h. $(\langle f \rangle, f)$ realisiert f.

Für jeden initialen Automaten $(\mathcal{A}, a)$, der f realisiert, gilt

$$|\langle f \rangle| = |\langle unfold^{\mathcal{A}}(a) \rangle| \overset{\text{(ii)}}{=} |unfold^{\mathcal{A}}(\langle a \rangle)| \leq |\langle a \rangle|.$$

Also ist $(\langle f \rangle, f)$ minimal. ❏

2.21* Baumautomaten [2, 5, 6, 37, 38, 46]

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

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

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 $\mathcal{B}$ (siehe Abschnitt 2.6).

$$out^{\mathcal{B}} \circ fold^{\mathcal{A}} : T_\Sigma \to Y$$

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

Im Fall $Y = 2$ nennt man

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

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

Wie man leicht sieht, ist die Funktion

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

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 $\chi(L) = out^{\mathcal{B}} \circ fold^{\mathcal{A}}$ 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

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

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

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

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

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

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 genau dann gilt, wenn $h(L)$ im Sinne von Wortsprachen regulär ist, also zum Bild von $fold^{Lang(X)} : T_{Reg(BL)} \to \mathcal{P}(X^*)$ gehört (siehe Abschnitte 3.7 und 10.3).

Sei $f : T_\Sigma \to Y$ und $\sim$ die **Nerode-Relation von** f, d.i. 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 $\{e_i\}_{i \in I} \subseteq \mathcal{T}_p(S)$,

$$
\begin{aligned}
R_{\prod_{i \in I} e_i} &= \{(a_1, \ldots, a_n) \in A^n_{\prod_{i \in I} e_i} \mid \forall\, i \in I : (\pi_i(a_1), \ldots, \pi_i(a_n)) \in R_{e_i}\}, \\
R_{\coprod_{i \in I} e_i} &= \{(\iota_i(a_1), \ldots, \iota_i(a_n)) \in A^n_{\coprod_{i \in I} e_i} \mid (a_1, \ldots, a_n) \in R_{e_i},\ i \in I\}.
\end{aligned}
$$

Lemma LIFT Seien $g, h : A \to B$ typverträgliche Funktionen und R eine typverträgliche Teilmenge von A and 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 $e \in \mathcal{T}_p(S)$. ❑

3.1 Invarianten, Unteralgebren und Induktion

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 $f : e \to e' \in F$ 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 $f : e \to e' \in F$ und $a \in B_e$, $f^{\mathcal{A}|_B}(a) =_{def} f^{\mathcal{A}}(a)$.

Beispiel 3.1

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

Für jede Signatur Σ, jede Σ-Algebra $\mathcal{A} = (A, Op)$ und jeden Σ-Homomorphismus $h : \mathcal{A} \to \mathcal{B}$ 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 $inc_\mathcal{B} = (inc_{B_s} : B_s \to A_s)_{s \in S}$ ist Σ-homomorph.

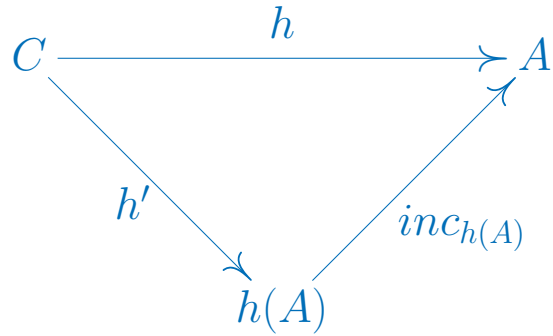
(2) (Homomorphiesatz) Sei $h : \mathcal{C} \to \mathcal{A}$ ein Σ-Homomorphismus. $h(\mathcal{C})$ ist eine Unteralgebra von A.

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

ist ein surjektiver Σ-Homomorphismus.

Ist h injektiv, dann ist h' bijektiv.

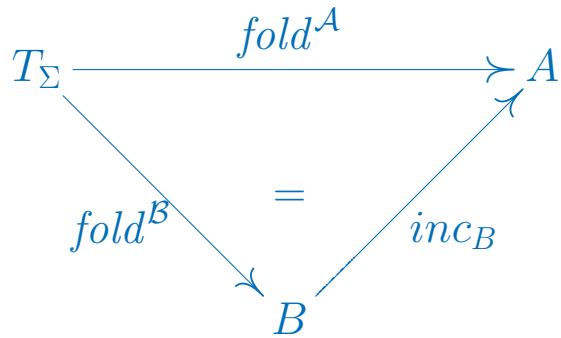
Alle Σ-Homomorphismen $h'' : \mathcal{C} \to h(\mathcal{C})$ mit $inc_{h(\mathcal{C})} \circ h'' = h$ stimmen mit h' überein.



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

(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 $h : A \to B$.

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



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

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

Also ist das Bild von $fold^{\mathcal{A}}$ in B enthalten.

Sei B eine Unteralgebra von T_Σ. Da T_Σ initial ist, folgt $inc_B \circ fold^{\mathcal{B}} = id_{T_\Sigma}$ 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 $g, h : A \to B$ Σ-Homomorphismen. Dann ist

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

eine Σ-Unteralgebra von A: Sei $f : e \to e' \in F$ 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 $g = h$. ❏

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

Nach Satz 3.2 (3) erfüllen alle zu T_Σ isomorphen Σ-Algebren $\mathcal{A} = (A, Op)$ das **Induktionsprinzip**, d.h. eine prädikatenlogische Formel φ gilt für alle Elemente von A, wenn φ von allen Elementen einer Σ-Unteralgebra $\mathcal{B} = (B, Op')$ von $\mathcal{A}$ erfüllt wird.

Um das zu zeigen, definiert man B zunächst als Menge aller Elemente von A, die φ erfüllen und prüft, ob B schon eine Unteralgebra von $\mathcal{A}$ bildet. Wenn nicht, wird B solange verkleinert, bis eine Unteralgebra erreicht ist (siehe [35], Kapitel 16).

3.2 Kongruenzen, Quotienten und Coinduktion

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 $f : e \to e' \in F$ und $(a, b) \in R_e$ $(f^{\mathcal{A}}(a), f^{\mathcal{B}}(b)) \in R_{e'}$ gilt.

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

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

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** $nat_R : \mathcal{A} \to \mathcal{A}/R$, die jedes Element $a \in A$ auf seine Äquivalenzklasse $[a]_R$ abbildet, ist Σ-homomorph.

(2) (Homomorphiesatz) Sei $h : A \to B$ ein Σ-Homomorphismus und $ker(h) \subseteq A^2$ der **Kern** von h, d.h. $ker(h) = \{(a, b) \mid h(a) = h(b)\}$.

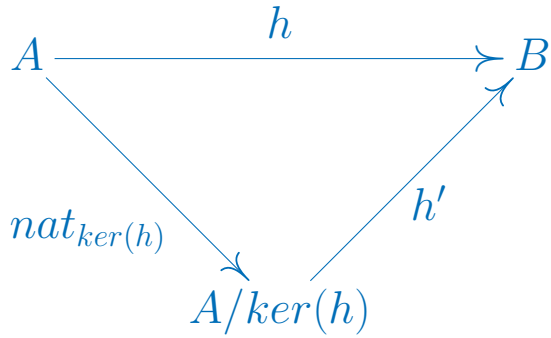
$ker(h)$ ist eine Σ-Kongruenz.

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

ist ein injektiver Σ-Homomorphismus.

Ist h surjektiv, dann ist h' bijektiv.

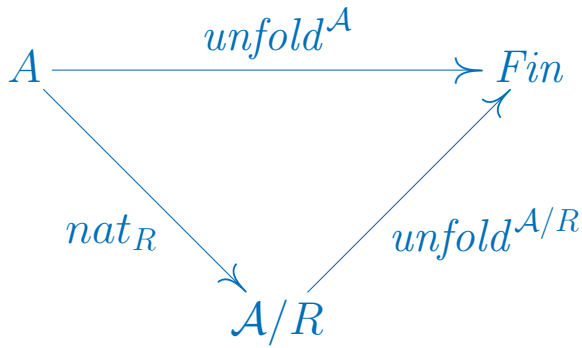
Alle Σ-Homomorphismen $h'' : A/ker(h) \to B$ mit $h'' \circ nat = h$ stimmen mit h' überein.



(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 $unfold^{\mathcal{A}} : A \to Fin$ ist die größte Σ-Kongruenz auf A und die Diagonale von Fin^2 die einzige Σ-Kongruenz R auf Fin.

(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 $h : A \to B$.

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



Daraus folgt für alle $a, b \in A$,

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

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

Sei R eine Kongruenz auf Fin. Da Fin final ist, folgt $unfold^{Fin/R} \circ nat_R = id_{Fin}$ 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 $g, h : A \to B$ Σ-Homomorphismen. Dann ist

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

eine Σ-Kongruenz auf B: Sei $f : e \to e' \in F$, $a \in A_e$ 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 $g = h$. ❑

Nach Satz 3.4 (3) erfüllen alle zu *Fin* isomorphen Σ-Algebren $\mathcal{A} = (A, Op)$ das **Coinduktionsprinzip**: Sei E eine Menge von Gleichungen zwischen Termen einer konstruktiven Signatur Σ'. E gilt in $\mathcal{A}$, wenn es eine Σ-Kongruenz R auf A gibt, die für alle $t = u \in E$ und $g : V \to A$ das Paar $(g^*(t), g^*(u))$ enthält.

Um das zu zeigen, definiert man R zunächst als als Menge aller Paare $(g^*(t), g^*(u))$ mit $t = u \in E$ und $g : V \to A$ und prüft, ob R schon eine Kongruenz ist. Wenn nicht, wird R solange vergrößert, bis eine Kongruenz erreicht ist (siehe [35], Kapitel 16).

Automatenminimierung durch Quotientenbildung

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

Im Beweis von Satz 2.20 (iii) wurde der $DAut(X,Y)$-Homomorphismus $h : \langle a \rangle \to \langle f \rangle$ 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 $ker(h) = ker(unfold^{\mathcal{A}}) \cap \langle a \rangle^2$.

Nach Satz 3.4 (3) ist $ker(unfold^{\mathcal{A}})$ 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:

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

Ist A_{state} endlich, dann lässt sich $ker(unfold^{\mathcal{A}})$ schnell und elegant mit dem in Abschnitt 2.3 von [30] und in [34] beschriebene (und in Haskell implementierte) **Paull-Unger-Verfahren** berechnen. Es startet mit $R' = A_{state}^2$ und benutzt die Kontraposition

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

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

Transitionsmonoid und syntaktische Kongruenz

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

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

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

Satz 3.5 (Transitionsmonoide und endliche Automaten)

Für alle $a \in A_{state}$ 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 $reach^{\langle a \rangle} : X^* \to (\langle a \rangle \to \langle a \rangle)$. Nach Satz 3.4 (2) ist das Transitionsmonoid von $\langle a \rangle$ Mon-isomorph zum Quotienten X^*/R. Außerdem ist die Abbildung

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

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

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

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

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

eine surjektive Abbildung von $reach^{\langle a \rangle}(X^*)$ 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 $A_{state} \to A_{state}$ ist. ❏

Sei $L \subseteq X^*$. Das Transitionsmonoid von $\langle L \rangle$ heißt **syntaktisches Monoid** und der Kern von $reach^{\langle L \rangle}$ **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 Abschnitt 3.7 für einen direkten Beweis oder Abschnitt 10.3 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 $unfold^{\mathcal{A}}(a) = L$.

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

Daraus würde jedoch

$$unfold^{\mathcal{A}}(x^i y^j) = \beta^{\mathcal{A}}(id_A^{\#}(a)(x^i y^j)) = \beta^{\mathcal{A}}(id_A^{\#}(id_A^{\#}(a)(x^i))(y^j)) = \beta^{\mathcal{A}}(id_A^{\#}(b_i)(y^j))$$
$$= \beta^{\mathcal{A}}(id_A^{\#}(b_j)(y^j)) = 1$$

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

3.3 Termsubstitution und -auswertung

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

$$\sigma^* : T_\Sigma(V) \to T_\Sigma(V)$$

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

Sei BT die Menge aller Basistypen, die in Σ vorkommen, und $X = \bigcup BT$.

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

$\sigma^*(t)$ 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 $\{t_1/x_1, \ldots, t_n/x_n\}$ für die Substitution σ

mit $\sigma(x_i) = t_i$ für alle $1 \leq i \leq n$ und $\sigma(x) = x$ für alle $x \in V \setminus \{x_1, \ldots, x_n\}$.

Satz 3.7

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

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

(2) Für alle Substitutionen $\sigma, \tau : V \to T_\Sigma(V)$ 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. ❑

3.4 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 $g : V \to A$ gilt $g^*(t) = g^*(t')$.

$Alg_{\Sigma,E}$ 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 $h : A \to B$ gilt:

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

Die Elemente der Menge

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

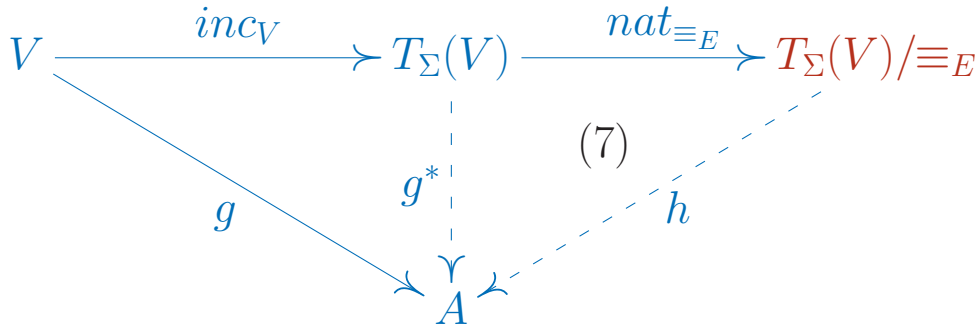
heißen **Instanzen von** E.

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 3.9

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

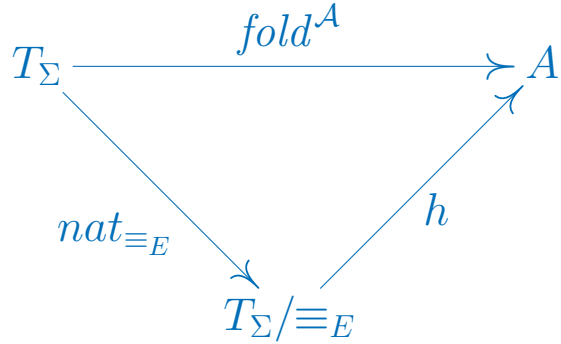


Beweis. Da der Kern von g^* eine Σ-Kongruenz ist, die $Inst(E)$ enthält (siehe [32], Satz GLK), $\equiv_E$ aber die kleinste derartige Relation auf $T_\Sigma(V)$ ist, ist letztere im Kern von g^* enthalten. Folglich ist $h : T_\Sigma(V)/\equiv_E \to A$ mit $h([t]_{\equiv_E}) = h(t)$ für alle $t \in T_\Sigma(V)$ wohldefiniert. (7) kommutiert, was zusammen mit der Surjektivität und Σ-Homomorphie von $nat_{\equiv_E}$ impliziert, dass auch h Σ-homomorph ist. ❏

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 3.9, dass $T_\Sigma/\equiv_E$ initial in $Alg_{\Sigma,E}$ ist, dass es also für alle $A \in Alg_{\Sigma,E}$ genau einen Σ-Homomorphismus $h : T_\Sigma/\equiv_E \to A$ gibt.

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



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

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

Die freie Mon-Algebra $T_{Mon}(V)$ ist kein Monoid, wohl aber ihr Quotient $T_{Mon}(V)/\equiv_E$. 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 $T_{Mon}(V)/\equiv_E$.

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** $\to_E$ 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 $\sigma : V \to T_\Sigma(V)$.

Die kleinste transitive Relation auf $T_\Sigma(V)$, die $\to_E$ enthält, wird mit $\overset{+}{\to}_E$ bezeichnet.

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

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

Eine Funktion $reduce : T_\Sigma(V) \to T_\Sigma(V)$ 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 $g : V \to A$. Wegen $\overset{+}{\to}_E \ \subseteq \ \equiv_E \ \subseteq \ ker(g^*)$ (siehe obige Aufgabe und den Beweis von Satz 3.9) gilt

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

Allgemeine Methoden zur Berechnung von Normalformen werden in [32], §5.7, behandelt.

3.5 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)
$par(base(\emptyset),x) = x$	$par(x,base(\emptyset)) = x$	(Neutralität von mt bzgl. par)
$seq(eps,x) = x$	$seq(x,eps) = x$	(Neutralität von eps bzgl. seq)
$seq(base(\emptyset),x) = mt$	$seq(x,base(\emptyset)) = mt$	(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 $Lang(X)$ E erfüllt, gilt (8) für $A = Lang(X)$.

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 $Beh(X, Y)$ (siehe 2.10) heißen **formale Potenzreihen**, wenn Y ein Semiring ist (siehe [41], Kapitel 9).

3.6 Die Brzozowski-Gleichungen

Die folgende Menge BRE von – **Brzozowski-Gleichungen** genannten – $Reg(BL)$-Gleichungen hat in $T_{Reg(BL)}$ genau eine Lösung, d.h. es gibt genau eine Erweiterung von $T_{Reg(BL)}$ zur $Acc(X)$-Algebra, die BRE erfüllt (siehe Beispiel 15.2).

In Abschnitt 2.10 wurde diese Lösung unter dem Namen $Bro(BL)$ eingeführt. Existenz und Eindeutigkeit folgen aus Satz 15.1 und werden dort aus dem in Satz 3.2 (3) eingeführten Induktionsprinzip abgeleitet.

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

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 15.3 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 15.4). ❑

Optimierter Brzozowski-Automat

Der Erkenner $Bro(BL)$ regulärer Sprachen (siehe Abschnitt 2.6) benötigt viel Platz, weil die wiederholten Aufrufe von $\delta^{Bro(BL)}$ aus t immer größere Ausdrücke erzeugen. Um das zu vermeiden, ersetzen wir $Bro(BL)$ durch die $Acc(X)$-Algebra $Norm(BL)$ (`norm` in Compiler.hs), die bis auf die Interpretation von δ mit $Bro(BL)$ übereinstimmt. $\delta^{Norm(BL)}$ normalisiert die von $\delta^{Bro(BL)}$ berechneten Folgezustände mit der in Abschnitt 3.5 beschriebenen Reduktionsfunktion

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

Für alle $t \in T_{Reg(BL)}$,

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

Gemäß Abschnitt 3.5 gilt

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

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

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

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

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

Für alle $x \in X$ und $w \in X^*$,

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

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. ❑

3.7 Erkenner regulärer Sprachen sind endlich

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

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

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

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

$$id^{\#}_{\mathcal{P}(X^*)}(mt^{Lang(X)})(w) = \emptyset, \tag{5}$$

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

$$id^{\#}_{\mathcal{P}(X^*)}(par^{Lang(X)}(L, L'))(w) = id^{\#}_{\mathcal{P}(X^*)}(L)(w) \cup id^{\#}_{\mathcal{P}(X^*)}(L')(w), \tag{7}$$

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

$$\begin{aligned} id^{\#}_{\mathcal{P}(X^*)}(iter^{Lang(X)}(L))(w) &= \{uv \mid u \in id^{\#}_{\mathcal{P}(X^*)}(L)(w),\ v \in iter^{Pow(X)}(L)\} \\ &\quad \cup \bigcup_{u_1 \ldots u_n v = w} (\mathit{if}\ \ \epsilon \in \textstyle\bigcap_{i=1}^{n} id^{\#}_{\mathcal{P}(X^*)}(L)(u_i) \\ &\qquad \mathit{then}\ \{uv' \mid u \in id^{\#}_{\mathcal{P}(X^*)}(L)(v), \\ &\qquad\quad v' \in iter^{Pow(X)}(L)\} \\ &\qquad \mathit{else}\ \emptyset) \end{aligned} \tag{9}$$

(siehe z.B. [41], 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 $(Pow(X), L)$ ein minimaler Erkenner von L ist. Ist L die Sprache eines regulären Ausdrucks t, ist also $L = fold^{Lang(X)}(t)$, dann ist

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

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

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

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

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

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

Damit ist ohne den üblichen Umweg über Potenzautomaten (siehe Beispiel 10.3) 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**)

$$G \;=\; (S, Z, BT, R)$$

besteht aus

- einer endlichen Menge S von **Sorten**, die auch *Nichtterminale* oder *Variablen* genannt werden,
- einer endlichen Menge Z von **Terminalen**,
- einer endlichen Menge BT von **Basistypen**, d.h. Mengen mit mindestens zwei Elementen,
- einer endlichen Menge R von **Regeln** $s \to w$ mit $s \in S$ und $w \in (S \cup Z \cup BT)^* \setminus \{s\}$, die auch *Produktionen* genannt werden.

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

Beispiel 4.1

Sei BL wie in Abschnitt 2.4, $syms(BL)$ wie in 2.9 und

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

$REG =_{def} (\{reg\}, syms(BL), \emptyset, R)$ ist eine CFG. ❑

$X = Z \cup \bigcup BT$ bildet das Alphabet der Wörter, die ein Compiler für G verarbeitet. Dementsprechend werden wir ihn als Funktion auf X^* definieren (siehe Kapitel 5). Deren Berechenbarkeit hängt u.a. von der Entscheidbarkeit der Basistypen von G ab, also von der Berechenbarkeit ihrer charakteristischen Funktionen $\chi(B) : X \to 2$, $B \in BT$ (siehe Abschnitt 2.1). Anders ausgedrückt: Ein Compiler für G benötigt Erkenner für die Basistypen.

Oft genügt es, bei der Definition einer CFG nur deren Regeln 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 von) Basistypen sind.

Beispiel 4.2

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

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

String, $\mathbb{Z}$, Rel und 2 (als Menge Boolescher Werte) sind die Basistypen von JavaLight.

String bezeichnet die Menge aller Zeichenfolgen außer Elementen anderer Basistypen 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 [22], Abschnitt 2)

$$
\begin{array}{lcl}
\mathit{Store} & \to & \langle \mathit{store} \rangle\ \langle \mathit{stock} \rangle\ \mathit{Stock}\ \langle /\mathit{stock} \rangle\ \langle /\mathit{store} \rangle\ \mid \\
 & & \langle \mathit{store} \rangle\ \mathit{Orders}\ \langle \mathit{stock} \rangle\ \mathit{Stock}\ \langle /\mathit{stock} \rangle\ \langle /\mathit{store} \rangle \\
\mathit{Orders} & \to & \mathit{Order}\ \mathit{Orders}\ \mid\ \mathit{Order} \\
\mathit{Order} & \to & \langle \mathit{order} \rangle\ \langle \mathit{customer} \rangle\ \mathit{Person}\ \langle /\mathit{customer} \rangle\ \mathit{Items}\ \langle /\mathit{order} \rangle \\
\mathit{Person} & \to & \langle \mathit{name} \rangle\ \mathit{String}\ \langle /\mathit{name} \rangle\ \mid\ \langle \mathit{name} \rangle\ \mathit{String}\ \langle /\mathit{name} \rangle\ \mathit{Emails} \\
\mathit{Emails} & \to & \mathit{Email}\ \mathit{Emails}\ \mid\ \epsilon \\
\mathit{Email} & \to & \langle \mathit{email} \rangle\ \mathit{String}\ \langle /\mathit{email} \rangle \\
\mathit{Items} & \to & \mathit{Item}\ \mathit{Items}\ \mid\ \mathit{Item} \\
\mathit{Item} & \to & \langle \mathit{item} \rangle\ \mathit{Id}\ \langle \mathit{price} \rangle\ \mathit{String}\ \langle /\mathit{price} \rangle\ \langle /\mathit{item} \rangle \\
\mathit{Stock} & \to & \mathit{ItemS}\ \mathit{Stock}\ \mid\ \mathit{ItemS} \\
\mathit{ItemS} & \to & \langle \mathit{item} \rangle\ \mathit{Id}\ \langle \mathit{quantity} \rangle\ \mathbb{Z}\ \langle /\mathit{quantity} \rangle\ \mathit{Suppliers}\ \langle /\mathit{item} \rangle \\
\mathit{Suppliers} & \to & \langle \mathit{supplier} \rangle\ \mathit{Person}\ \langle /\mathit{supplier} \rangle\ \mid\ \mathit{Stock} \\
\mathit{Id} & \to & \langle \mathit{id} \rangle\ \mathit{String}\ \langle /\mathit{id} \rangle
\end{array}
$$

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>
```

4.4 Linksrekursive CFGs und abstrakte Syntax

Sei $G = (S, Z, BT, R)$ eine CFG und $W = (S \cup Z \cup BT)^*$.

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**

$$\rightarrow_G \;=\; \{(vsw, v\alpha w) \mid s \rightarrow \alpha \in R,\; v, w \in W\}.$$

$\overset{+}{\rightarrow}_G$ und $\overset{*}{\rightarrow}_G$ bezeichnen den transitiven bzw. reflexiv-transitiven Abschluss von $\rightarrow_G$.

G heißt **linksrekursiv**, falls $v \in W^*$ mit $s \overset{+}{\to}_G sv$ existiert.

G heißt **LL-kompilierbar**, falls $R_G =_{def} \{(s, s') \in S \times S \mid \exists\, v, w \in W : sv \overset{+}{\to}_G s'w\}$ antisymmetrisch ist.

Z.B. sind REG, JavaLight und XMLstore LL-kompilierbar (siehe Beispiele 4.1-4.3).

Enthält G eine **linksrekursive Regel** $s \to sw$, dann ist G linksrekursiv.

Für LL-kompilierbare Grammatiken gilt auch die Umkehrung: Sei G linksrekursiv, enthalte aber keine linksrekursive Regel. Dann gäbe es eine Ableitung der Form

$$s \to_G s'v \overset{+}{\to}_G sw, \tag{1}$$

womit R_G sowohl (s, s') als auch (s', s) enthielte und daher $s = s'$ wäre, weil G LL-kompilierbar ist. Aus (1) würde $s \to_G sv$ folgen. G müsste also die linksrekursive Regel $s \to sv$ enthalten. ↯

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 Auswertung 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. $(*5) : \mathbb{Z} \to \mathbb{Z}$: $(*5)$ bildet $x \in \mathbb{Z}$ auf $x * 5$ ab.

Die linksassoziative Auswertung $((x+y)-z)-z'$ bzw. $((x/y)*z)/z'$ der obigen Ausdrücke liefert dieselben Ergebnisse wie die – mit den Regeln für *Sumsect* und *Prodsect* erreichte – rechtsassoziative Auswertung $(x)((+y) \cdot ((-z) \cdot (-z')))$ bzw. $(x)((/y) \cdot ((*z) \cdot (/z')))$. $\cdot$ ist $\circ$ mit vertauschten Argumenten: $\cdot = \mathit{flip}(\circ)$.

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

Verfahren zur Eliminierung von Linksrekursion

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

- Wiederhole zunächst sooft wie möglich folgenden Schritt:
  Für alle Regelpaare $(s \to s'v, s' \to w)$ mit $s \neq s'$ ersetze $s \to s'v$ durch die neue Regel $s \to wv$.
- Streiche alle Regeln der Form $s \to s$.

Mit diesen Schritten werden aus Ableitungen der Form $s \overset{+}{\to}_G s'v \overset{+}{\to}_G sw$ linksrekursive Regeln, d.h. G wird LL-kompilierbar. Gemäß obiger Aufgabe müssen nun noch die linksrekursiven Regeln aus G entfernt werden. Um dabei die von G erzeugte Sprache nicht zu verändern, müssen neue Sorten und Regeln eingeführt werden: Sei $s \in S$.

$$
\begin{aligned}
recs(S) &= \{s \in S \mid \exists\, s \to sw \in R\},\\
nonrecs(s) &= \{w \mid s \in recs(S),\ s \to w \in R,\ w \notin \{s\} \times (S \cup Z \cup BT)^*\},\\
S' &= S \cup \{s' \mid s \in recs(S)\},\\
R' &= R \setminus \{s \to w \in R \mid s \in recs(S)\}\\
&\quad \cup \{s' \to ws' \mid s \to sw \in R\}\\
&\quad \cup \{s \to vs' \mid v \in nonrecs(s),\ s \in S\}\\
&\quad \cup \{s' \to \epsilon \mid s \in recs(S)\}.
\end{aligned}
$$

$G' = (S', Z, BT, R')$ ist nicht-linksrekursiv und erzeugt dieselbe Sprache wie G.

Beispiel 4.2 (Fortsetzung)

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

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

❏

Die – auch **rekursiver Abstieg** genannte – LL-Kompilation (siehe Kapitel 6) terminiert, wenn die zugrundeliegende Grammatik nicht-linksrekursiv ist (Satz 6.4). Darüberhinaus können unter dieser Voraussetzung Erkenner regulärer Sprachen leicht zu Erkennern kontextfreier Sprachen erweitert werden (siehe Abschnitt 16.5).

Abstrakte Syntax

Sei $G = (S, Z, BT, R)$ eine CFG und $W = (S \cup Z \cup BT)^*$. Die folgende Funktion $typ : W \to \mathcal{T}_p(S)$ streicht alle Elemente von Z aus Wörtern von W und überführt diese in die durch sie bezeichneten Produkttypen:

- $typ(\epsilon) = 1$.
- Für alle $x \in Z$ und $w \in W$, $typ(xw) = typ(w)$.
- Für alle $s \in S \cup BT$ und $w \in W^*$, $typ(sw) = s \times typ(w)$.

Die konstruktive Signatur

$$\Sigma(G) = (S, \{f_{s \to w} \colon typ(w) \to s \mid s \to w \in R\})$$

heißt **abstrakte Syntax von** G.

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

Beispiel

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

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 $r = (s \to w_0 e_1 w_1 \dots e_n w_n)$ mit $w_0, \dots, w_n \in Z^*$ und $e_1, \dots, e_n \in S \cup BT$ entstandenen Konstruktor f_r die (Mixfix-)Darstellung $w_0\_w_1 \dots \_w_n$ gewählt.

So könnte beispielsweise der Konstruktor

$$f : \mathit{Disjunct} \times \mathit{Command} \times \mathit{Command} \to \mathit{Commands}$$

für die JavaLight-Regel

$$\mathit{Commands} \;\to\; \texttt{if}\ \mathit{Disjunct}\ \mathit{Command}\ \texttt{else}\ \mathit{Command}$$

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

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

$$G(\Sigma) \;=\; (S, F \cup \{(,),,\}, BT, R)$$

mit $\Sigma(G(\Sigma)) = \Sigma$ überführt werden, wobei BT die Menge der in Σ vorkommenden Basistypen ist und

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

Beispiel 4.5 SAB

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

$$r_1 = S \to aB, \qquad r_2 = S \to bA, \qquad r_3 = S \to \epsilon,$$
$$r_4 = A \to aS, \qquad r_5 = A \to bAA, \qquad r_6 = B \to bS, \qquad r_7 = B \to aBB.$$

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

$$f_1 : B \to S, \qquad f_2 : A \to S, \qquad f_3 : 1 \to S,$$
$$f_4 : S \to A, \qquad f_5 : A \times A \to A, \qquad f_6 : S \to B, \qquad f_7 : B \times B \to B.$$

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

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

$$\begin{aligned}
f_1^{SABcount} = f_4^{SABcount} &= \lambda(i,j).(i+1,j),\\
f_2^{SABcount} = f_6^{SABcount} &= \lambda(i,j).(i,j+1),\\
f_3^{SABcount} &= (0,0),\\
f_5^{SABcount} &= \lambda((i,j),(k,l)).(i+k,j+l+1),\\
f_7^{SABcount} &= \lambda((i,j),(k,l)).(i+k+1,j+l). \qquad \text{❑}
\end{aligned}$$

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

$$
\begin{aligned}
S \;=\; \{\; & \mathit{Commands}, \mathit{Command}, \mathit{Sum}, \mathit{Prod}, \mathit{Factor}, \mathit{Disjunct}, \mathit{Conjunct}, \mathit{Literal} \;\} \\
F \;=\; \{\; & \mathit{seq} : \mathit{Command} \times \mathit{Commands} \to \mathit{Commands}, \\
& \mathit{embed} : \mathit{Command} \to \mathit{Commands}, \\
& \mathit{block} : \mathit{Commands} \to \mathit{Command}, \\
& \mathit{assign} : \mathit{String} \times \mathit{Sum} \to \mathit{Command}, \\
& \mathit{cond} : \mathit{Disjunct} \times \mathit{Command} \times \mathit{Command} \to \mathit{Command}, \\
& \mathit{cond}1, \mathit{loop} : \mathit{Disjunct} \times \mathit{Command} \to \mathit{Command}, \\
& \mathit{sum} : \mathit{Prod} \to \mathit{Sum}, && (1) \\
& \mathit{plus}, \mathit{minus} : \mathit{Sum} \times \mathit{Prod} \to \mathit{Sum}, && (2) \\
& \mathit{prod} : \mathit{Factor} \to \mathit{Prod}, && (3) \\
& \mathit{times}, \mathit{div} : \mathit{Prod} \times \mathit{Factor} \to \mathit{Prod}, && (4) \\
& \mathit{embedI} : \mathbb{Z} \to \mathit{Factor}, \\
& \mathit{var} : \mathit{String} \to \mathit{Factor}, \\
& \mathit{encloseS} : \mathit{Sum} \to \mathit{Factor}, \\
& \mathit{disjunct} : \mathit{Conjunct} \times \mathit{Disjunct} \to \mathit{Disjunct}, \\
& \mathit{embedC} : \mathit{Conjunct} \to \mathit{Disjunct}, \\
& \mathit{conjunct} : \mathit{Literal} \times \mathit{Conjunct} \to \mathit{Conjunct}, \\
& \mathit{embedL} : \mathit{Literal} \to \mathit{Conjunct}, \\
& \mathit{not} : \mathit{Literal} \to \mathit{Literal},
\end{aligned}
$$

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

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

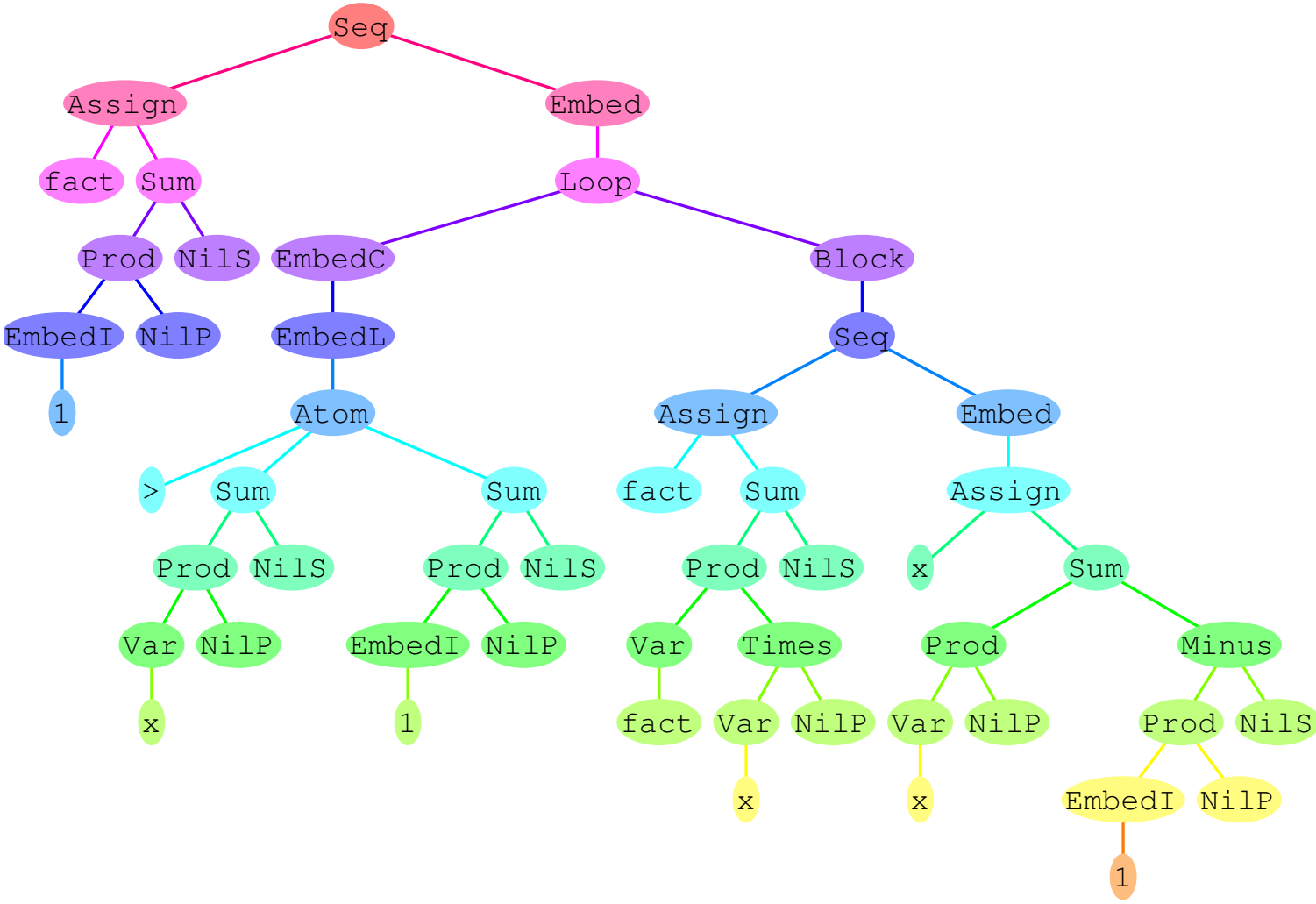
Die abstrakte Syntax der nach obigem Verfahren aus JavaLight gebildeten nicht-linksrekursiven Grammatik JavaLight′ enthält zusätzlich die Sorten *Sumsect* und *Prodsect* sowie folgende Konstruktoren anstelle von (1)-(4):

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

Z.B. hat das JavaLight-Programm

$$\texttt{fact} = \texttt{1};\ \texttt{while}\ \texttt{x} > \texttt{1}\ \{\texttt{fact} = \texttt{fact} * \texttt{x};\ \texttt{x} = \texttt{x} - \texttt{1};\}$$

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 http://cloudconvert.org kann diese in ein anderes Format konvertiert werden. ❏

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

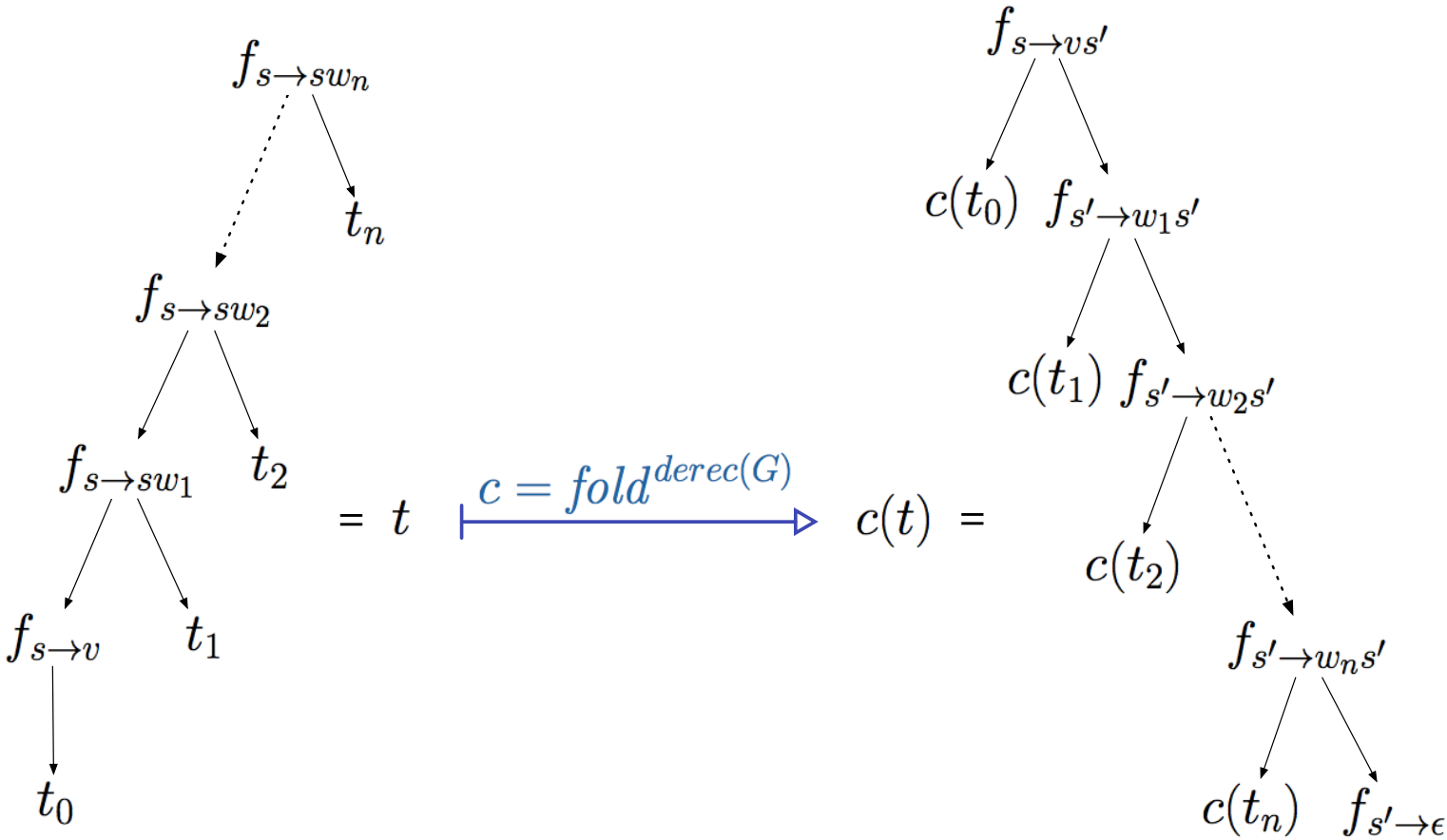
Die Faltung eines Syntaxbaums t von G in folgender $\Sigma(G)$-Algebra $derec(G)$ liefert den t entsprechenden Syntaxbaum von G':

- Für alle $s \in S \cup BT$, $derec(G)_s = T_{\Sigma(G'),s}$. (1)
- Für alle $s \in S \setminus recs(S)$, $s \to v \in R$ und $t \in T_{\Sigma(G'),typ(v)}$, $f_{s\to v}^{derec(G)}(t) = f_{s\to v}(t)$. (2)
- Für alle $s \to sw \in R$, $v \in nonrecs(s)$, $t \in T_{\Sigma(G'),typ(v)}$, $t' \in T_{\Sigma(G'),s'}$ und $u \in T_{\Sigma(G'),typ(w)}$,

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

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

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

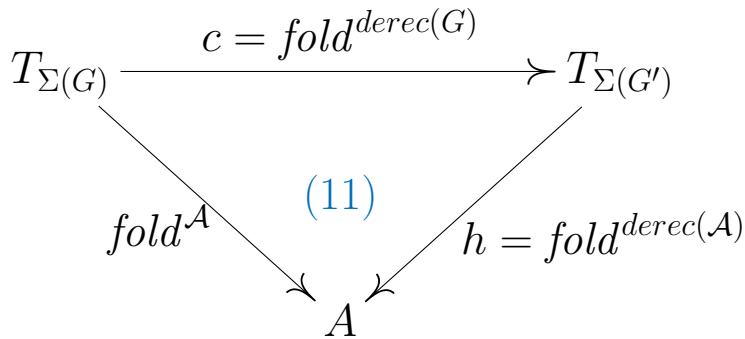


Ein Parser für G braucht die nicht-linkskursive Version G', um die Sprache von G zu entscheiden. Also erzeugt er gezwungenermaßen Syntaxbäume von G'.

Am *back end* des Compilers können wir jedoch zu G zurückkehren, d.h. als Zielsprachen sind auch $\Sigma(G)$-Algebren erlaubt (siehe Kapitel 1). Um Syntaxbäume in einer $\Sigma(G)$-Algebra $\mathcal{A} = (A, Op)$ falten zu können, muss diese in eine $\Sigma(G')$-Algebra transformiert werden, die wir mit $derec(\mathcal{A}) = (A', Op')$ bezeichnen und die sich wie folgt aus $\mathcal{A}$ ergibt:

- Für alle $s \in S \cup BT$, $A'_s = A_s$. (5)
- Für alle $s \in S \setminus recs(S)$ und $s \to v \in R$, $f^{derec(\mathcal{A})}_{s \to v} = f^{\mathcal{A}}_{s \to v}$. (6)
- Für alle $s' \in S'$, $s \to sw \in R$, $v \in nonrecs(s)$, $a \in A_{typ(v)}$, $b \in A_{typ(w)}$, $g : A_s \to A_s$ und $x \in A_s$,

$$A'_{s'} = (A_s \to A_s), \tag{7}$$
$$f^{derec(\mathcal{A})}_{s' \to \epsilon}(\epsilon) = id_{A_s}, \tag{8}$$
$$f^{derec(\mathcal{A})}_{s \to vs'}(a, g) = g(f^{\mathcal{A}}_{s \to v}(a)), \tag{9}$$
$$f^{derec(\mathcal{A})}_{s' \to ws'}(b, g)(x) = g(f^{\mathcal{A}}_{s \to sw}(x, b)). \tag{10}$$



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

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

$$\begin{aligned}
&h(c(t)) = h(c(f_{s\to v}(u))) \overset{c\ hom.}{=} h(f^{derec(G)}_{s\to v}(c(u))) \overset{(2)}{=} h(f_{s\to v}(c(u))) \\
&\overset{h\ hom.}{=} f^{derec(\mathcal{A})}_{s\to v}(h(c(u))) \overset{(6)}{=} f^{\mathcal{A}}_{s\to v}(h(c(u))) \overset{ind.\ hyp.}{=} f^{\mathcal{A}}_{s\to v}(fold^{\mathcal{A}}(u)) \\
&\overset{fold^{\mathcal{A}}\ hom.}{=} fold^{\mathcal{A}}(f_{s\to v}(u)) = fold^{\mathcal{A}}(t).
\end{aligned}$$

Sei $s \in recs(S)$ und $t \in T_{\Sigma(G),s}$. Dann gibt es $n \in \mathbb{N}$, $v \in nonrecs(s)$, $t_0 \in T_{\Sigma(G),typ(v)}$ und für alle $1 \le i \le n$ $s \to sw_i \in R$ und $t_i \in T_{\Sigma(G),typ(w_i)}$ mit

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

Daraus folgt

$$\begin{aligned}
&h(c(t)) \overset{(12)}{=} h(c(f_{s\to sw_n}(\dots(f_{s\to sw_1}(f_{s\to v}(t_0), t_1)\dots), t_n))) \\
&\overset{c\ hom.}{=} h(f^{derec(G)}_{s\to sw_n}(\dots(f^{derec(G)}_{s\to sw_1}(f^{derec(G)}_{s\to v}(c(t_0))), c(t_1))\dots), c(t_n)) \\
&\overset{(3)}{=} h(f^{derec(G)}_{s\to sw_n}(\dots(f^{derec(G)}_{s\to sw_1}(f_{s\to vs'}(c(t_0), f_{s'\to\epsilon})), c(t_1))\dots), c(t_n)) \\
&\overset{(4)}{=} h(f^{derec(G)}_{s\to sw_n}(\dots(f_{s\to vs'}(c(t_0), f_{s'\to w_1 s'}(c(t_1), f_{s'\to\epsilon}))\dots), c(t_n)))
\end{aligned}$$

$$\overset{n-1 \text{ applications of } (4)}{=} h(f_{s\to vs'}(c(t_0), f_{s'\to w_1 s'}(c(t_1), \ldots, f_{s'\to w_n s'}(c(t_n), f_{s'\to\epsilon})\ldots))$$

$$\overset{h \text{ hom.}}{=} f^{derec(\mathcal{A})}_{s\to vs'}(h(c(t_0)), f^{derec(\mathcal{A})}_{s'\to w_1 s'}(h(c(t_1)), \ldots, f^{derec(\mathcal{A})}_{s'\to w_n s'}(h(c(t_n)), f^{derec(\mathcal{A})}_{s'\to\epsilon})\ldots))$$

$$\overset{(8)}{=} f^{derec(\mathcal{A})}_{s\to vs'}(h(c(t_0)), f^{derec(\mathcal{A})}_{s'\to w_1 s'}(h(c(t_1)), \ldots, f^{derec(\mathcal{A})}_{s'\to w_n s'}(h(c(t_n)), id)\ldots))$$

$$\overset{(9)}{=} f^{derec(\mathcal{A})}_{s'\to w_1 s'}(h(c(t_1)), \ldots, f^{derec(\mathcal{A})}_{s'\to w_n s'}(h(c(t_n)), id)\ldots)(f^{\mathcal{A}}_{s\to v}(h(c(t_0))))$$

$$\overset{(10)}{=} (\ldots, f^{derec(\mathcal{A})}_{s'\to w_n s'}(h(c(t_n)), id)\ldots)(f^{\mathcal{A}}_{s\to sw_1}(f^{\mathcal{A}}_{s\to v}(h(c(t_0))), h(c(t_1)))) = \ldots$$

$$\overset{(10)}{=} f^{derec(\mathcal{A})}_{s'\to w_n s'}(h(c(t_n)), id)(\ldots(f^{\mathcal{A}}_{s\to sw_1}(f^{\mathcal{A}}_{s\to v}(h(c(t_0))), h(c(t_1))))\ldots)$$

$$\overset{(10)}{=} id(f^{\mathcal{A}}_{s\to sw_n}(\ldots(f^{\mathcal{A}}_{s\to sw_1}(f^{\mathcal{A}}_{s\to v}(h(c(t_0))), h(c(t_1))))\ldots, h(c(t_n))))$$

$$= f^{\mathcal{A}}_{s\to sw_n}(\ldots(f^{\mathcal{A}}_{s\to sw_1}(f^{\mathcal{A}}_{s\to v}(h(c(t_0))), h(c(t_1))))\ldots, h(c(t_n)))$$

$$\overset{ind.\ hyp.}{=} f^{\mathcal{A}}_{s\to sw_n}(\ldots(f^{\mathcal{A}}_{s\to sw_1}(f^{\mathcal{A}}_{s\to v}(fold^{\mathcal{A}}(t_0)), fold^{\mathcal{A}}(t_1)))\ldots, fold^{\mathcal{A}}(t_n))$$

$$\overset{fold^{\mathcal{A}} \text{ hom.}}{=} fold^{\mathcal{A}}(f_{s\to sw_n}(\ldots(f_{s\to sw_1}(f_{s\to v}(t_0), t_1)\ldots), t_n)) \overset{(12)}{=} fold^{\mathcal{A}}(t). \qquad \square$$

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

$$\begin{aligned} S' &= S \cup \{Sumsect, Prodsect\}, \\ F' &= F \setminus \{sum, plus, minus, prod, times, div\} \\ &\quad \cup \{sum', plus', minus', nilS, prod', times', div', nilP\}. \end{aligned}$$

Damit ist die $\Sigma(G)$-Algebra $derec(G)$ – neben (1) und (2) – wie folgt definiert:

Für alle $t, u \in T_{\Sigma(G),Prod}$ und $t' \in T_{\Sigma(G'),Sumsect}$,

$$\begin{aligned} sum^{derec(G)}(t) &= sum'(t, nilS), \\ plus^{derec(G)}(sum'(t,t'),u) &= sum'(t, plus'(u,t')), \\ minus^{derec(G)}(sum'(t,t'),u) &= sum'(t, minus'(u,t')). \end{aligned}$$

Für alle $t, u \in T_{\Sigma(G),Factor}$ und $t' \in T_{\Sigma(G'),Prodsect}$,

$$\begin{aligned} prod^{derec(G)}(t) &= prod'(t, nilP), \\ times^{derec(G)}(prod'(t,t'),u) &= prod'(t, times'(u,t')), \\ div^{derec(G)}(prod'(t,t'),u) &= prod'(t, div'(u,t')) \end{aligned}$$

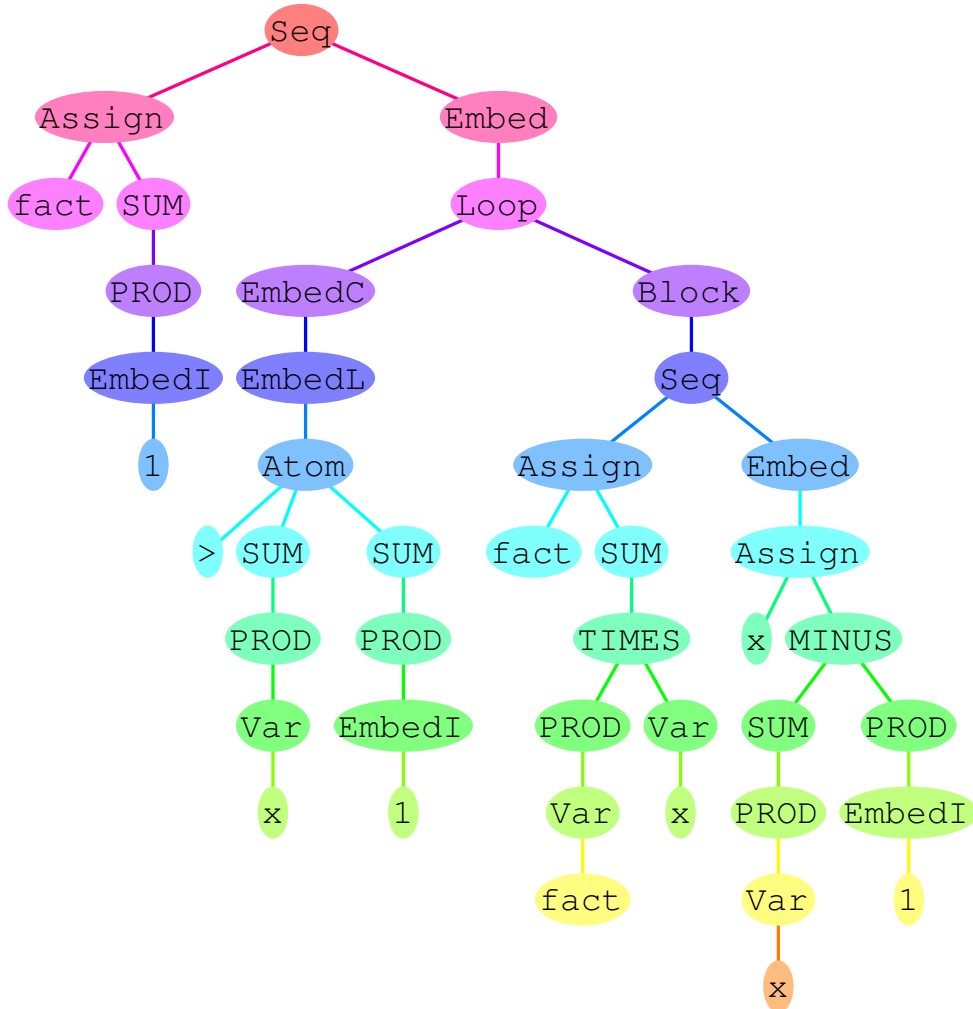
Sei $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$-Algebra. Die $\Sigma(G')$-Algebra $derec(\mathcal{A}) = (A', Op')$ ist – neben (3) und (4) – wie folgt definiert:

- $A'_{Sumsect} = A_{Sum} \to A_{Sum}$ und $A'_{Prodsect} = A_{Prod} \to A_{Prod}$.
- Für alle $a \in A_{Prod}$, $g : A_{Sum} \to A_{Sum}$ und $x \in A_{Sum}$,

$$\begin{aligned} sum'^{derec(\mathcal{A})}(a, g) &= g(sum^{\mathcal{A}}(a)), \\ plus'^{derec(\mathcal{A})}(a, g)(x) &= g(plus^{\mathcal{A}}(x, a)), \\ minus'^{derec(\mathcal{A})}(a, g)(x) &= g(minus^{\mathcal{A}}(x, a)), \\ nilS^{derec(\mathcal{A})} &= id_{A_{Sum}}. \end{aligned}$$

- Für alle $a \in A_{Factor}$, $g : A_{Prod} \to A_{Prod}$ und $x \in A_{Prod}$,

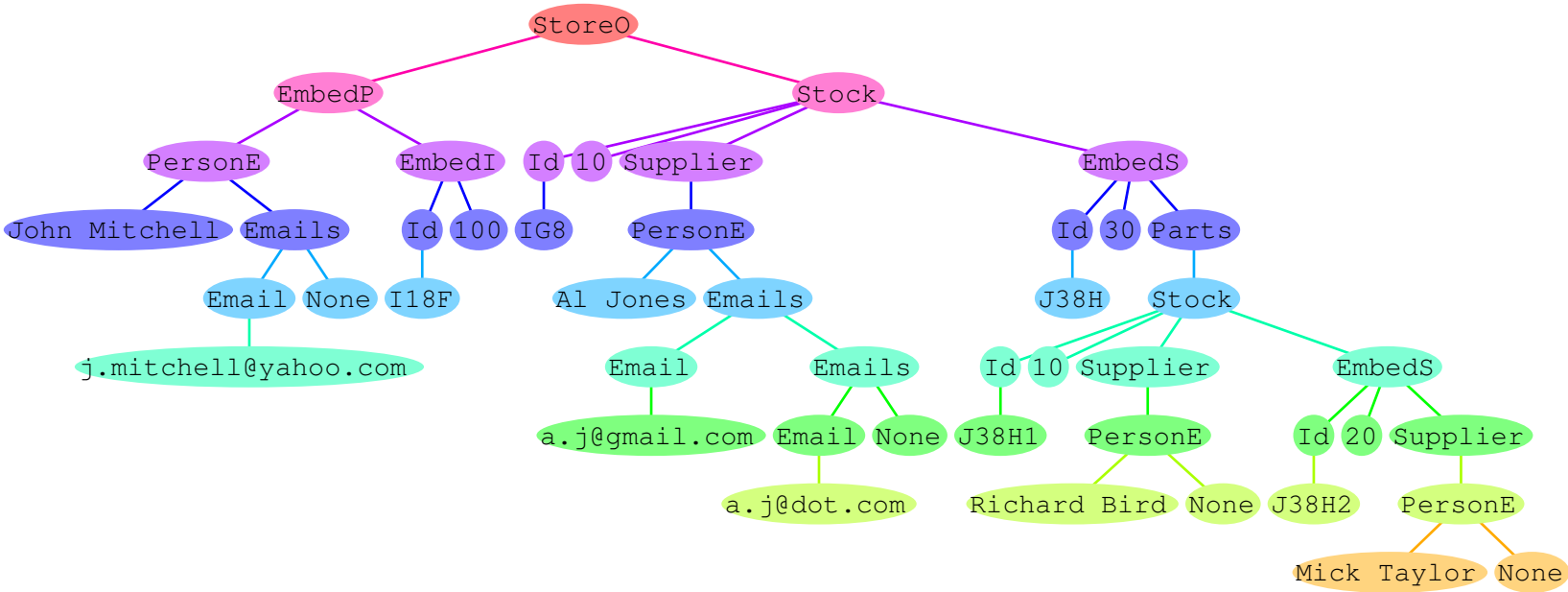
$$\begin{aligned} prod'^{derec(\mathcal{A})}(a, g) &= g(prod^{\mathcal{A}}(a)), \\ times'^{derec(\mathcal{A})}(a, g)(x) &= g(times^{\mathcal{A}}(x, a)), \\ div'^{derec(\mathcal{A})}(a, g)(x) &= g(div^{\mathcal{A}}(x, a)), \\ nilP^{derec(\mathcal{A})} &= id_{A_{Prod}}. \end{aligned}$$



Syntaxbaum von G, dessen G'-Version (= Faltung in 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:

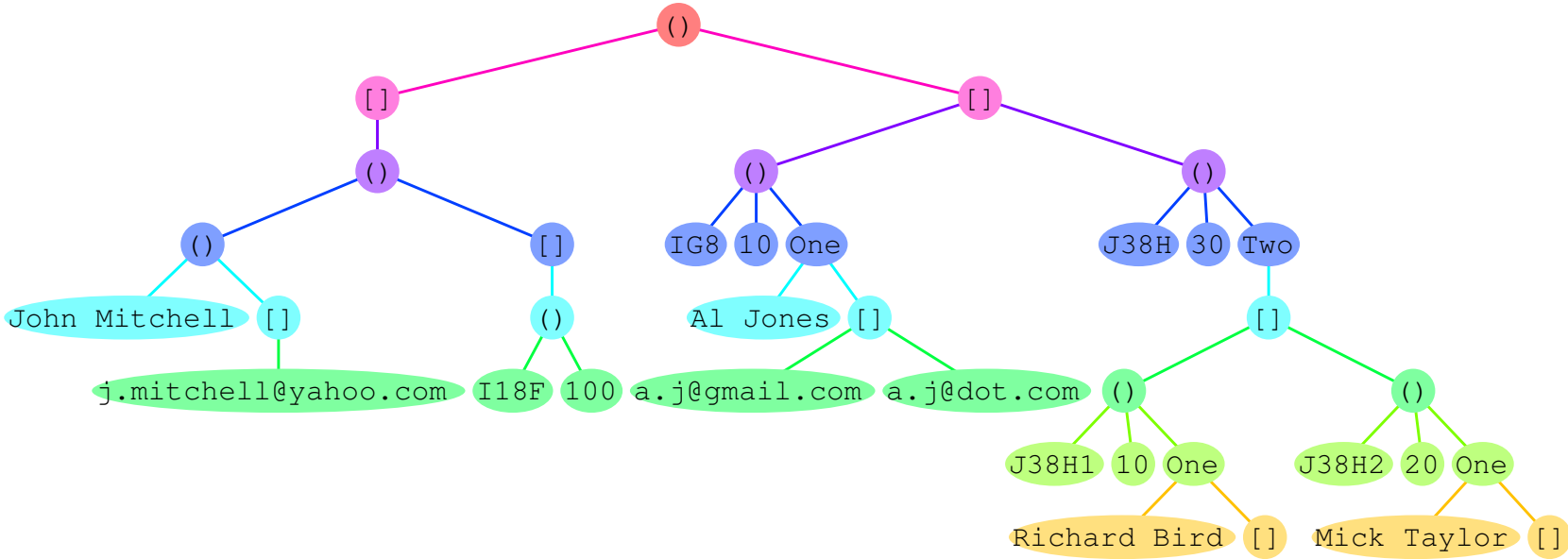
$$
\begin{aligned}
S &= \{\ \mathit{Store}, \mathit{Orders}, \mathit{Order}, \mathit{Person}, \mathit{Emails}, \mathit{Email}, \mathit{Items}, \mathit{Item}, \mathit{Stock}, \\
&= \{\ \mathit{ItemS}, \mathit{Suppliers}, \mathit{Id}\ \} \\
F &= \{\ \mathit{store} && : \mathit{Stock} \to \mathit{Store}, \\
&\quad \mathit{storeO} && : \mathit{Orders} \times \mathit{Stock} \to \mathit{Store}, \\
&\quad \mathit{orders} && : \mathit{Person} \times \mathit{Items} \times \mathit{Orders} \to \mathit{Orders}, \\
&\quad \mathit{embedP} && : \mathit{Person} \times \mathit{Items} \to \mathit{Orders}, \\
&\quad \mathit{person} && : \mathit{String} \to \mathit{Person}, \\
&\quad \mathit{personE} && : \mathit{String} \times \mathit{Emails} \to \mathit{Person}, \\
&\quad \mathit{emails} && : \mathit{Email} \times \mathit{Emails} \to \mathit{Emails}, \\
&\quad \mathit{none} && : 1 \to \mathit{Emails}, \\
&\quad \mathit{email} && : \mathit{String} \to \mathit{Email}, \\
&\quad \mathit{items} && : \mathit{Id} \times \mathit{String} \times \mathit{Items} \to \mathit{Items}, \\
&\quad \mathit{embedI} && : \mathit{Id} \times \mathit{String} \to \mathit{Items}, \\
&\quad \mathit{stock} && : \mathit{Id} \times \mathbb{Z} \times \mathit{Supplier} \times \mathit{Stock} \to \mathit{Stock}, \\
&\quad \mathit{embedS} && : \mathit{Id} \times \mathbb{Z} \times \mathit{Supplier} \to \mathit{Stock}, \\
&\quad \mathit{supplier} && : \mathit{Person} \to \mathit{Suppliers}, \\
&\quad \mathit{parts} && : \mathit{Stock} \to \mathit{Suppliers}, \\
&\quad \mathit{id} && : \mathit{String} \to \mathit{Id}\ \}
\end{aligned}
$$



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 $G = (S, Z, BT, R)$ eine LL-kompilierbare CFG und $X = Z \cup \bigcup BT$.

4.8 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.

$Word(G)$, die Wortalgebra von G

- Für alle $s \in S$, $Word(G)_s =_{def} X^*$.
- Für alle $w_0 \dots w_n \in Z^*$, $s_1, \dots, s_n \in S \cup BT$, $r = (s \to w_0 s_1 w_1 \dots s_n w_n) \in R$ und $(v_1, \dots, v_n) \in Word(G)_{s_1 \times \dots \times s_n} = (X^*)^n$,

$$f_r^{Word(G)}(v_1, \dots, v_n) =_{def} w_0 v_1 w_1 \dots v_n w_n.$$

$t \in T_{\Sigma(G)}$ heißt **G-Syntaxbaum für** $w \in X^*$, falls $fold^{Word(G)}(t) = w$ gilt.

Die **Sprache** $L(G) = (L(G)_s)_{s \in S}$ **von** G ist die S-sortige Menge der Wörter über X, die sich aus der Faltung eines Syntaxbaums in $Word(G)$ ergeben: Für alle $s \in S$,

$$L(G)_s \ =_{def} \ fold_s^{Word(G)}(T_{\Sigma(G),s}).$$

G heißt **eindeutig**, wenn für alle $s \in S$ $fold_s^{Word(G)}$ injektiv ist.

Da die Faltung in $Word(G)$ zur Eingabe zurückführt, nennt man sie auch **Unparser**.

Zwei Grammatiken mit derselben Sortenmenge heißen **äquivalent**, wenn ihre Sprachen übereinstimmen.

Beispiel Nochmal die Regeln von SAB (siehe Beispiel 4.5):

$$r_1 = S \to aB, \ r_2 = S \to bA, \quad r_3 = S \to \epsilon,$$
$$r_4 = A \to aS \quad r_5 = A \to bAA, \ r_6 = B \to bS, \ r_7 = B \to aBB.$$

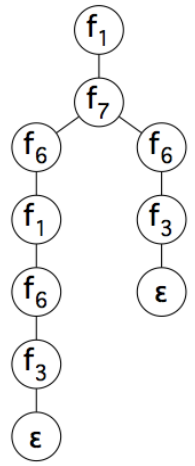
Alle drei Trägermengen der Wortalgebra $Word(\mathrm{SAB})$ sind durch $\{a,b\}^*$ gegeben.

Die Konstruktoren von $\Sigma(\mathrm{SAB})$ werden in $Word(\mathrm{SAB})$ wie folgt interpretiert:

Für alle $v, w \in \{a,b\}^*$,

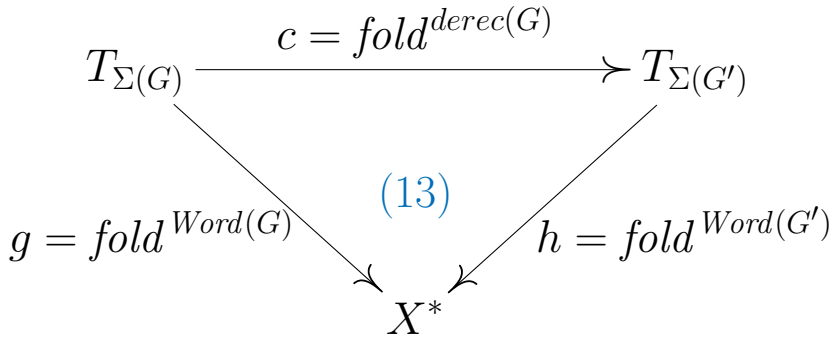
$$\begin{array}{lcl} f_1^{Word(\mathrm{SAB})}(w) & = & f_4^{Word(\mathrm{SAB})}(w) = aw, \\ f_2^{Word(\mathrm{SAB})}(w) & = & f_6^{Word(\mathrm{SAB})}(w) = bw, \\ f_3^{Word(\mathrm{SAB})} & = & \epsilon, \\ f_5^{Word(\mathrm{SAB})}(v,w) & = & bvw, \\ f_7^{Word(\mathrm{SAB})}(v,w) & = & avw. \end{array}$$

Syntaxbaum für `aababb`:

$fold^{Word(G)}($  $) = \texttt{aababb}$

Mit ϵ markierte Blätter eines Syntaxbaums werden künftig weglassen. ❑

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:



Aufgabe Aus (der Kommutativität von) (13) und der Bijektivität von c (!) folgt

$$g \circ c^{-1} = h \circ c \circ c^{-1} = h, \tag{14}$$

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 `javasource`. Die Inhalte von `prog` und `javasource` stimmen also miteinander überein!

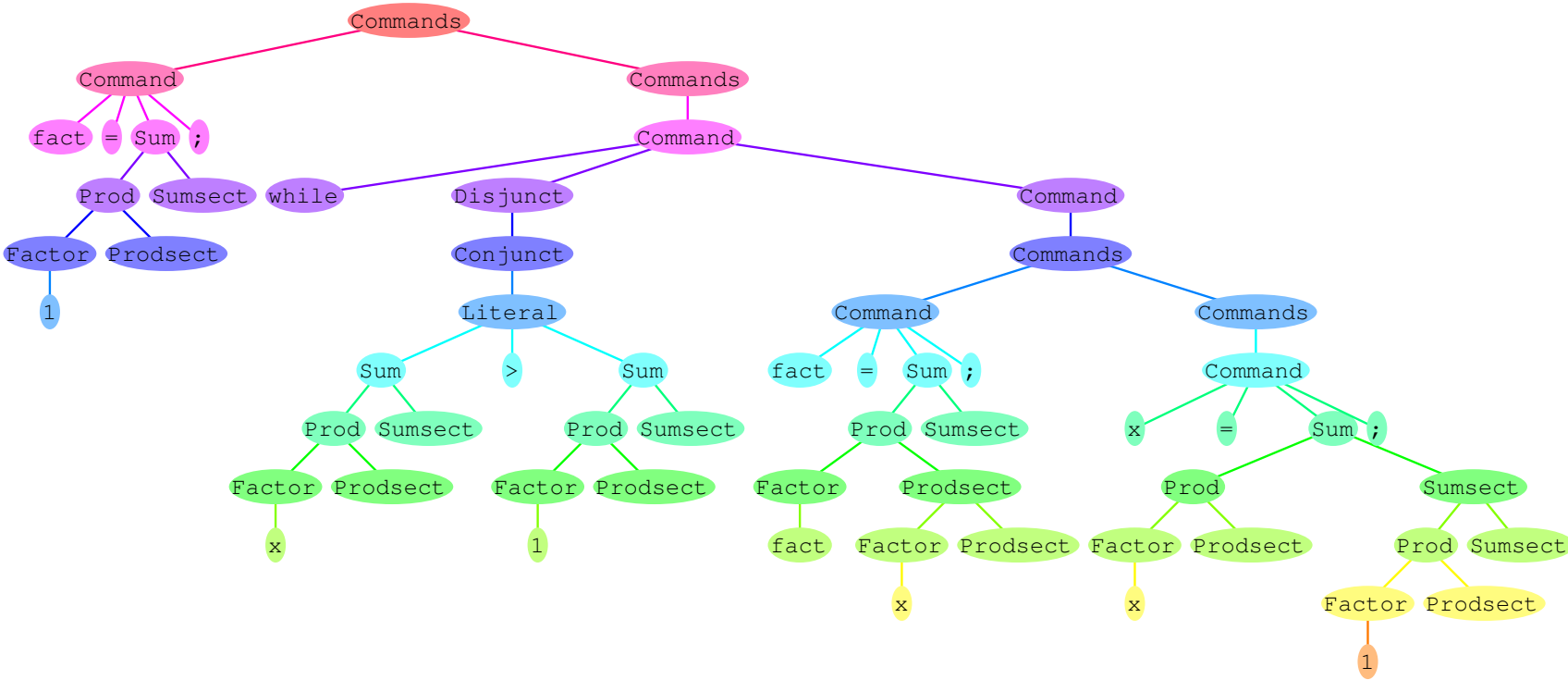
$Abl(G)$, die Ableitungsbaumalgebra von G

- Für alle $s \in S$, $Abl(G)_s = wtr(\mathbb{N}, S \cup Z \cup BT)$ (siehe Kapitel 2).
- Für alle $z_0, \ldots, z_{n_0}, z_{n_0+2}, \ldots, z_{n_{k-1}+2}, \ldots, z_{n_k} \in Z$, $s_1, \ldots, s_k \in S \cup BT$,

$$r = (s \to z_0 \ldots z_{n_0} s_1 z_{n_0+2} \ldots z_{n_{k-1}} s_k z_{n_{k-1}+2} \ldots z_{n_k}) \in R$$

und $(t_1, \ldots, t_k) \in Abl(G)_{s_1 \times \ldots \times s_k} = wtr(\mathbb{N}, S \cup Z \cup BT)^k$,

$$f_r^{Abl(G)}(t_1, \ldots, t_k) =_{def} s\{\ 0 \to z_0, \ldots,$$
$$n_0 \to z_{n_0}, n_0 + 1 \to t_1, n_0 + 2 \to z_{n_0+2},$$
$$\ldots,$$
$$n_{k-1} \to z_{n_{k-1}}, n_{k-1} + 1 \to t_k, n_{k-1} + 2 \to z_{n_{k-1}+2},$$
$$\ldots, n_k \to z_{n_k}\ \}$$



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 $Store = String \to \mathbb{Z}$ (Menge der *Speicherzustände*; hier: Belegungen von Programmvariablen durch ganze Zahlen).

Das Zustandsmodell von JavaLight ist durch folgende Σ(JavaLight)-Algebra $\mathcal{A} = (A, Op)$ gegeben (siehe Beispiel 4.6):

$$A_{Commands} = A_{Command} = Store \rightharpoonup Store \quad \text{(Menge der partiellen Funktionen von } Store \text{ nach } Store\text{)}$$

$$A_{Sum} = A_{Prod} = A_{Factor} = Store \to \mathbb{Z}$$

$$A_{Disjunct} = A_{Conjunct} = A_{Literal} = Store \to 2$$

Für alle $f, g : Store \rightharpoonup Store$, $x \in String$, $e : Store \to \mathbb{Z}$, $st \in Store$ und $p : Store \to 2$,

$$\begin{aligned}
seq^{\mathcal{A}}(f,g) &= g \circ f, \\
embed^{\mathcal{A}}(f) &= block^{\mathcal{A}}(f) = f, \\
assign^{\mathcal{A}}(x,e)(st) &= st[e(st)/x], \\
cond^{\mathcal{A}}(p,f,g)(st) &= \mathit{if}\ p(st)\ \mathit{then}\ f(st)\ \mathit{else}\ g(st), \\
cond1^{\mathcal{A}}(p,f)(st) &= \mathit{if}\ p(st)\ \mathit{then}\ f(st)\ \mathit{else}\ st, \\
loop^{\mathcal{A}}(p,f)(st) &= \mathit{if}\ p(st)\ \mathit{then}\ loop^{\mathcal{A}}(p,f)(f(st))\ \mathit{else}\ st.
\end{aligned}$$

Offenbar können mit der letzten Gleichung unendliche Berechnungen erzeugt werden. In solchen Fällen bleibt $loop^{\mathcal{A}}(p, f)(st)$ undefiniert. Aus diesem Grund mussten wir *Commands* und *Command* durch Mengen *partieller* Funktionen interpretieren.

Darüberhinaus liefert die Gleichung für $loop^{\mathcal{A}}$ keine Definition, sondern nur eine Anforderung an eine Funktion, deren Existenz noch zu zeigen ist. Der Beweis wendet den Fixpunktsatz von Kleene auf den CPO $Store \multimap Store$ an (siehe Abschnitt 17.1).

Für alle $f, g : Store \to \mathbb{Z}$, $st \in Store$, $x \in String$ und $i \in \mathbb{Z}$,

$$\begin{aligned}
sum^{\mathcal{A}}(f) &= prod^{\mathcal{A}}(f) = f, \\
plus^{\mathcal{A}}(f, g) &= \lambda st.f(st) + g(st), \\
minus^{\mathcal{A}}(f, g) &= \lambda st.f(st) - g(st), \\
times^{\mathcal{A}}(f, g) &= \lambda st.f(st) * g(st), \\
div^{\mathcal{A}}(f, g) &= \lambda st.f(st)/g(st), \\
embedI^{\mathcal{A}}(i)(st) &= i, \\
var^{\mathcal{A}}(x)(st) &= st(x), \\
encloseS^{\mathcal{A}}(f) &= f.
\end{aligned}$$

Für alle $f, g : Store \to 2$, $rel \in Rel$, $e, e' : Store \to \mathbb{Z}$ und $b \in 2$,

$$\begin{aligned}
disjunct^{\mathcal{A}}(f,g) &= \lambda st.f(st) \vee g(st),\\
embedC^{\mathcal{A}}(f) &= embedL^{\mathcal{A}}(f) = encloseD^{\mathcal{A}}(f) = f,\\
conjunct^{\mathcal{A}}(f,g) &= \lambda st.f(st) \wedge g(st),\\
not^{\mathcal{A}}(f) &= \neg \circ f,\\
atom^{\mathcal{A}}(rel,e,e') &= \lambda st.rel(e(st),e'(st)),\\
embedB^{\mathcal{A}}(b)(st) &= b.
\end{aligned}$$

In Abschnitt 9.5 wird $\mathcal{A}$ unter dem Namen *javaState* in Haskell implementiert.

Laut Abschnitt 4.4 werden die zusätzlichen Sorten bzw. Konstruktoren von $\Sigma(\text{JavaLight}')$ in $\mathcal{A}' = derec(\mathcal{A}) = (A', Op')$ wie folgt interpretiert:

$$\begin{aligned}
A'_{Sumsect} &= A_{Sum} \to A_{Sum} = (Store \to \mathbb{Z}) \to (Store \to \mathbb{Z}),\\
A'_{Prodsect} &= A_{Prod} \to A_{Prod} = (Store \to \mathbb{Z}) \to (Store \to \mathbb{Z}).
\end{aligned}$$

Für alle $f, g : Store \to \mathbb{Z}$ und $h : (Store \to \mathbb{Z}) \to (Store \to \mathbb{Z})$,

$$\begin{aligned}
sum'^{\mathcal{A}'}(f,h) &= h(sum^{\mathcal{A}}(f)) = h(f),\\
plus'^{\mathcal{A}'}(f,h)(g) &= h(plus^{\mathcal{A}}(g,f)) = h(\lambda st.g(st) + f(st)),\\
minus'^{\mathcal{A}'}(f,h)(g) &= h(minus^{\mathcal{A}}(g,f)) = h(\lambda st.g(st) - f(st)),
\end{aligned}$$

$$
\begin{aligned}
nilS^{\mathcal{A}'} &= id_{Store\to\mathbb{Z}},\\
prod'^{\mathcal{A}'}(f,h) &= h(prod^{\mathcal{A}}(f)) = h(f),\\
times'^{\mathcal{A}'}(f,h)(g) &= h(div^{\mathcal{A}}(g,f)) = h(\lambda st.g(st) * f(st)),\\
div'^{\mathcal{A}'}(f,h)(g) &= h(div^{\mathcal{A}}(g,f)) = h(\lambda st.g(st)/f(st)),\\
nilP^{\mathcal{A}'} &= id_{Store\to\mathbb{Z}}.
\end{aligned}
$$

Alternative Interpretation der Sektionssorten

Die Faltung eines $\Sigma(\text{JavaLight}')$-Terms t der Sorte *Sum* oder *Prod* führt zum gleichen Ergebnis wie die Faltung von t in der $\Sigma(\text{JavaLight}')$-Algebra $\mathcal{A}'' = (A'', Op'')$, deren $\Sigma(\text{JavaLight})$-Redukt mit $\mathcal{A}$ übereinstimmt und die $Sumsect, \ldots, nilP$ wie folgt interpretiert:

$$A''_{Sumsect} = A''_{Prodsect} = Store \to (\mathbb{Z} \to \mathbb{Z}).$$

Für alle $f : Store \to \mathbb{Z}$, $h : Store \to (\mathbb{Z} \to \mathbb{Z})$, $st \in Store$ und $i \in \mathbb{Z}$,

$$
\begin{aligned}
sum'^{\mathcal{A}''}(f,h) &= \lambda st.h(st)(f(st)),\\
plus'^{\mathcal{A}''}(f,h)(st)(i) &= h(st)(i + f(st)),\\
minus'^{\mathcal{A}''}(f,h)(st)(i) &= h(st)(i - f(st)),\\
nilS^{\mathcal{A}''} &= \lambda st.id_{\mathbb{Z}},
\end{aligned}
$$

$$
\begin{aligned}
prod'^{\mathcal{A}''}(f,h) &= \lambda st.h(st)(f(st)),\\
times'^{\mathcal{A}''}(f,h)(st)(i) &= h(st)(i * f(st)),\\
div'^{\mathcal{A}''}(f,h)(st)(i) &= h(st)(i/f(st)),\\
nilP^{\mathcal{A}''} &= \lambda st.id_{\mathbb{Z}}.
\end{aligned}
$$

Die S-sortige Funktion $\Phi : A'' \to A'$ sei wie folgt definiert:

- Für alle $h : Store \to (\mathbb{Z} \to \mathbb{Z})$, $f : Store \to \mathbb{Z}$ und $st \in Store$,

$$\Phi_{Sumsect}(h)(f)(st) = \Phi_{Prodsect}(h)(f)(st) = h(st)(f(st)).$$

- Für alle $s \in S \setminus \{Sumsect, Prodsect\}$ und $a \in A_s$, $\Phi_s(a) = a$.

Da Φ ein $\Sigma(\text{JavaLight}')$-Homomorphismus von $\mathcal{A}''$ nach $\mathcal{A}'$ ist (Beweis!), gilt

$$fold^{\mathcal{A}'} = \Phi \circ fold^{\mathcal{A}''}.$$

also insbesondere $fold_s^{\mathcal{A}'} = fold_s^{\mathcal{A}''}$ für alle $s \in S \setminus \{Sumsect, Prodsect\}$.

5 Parser und Compiler für CFGs

Sei $G = (S, Z, BT, R)$ eine LL-kompilierbare CFG, $X = Z \cup \bigcup BT$ und *Ziel* eine $\Sigma(G)$-Algebra, in die Wörter über X übersetzt werden sollen.

In Anlehnung an den klassischen Begriff eines semantisch korrekten Compilers [25, 45] verlangen wir, dass die Semantiken *Sem* und *Mach* seiner Quell- bzw. Zielsprache in der durch das folgende Funktionsdiagramm ausgedrückten Weise miteinander zusammenhängen:

$$\begin{array}{ccc} T_{\Sigma(G)} & \xrightarrow{fold^{Ziel}} & Ziel \\ \Big\downarrow{\scriptstyle fold^{Sem}} & (1) & \Big\downarrow{\scriptstyle evaluate} \\ Sem & \xrightarrow[encode]{} & Mach \end{array}$$

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 [45] 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 $evaluate \circ fold^{Ziel}$ und $encode \circ fold^{Mach}$ $\Sigma(G)$-homomorph. Da $T_{\Sigma(G)}$ eine initiale $\Sigma(G)$-Algebra ist, sind beide Kompositionen gleich. Also kommutiert (1).

Während $fold^{Ziel}$ Syntaxbäume in der *Zielsprache* auswertet, ordnet $fold^{Word(G)}$ (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

$$parse_G : X^* \to M(T_{\Sigma(G)}),$$

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**

$$\circ : \mathcal{K}(A,B) \times \mathcal{K}(B,C) \to \mathcal{K}(A,C)$$

$$(f,g) \longmapsto g \circ f,$$

- 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 $f : A \to B$
anstelle von $f \in \mathcal{K}(A,B)$.

Wir haben hier mit vier Kategorien zu tun: Set, Set^2 und 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 Alg_Σ von Set^S, deren Objekte alle Σ-Algebren und deren Morphismen alle Σ-Homomorphismen sind.

Seien $\mathcal{K}, \mathcal{L}$ Kategorien. Ein **Funktor** $F : \mathcal{K} \to \mathcal{L}$ ist eine Funktion, die jedem $\mathcal{K}$-Objekt ein $\mathcal{L}$-Objekt und jedem $\mathcal{K}$-Morphismus $f : A \to B$ einen $\mathcal{L}$-Morphismus $F(f) : F(A) \to F(B)$ zuordnet sowie folgende Gleichungen erfüllt:

- Für alle $\mathcal{K}$-Objekte A, $F(id_A) = id_{F(A)}$, (2)
- Für alle $\mathcal{K}$-Morphismen $f : A \to B$ and $g : B \to C$, $F(g \circ f) = F(g) \circ F(f)$. (3)

Zwei Funktoren $F : \mathcal{K} \to \mathcal{L}$ und $G : \mathcal{L} \to \mathcal{M}$ kann man wie Funktionen zu weiteren Funktoren komponieren: Für alle $\mathcal{K}$-Objekte A und $\mathcal{K}$-Morphismen f,

$$\begin{aligned}(G \circ F)(A) &=_{def} G(F(A)),\\ (G \circ F)(f) &=_{def} G(F(f)).\end{aligned}$$

Meistens schreibt man GF anstelle von $G \circ F$.

Beispiele

Sei $B \in \mathcal{L}$. Der **konstante Funktor** $const(B) : \mathcal{K} \to \mathcal{L}$ ordnet jedem $\mathcal{K}$-Objekt das $\mathcal{L}$-Objekt B zu und jedem $\mathcal{K}$-Morphismus die Identität auf B.

Der **Identitätsfunktor** $Id_{\mathcal{K}} : \mathcal{K} \to \mathcal{K}$ ordnet jedem $\mathcal{K}$-Objekt und jedem $\mathcal{K}$-Morphismus sich selbst zu.

Der **Diagonalfunktor** $\Delta_{\mathcal{K}} : \mathcal{K} \to \mathcal{K}^2$ ordnet jedem $\mathcal{K}$-Objekt A das Objektpaar (A, A) und jedem $\mathcal{K}$-Morphismus f das Morphismenpaar (f, f) zu.

Der **Produktfunktor** $\_ \times \_ : Set^2 \to Set$ ordnet jedem Mengenpaar (A, B) die Menge $A \times B$ und jedem Funktionspaar $(f : A \to B, g : C \to D)$ die Funktion $f \times g =_{def} \lambda(a, c).(f(a), g(c))$ zu.

Der **Listenfunktor** $\_^* : Set \to Set$ ordnet jeder Menge A die Menge A^* der Wörter über A zu und jeder Funktion $f : A \to B$ die Funktion

$$\begin{aligned} f^* = map(f) : A^* &\to B^* \\ \epsilon &\mapsto \epsilon \\ (a_1, \ldots, a_n) &\mapsto (f(a_1), \ldots, f(a_n)) \end{aligned}$$

Der **Mengenfunktor** $\mathcal{P} : Set \to Set$ ordnet jeder Menge A die Potenzmenge $\mathcal{P}(A)$ zu und jeder Funktion $f : A \to B$ die Funktion

$$\begin{aligned} \mathcal{P}(f) : \mathcal{P}(A) &\to \mathcal{P}(B) \\ C &\mapsto \{f(c) \mid c \in C\} \end{aligned}$$

Sei E eine Menge von Fehlermeldungen, Ausnahmewerten o.ä.

Der **Ausnahmefunktor** $\_ + E : Set \to Set$ ordnet jeder Menge A die Menge $A + E$ zu (siehe Kapitel 2) und jeder Funktion $f : A \to B$ die Funktion

$$\begin{aligned} f + E : A + E &\to B + E \\ (a, 1) &\mapsto (f(a), 1) \\ (e, 2) &\mapsto (e, 2) \end{aligned}$$

Sei S eine Menge.

Der **Potenz-** oder **Leserfunktor** $\_^S : Set \to Set$ ordnet jeder Menge A die Menge $S \to A$ zu und jeder Funktion $f : A \to B$ die Funktion

$$\begin{aligned} f^S : A^S &\to B^S \\ g &\mapsto f \circ g \end{aligned}$$

Der **Copotenz-** oder **Schreiberfunktor** $\_ \times S : Set \to Set$ kombiniert den Identitätsfunktor mit einem Produktfunktor: Er ordnet jeder Menge A die Menge $A \times S$ zu und jeder Funktion $f : A \to B$ die Funktion

$$\begin{aligned} f \times S : A \times S &\to B \times S \\ (a, s) &\mapsto (f(a), s) \end{aligned}$$

Der **Zustandsfunktor** $(\_ \times S)^S : Set \to Set$ kombiniert einen Leser- mit einem Schreiberfunktor: Er ordnet jeder Menge A die Menge $(A \times S)^S$ zu und jeder Funktion $f : A \to B$ die Funktion

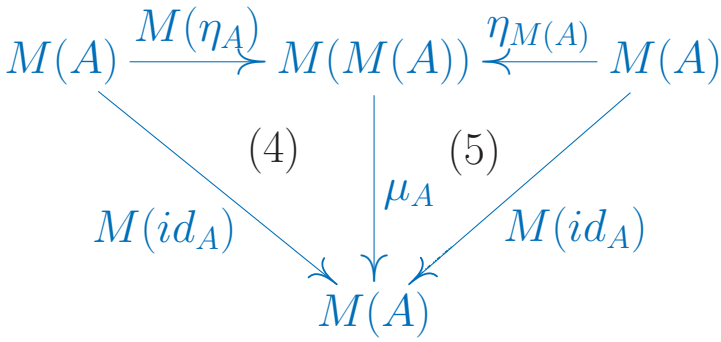
$$\begin{aligned} (f \times S)^S : (A \times S)^S &\to (B \times S)^S \\ g &\mapsto (\lambda(a, s).(f(a), s)) \circ g \end{aligned}$$

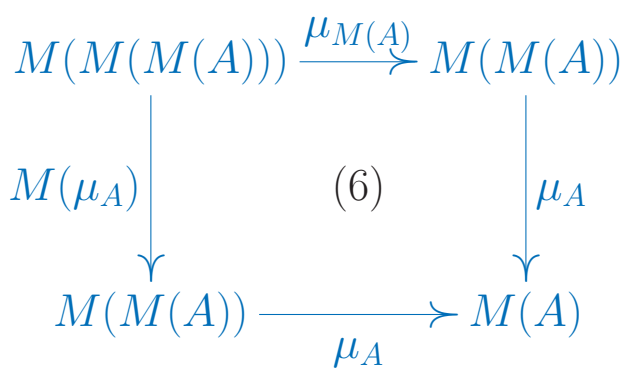
Aufgabe Zeigen Sie, dass die hier definierten Funktionen tatsächlich Funktoren sind, also (2) und (3) erfüllen, und dass sie sich von Set auf Set^S fortsetzen lassen. ❑

Seien $F, G : \mathcal{K} \to \mathcal{L}$ Funktoren. Eine **natürliche Transformation** $\tau : F \to G$ ordnet jedem $\mathcal{K}$-Objekt A einen $\mathcal{L}$-Morphismus $\tau_A : F(A) \to G(A)$ derart, dass für alle $\mathcal{K}$-Morphismen $f : A \to B$ folgendes Diagramm kommutiert:

$$\begin{array}{ccc} F(A) & \xrightarrow{\tau_A} & G(A) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(B) & \xrightarrow{\tau_B} & G(B) \end{array}$$

Ein Funktor $M : \mathcal{K} \to \mathcal{K}$ heißt **Monade**, wenn es zwei natürliche Transformationen $\eta : Id_{\mathcal{K}} \to M$ (**Einheit**) und $\mu : MM \to M$ (**Multiplikation**) gibt, die für alle $A \in \mathcal{K}$ das folgende Diagramm kommutativ machen:





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:

$$\begin{array}{rcl} \eta_A : A & \to & A^* \\ a & \mapsto & a \end{array} \qquad \begin{array}{rcl} \mu_A : (A^*)^* & \to & A^* \\ (w_1, \ldots, w_n) & \mapsto & w_1 \ldots w_n \end{array}$$

- Mengenfunktor:

$$\begin{array}{rcl} \eta_A : A & \to & \mathcal{P}(A) \\ a & \mapsto & \{a\} \end{array} \qquad \begin{array}{rcl} \mu_A : \mathcal{P}(\mathcal{P}(A)) & \to & \mathcal{P}(A) \\ S & \mapsto & \bigcup S \end{array}$$

- Ausnahmefunktor:

$$\begin{array}{rcl} \eta_A : A & \to & A + E \\ a & \mapsto & (a, 1) \end{array} \qquad \begin{array}{rcl} \mu_A : (A + E) + E & \to & A + E \\ ((a, 1), 1) & \mapsto & (a, 1) \\ ((e, 2), 1) & \mapsto & (e, 2) \\ (e, 2) & \mapsto & (e, 2) \end{array}$$

- Leserfunktor:

$$\begin{array}{rcl} \eta_A : A & \to & A^S \\ a & \mapsto & \lambda s.a \end{array} \qquad \begin{array}{rcl} \mu_A : (A^S)^S & \to & A^S \\ f & \mapsto & \lambda s.f(s)(s) \end{array}$$

- Schreiberfunktor: Hier muss S die Trägermenge eines Monoids $M = (S, *, e)$ sein.

$$\begin{array}{rcl} \eta_A : A & \to & A \times S \\ a & \mapsto & (a, e) \end{array} \qquad \begin{array}{rcl} \mu_A : (A \times S) \times S & \to & A \times S \\ ((a, s), s') & \mapsto & (a, s * s') \end{array}$$

- Zustandsfunktor:

$$\begin{array}{rcl} \eta_A : A & \to & (A \times S)^S \\ a & \mapsto & \lambda s.(a, s) \end{array} \qquad \begin{array}{rcl} \mu_A : ((A \times S)^S \times S)^S & \to & (A \times S)^S \\ f & \mapsto & (\lambda(h, s).h(s)) \circ f \end{array}$$

Aufgabe Zeigen Sie, dass für diese Beispiele (4)-(6) gilt. ❏

Seien A und B Mengen. Der **bind-Operator**

$$\gg= \ : \ M(A) \times (A \to M(B)) \ \to \ M(B)$$

wird wie folgt aus M und der Multiplikation von M abgeleitet:

Für alle $A \in \mathcal{K}$ und $f : A \to M(B)$,

$$(\gg= f) \ = \ \mu_B \circ M(f). \tag{7}$$

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 $m \gg= f$ 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 Zustandsmonade begründen.

Aus der Multiplikation der Zustandsmonade und der allgemeinen Definition des bind-Operators ergibt sich nämlich die folgende Charakterisierung des bind-Operators der Zustandsmonade:

Für alle $g : S \to A \times S$ und $f : A \to (S \to B \times S)$,

$$g \gg\!= f = \mu_B(M(f)(g)) = \mu_B((f \times S)^S(g)) = \mu_B((\lambda(a,s).(f(a),s)) \circ g)$$
$$= (\lambda(h,s).h(s)) \circ (\lambda(a,s).(f(a),s)) \circ g = (\lambda(a,s).f(a)(s)) \circ g.$$

Die Anwendung der Funktion $(g \gg\!= f) : S \to B \times S$ 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, $a \in A$, $m \in M(A)$, $m' \in M(M(A))$, $f : A \to M(B)$, $g : B \to M(C)$ und $h : A \to B$. 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

$$\oplus \; : \; M \times M \; \rightarrow \; M$$

definiert werden kann, wobei $M \times M =_{def} \_ \times \_ \circ \Delta \circ M$. Damit lassen sich u.a. Backtracking und Nichtdeterminismus monadischer Compiler realisieren. Wir werden $\oplus$ deshalb in die Definition einer Compilermonade aufnehmen (s.u.).

Sei S eine Menge und M eine Plusmonade mit Einheit η, Multiplikation μ und paralleler Komposition $\oplus$. Die Einbettung von M in die Leser- oder Zustandsmonade liefert eine weitere Plusmonade mit Einheit η', Multiplikation μ' und paralleler Komposition $\oplus'$.

- Lesermonade über $(M, \oplus)$: Sei $h : A \rightarrow B$.

$$\begin{aligned} M(h)^S : M(A)^S &\rightarrow M(B)^S \\ f &\mapsto M(h) \circ f \\ \eta'_A : A &\rightarrow M'(A) \\ a &\mapsto \lambda s.\eta_A(a) \\ \mu'_A : M(M(A)^S)^S &\rightarrow M(A)^S \\ f &\mapsto \lambda s.f(s) \gg\!= \lambda g.g(s) \\ \oplus' : M(A)^S \times M(A)^S &\rightarrow M(A)^S \\ (f, g) &\mapsto \lambda s.f(s) \oplus g(s) \end{aligned}$$

- Zustandsmonade über $(M, \oplus)$: Sei $h : A \to B$.

$$\begin{aligned}
M(h \times S)^S : M(A \times S)^S &\to M(B \times S)^S \\
f &\mapsto \lambda s.f(s) \gg\!= \lambda(a, s).\eta(h(a), s) \\
\eta'_A : A &\to M(A \times S)^S \\
a &\mapsto \lambda s.\eta_{A\times S}(a, s) \\
\mu'_A : M((M(A \times S)^S) \times S)^S &\to M(A \times S)^S \\
f &\mapsto \lambda s.f(s) \gg\!= \lambda(g, s).g(s) \\
\oplus' : M(A \times S)^S \times M(A \times S)^S &\to M(A \times S)^S \\
(f, g) &\mapsto \lambda s.f(s) \oplus g(s)
\end{aligned}$$

Monadische Compiler des Typs $M(\_ \times S)^S$ werden in Kapitel 6 behandelt und in Kapitel 14 implementiert. Dort ist S die Menge X^* der zu verarbeitenden Wörter.

Zunächst müssen weitere Anforderungen an M gestellt werden.

5.2 Compilermonaden

Sei $M : Set^S \to Set^S$ eine Monade mit Einheit η, bind-Operator $\gg\!=$ und paralleler Komposition $\oplus$, $set : M \to \mathcal{P}$ eine weitere natürliche Transformation und

$$E_M = \{m \in M(A) \mid A \in Set^S,\ set(m) = \emptyset\}$$

("Menge der Ausnahmewerte").

M heißt **Compilermonade**, wenn für alle Mengen A und B, $m, m', m'' \in M(A)$, $e \in E_M$, $f : A \to M(B)$, $h : A \to B$ und $a \in A$ Folgendes gilt:

$$\begin{aligned}
(m \oplus m') \oplus m'' &= m \oplus (m' \oplus m''), \\
M(h)(e) &= e, && (CM1)\\
M(h)(m \oplus m') &= M(h)(m) \oplus M(h)(m'), && (CM2)\\
set_A(m \oplus m') &\subseteq set_A(m) \cup set_A(m'), && (CM3)\\
set_A(\eta_A(a)) &= \{a\}, \\
set_B(m \gg\!= f) &= \bigcup\{set_B(f(a)) \mid a \in set_A(m)\}.
\end{aligned}$$

Nach Definition der Einheit $\eta^{\mathcal{P}}$ und des bind-Operators $\gg\!=^{\mathcal{P}}$ der Mengenmonade $\mathcal{P}$ (s.o.) machen die letzten beiden Gleichungen *set* zum Monadenmorphismus, d.h. *set* ist mit den Einheiten und den bind-Operatoren von M bzw. $\mathcal{P}$ verträglich:

$$\begin{aligned} set_A \circ \eta_A &= \eta_A^{\mathcal{P}}, \\ set_B(m \gg\!= f) &= set_A(m) \gg\!=^{\mathcal{P}} set_B \circ f. \end{aligned}$$

Außerdem erhält man für alle S-sortigen Mengen A, S-sortigen Funktionen $h : A \to B$ und $m \in M(A)$:

$$set_B(M(h)(m)) = set_B(m \gg\!= \eta_B \circ h) = \bigcup\{set_B(\eta_B(h(a)) \mid a \in set_A(m)\}$$
$$= \bigcup\{\{h(a)\} \mid a \in set_A(m)\} = h(set_A(m)).$$

Satz 5.3 Listen-, Mengen- und Ausnahmefunktoren sind Compilermonaden.

Beweis. Seien A und E disjunkte Mengen.

Sei $\oplus = \cdot$ und $set_A : A^* \to \mathcal{P}(A)$ definiert durch $set_A(\epsilon) = \emptyset$ und $set_A(s) = \{a_1, \ldots, a_n\}$ für alle $s = (a_1, \ldots, a_n) \in A^+$. Damit ist $\_^*$ eine Compilermonade.

Sei $\oplus = \cup$ und $set_A : \mathcal{P}(A) \to \mathcal{P}(A)$ definiert durch $id_{\mathcal{P}(A)}$. Damit ist $\mathcal{P}$ eine Compilermonade.

Seien $\oplus_A : (A + E) \times (A + E) \to A + E$ und $set_A : A + E \to \mathcal{P}(A)$ definiert durch $\oplus_A((a, 1), x) = (a, 1)$ und $\oplus_A((e, 2), x) = x$ bzw. $set_A(a, 1) = \{a\}$ und $set_A(e, 2) = \emptyset$ für alle $a \in A$, $e \in E$ und $x \in A + E$. Damit ist $\lambda A.A + E$ eine Compilermonade. ❑

5.4 Monadenbasierte Parser und Compiler

Sei $G = (S, Z, BT, R)$ eine LL-kompilierbare CFG, G' die daraus gebildete nicht-linksrekursive CFG, $X = Z \cup \bigcup BT$, $\Sigma(G) = (S, F)$ und $M : Set^S \to Set^S$ eine Compilermonade.

Wie bereits in Kapitel 1 informell beschrieben wurde, kombiniert 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. Da der Parser der Compiler-Instanz, die Syntaxbäume erzeugt, entsprechen soll, nennen wir

$$parse_G = compile_G^{T_{\Sigma(G)}} \tag{12}$$

den **Parser von** $compile_G$ und gelangen zur Instanz von $compile_G$ für eine beliebige Zielalgebra $\mathcal{A}$, indem wir $parse_G$ mit $M(fold^{\mathcal{A}})$ komponieren:

$$compile_G^{\mathcal{A}} \;=\; X^* \overset{parse_G}{\longrightarrow} M(T_{\Sigma(G)}) \overset{M(fold^{\mathcal{A}})}{\longrightarrow} M(A). \tag{13}$$

Ist G linksrekursiv, dann wird die Implementierung von $parse_G$ als LL-Compiler (siehe Kapitel 6) nicht immer terminieren. Deshalb fordern wir in diesem Fall

$$compile_G^{\mathcal{A}} \;=\; X^* \overset{parse_{G'}}{\longrightarrow} M(T_{\Sigma(G')}) \overset{M(fold^{derec(\mathcal{A})})}{\longrightarrow} M(A)$$

anstelle von (13).

Da $T_{\Sigma(G)}$ eine initiale $\Sigma(G)$-Algebra ist, stimmt $fold^{T_{\Sigma(G)}}$ mit der Identität auf $T_{\Sigma(G)}$ überein. Deshalb gilt (13) für $\mathcal{A} = T_{\Sigma(G)}$:

$$M(fold^{T_{\Sigma(G)}}) \circ parse_G = M(id_{T_{\Sigma(G)}}) \circ parse_G \overset{M \text{ ist Funktor}}{=} id_{M(T_{\Sigma(G)})} \circ parse_G$$
$$= parse_G = compile_G^{T_{\Sigma(G)}}.$$

Aus (13) und der Kommutativität von (1) folgt die Kommutativität des folgenden Diagramms:

$$\begin{array}{ccc} X^* & \xrightarrow{compile_G^{\mathcal{A}}} & M(A) \\ {\scriptstyle compile_G^{Sem}}\downarrow & & \downarrow{\scriptstyle M(evaluate)} \\ M(Sem) & \xrightarrow[M(encode)]{} & M(Mach) \end{array}$$

Beweis.

$$M(evaluate) \circ compile_G^{\mathcal{A}} \overset{(13)}{=} M(evaluate) \circ M(fold^{\mathcal{A}}) \circ parse_G$$
$$\overset{M \text{ ist Funktor}}{=} M(evaluate \circ fold^{\mathcal{A}}) \circ parse_G \overset{(1)}{=} M(encode \circ fold^{Sem}) \circ parse_G$$
$$\overset{M \text{ ist Funktor}}{=} M(encode) \circ M(fold^{Sem}) \circ parse_G \overset{(13)}{=} M(encode) \circ compile_G^{Sem}. \quad \square$$

(13) ist ein Spezialfall folgender Bedingung: Für alle $\Sigma(G)$-Homomorphismen $h : \mathcal{B} \to \mathcal{A}$,

$$compile_G^{\mathcal{A}} = M(h) \circ compile_G^{\mathcal{B}}. \tag{14}$$

Im Fall $\mathcal{B} = T_{\Sigma(G)}$ und $h = fold^{\mathcal{A}} : T_{\Sigma(G)} \to \mathcal{A}$ folgt nämlich

$$compile_G^{\mathcal{A}} = M(fold^{\mathcal{A}}) \circ compile_G^{T_{\Sigma(G)}} \overset{(12)}{=} M(fold^{\mathcal{A}}) \circ parse_G$$

aus (14).

Wir werden $compile_G^{\mathcal{A}}$ im folgenden Kapitel für LL-Compiler direkt definieren und nicht als Komposition $M(fold^{\mathcal{A}}) \circ parse_G$ von Parser und Baumentfaltung. Dabei wenden wir implizit die Methode des **Deforestation** an (siehe z.B. [1], Abschnitt 6.6.1).

(14) gilt übrigens genau dann, wenn $compile_G : X^* \to MU$ eine natürliche Transformation ist (s.o.), wobei

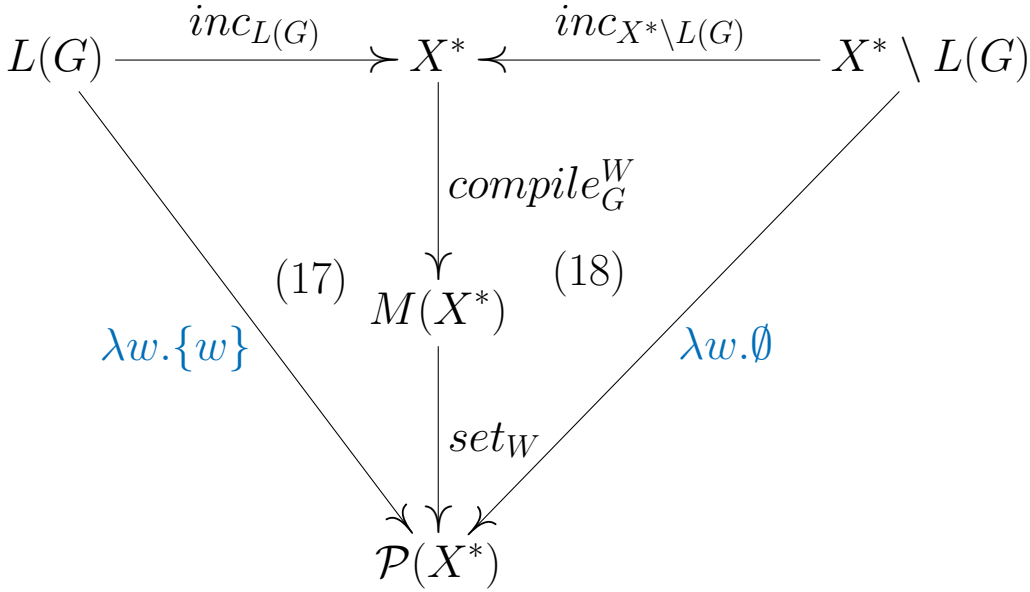
- X^* als Funktor von $Alg_{\Sigma(G)}$ nach Set jeder $\Sigma(G)$-Algebra die Menge X^* und jedem $\Sigma(G)$-Homomorphismus die Funktion id_{X^*} zuordnet,
- der *Vergissfunktor* $U : Alg_{\Sigma(G)} \to Set$ jede $\Sigma(G)$-Algebra auf die Vereinigung ihrer Trägermengen und jeden $\Sigma(G)$-Homomorphismus auf die ihm zugrundeliegende Funktion abbildet.

Sei $W = Word(G)$. $parse_G$ ist **korrekt**, wenn für alle $w \in X^*$ Folgendes gilt:

- $set_W(compile_G^W(w)) \subseteq \{w\}$, (15)
- $compile_G^W(w)) \notin E_M \Rightarrow w \in L(G)$, (16)

$parse_G$ ist **vollständig**, wenn für alle $w \in L(G)$ $compile_G^W(w)$ nicht zu E_M gehört.

Offenbar ist $parse_G$ genau dann korrekt und vollständig, wenn die beiden Dreiecke des folgenden Diagramms kommutieren, wenn also $set_W \circ compile_G^W$ die Summenextension von $(\lambda w.\{w\}, \lambda w.\emptyset)$ ist (siehe Kapitel 2):



Aus (17) folgt, dass $parse_G$ auf der Sprache von G rechtsinvers zur Faltung in der Wortalgebra von G ist:

$$set_W \circ M(fold^W) \circ parse_G \circ inc_{L(G)} \overset{(13)}{=} set_W \circ compile_G^W \circ inc_{L(G)} = \lambda w.\{w\}.$$

Zusammenfassend ergibt sich folgende Definition:

Ein **generischer Compiler für** G ist eine natürliche Transformation

$$compile_G : X^* \to MU$$

derart, dass $parse_G = compile_G^{T_{\Sigma(G)}}$ korrekt und vollständig ist.

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 $G = (S, Z, BT, R)$ eine nicht-linksrekursive CFG, $X = Z \cup \bigcup BT$, M eine Compilermonade, $errmsg : X^* \to E_M$ und $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$-Algebra. $compile_G$ heißt **LL-Compiler**, wenn

$$(compile^{\mathcal{A}}_{G,s} : X^* \to M(A_s))_{s \in S}$$

wie folgt definiert ist: Für alle $s \in S$ und $w \in X^*$,

$$compile^{\mathcal{A}}_{G,s}(w) \;=\; trans^{\mathcal{A}}_s(w) \gg\!= \lambda(a, v).\mathit{if}\ v = \epsilon\ \mathit{then}\ \eta_A(a)\ \mathit{else}\ errmsg(v), \qquad (1)$$

wobei für alle für alle $s \in S \cup Z \cup BT$

$$trans^{\mathcal{A}}_s : X^* \to M(A_s \times X^*)$$

wie folgt definiert ist:

Fall 1: $s \in Z \cup BT$. Für alle $x \in X$ und $w \in X^*$,

$$\begin{aligned} trans_s^{\mathcal{A}}(xw) &= \mathit{if}\ x \in s\ \mathit{then}\ \eta_{A \times X^*}(x, w)\ \mathit{else}\ errmsg(xw) \\ trans_s^{\mathcal{A}}(\epsilon) &= errmsg(\epsilon) \end{aligned}$$

Im Fall $s \in Z$ steht $x \in s$ für $x = s$. Im Fall $s \in BT$ erfordert die Berechenbarkeit von $trans_s^{\mathcal{A}}$ einen Erkenner für s (siehe Bemerkung am Anfang von Kapitel 4).

Fall 2: $s \in S$. Für alle $w \in X^*$,

$$trans_s^{\mathcal{A}}(w) = \bigoplus_{r=(s \to e) \in R} try_r^{\mathcal{A}}(w). \tag{2}$$

Für alle $r = (s \to (e_1, \ldots, e_n)) \in R$ und $w \in X^*$,

$$try_r^{\mathcal{A}}(w) = \begin{cases} trans_{e_1}^{\mathcal{A}}(w) \gg\!= \lambda(a_1, w_1). \\ trans_{e_2}^{\mathcal{A}}(w_1) \gg\!= \lambda(a_2, w_2). \\ \qquad \vdots \\ trans_{e_n}^{\mathcal{A}}(w_{n-1}) \gg\!= \lambda(a_n, w_n).\eta_{A \times X^*}(f_r^{\mathcal{A}}(a_{i_1}, \ldots, a_{i_k}), w_n), \end{cases} \tag{3}$$

wobei $\{i_1, \ldots, i_k\} = \{1 \leq i \leq n \mid e_i \in S \cup BT\}$.

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 $trans_{e_j}^{\mathcal{A}}(w)$ scheitert, d.h. existiert kein aus e_j ableitbares Präfix von w, dann übergibt $try_r^{\mathcal{A}}$ das *gesamte* Eingabewort zur Parsierung an den nächsten Teilcompiler $try_{r'}^{\mathcal{A}}$ der Argumentliste von $\oplus$ in (2) (**Backtracking**).

Ist $trans_{e_i}^{\mathcal{A}}(w)$ jedoch für alle $1 \leq i \leq n$ erfolgreich, dann schließt der Aufruf von $try_r^{\mathcal{A}}$ mit der Anwendung von $f_r^{\mathcal{A}}$ auf die Zwischenergebnisse $a_{i_1}, \ldots, a_{i_k} \in A$.

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 $try_{s \to e}^{\mathcal{A}}$ 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 $m' \in M(B)$.

$$\begin{aligned} do\ a \leftarrow m;\ m' &= m \gg\!= \lambda a.\ do\ m' \\ do\ m;\ m' &= m \gg do\ m' \end{aligned}$$

Gleichung (10) von Kapitel 5 impliziert die Assoziativität des Sequentialisierungsoperators (;), d.h. $do\ m; m'; m''$ ist gleichbedeutend mit $do\ (do\ m; m'); m''$ und $do\ m; (do\ m'; m'')$.

Beispiel 6.1 SAB (siehe Beispiel 4.5)

Die Regeln

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

von SAB liefern nach obigem Schema die folgenden Transitionsfunktionen des LL-Compilers für SAB:

Für alle SAB-Algebren alg, $x, z \in \{a, b\}$ und $w \in \{a, b\}^*$,

```
trans_z^alg(xw) = if x = z then η(z,w) else errmsg(xw)
trans_z^alg(ϵ)  = errmsg(ϵ)
```

```
trans_S^alg(w)  = try_r1(w) ⊕ try_r2(w) ⊕ try_r3(w)
trans_A^alg(w)  = try_r4(w) ⊕ try_r5(w)
trans_B^alg(w)  = try_r6(w) ⊕ try_r7(w)

try_r1(w)  = do (x,w) <- trans_a^alg(w);
                (c,w) <- trans_B^alg(w);  η(f_r1^alg(c),w)
try_r2(w)  = do (x,w) <- trans_b^alg(w);
                (c,w) <- trans_A^alg(w);  η(f_r2^alg(c),w)
try_r3(w)  = η(f_r3^alg,w)

try_r4(w)  = do (x,w) <- trans_a^alg(w);
                (c,w) <- trans_S^alg(w);  η(f_r4^alg(c),w)
try_r5(w)  = do (x,w) <- trans_b^alg(w);
                (c,w) <- trans_A^alg(w);
                (d,w) <- trans_A^alg(w);  η(f_r5^alg(c,d),w)

try_r6(w)  = do (x,w) <- trans_b^alg(w);
                (c,w) <- trans_S^alg(w);  η(f_r6^alg(c),w)
try_r7(w)  = do (x,w) <- trans_a^alg(w);
                (c,w) <- trans_B^alg(w);
                (d,w) <- trans_B^alg(w);  η(f_r7^alg(c,d),w)
```

❑

Satz 6.4 $trans^{\mathcal{A}}$ ist wohldefiniert.

Beweis durch Noethersche Induktion. Da S endlich und G nicht linksrekursiv ist, ist die folgende Ordnung $>$ auf $S \cup Z \cup BT$ Noethersch, d.h., es gibt keine unendlichen Ketten $s_1 > s_2 > s_3 > \cdots$:

$$s > s' \quad \Leftrightarrow_{def} \quad \exists\, w \in (S \cup Z \cup BT)^* : s \xrightarrow{+}_G s'w.$$

Wir erweitern $>$ zu einer ebenfalls Noetherschen Ordnung $\succ$ auf $X^* \times (S \cup Z \cup BT)$:

$$(v, s) \succ (w, s') \quad \Leftrightarrow_{def} \quad |v| > |w| \text{ oder } (|v| = |w| \text{ und } s > s').$$

Nach Definition von $trans_s(w)$ gibt es $r = (s \to e_1 \dots e_n) \in R$ mit

$$trans_s^{\mathcal{A}}(w) \;=\; \dots trans_{e_1}^{\mathcal{A}}(w) \dots trans_{e_2}^{\mathcal{A}}(w_1) \dots trans_{e_n}^{\mathcal{A}}(w_{n-1}) \dots$$

Da $w_1, \dots, w_n$ echte Suffixe von w sind, gilt $s > e_1$ und $|w| > |w_i|$ für alle $1 \le i < n$, also $(w, s) \succ (w, e_1)$ und $(w, s) \succ (w_{i-1}, e_i)$ für alle $1 < i \le n$. Die rekursiven Aufrufe von $trans^{\mathcal{A}}$ haben also bzgl. $\succ$ kleinere Argumente.

Folglich terminiert jeder Aufruf von $trans^{\mathcal{A}}$, m.a.W.: $trans^{\mathcal{A}}$ ist wohldefiniert. ❑

Der Beweis, dass $compile_G$ Gleichung (13) von Kapitel 5 erfüllt, verwendet die folgenden Zusammenhänge zwischen einer Monade M und ihrem bind-Operator:

Lemma 6.5

Sei $M : Set^S \to Set^S$ eine Monade und $h : A \to B$ eine S-sortige Funktion. Für alle $m \in M(A)$, $f : A \to M(A)$ und $g : B \to M(B)$,

$$M(h)(m \gg\!= f) \;=\; m \gg\!= M(h) \circ f, \tag{4}$$

$$M(h)(m) \gg\!= g \;=\; m \gg\!= g \circ h. \tag{5}$$

Sei Σ eine Signatur, $m_1, \ldots, m_n \in M(A)$, $h : A \to B$ Σ-homomorph, für alle $1 \leq i \leq n$,

$$f_i : A \to \cdots \to A \to M(A), \quad g_i : B \to \cdots \to B \to M(B),$$

und $t \in T_\Sigma(V)$ mit $var(t) = \{x_1, \ldots, x_n\}$,

$$\begin{aligned} f_i(a_1) \ldots (a_{i-1}) &= m_i \gg f_{i+1}(a_1) \ldots (a_{i-1}), \\ f_{n+1}(a_1) \ldots (a_n) &= \eta_A(f_a^*(t)) \end{aligned}$$

für alle $a = (a_1, \ldots, a_n) \in A$ und

$$\begin{aligned} g_i(b_1) \ldots (b_{i-1}) &= M(h)(m_i) \gg g_{i+1}(b_1) \ldots (b_{i-1}), \\ g_{n+1}(b_1) \ldots (b_n) &= \eta_B(g_b^*(t)) \end{aligned}$$

für alle $b = (b_1, \ldots, b_n) \in B$, wobei $f_a : V \to A$ und $g_b : V \to B$ 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).

$$M(h)(m \gg\!= f) \overset{(11)\ \text{in Kap. 5}}{=} (m \gg\!= f) \gg\!= \eta_B \circ h$$
$$\overset{(10)\ \text{in Kap. 5}}{=} m \gg\!= \lambda a.f(a) \gg\!= \eta_B \circ h$$
$$\overset{(11)\ \text{in Kap. 5}}{=} m \gg\!= \lambda a.M(h)(f(a)) \;=\; m \gg\!= M(h) \circ f.$$

Beweis von (5).

$$M(h)(m) \gg\!= g \;=\; (m \gg\!= \eta_B \circ h) \gg\!= g \overset{(10)\ \text{in Kap. 5}}{=} m \gg\!= \lambda a.\eta_B(h(a)) \gg\!= g$$
$$\overset{(9)\ \text{in Kap. 5}}{=} m \gg\!= \lambda a.g(h(a)) \;=\; m \gg\!= g \circ h.$$

Beweis von (6).

$$M(h)(f_1) = M(h)(m_1 \gg f_2) \overset{(4)}{=} m_1 \gg\!= M(h) \circ f_2 = m_1 \gg\!= \lambda a_1.M(h)(f_2(a_1))$$
$$= m_1 \gg\!= \lambda a_1.M(h)(m_2 \gg\!= f_3(a_1)) \overset{(4)}{=} m_1 \gg\!= \lambda a_1.m_2 \gg\!= M(h) \circ f_3(a_1)$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \lambda a_2.M(h)(f_3(a_1)(a_2))$$
$$= \cdots = m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.M(h)(f_{n+1}(a_1)\ldots(a_n))$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.M(h)(\eta_A(f_a^*(t)))$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.\eta_B(h(f_a^*(t)))$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.\eta_B((h \circ f_a)^*(t))$$

$$\overset{h\circ f_a = g_{h(a)}}{=} m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.\eta_B(g^*_{h(a)}(t)) \qquad (7)$$

$$g_1 = M(h)(m_1) \gg\!= g_2 \overset{(5)}{=} m_1 \gg\!= g_2 \circ h = m_1 \gg\!= \lambda a_1.g_2(h(a_1))$$
$$= m_1 \gg\!= \lambda a_1.M(h)(m_2) \gg\!= g_3(h(a_1)) = m_1 \gg\!= \lambda a_1.m_2 \gg\!= g_3(h(a_1)) \circ h$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \lambda a_2.g_3(h(a_1))(h(a_2))$$
$$= \cdots = m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.g_{n+1}(h(a_1)) \ldots, (h(a_n))$$
$$= m_1 \gg\!= \lambda a_1.m_2 \gg\!= \cdots \gg\!= \lambda a_n.\eta_B(g^*_{h(a)}(t)) \qquad (8)$$

Aus (7) und (8) folgt (6). ❑

Satz 6.6

Sei $h : \mathcal{A} \to \mathcal{B} = (B, Op')$ ein $\Sigma(G)$-Homomorphismus. Der oben definierte LL-Compiler ist mit Homomorphismen verträglich, d.h.

$$compile^{\mathcal{B}}_G \;=\; M(h) \circ compile^{\mathcal{A}}_G. \qquad (9)$$

Beweis. Sei

$$trans^{\mathcal{B}} \;=\; M(h \times id) \circ trans^{\mathcal{A}}. \qquad (10)$$

Dann gilt (9):

Für alle $w \in X^*$,

$$
\begin{aligned}
compile_G^{\mathcal{B}}(w) &\overset{(1)}{=} trans^{\mathcal{B}}(w) \gg\!= \lambda(b,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(b)\ \mathit{else}\ errmsg(w) \\
&\overset{(10)}{=} M(h \times id)(trans^{\mathcal{A}}(w)) \gg\!= \lambda(b,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(b)\ \mathit{else}\ errmsg(w) \\
&\overset{(11)\ in\ Kap.\ 5}{=} (trans^{\mathcal{A}}(w) \gg\!= (\eta_{B\times X^*} \circ (h \times id))) \\
&\qquad \gg\!= \lambda(b,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(b)\ \mathit{else}\ errmsg(w) \\
&= (trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\eta_{B\times X^*}(h(a),w)) \\
&\quad \gg\!= \lambda(b,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(b)\ \mathit{else}\ errmsg(w) \\
&\overset{(10)\ in\ Kap.\ 5}{=} trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\eta_{B\times X^*}(h(a),w) \\
&\qquad \gg\!= \lambda(b,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(b)\ \mathit{else}\ errmsg(w) \\
&\overset{(9)\ in\ Kap.\ 5}{=} trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(h(a))\ \mathit{else}\ errmsg(w) \\
&\overset{(CM1)}{=} trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_B(h(a))\ \mathit{else}\ M(h)(errmsg(w)) \\
&\overset{(9),(11)\ in\ Kap.\ 5}{=} trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_A(a) \gg\!= \eta_B \circ h \\
&\qquad \mathit{else}\ errmsg(w) \gg\!= \eta_B \circ h \\
&= trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).(\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_A(a)\ \mathit{else}\ errmsg(w)) \gg\!= \eta_B \circ h \\
&\overset{(10)\ in\ Kap.\ 5}{=} (trans^{\mathcal{A}}(w) \gg\!= \lambda(a,w).\mathit{if}\ w = \epsilon\ \mathit{then}\ \eta_A(a)\ \mathit{else}\ errmsg(w)) \gg\!= \eta_B \circ h
\end{aligned}
$$

$$= compile^{\mathcal{A}}_G(w) \gg\!= \eta_B \circ h \overset{(11)\ in\ Kap.\ 5}{=} M(h)(compile^{\mathcal{A}}_G(w)).$$

Beweis von (10) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung $\succ$:

Fall 1: $s \in Z \cup BT$. Für alle $x \in X$ und $w \in X^*$,

$$M(h \times id)(trans^{\mathcal{A}}_s(xw)) = if\ x \in s\ then\ M(h \times id)(\eta_{A \times X^*}(x, w))$$
$$else\ M(h \times id)(errmsg(xw))$$
$$\overset{\eta\ ist\ nat.\ Transformation,\ (CM1)}{=} if\ x \in s\ then\ \eta_{B \times X^*}((h \times id)(x, w))\ else\ errmsg(xw)$$
$$\overset{(h \times id)(x,w) = (h(x),w) = (x,w)}{=} if\ x \in s\ then\ \eta_{B \times X^*}(x, w)\ else\ errmsg(xw) = trans^{\mathcal{B}}_s(xw),$$
$$M(h \times id)(trans^{\mathcal{A}}_s(\epsilon)) = M(h \times id)(errmsg(\epsilon)) \overset{(CM1)}{=} errmsg(\epsilon) = trans^{\mathcal{B}}_s(\epsilon).$$

Fall 2: $s \in S$. Sei $r = (s \to w') \in R$. Für alle $w \in X^*$,

$$M(h \times id)(try^{\mathcal{A}}_r(w))$$
$$\overset{(3)}{=} M(h \times id)\left(\left\{\begin{array}{l} trans^{\mathcal{A}}_{e_1}(w) \gg\!= \lambda(a_1, w_1). \\ \vdots \\ trans^{\mathcal{A}}_{e_n}(w_{n-1}) \gg\!= \lambda(a_n, w_n).\eta_{A \times X^*}(f^{\mathcal{A}}_r(a_{j_1}, \ldots, a_{j_k}), w_n) \end{array}\right\}\right)$$

$$
\begin{aligned}
&= M(h \times id)(\left\{ \begin{array}{l} trans^{\mathcal{A}}_{e_1}(w) \gg\!= \lambda x_1. \\ \vdots \\ trans^{\mathcal{A}}_{e_n}(w_{n-1}) \gg\!= \lambda x_n.\eta_{A\times X^*}(f^{\mathcal{A}}_r(\pi_1(x_{j_1}),\ldots,\pi_1(x_{j_k})),\pi_2(x_n)) \end{array} \right\}) \\
&\overset{6.5\ (6)}{=} \left\{ \begin{array}{l} M(h \times id)(trans^{\mathcal{A}}_{e_1}(w)) \gg\!= \lambda x_1. \\ \vdots \\ M(h \times id)(trans^{\mathcal{A}}_{e_n}(w_{n-1})) \\ \qquad \gg\!= \lambda x_n.\eta_{B\times X^*}(f_r^{derec(\mathcal{B})}(\pi_1(x_{j_1}),\ldots,\pi_1(x_{j_k})),\pi_2(x_n)) \end{array} \right\}) \\
&= \left\{ \begin{array}{l} M(h \times id)(trans^{\mathcal{A}}_{e_1}(w)) \gg\!= \lambda(b_1,w_1). \\ \vdots \\ M(h \times id)(trans^{\mathcal{A}}_{e_n}(w_{n-1})) \gg\!= \lambda(b_n,w_n).\eta_{B\times X^*}(f_r^{derec(\mathcal{B})}(b_{j_1},\ldots,b_{j_k}),w_n) \end{array} \right\} \\
&\overset{ind.\ hyp.}{=} \left\{ \begin{array}{l} trans^{\mathcal{B}}_{e_1}(w) \gg\!= \lambda(b_1,w_1). \\ \vdots \\ trans^{\mathcal{B}}_{e_n}(w_{n-1}) \gg\!= \lambda(b_n,w_n).\eta_{B\times X^*}(f_r^{derec(\mathcal{B})}(b_{j_1},\ldots,b_{j_k}),w_n) \end{array} \right\} \\
&\overset{(3)}{=} try^{\mathcal{B}}_r(w). \qquad (11)
\end{aligned}
$$

Daraus folgt (10):

$$M(h \times id)(trans_s^{\mathcal{A}}(w)) \overset{(2)}{=} M(h \times id)(\oplus_{r=(s\to e)\in R} try_r^{\mathcal{A}}(w))$$
$$\overset{(CM2)}{=} \oplus_{r=(s\to e)\in R} M(h \times id)(try_r^{\mathcal{A}}(w)) \overset{(11)}{=} \oplus_{r=(s\to e)\in R} try_r^{\mathcal{B}}(w) \overset{(2)}{=} trans_s^{\mathcal{B}}(w). \quad \square$$

Lemma 6.7

Sei $M : Set^S \to Set^S$ eine Compilermonade, $f : A^n \to A$ und $m_1, \ldots, m_n \in M(A)$.

$$set_A(m_1 \gg\!= \lambda a_1 \ldots m_n \gg\!= \lambda a_n.\eta_A(f(a_1, \ldots, a_n)))$$
$$= \{f(a_1, \ldots, a_n) \mid \textstyle\bigwedge_{i=1}^n a_i \in set_A(m_i)\}.$$

Beweis.

$$set_A(m_1 \gg\!= \lambda a_1 \ldots m_n \gg\!= \lambda a_n.\eta_A(f(a_1, \ldots, a_n)))$$
$$= \bigcup\{set_A(m_2 \gg\!= \lambda a_2 \ldots m_n \gg\!= \lambda a_n.\eta_A(f(a_1, \ldots, a_n))) \mid a_1 \in set_A(m_1)\}$$
$$= \bigcup\{\bigcup\{set_A(m_3 \gg\!= \lambda a_3 \ldots m_n \gg\!= \lambda a_n.\eta_A(f(a_1, \ldots, a_n))) \mid a_2 \in set_A(m_2)\}$$
$$\quad \mid a_1 \in set_A(m_1)\}$$
$$= \bigcup\{set_A(m_3 \gg\!= \lambda a_3 \ldots m_n \gg\!= \lambda a_n.\eta_A(f(a_1, \ldots, a_n)))$$
$$\quad \mid a_2 \in set_A(m_2),\ a_1 \in set_A(m_1)\}$$
$$= \ldots$$

$$= \bigcup\{set_A(\eta_A(f(a_1,\ldots,a_n))) \mid \bigwedge_{i=1}^{n} a_i \in set_A(m_i)\}$$
$$= \bigcup\{\{f(a_1,\ldots,a_n)\} \mid \bigwedge_{i=1}^{n} a_i \in set_A(m_i)\}$$
$$= \{f(a_1,\ldots,a_n) \mid \bigwedge_{i=1}^{n} a_i \in set_A(m_i)\}. \qquad \square$$

Satz 6.8

Sei $W = Word(G)$. Für den oben definierten LL-Compiler und alle $w \in X^*$ gilt:

$$set_W(compile_G^W(w)) \subseteq \{w\}. \tag{12}$$

Beweis. Für alle $w \in X^*$ und $s \in S \cup Z \cup BT$ gelte folgende Bedingung:

$$\forall\ (v,v') \in set_{W^2}(trans_s^W(w)) : vv' = w. \tag{13}$$

Dann gilt (12):

$$set_W(compile_{G,s}^W(w))$$
$$\overset{(1)}{=} set_W(trans_s^W(w) \gg\!= \lambda(v,v').\, if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v'))$$
$$= \bigcup\{set_W(if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v')) \mid (v,v') \in set_{W^2}(trans_s^W(w))\}$$
$$\overset{(13)}{\subseteq} \bigcup\{set_W(if\ v' = \epsilon\ then\ \eta_W(v)\ else\ errmsg(v')) \mid vv' = w\}$$
$$= \bigcup\{if\ v' = \epsilon\ then\ set_W(\eta_W(v))\ else\ set_W(errmsg(v')) \mid vv' = w\}$$
$$= \bigcup\{if\ v' = \epsilon\ then\ \{v\}\ else\ \emptyset \mid vv' = w\} = \{w\}.$$

Beweis von (13) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung $\succ$.

Fall 1: $s \in Z \cup BT$. Sei $x \in X$, $w \in X^*$ und $(v, v') \in set_{W^2}(trans_s^W(xw))$. Dann gilt

$$\begin{aligned}(v, v') \;\in\; & set_{W^2}(\mathit{if}\ x \in s\ \mathit{then}\ \eta_{W^2}(x, w)\ \mathit{else}\ \mathit{errmsg}(xw)) \\ & = \mathit{if}\ x \in s\ \mathit{then}\ set_{W^2}(\eta_{W^2}(x, w))\ \mathit{else}\ set_{W^2}(\mathit{errmsg}(xw)) \\ & = \mathit{if}\ x \in s\ \mathit{then}\ \{(x, w)\}\ \mathit{else}\ \emptyset,\end{aligned}$$

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

$$\begin{aligned}(v, v') \;\in\; & set_{W^2}(trans_s^W(w))) \overset{(2)}{=} set_{W^2}(\textstyle\bigoplus_{r=(s\to e)\in R} try_r^W(w)) \\ & \overset{(CM3)}{\subseteq} \textstyle\bigcup_{r=(s\to e)\in R} set_{W^2}(try_r^W(w)).\end{aligned}$$

Also gibt es $r = (s \to e_1 \dots e_n) \in R$ mit $(v, v') \in set_{W^2}(try_r^W(w))$. Sei $w_0 = w$. Dann gilt

$$\begin{aligned}(v, v') \;\in\; & set_{W^2}(try_r^W(w_0)) \\ & \overset{(3)}{=} set_{W^2}(trans_{e_1}^W(w_0) \gg\!= \lambda(v_1, w_1). \\ & \qquad\vdots \\ & \qquad trans_{e_n}^W(w_{n-1}) \gg\!= \lambda(v_n, w_n).\eta_{W^2}(f_r^W(v_{j_1}, \dots, v_{j_k}), w_n))\end{aligned}$$

$$\overset{\text{Lemma 6.7}}{=} \{(f_r^W(v_{j_1},\ldots,v_{j_k}),w_n) \mid \bigwedge_{i=1}^n (v_i,w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\}$$
$$= \{(v_1\ldots v_n,w_n) \mid \bigwedge_{i=1}^n (v_i,w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\}. \qquad (14)$$

Daher existieren $v_1,\ldots,v_n,w_1,\ldots,w_n \in X^*$ mit $v = v_1\ldots v_n$, $v' = w_n$ und $(v_i,w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))$ für alle $1 \le i \le n$. Nach Induktionsvoraussetzung folgt $v_iw_i = w_{i-1}$, also

$$vv' = v_1\ldots v_nw_n = w_0 = w. \qquad \text{❑}$$

Satz 6.9 Sei $W = Word(G)$. Für den oben definierten LL-Compiler und alle $w \in X^*$ gilt:

$$compile_G^W(w)) \notin E_M \;\Rightarrow\; w \in L(G). \qquad (15)$$

Beweis. Für alle $s \in S \cup Z \cup BT$ und $v,w,w' \in X^*$ gelte:

$$(v,w') \in set_{W^2}(trans_s^W(w)) \;\Rightarrow\; v \in L(G)_s. \qquad (16)$$

Für alle $w \in X^*$ erhalten wir:

$$set_W(compile_{G,s}^W(w))$$
$$\overset{(1)}{=} set_W(trans_s^W(w) \gg\!= \lambda(v,v').\mathit{if}\ v' = \epsilon\ \mathit{then}\ \eta_W(v)\ \mathit{else}\ errmsg(v'))$$
$$= \bigcup\{set_W(\mathit{if}\ v' = \epsilon\ \mathit{then}\ \eta_W(v)\ \mathit{else}\ errmsg(v')) \mid (v,v') \in set_{W^2}(trans_s^W(w))\}$$

$$= \bigcup\{set_W(\eta_W(v)) \mid (v,\epsilon) \in set_{W^2}(trans_s^W(w))\} \cup$$
$$\bigcup\{set_W(errmsg(v')) \mid (v,v') \in set_{W^2}(trans_s^W(w)),\ v' \neq \epsilon\}$$
$$= \bigcup\{\{v\} \mid (v,\epsilon) \in set_{W^2}(trans_s^W(w))\} \cup \bigcup\{\emptyset \mid (v,v') \in set_{W^2}(trans_s^W(w)),\ v' \neq \epsilon\}$$
$$= \{v \mid (v,\epsilon) \in set_{W^2}(trans_s^W(w))\}. \tag{17}$$

Beweis von (15). Sei $w \in X^*$ und $compile_G^W(w)) \notin E_M$. Dann ist $set_W(compile_G^W(w))$ nichtleer. Daraus folgt

$$\{w\} \overset{(12)}{=} set_W(compile_G^W(w)) \overset{(17)}{=} \{v \mid (v,\epsilon) \in set_{W^2}(trans_s^W(w))\},$$

also $(w,\epsilon) \in set_{W^2}(trans_s^W(w))$. (16) impliziert $w \in L(G)_s$.

Beweis von (16) durch Noethersche Induktion bzgl. der im Beweis von Satz 6.4 definierten Ordnung $\succ$.

Fall 1: $s \in Z \cup BT$, $x \in X$ und $w \in X^*$. Dann gilt:

$$set_{W^2}(trans_s^W(xw)) = set_{W^2}(if\ x \in s\ then\ \eta_{W^2}(x,w)\ else\ errmsg(xw))$$
$$= if\ x \in s\ then\ set_{W^2}(\eta_{W^2}(x,w))\ else\ set_{W^2}(errmsg(xw))$$
$$= if\ x \in s\ then\ \{(x,w)\}\ else\ \emptyset. \tag{18}$$

Sei $(v,w') \in set_{W^2}(trans_s^W(xw))$.

Aus (18) folgt $(v, w') = (x, w)$ und $x \in s$, also $v = x \in s = L(G)_s$.

Fall 2: $s \in S$ und $w \in X^*$. Dann gilt

$$set_{W^2}(trans_s^W(w)) \overset{(2)}{=} set_{W^2}(\bigoplus_{r=(s\to e)\in R} try_r^W(w)) \overset{(CM3)}{\subseteq} \bigcup_{r=(s\to e)\in R} set_{W^2}(try_r^W(w)). \quad (19)$$

Sei $(v, w') \in set_{W^2}(trans_s^W(w))$. Wegen (19) gibt es $r = (s \to (e_1, \ldots, e_n)) \in R$ mit

$$\begin{aligned}
(v, w') &\in set_{W^2}(try_r^W(w)) \\
&\overset{(3)}{=} set_{W^2}(trans_{e_1}^W(w) \gg\!= \lambda(v_1, w_1). \\
&\qquad \vdots \\
&\qquad trans_{e_n}^W(w_{n-1}) \gg\!= \lambda(v_n, w_n).\eta_{W^2}(f_r^W(v_{j_1}, \ldots, v_{j_k}), w_n)) \\
&\overset{Lemma\ 6.7}{=} \{(f_r^W(v_{j_1}, \ldots, v_{j_k}), w_n) \mid \textstyle\bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\} \\
&= \{(v_1 \ldots v_n, w_n) \mid \textstyle\bigwedge_{i=1}^n (v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))\}.
\end{aligned}$$

Also gibt es $v_1, \ldots, v_n w_1, \ldots, w_n \in X^*$ mit $v = v_1 \ldots v_n$ und

$$(v_i, w_i) \in set_{W^2}(trans_{e_i}^W(w_{i-1}))$$

für alle $1 \leq i \leq n$. Nach Induktionsvoraussetzung gilt $v_i \in L(G)_{e_i}$, also $v = v_1 \ldots v_n \in L(G)_s$ wegen $s \to (e_1, \ldots, e_n) \in R$. ❑

Korollar 6.10

Sei $G = (S, Z, BT, R)$ eine nicht-linksrekursive CFG und $W = Word(G)$. Der oben definierte LL-Compiler ist ein generischer Compiler für G, falls $parse_G$ ist vollständig ist, falls also für alle $w \in L(G)$ $compile_G^W(w)$ nicht zu E_M gehört.

Beweis. Sätze 6.6, 6.8 und 6.9. ❏

Ob $parse_G$ ist vollständig ist oder nicht, hängt u.a. davon ab, welche Regeln von G die parallele Komposition $\oplus$ von *try*-Funktionsaufrufen berücksichtigt (siehe (2)). Da $\oplus$ nicht kommutativ ist, kann die Reihenfolge, in der sie ausgeführt werden, entscheidend sein. Manchmal lässt sich auch durch eine andere Reihenfolge $parse_G$ nicht vollständig machen. Wie das folgende Beispiel zeigt, kann dann evtl. eine Änderung der Grammatik helfen:

Beispiel 6.2 Eine solche Änderung führt z.B. zu folgenden Regeln von JavaLight+ (siehe Kapitel 11):

$$
\begin{array}{lcl}
Command & \rightarrow & String = ExpSemi \mid write\ ExpSemi \mid \ldots \\
ExpSemi & \rightarrow & Sum; \mid Disjunct; \\
ExpBrac & \rightarrow & Sum) \mid Disjunct) \\
ExpComm & \rightarrow & Sum, \mid Disjunct, \\
Actuals' & \rightarrow & ExpBrac \mid ExpComm\ Actuals'
\end{array}
$$

In einer äquivalenten – naheliegenderen – Grammatik G gäbe es anstelle der Sorten *ExpSemi*, *ExpBrac* und *ExpComm* nur *Exp* und die Kommandos von JavaLight+ wären mit folgenden Regeln ableitbar:

$$
\begin{array}{lcll}
\mathit{Command} & \to & \mathit{String} = \mathit{Exp}; \mid \mathit{write}\ \mathit{Exp}; \mid \ldots & \text{(A)} \\
\mathit{Exp} & \to & \mathit{Sum} \mid \mathit{Disjunct} & \text{(B)} \\
\mathit{Sum} & \to & \ldots \mid \mathit{Prod} & \\
\mathit{Prod} & \to & \ldots \mid \mathit{Factor} & \\
\mathit{Factor} & \to & \mathit{String} \mid \mathit{String}\ \mathit{Actuals} \mid \ldots & \\
\mathit{Disjunct} & \to & \ldots \mid \mathit{Conjunct} & \\
\mathit{Conjunct} & \to & \ldots \mid \mathit{Literal} & \\
\mathit{Literal} & \to & \mathit{String} \mid \mathit{String}\ \mathit{Actuals} \mid \ldots & \\
\mathit{Actuals} & \to & () \mid (\mathit{Actuals}' & \\
\mathit{Actuals}' & \to & \mathit{Exp}) \mid \mathit{Exp},\ \mathit{Actuals}' & \text{(C)}
\end{array}
$$

Der Parser des LL-Compilers für G wäre unvollständig, weil er z.B. das mit obigen Regeln ableitbare Kommando `z = x<=11;` nicht als solches erkennt: Der Teilcompiler für *Exp* sucht nach einem aus *Sum* ableitbaren Präfix von `x<=11` sucht, erkennt `x` als solches und gibt deshalb die Kontrolle an den Compiler für *Command* zurück, der als nächstes Zeichen das Semikolon erwartet, stattdessen aber `<=` vorfindet.

Vertauscht man *Sum* und *Disjunct* in Regel (B), dann wird z.B. das mit obigen Regeln ableitbare Kommando `z = x<=11;` nicht als solches erkannt: Jetzt sucht der Teilcompiler für *Exp* nämlich nach einem aus *Disjunct* ableitbaren Präfix von `x+11`, erkennt wie oben `x` als solches und gibt wieder die Kontrolle an den Compiler für *Command* zurück, der als nächstes Zeichen das Semikolon erwartet, stattdessen aber `+` vorfindet.

Demgegenüber erzwingt in den JavaLight+-Regeln das auf *Sum* oder *Disjunct* folgende Symbol (Semikolon, schließende Klammer bzw. Komma), dass die Teilcompiler für *ExpSemi*, *ExpBrac* und *ExpComm* die jeweilige Resteingabe *einschließlich* des abschließenden Symbols verarbeiten.

Bei der Implementierung des JavaLight+-Compilers genügt *ein* Teilcompiler für *ExpSemi*, *ExpBrac* und *ExpComm*. Er muss lediglich mit dem jeweiligen Abschlusssymbol parametrisiert werden (siehe Java2.hs). ❑

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.

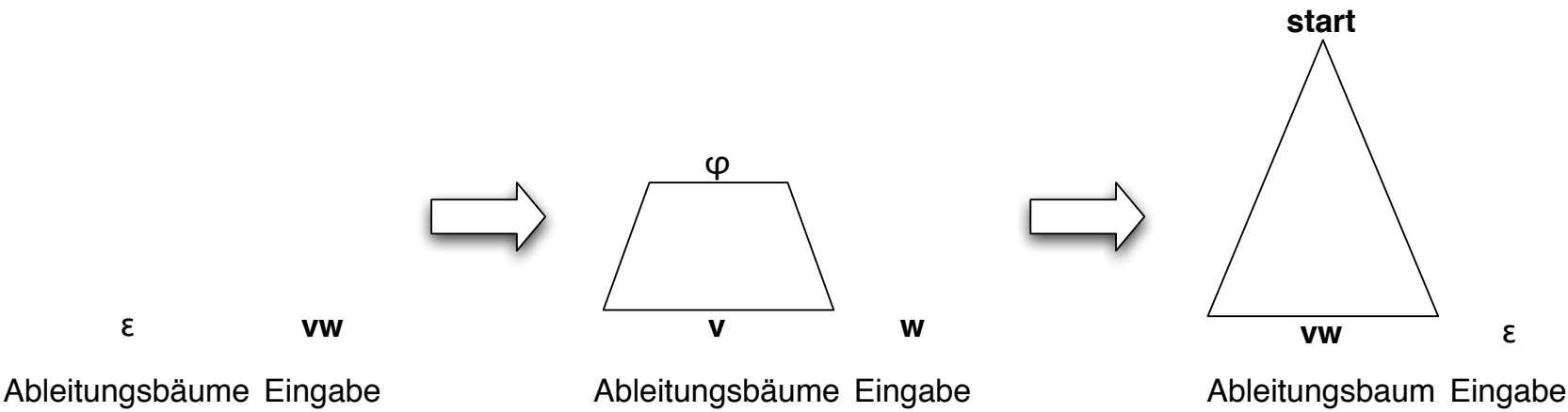
Im Gegensatz zum LL-Compiler ist die entsprechende Übersetzungsfunktion eines LR-Compilers iterativ. Die Umkehrung der Ableitungsrichtung (Reduktion statt Ableitung) macht es möglich, die Compilation durch einen Automaten, also eine iterativ definierte Funktion, zu steuern.

Während LL-Compiler keine linksrekursiven CFGs verarbeiten können, sind LR-Compiler auf LR(k)-Grammatiken beschränkt (s.u.).

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

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 Basistypen von G paarweise disjunkt sein.

Beispiel 7.1 Die CFG $G = (\{S, A\}, \{*, b\}, \emptyset, \{S \to A,\ A \to A * A,\ A \to b\})$ ist für kein k eine LR(k)-Grammatik.

Z.B. liegt nach der Reduktion des Präfixes $b * b$ des Eingabewortes $b * b * b$ zu $A * A$ ein **shift-reduce-Konflikt** vor:

Soll zunächst $A * A$ mit der Regel $A \to A * A$ zu A reduziert oder erst die Resteingabe $*b$ eingelesen und dann b mit $A \to b$ zu A reduziert werden? Die Resteingabe liefert keine eindeutige Antwort. ❏

Mengen von Wörtern über $BS = Z \cup BT$:

Für alle $k > 0$, $\alpha \in (S \cup BS)^*$ und $s \in S$,

$$
\begin{aligned}
\mathit{first}_k(\alpha) &= \{\beta \in BS^k \mid \exists\, \gamma \in BS^* : \alpha \xrightarrow{*}_G \beta\gamma\} \cup \{\beta \in BS^{<k} \mid \alpha \xrightarrow{*}_G \beta\} \\
\mathit{follow}_k(s) &= \{\beta \in BS^k \mid \exists\, \alpha, \gamma \in BS^* : \mathit{start} \xrightarrow{*}_G \alpha s \beta \gamma\} \cup \\
&\quad\ \{\beta \in BS^{<k} \mid \exists\, \alpha \in BS^* : \mathit{start} \xrightarrow{*}_G \alpha s \beta\} \\
\mathit{first}(\alpha) &= \mathit{first}_1(\alpha) \\
\mathit{follow}(s) &= \mathit{follow}_1(s)
\end{aligned}
$$

Simultane induktive Definition der first-Mengen

$$
\begin{aligned}
\mathit{first}(\epsilon) &= \{\epsilon\} && (1) \\
B \in BS \wedge \alpha \in (S \cup BS)^* &\Rightarrow \mathit{first}(B\alpha) = \{B\} && (2) \\
(s \to \alpha) \in R \wedge \beta \in (S \cup BS)^* \wedge B \in \mathit{first}(\alpha\beta) &\Rightarrow B \in \mathit{first}(s\beta) && (3)
\end{aligned}
$$

Sei G eine CFG. Wir definieren den LR(1)-Compiler für G in mehreren Schritten.

Zunächst definieren wir einen Erkenner für $L(G)_{start}$.

Während Erkenner für reguläre Sprachen reguläre Ausdrücke auf Funktionen des Typs $X^* \to 2$ abbilden (siehe Kapitel 2 und 3), bildet der folgende Erkenner (*recognizer*) Wörter über den Sorten und Basismengen von G auf jene Funktionen ab:

$$recog_1 : (S \cup BS)^* \to 2^{X^*}$$

formuliert. Sei $\gamma, \alpha, \varphi \in (S \cup BS)^*$, $x \in X$ und $w \in X^*$.

$$recog_1(\gamma\alpha)(xw) = recog_1(\gamma\alpha x)(w) \quad \text{falls} \quad \exists\, s \to \alpha\beta \in R,\ v \in (S \cup BS)^* : start \xrightarrow{*}_G \gamma s v,\ \beta \neq \epsilon,\ x \in \mathit{first}(\beta) \cap Z$$

shift (read)

$$recog_1(\gamma\alpha)(xw) = recog_1(\gamma\alpha B)(w) \quad \text{falls} \quad \exists\, s \to \alpha\beta \in R,\ B \in BT,\ v \in (S \cup BS)^* : start \xrightarrow{*}_G \gamma s v,\ \beta \neq \epsilon,\ x \in B \in \mathit{first}(\beta)$$

shift (read)

$$
\begin{array}{lcll}
recog_1(\gamma\alpha)(w) & = & recog_1(\gamma s)(w) & \text{falls} \quad \exists\, s \to \alpha \in R,\ B \in BT,\ v \in (S \cup BS)^* : \\
 & & & \quad start \overset{*}{\to}_G \gamma s v,\ s \neq start, \\
 & & & \quad w = \epsilon \in follow(s) \vee \\
 & & & \quad head(w) \in follow(s) \cap Z \vee \\
 & & & \quad head(w) \in B \in follow(s) \\
 & & & \textbf{reduce} \\
recog_1(\varphi)(\epsilon) & = & 1 & \text{falls} \quad start \to \varphi \in R \\
 & & & \textbf{accept} \\
recog_1(\varphi)(w) & = & 0 & \text{sonst} \\
 & & & \textbf{reject}
\end{array}
$$

$recog_1$ 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 $w \in X^*$,

$$recog_1(\varphi)(w) = 1 \iff start \overset{*}{\to}_G \varphi w. \tag{2}$$

(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 Q ist die Bildmenge der Funktion

$$\begin{aligned} state : (S \cup BS)^* &\to \mathcal{P}(Quad) \\ \varphi &\mapsto \{(s,\alpha,\beta,u) \in Quad \mid \exists\, \gamma, v : start \overset{*}{\to}_G \gamma s v,\ \varphi = \gamma\alpha, \\ &\qquad u = v = \epsilon \vee u = head(v) \in BS\}, \end{aligned}$$

wobei $Quad$ die Menge aller Quadrupel $(s,\alpha,\beta,u) \in S \times ((S \cup BS)^*)^2 \times (BS \cup 1)$ mit $s \to \alpha\beta \in R$ ist.

Die Bedingungen in der Definition von $recog_1$ werden durch Abfragen der Form $(s,\alpha,\beta,x) \in state(\varphi)$ ersetzt:

$$\begin{aligned} recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha x)(w) \quad \text{falls} \quad \exists\, (s,\alpha,\beta,\epsilon) \in state(\gamma\alpha) : \\ &\qquad \beta \neq \epsilon,\ x \in first(\beta) \cap Z \\ recog_1(\gamma\alpha)(xw) &= recog_1(\gamma\alpha B)(w) \quad \text{falls} \quad \exists\, (s,\alpha,\beta,\epsilon) \in state(\gamma\alpha),\ B \in BT : \\ &\qquad \beta \neq \epsilon,\ x \in B \in first(\beta) \\ recog_1(\gamma\alpha)(w) &= recog_1(\gamma s)(w) \quad \text{falls} \quad \exists\, (s,\alpha,\epsilon,u) \in state(\gamma\alpha) :\ s \neq start, \\ &\qquad w = \epsilon = u \ \vee\ head(w) = u \in Z \vee \\ &\qquad head(w) \in u \in BT \end{aligned}$$

$$
\begin{aligned}
recog_1(\varphi)(\epsilon) &= 1 \quad \text{falls} \quad (start, \varphi, \epsilon, \epsilon) \in state(\varphi) \\
recog_1(\varphi)(w) &= 0 \quad \text{sonst}
\end{aligned}
$$

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

$$
\begin{aligned}
\delta_G : Q_G &\multimap Q_G^{S \cup BS} \\
state(\varphi) &\mapsto \lambda s.state(\varphi s)
\end{aligned}
$$

In der Definition von $recog_1$ wird jedes Wort von $(S \cup BS)^*$ durch eine gleichlange Liste von Zuständen des LR-Automaten ersetzt:

$$
\begin{aligned}
h : (S \cup BS)^* &\to Q_G^* \\
\epsilon &\mapsto q_0 =_{def} state(\epsilon) \\
(s_1, \ldots, s_n) &\mapsto (state(s_1 \ldots s_n), state(s_1 \ldots s_{n-1}), \ldots, state(s_1), state(\epsilon))
\end{aligned}
$$

Simultane induktive Definition von Q_G und δ_G

$$start \to \alpha \in R \;\Rightarrow\; (start, \epsilon, \alpha, \epsilon) \in q_0 \tag{3}$$

$$(s, \alpha, s'\beta, u) \in q \wedge s' \to \gamma \in R \wedge v \in \mathit{first}(\beta u) \;\Rightarrow\; (s', \epsilon, \gamma, v) \in q \tag{4}$$

$$(s, \alpha, s'\beta, u) \in q,\; s' \in S \cup BS \;\Rightarrow\; (s, \alpha s', \beta, u) \in \delta_G(q)(s') \tag{5}$$

Im Folgenden benutzen wir gelegentlich die Haskell-Funktionen (:) und (++), um auf Wörtern zu operieren (siehe Kapitel 9).

Aus $recog_1$ wird $recog_2 : Q_G^* \to 2^{X^*}$:

Für alle $q, q_i \in Q_G$, $qs \in Q_G^*$, $x \in X$ und $w \in X^*$,

$$\begin{aligned} recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(x) : q : qs)(w) \\ &\quad \text{falls } \exists\, (s, \alpha, \beta, \epsilon) \in q : \beta \neq \epsilon,\; x \in \mathit{first}(\beta) \cap Z \\ recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(B) : q : qs)(w) \\ &\quad \text{falls } \exists\, (s, \alpha, \beta, \epsilon) \in q,\; B \in BT : \\ &\qquad \beta \neq \epsilon,\; x \in B \in \mathit{first}(\beta) \end{aligned}$$

$$
\begin{array}{lcl}
recog_2(q_1 : \cdots : q_{|\alpha|} : q : qs)(w) & = & recog_2(\delta_G(q)(s) : q : qs)(w) \\
 & & \text{falls } \exists\ (s, \alpha, \epsilon, u) \in q_1 : s \neq start, \\
 & & \quad w = \epsilon = u \ \vee \ head(w) = u \in Z \vee \\
 & & \quad head(w) \in u \in BT \\
recog_2(q : qs)(\epsilon) & = & 1 \quad \text{falls } \exists\ \varphi : (start, \varphi, \epsilon, \epsilon) \in q \\
recog_2(qs)(w) & = & 0 \quad \text{sonst}
\end{array}
$$

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 – **Aktionstabelle für** G genannte – Funktion

$$act_G : Q_G \times (BS \cup 1) \to R \cup \{shift, error\},$$

die wie folgt definiert ist:

Für alle $u \in BS \cup 1$,

$$act_G(q,u) = \begin{cases} shift & \text{falls } \exists\ (s,\alpha,\beta,\epsilon) \in q : \beta \neq \epsilon,\ u \in first(\beta), \\ s \to \alpha & \text{falls } \exists\ (s,\alpha,\epsilon,u) \in q, \\ error & \text{sonst.} \end{cases}$$

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 $w \in X^*$,

$$\begin{aligned}
recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(x) : q : qs)(w) \\
&\quad \text{falls } x \in Z,\ act_G(q,x) = shift \\
recog_2(q : qs)(xw) &= recog_2(\delta_G(q)(B) : q : qs)(w) \\
&\quad \text{falls } \exists\ B \in BT : x \in B,\ act_G(q,B) = shift \\
recog_2(q_1 : \cdots : q_{|\alpha|} : q : qs)(w) &= recog_2(\delta_G(q)(s) : q : qs)(w) \\
&\quad \text{falls } \exists\ u \in BS \cup 1 : act_G(q_1,u) = s \to \alpha, \\
&\qquad w = \epsilon = u \ \vee\ head(w) = u \in Z \vee \\
&\qquad head(w) \in u \in BT \\
recog_2(q : qs)(\epsilon) &= 1 \quad \text{falls } act_G(q,\epsilon) = start \to \alpha \\
recog_2(qs)(w) &= 0 \quad \text{sonst}
\end{aligned}$$

Beispiel 7.2 SAB2

$$
\begin{aligned}
G &= (\{S, A, B\}, \{c, d, *\}, R) \\
R &= \{S \to A, \quad A \to A * B, \quad A \to B, \quad B \to c, \quad B \to d\}
\end{aligned}
$$

LR-Automat für G

$$
\begin{aligned}
q_0 &= \{(S, \epsilon, A, \epsilon), (A, \epsilon, A * B, \epsilon), (A, \epsilon, B, \epsilon), (A, \epsilon, A * B, *), (A, \epsilon, B, *), \\
&\quad (B, \epsilon, c, \epsilon), (B, \epsilon, d, \epsilon), (B, \epsilon, c, *), (B, \epsilon, d, *)\} \\
q_1 &= \delta_G(q_0)(A) = \{(S, A, \epsilon, \epsilon), (A, A, *B, \epsilon), (A, A, *B, *)\} \\
q_2 &= \delta_G(q_0)(B) = \{(A, B, \epsilon, \epsilon), (A, B, \epsilon, *)\} \\
q_3 &= \delta_G(q_0)(c) = \{(B, c, \epsilon, \epsilon), (B, c, \epsilon, *)\} = \delta_G(q_5)(c) \\
q_4 &= \delta_G(q_0)(d) = \{(B, d, \epsilon, \epsilon), (B, d, \epsilon, *)\} = \delta_G(q_5)(d) \\
q_5 &= \delta_G(q_1)(*) = \{(A, A*, B, \epsilon), (A, A*, B, *), \\
&\quad (B, \epsilon, c, \epsilon), (B, \epsilon, d, \epsilon), (B, \epsilon, c, *), (B, \epsilon, d, *)\} \\
q_6 &= \delta_G(q_5)(B) = \{(A, A * B, \epsilon, \epsilon), (A, A * B, \epsilon, *)\}
\end{aligned}
$$

Für alle $q \in \{q_0, \ldots, q_6\}$ und $s \in S \cup BS$ definieren wir $\delta_G(q)(s)$ als leere Menge.

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	

Aktionstabelle für G

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
c	*shift*	*error*	*error*	*error*	*error*	*shift*	*error*
d	*shift*	*error*	*error*	*error*	*error*	*shift*	*error*
$*$	*error*	*shift*	$A \to B$	$B \to c$	$B \to d$	*error*	$A \to A * B$
ϵ	*error*	$S \to A$	$A \to B$	$B \to c$	$B \to d$	*error*	$A \to A * B$

Ein Erkennerlauf

$$
\begin{array}{lll}
 & recog_2(q_0)(c * d) & \\
= & recog_2(q_3 q_0)(*d) & \text{wegen } act_G(q_0, c) = \mathit{shift} \quad \text{und } \delta_G(q_0)(c) = q_3 \\
= & recog_2(q_2 q_0)(*d) & \text{wegen } act_G(q_3, *) = B \to c \quad \text{und } \delta_G(q_0)(B) = q_2 \\
= & recog_2(q_1 q_0)(*d) & \text{wegen } act_G(q_2, *) = A \to B \quad \text{und } \delta_G(q_0)(A) = q_1 \\
= & recog_2(q_5 q_1 q_0)(d) & \text{wegen } act_G(q_1, *) = \mathit{shift} \quad \text{und } \delta_G(q_1)(*) = q_5 \\
= & recog_2(q_4 q_5 q_1 q_0)(\epsilon) & \text{wegen } act_G(q_5, d) = \mathit{shift} \quad \text{und } \delta_G(q_5)(d) = q_4 \\
= & recog_2(q_6 q_5 q_1 q_0)(\epsilon) & \text{wegen } act_G(q_4, \epsilon) = B \to d \quad \text{und } \delta_G(q_5)(B) = q_6 \\
= & recog_2(q_1 q_0)(\epsilon) & \text{wegen } act_G(q_6, \epsilon) = A \to A * B \quad \text{und } \delta_G(q_0)(A) = q_1 \\
= & 1 & \text{wegen } act_G(q_1, \epsilon) = S \to A \qquad \text{❑}
\end{array}
$$

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

$$compile_G^{\mathcal{A}} : Q_G^* \times A^* \ \to \ M(A)^{X^*}.$$

Er startet mit $q_0 = state(\epsilon)$ im ansonsten leeren Zustandskeller. Die Bedeutung der Liste von A-Elementen im Definitionsbereich von $compile_G^{\mathcal{A}}$ entnimmt man am besten der folgenden Formalisierung der Korrektheit von $compile_G$:

- Für alle $w \in X^*$ und $t \in T_{\Sigma(G),start}$,

$$compile_G^{\mathcal{A}}(q_0, \epsilon)(w) = \eta(fold^{\mathcal{A}}(t)) \;\Rightarrow\; fold^{Word(G)}(t) = w, \tag{1}$$

$$compile_G^{\mathcal{A}}(q_0, \epsilon)(w) = error(w) \;\Rightarrow\; w \notin L(G)_{start}. \tag{2}$$

Die Invariante von $compile_G$, aus der (1) folgt, ist komplizierter als die des Erkenners $recog_1$ (s.o.):

- Für alle $\alpha = w_0 s_1 w_1 \ldots s_n w_n$ mit $w_i \in Z^*$ und $s_i \in S \cup BT$, $qs \in Q_G^*$, $a_i \in A$, $w \in X^*$, und $t \in T_{\Sigma(G)}(\{x_1, \ldots, x_n\})$,

$$compile_G^{\mathcal{A}}(state(\alpha) : qs, [a_1, \ldots, a_n])(w) = \eta(g_A^*(t)) \;\Rightarrow\; g_W^*(t) = \alpha w, \tag{3}$$

  wobei $g_A : \{x_1, \ldots, x_n\} \to A$ und $g_W : \{x_1, \ldots, x_n\} \to Word(G)$ x_i auf a_i bzw. s_i abbilden.

Aus (3) ergibt sich die folgende iterative Definition von $compile_G^{\mathcal{A}}$:

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 $w \in X^*$,

$$compile_G^{\mathcal{A}}(q : qs, as)(xw) = compile_G^{\mathcal{A}}(\delta_G(q)(x) : q : qs, as)(w)$$
$$\text{falls } x \in Z,\ act_G(q, x) = shift$$

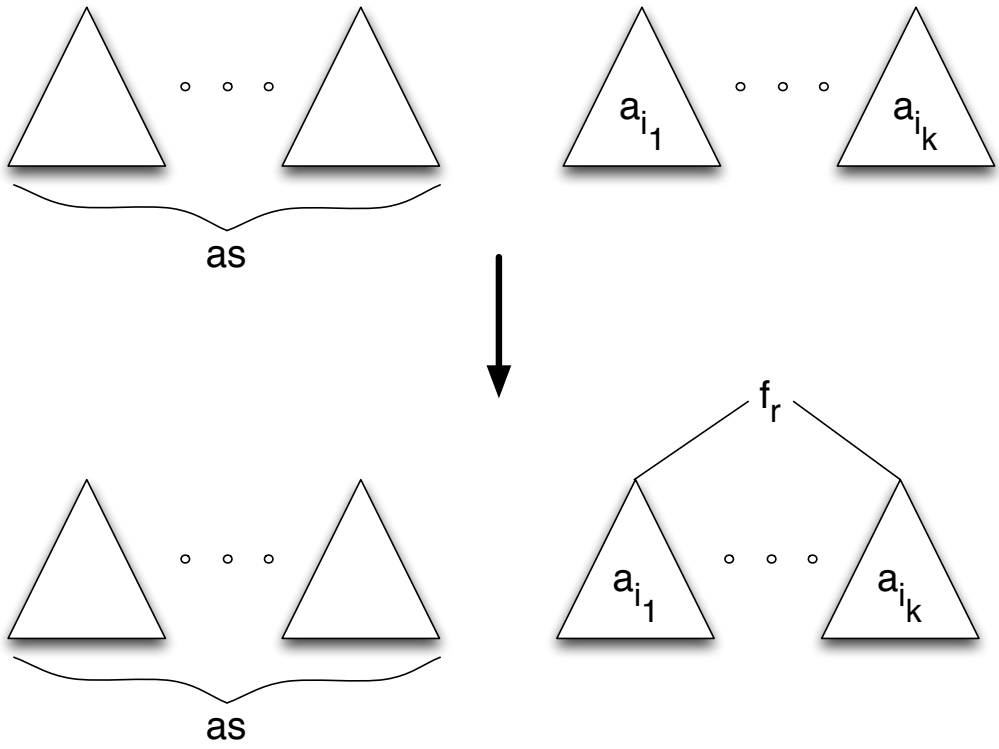
$$compile_G^{\mathcal{A}}(q : qs, as)(xw) = compile_G^{\mathcal{A}}(\delta_G(q)(B) : q : qs, as +\!\!+ [x])(w)$$
$$\text{falls } \exists\ B \in BT : x \in B,\ act_G(q, B) = shift$$

$$compile_G^{\mathcal{A}}(q_1 : \cdots : q_n : q : qs, as +\!\!+ [a_{i_1}, \ldots, a_{i_k}])(w) =$$
$$compile_G^{\mathcal{A}}(\delta_G(q)(s) : q : qs, as +\!\!+ [f_r^{\mathcal{A}}(a_{i_1}, \ldots, a_{i_k})])(w)$$
$$\text{falls } \exists\ u \in BS \cup 1 : act_G(q_1, u) = r = (s \to e_1 \ldots e_n),$$
$$\{i_1, \ldots, i_k\} = \{1 \le i \le n \mid e_i \in S \cup BT\},$$
$$w = \epsilon = u \ \vee\ head(w) = u \in Z \vee$$
$$head(w) \in u \in BT$$

$$compile_G^{\mathcal{A}}(q : qs, [a_{i_1}, \ldots, a_{i_k}])(\epsilon) = \eta(f_r^{\mathcal{A}}(a_{i_1}, \ldots, a_{i_k}))$$
$$\text{falls } act_G(q, \epsilon) = r = (start \to e_1 \ldots e_n),$$
$$\{i_1, \ldots, i_k\} = \{1 \le i \le n \mid e_i \in S \cup BT\}$$

$$compile_G^{\mathcal{A}}(qs, as)(w) = errmsg(w)$$
$$\text{sonst}$$

Im Fall $A = T_{\Sigma(G)}$ bestehen die Listen as, $[a_{i_1}, \ldots, a_{i_k}]$ und $[f_r^{\mathcal{A}}(a_{i_1}, \ldots, a_{i_k})]$ 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:

$$
\begin{aligned}
& compile_G^{T_\Sigma(G)}([0],[])(c*d) \\
= \; & compile_G^{T_\Sigma(G)}([3,0],[])(*d) \\
= \; & compile_G^{T_\Sigma(G)}([2,0],[f_{B\to c}])(*d) \\
= \; & compile_G^{T_\Sigma(G)}([1,0],[f_{A\to B}(f_{B\to c})])(*d) \\
= \; & compile_G^{T_\Sigma(G)}([5,1,0],[f_{A\to B}(f_{B\to c})])(d) \\
= \; & compile_G^{T_\Sigma(G)}([4,5,1,0],[f_{A\to B}(f_{B\to c})])(\epsilon) \\
= \; & compile_G^{T_\Sigma(G)}([6,5,1,0],[f_{A\to B}(f_{B\to c}),f_{B\to d}])(\epsilon) \\
= \; & compile_G^{T_\Sigma(G)}([1,0],[f_{A\to A*B}(f_{A\to B}(f_{B\to c}),f_{B\to d})])(\epsilon) \\
= \; & \eta(f_{S\to A}(f_{A\to A*B}(f_{A\to B}(f_{B\to c}),f_{B\to d})))
\end{aligned}
$$

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 $statems \to \epsilon$. 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	statems	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	statems	statems	statems	statems	statems	statems
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	exp NUM	exp NUM	exp NUM	exp NUM	exp NUM
8	prog statems exp dot	prog statems exp dot	prog statems exp dot	prog statems exp dot	prog statems exp dot	prog statems exp dot
9	error	error	error	error	shift	error
10	error	error	error	error	shift	error
11	statems statems assign semi	statems statems assign semi	statems statems assign semi	statems statems assign semi	statems statems assign semi	statems statems assign semi
12	error	error	error	error	error	shift
13	exp ID	exp ID	exp ID	exp ID	exp ID	exp ID
14	error	error	error	error	error	error
15	exp exp add exp	exp exp add exp	exp exp add exp	exp exp add exp	exp exp add exp	exp exp add exp
16	exp exp mul exp	exp exp mul exp	exp exp mul exp	exp exp mul exp	exp exp mul exp	exp exp mul exp
17	exp op exp cl	exp op exp cl	exp op exp cl	exp op exp cl	exp op exp cl	exp op exp cl
18	error	error	error	error	shift	error
19	assign ID col eq exp	assign ID col eq exp	assign ID col eq exp	assign ID col eq exp	assign ID col eq exp	assign ID col eq exp

	ID	NUM	add	mul	eq
0	statems	statems	statems	statems	statems
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	exp ID	exp ID	exp ID	exp ID
6	exp NUM	exp NUM	exp NUM	exp NUM	exp NUM
8	prog statems exp dot	prog statems exp dot	prog statems exp dot	prog statems exp dot	prog statems exp dot
9	shift	shift	error	error	error
10	shift	shift	error	error	error
11	statems statems assign semi	statems statems assign semi	statems statems assign semi	statems statems assign semi	statems statems assign semi
12	error	error	shift	shift	error
13	exp ID	exp ID	exp ID	exp ID	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	exp exp mul exp	exp exp mul exp	exp exp mul exp	exp exp mul exp
17	exp op exp cl	exp op exp cl	exp op exp cl	exp op exp cl	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

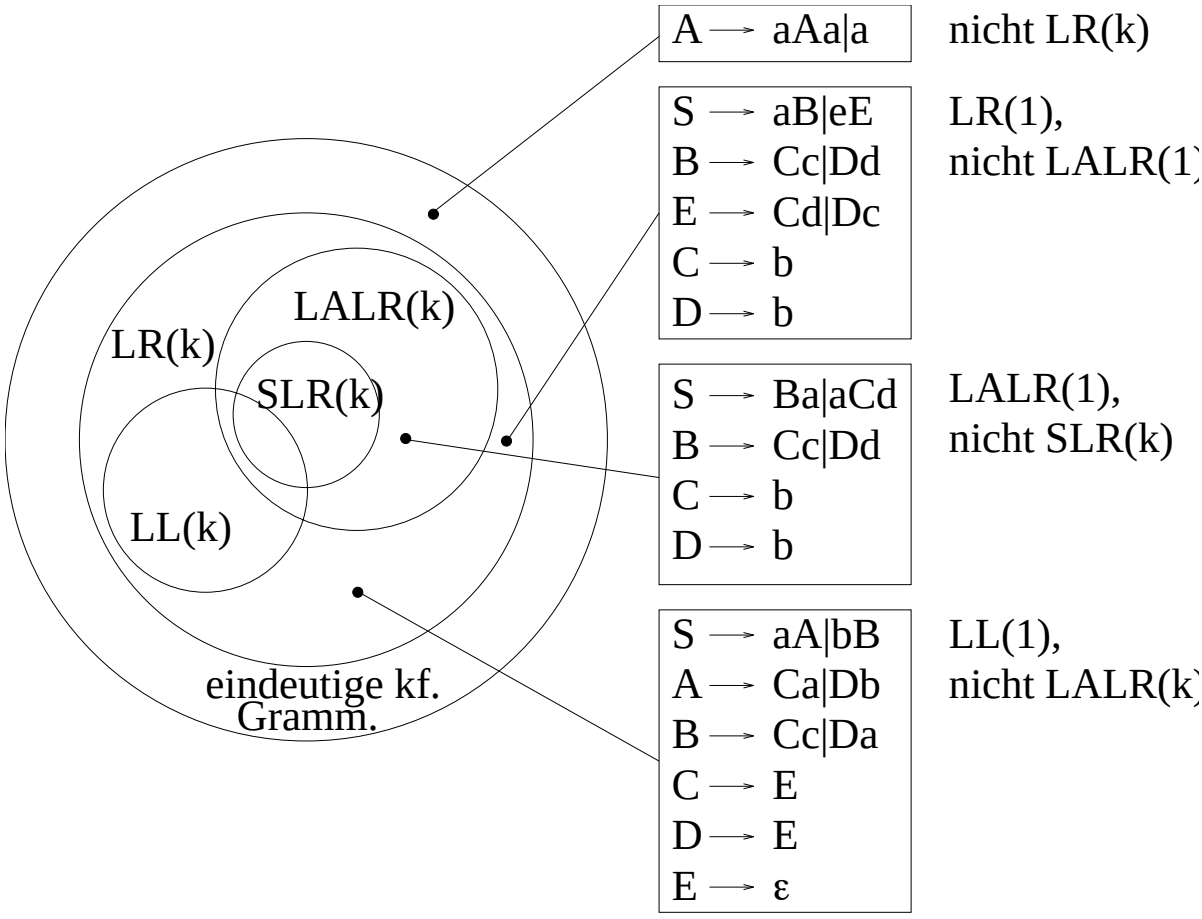
Im Folgenden werden die Regeln von G um C-Code erweitert, der eine $\Sigma(G)$-Algebra implementiert, die dem Zustandsmodell von JavaLight (siehe 4.9) ähnelt.

```
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 '*'
%%
```

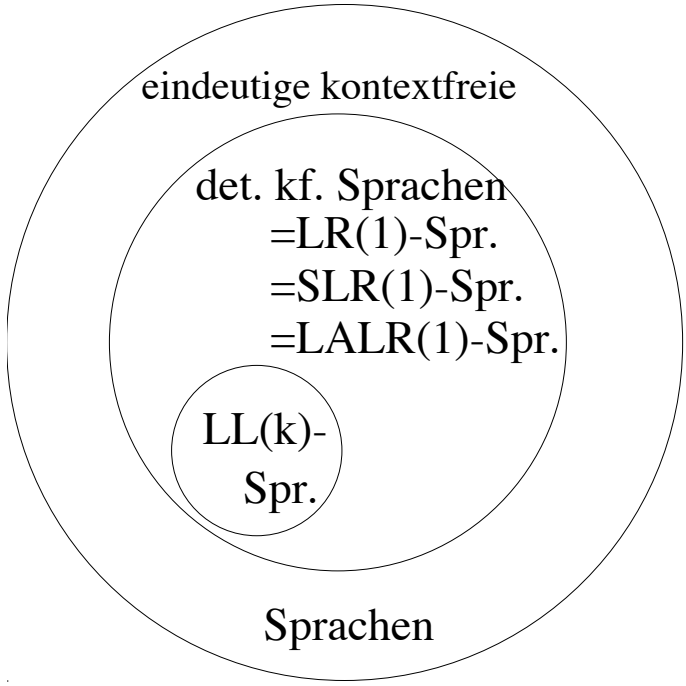
```
prog    :   statems exp '.' {printf("The final result
                                is %d\n", $2);}

statems :   statems 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 (deterministischer) 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 wegfiele, bräuchten wir keinen Lookahead beim Verarbeiten der Eingabe, womit die Zustände generell nur aus Tripeln bestünden – wie im Beispiel 7.4.

8 Datentypen in Haskell

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
```

$typ_{11}, \ldots, typ_{kn_k}$ sind beliebige Typen, die außer $x_1, \ldots, x_m$ 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 ... 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 -> ... -> typ_in_i -> DT a_1 ... a_m.
```

Alle mit einem Großbuchstaben beginnenden Funktionssymbole und alle mit einem Doppelpunkt beginnenden aus Sonderzeichen bestehenden Strings werden vom Haskell-Compiler als Konstruktoren eines Datentyps aufgefasst und müssen deshalb irgendwo im Programm in einer Datentypdefinition vorkommen.

8.1 Beispiele (siehe 2.8)

$T_{List(X)}$ ist isomorph zur kleinsten und $CT_{List(X)}$ zur größten Lösung der Gleichung

$$M \;=\; \{[]\} \;\cup\; \{a : s \mid a \in A,\; s \in M\} \tag{1}$$

in der Mengenvariablen M. Beide Mengen werden durch den Standard-Datentyp für Listen implementiert:

```
data [x] = [] | x : [x]
```

In dieser Darstellung besteht die kleinste bzw. größte Lösung von (1) aus allen endlichen bzw. allen endlichen oder unendlichen Listen über X.

$T_{Reg(BL)}$ ist isomorph zur kleinsten und $CT_{Reg(BL)}$ zur größten Lösung der Gleichung

$$\begin{aligned} M \;=\; &\{par(t,u) \mid t,u \in M\} \;\cup\; \{seq(t,u) \mid t,u \in M\} \;\cup\; \{iter(t) \mid t \in M\} \;\cup \\ &\{base(B) \mid B \in BL\} \end{aligned} \tag{2}$$

in der Mengenvariablen M. Beide Mengen werden durch folgenden Datentyp implementiert:

```
data RegT bs = Par (RegT bs) (RegT bs) | Seq (RegT bs) (RegT bs) |
               Iter (RegT bs) | Base bs
```

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. Dazu wird

```
C_i typ_i1 ... typ_in_i
```

in obiger Definition von DT erweitert zu:

```
C_i {d_i1 :: typ_i1, ..., d_in_i :: typ_in_i}
```

Wie C_i, so ist auch d_{ij} eine Funktion. Als solche hat sie den Typ

```
DT a1 ... am -> typ_ij
```

Kommt DT in typ_{ij} nicht vor, dann wird d_{ij} auch **Attribut** oder **Selektor** genannt.

Nicht-rekursive Datentypen mit genau einem Konstruktor entsprechen den in Kapitel 2 behandelten Produkttypen. Folglich sind alle Destruktoren eines Produkttyps Attribute.

Rekursive Datentypen mit genau einem Konstruktor liefern die Haskell-Realisierung der aus imperativen und objektorientierten Programmiersprachen bekannten **Records** und **Objektklassen**. Deren Destruktoren, die keine Attribute sind, werden dort **Methoden** genannt.

Semantisch entsprechen sie Zustandstransformationen, sofern man die Elemente des Datentyps als Zustände auffasst. Außerdem notiert man in objektorientierten Sprachen in der Regel den Aufruf $d_{ij}(x)$ eines Destruktors in der "qualifizierten" Form $x.d_{ij}$.

Destruktoren sind invers zu Konstruktoren. Z.B. hat der folgende Ausdruck den Wert e_j:

```
d_ij (C_i e_1 ... e_ni)
```

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 $d_{ij_1}, \ldots, d_{ij_m}$ neue Werte, nämlich $e_1, \ldots, e_m$ 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 man in der zweiten Definition von obj (s.o.) die Gleichung $d_{ij} = e_j$ durch $d_{ij} = d_{ij}$ ersetzt und die neue Funktion auf der rechten Seite lokal definiert:

```
obj = C_i {d_i1 = e_1,..., d_ij = d_ij,..., d_in_i = en_i}
      where d_ij ... = ... d_ij ...
```

Ein Konstruktor darf nicht zu mehreren Datentypen gehören.
Ein Destruktor darf nicht zu mehreren Konstruktoren unterschiedlicher Datentypen gehören.

8.2 Beispiele (siehe 2.8)

$DT_{coList(A)}$ ist isomorph zur größten Lösung der Gleichung

$$M = \{coListC(Nothing)\} \cup \{coListC(Just(a, s)) \mid a \in A,\ s \in M\} \qquad (3)$$

in der Mengenvariablen M und wird 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 durch folgenden Datentyp implementiert:

```
data StreamC a = (:<) {hd :: a, tl :: StreamC a}
```

Als Elemente von $coList(\mathbb{Z})$ lassen sich z.B. die Folgen $(0, 1, 0, 1, \dots)$ und $(1, 0, 1, 0, \dots)$ wie folgt implementieren:

```
blink,blink' :: StreamC Int
blink  = 0:<blink'
blink' = 1:<blink
```

$DT_{DAut(X,Y)}$ ist isomorph zur größten Lösung der Gleichung

$$M \;=\; \{DA(f)(y) \mid f : X \to M,\; y \in Y\}$$

in der Mengenvariablen M und wird durch folgenden Datentyp implementiert:

```
data DAutC x y = DA {deltaC :: x -> DAutC x y, betaC :: y}
```

9 Algebren in Haskell

Sei $\Sigma = (S, F)$ eine Signatur, $BT = \{x_1, \ldots, x_k\}$ die Menge der Basistypen von Σ, $S = \{s_1, \ldots, s_m\}$ und $F = \{f_1 : e_1 \to e_1', \ldots, f_n : e_n \to e_n'\}$.

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 instanziiert.

Um eine Signatur Σ in Haskell zu implementieren, genügt es daher, den Datentyp ihrer Algebren nach obigem Schema zu formulieren.

Der Datentyp $Sigma(x_1) \ldots (x_k)$ repräsentiert im Gegensatz zu den Datentypen der Beispiele 8.1 und 8.2, die Trägermengen einzelner Algebren implementieren, die Klasse aller Σ-Algebren.

9.1 Beispiele (siehe 2.8 und 2.9)

```
data Nat nat = Nat {zero :: nat, succ :: nat -> nat}
```

$natT$ implementiert $T_{Nat} \cong \mathbb{N}$:

```
natT :: Nat Int
natT = Nat {zero = 0, succ = (+1)}
```

```
data List x list = List {nil :: list, cons :: x -> list -> list}
```

$listT$ implementiert $T_{List(X)} \cong X^*$ (siehe 8.1):

```
listT :: List x [x]
listT = List {nil = [], cons = (:)}
```

Die folgende Funktion implementiert die Faltung $fold^{alg} : T_{List(X)} \to alg$ von $List(X)$-Grundtermen 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 $T_{Reg(BL)}$:

```
regT :: bs -> Reg bs (RegT bs)                                    (siehe 8.1)
regT = Reg {par = Par, seq = Seq, iter = Iter, base = Base}
```

regWord implementiert $Regword(BL)$ (siehe 2.9):

```
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
```

Die folgende Funktion implementiert die Faltung $fold^{alg} : T_{Reg(BL)} \to alg$ 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` steht 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 8.3)
streamC = Stream {head = hd, tail = tl}
```

Implementierung der $Stream(\mathbb{Z})$-Algebra *zo* (siehe 2.6):

```
data ZO = Blink | Blink'
zo :: Stream Int ZO
zo = Stream {head = \case Blink -> 0; Blink' -> 1,
             tail = \case Blink -> Blink'; Blink' -> Blink}
```

Implementierung der finalen $Stream(X)$-Algebra $InfSeq(X)$ (siehe 2.6):

```
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                (siehe 8.2)
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:

```
data DAut x y state = DAut {delta :: state -> x -> state,
                            beta :: state -> y}
```

Implementierung der finalen $DAut(X,Y)$-Algebra $DT_{DAut(X,Y)}$:

```
dAutC :: DAut x y (DAutC x y)                                     (siehe 8.2)
```

```
dAutC = DAut {delta = deltaC, beta = betaC}
```

Implementierung der $Acc(\mathbb{Z})$-Algebra *eo* (siehe 2.6):

```
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
```

Implementierung der finalen $DAut(X, Y)$-Algebra $Beh(X, Y)$ (siehe 2.10):

```
behFun :: DAut x y ([x] -> y)
behFun = DAut {delta = \f x -> f . (x:), beta = ($ [])}
```

Die folgenden zwei 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 Abschnitt 2.16 realisieren die initialen Automaten $(eo, Esum)$ und $(eo, Osum)$ die Verhaltensfunktion $even \circ sum : \mathbb{Z}^* \to 2$ bzw. $odd \circ sum : \mathbb{Z}^* \to 2$.

Da eo $Acc(\mathbb{Z})$-isomorph zur Unteralgebra von $DT_{Acc(\mathbb{Z})}$ mit der Trägermenge $\{esum, osum\}$ ist (siehe 2.8), werden $even \circ sum$ und $odd \circ sum$ auch von den initialen Automaten $(DT_{Acc(\mathbb{Z})}, esum)$ und $(DT_{Acc(\mathbb{Z})}, osum)$ realisiert. Es gelten also folgende Gleichungen:

```
unfoldDAutF eo Esum = even . sum = unfoldDAutF dAutC esum
unfoldDAutF eo Osum =  odd . sum = unfoldDAutF dAutC osum
```

9.2 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}
```

9.3 Die Termalgebra von JavaLight

Zunächst wird die Menge der JavaLight-Terme analog zu den $T_{List(X)}$ und $T_{Reg(BL)}$ (siehe 8.1 bzw. 8.2) 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, encloseD = EncloseD}

curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f a b c = f(a,b,c)
```

9.4 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++")"}
```

9.5 Das Zustandsmodell von JavaLight (siehe 4.9)

```
type Store  = String -> Int
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 lift :: (a -> b -> c) -> (st -> a) -> (st -> b) -> (st -> c)
          lift op f g st = f st `op` g st

          cond :: St Bool -> St Store -> St Store -> St Store
          cond p f g st = if p st then f st else g st

          loop :: St Bool -> St Store -> St Store
          loop p f st = if p st then loop p f $ f st else st
```

Z.B. übersetzt $compile^{javaState}_{JavaLight}$ das JavaLight-Programm

$$prog \;=\; \texttt{fact = 1; while x > 1 \{fact = fact*x; x = x-1;\}}$$

in die folgende Zustandstransformation:

$$\begin{aligned} trans : Store &\to Store \\ store &\mapsto \lambda z.if\ z = x\ then\ 0\ else\ if\ z = fact\ then\ store(x)!\ else\ store(z) \end{aligned}$$

9.6* Die Ableitungsbaumalgebra von JavaLight

```
type TS = Tree String

javaDeri :: JavaLight 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 []
```

9.7 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,
               storeO    :: orders -> stock -> store,
               orders    :: person -> items -> orders -> orders,
               embedO    :: 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}
```

10 Attributierte Übersetzung

Um die Operationen der Zielalgebra einer Übersetzung induktiv definieren zu können, müssen Trägermengen oft parametrisiert werden oder die Wertebereiche bereits parametrisierter Trägermengen um zusätzliche Komponenten erweitert werden, die zur Berechnung von Parametern rekursiver Aufrufe des Übersetzers benötigt werden. Die Zielalgebra A hat dann die Form

$$A_{v_1} \times \ldots \times A_{v_m} \;\rightarrow\; A_{a_1} \times \ldots \times A_{a_n}. \tag{1}$$

Die Indizes $v_1, \ldots, v_m$ und $a_1, \ldots, a_n$ 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.

10.1* Binärdarstellung rationaler Zahlen

(siehe auch Compiler.hs)

konkrete Syntax G	abstrakte Syntax $\Sigma(G)$
$rat \to nat. \mid rat0 \mid rat1$	$mkRat : nat \to rat$
	$app0, app1 : rat \to rat$
$nat \to 0 \mid 1 \mid nat0 \mid nat1$	$0, 1 : 1 \to nat$
	$app0, app1 : nat \to nat$

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.

$A_{nat} = \mathbb{N}$

$0^A : A_{nat}$

$0^A = 0$

$app0^{\mathcal{A}} : A_{nat} \to A_{nat}$

$app0^{\mathcal{A}}(n) = n * 2$

$mkRat^{\mathcal{A}} : A_{nat} \to A_{rat}$

$mkRat^{\mathcal{A}}(val) = (val, 1)$

$app0^{\mathcal{A}} : A_{rat} \to A_{rat}$

$app0^{\mathcal{A}}(val, inc) = (val, inc/2)$

$A_{rat} = \mathbb{Q} \times \mathbb{Q}$

$1^A : A_{nat}$

$1^A = 1$

$app1^{\mathcal{A}} : A_{nat} \to A_{nat}$

$app1^{\mathcal{A}}(n) = n * 2 + 1$

$app1^{\mathcal{A}} : A_{rat} \to A_{rat}$

$app1^{\mathcal{A}}(val, inc)) = (val + inc/2, inc/2)$

10.2* Strings mit Hoch- und Tiefstellungen

konkrete Syntax G	abstrakte Syntax $\Sigma(G)$
$string \rightarrow string\ box \mid$	$app : string \times box \rightarrow string$
$\quad string \uparrow box \mid$	$up : string \times box \rightarrow string$
$\quad string \downarrow box \mid$	$down : string \times box \rightarrow string$
$\quad box$	$mkString : box \rightarrow string$
$box \rightarrow (string) \mid$	$mkBox : string \rightarrow box$
$\quad Char$	$embed : Char \rightarrow box$

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.

$$A_{string} = A_{box} = \textit{Strings mit Hoch- und Tiefstellungen} \times \mathbb{N}^2 \rightarrow \mathbb{N}^3$$

$$app^{\mathcal{A}} : A_{string} \times A_{box} \to A_{string}$$

$$app^{\mathcal{A}}(f,g)(x,y) = (str\ str', l + l', max\ h\ h', max\ t\ t')$$
$$\text{where } (str, l, h, t) = f(x,y)$$
$$(str', l', h', t') = g(x + l, y)$$

$$up^{\mathcal{A}} : A_{string} \times A_{box} \to A_{string}$$

$$up^{\mathcal{A}}(f,g)(x,y) = (str^{str'}, l + l', h + h' - 1, max\ t\ (t' - h + 1))$$
$$\text{where } (str, l, h, t) = f(x,y)$$
$$(str', l', h', t') = g(x + l, y + h - 1)$$

$$down^{\mathcal{A}} : A_{string} \times A_{box} \to A_{string}$$

$$down^{\mathcal{A}}(f,g)(x,y) = (str_{str'}, l + l', max\ h\ (h' - t - 1), t + t' - 1)$$
$$\text{where } (str, l, h, t) = f(x,y)$$
$$(str', l', h', t') = g(x + l, y - t + 1)$$

$$mkString^{\mathcal{A}} : A_{box} \to A_{string}$$

$$mkString^{\mathcal{A}}(f) = f$$

$$mkBox^{\mathcal{A}} : A_{string} \to A_{box}$$

$$mkBox^{\mathcal{A}}(f) = f$$

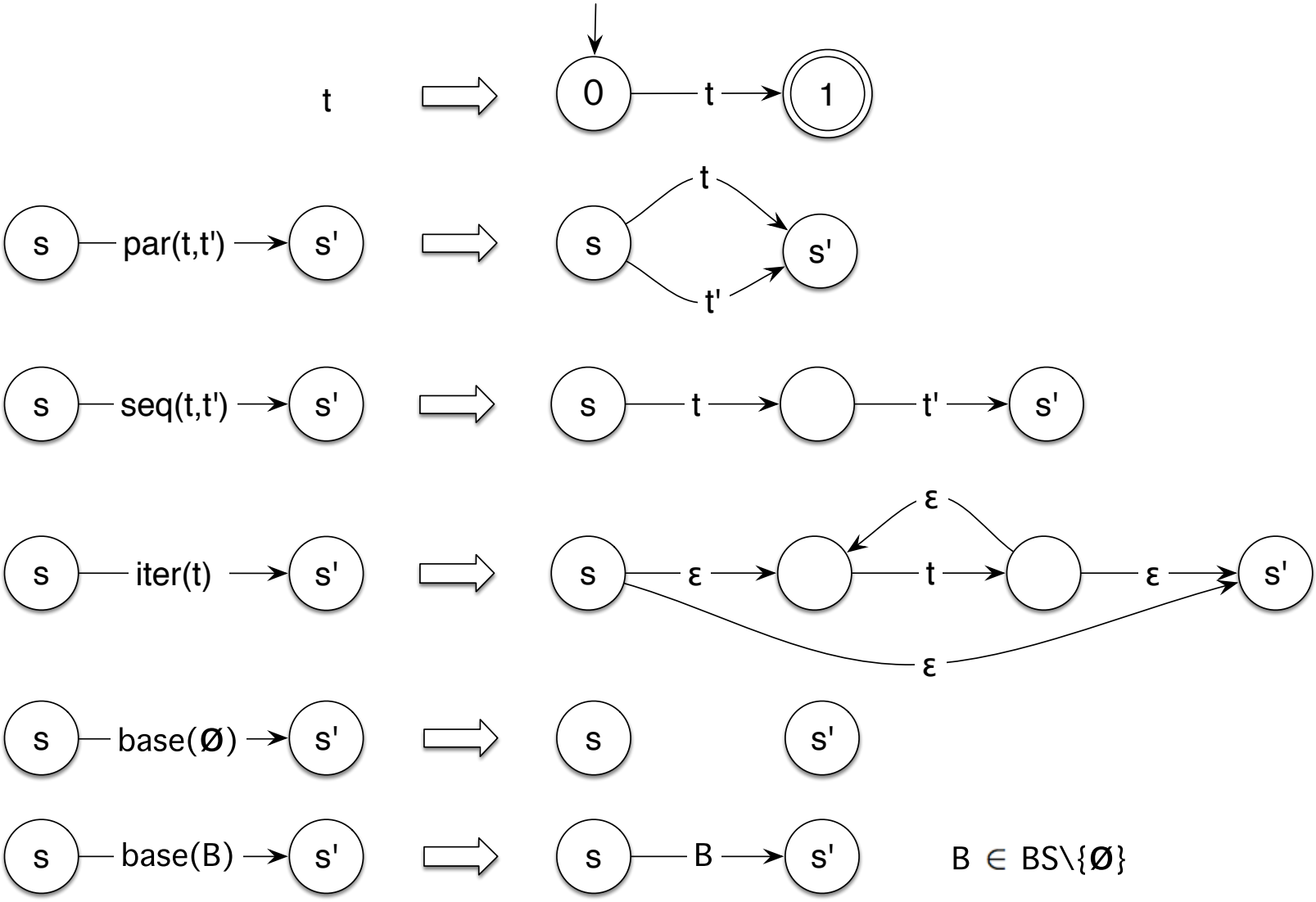
$$embed^{\mathcal{A}} : Char \to A_{box}$$

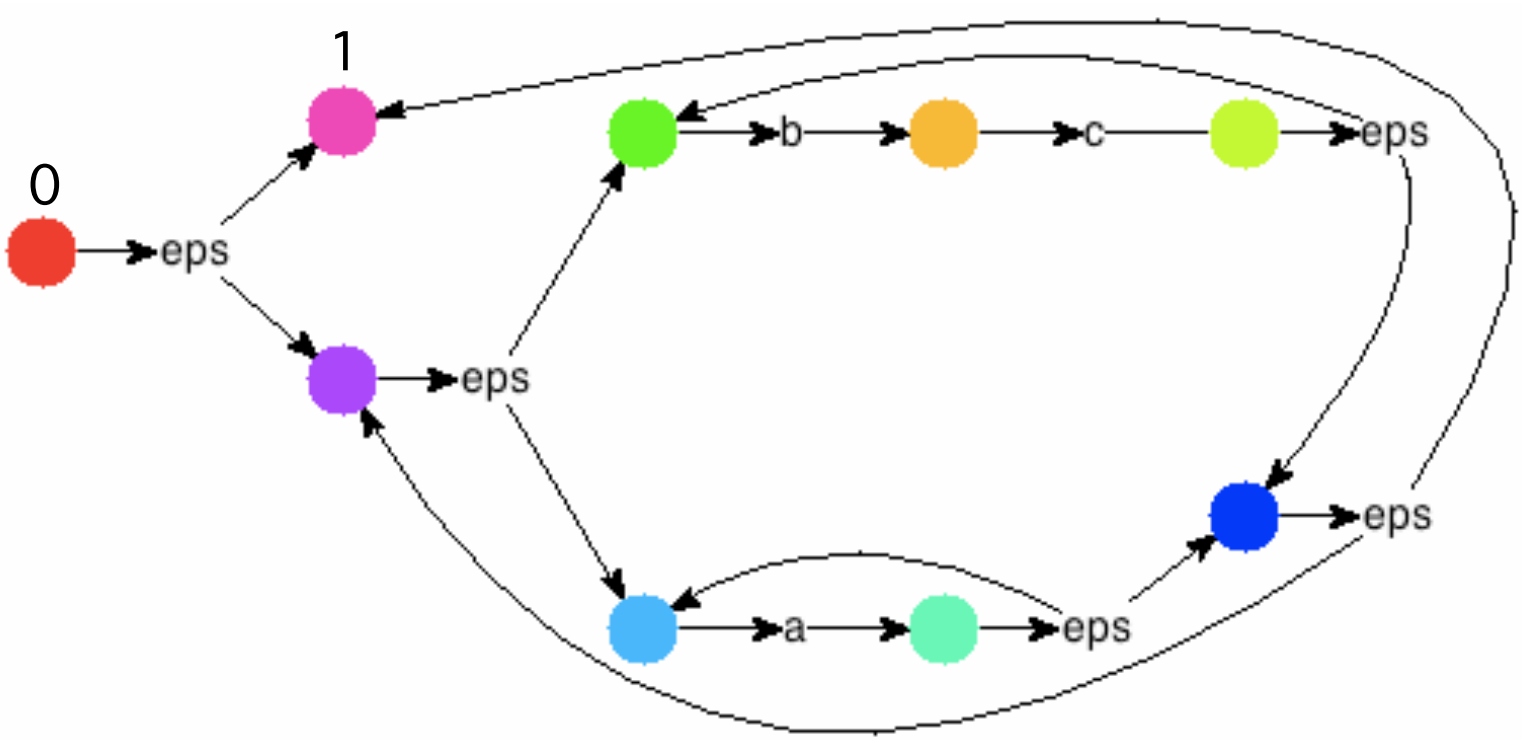
$$emded^{\mathcal{A}}(c)(x,y) = (c, 1, 2, 0)$$

10.3* 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 [12], 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.


t
0
t
1
s
par(t,t')
s'
t
t'
seq(t,t')
iter(t)
ε
base(Ø)
base(B)
B ∈ BS\{Ø}



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.

$$A_{reg} \quad = \quad (NDA \times \mathbb{N}^3 \ \rightarrow \ NDA \times \mathbb{N})$$

Hierbei ist $NDA \quad = \quad \mathbb{N} \rightarrow (BL \ \rightarrow \ \mathbb{N}^*)$ 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:

$$par^{regNDA} : regNDA_{reg} \times regNDA_{reg} \rightarrow regNDA_{reg}$$
$$par^{regNDA}(f, g)(\delta, s, s', next) = g(\delta', s, s', next')$$
$$\text{where } (\delta', next') = f(\delta, s, s', next)$$

$$seq^{regNDA}\_ : regNDA_{reg} \times regNDA_{reg} \to regNDA_{reg}$$

$$seq^{regNDA}(f,g)(\delta,s,s',next) = g(\delta',next,s',next') \quad \text{where } (\delta',next') = f(\delta,s,next,next+1)$$

$$iter^{regNDA} : regNDA_{reg} \to regNDA_{reg}$$

$$iter^{regNDA}(f)(\delta,s,s',next) = (addTo(\delta_4)(s)(\epsilon)(s'),next_3)$$

$$\begin{aligned} \text{where } next_1 &= next+1 \\ next_2 &= next_1+1 \\ (\delta_1,next_3) &= f(\delta,next,next_1,next_2) \\ \delta_2 &= addTo(\delta_1)(s)(\epsilon)(next) \\ \delta_3 &= addTo(\delta_2)(next_1)(\epsilon)(next) \\ \delta_4 &= addTo(\delta_3)(next_1)(\epsilon)(s') \end{aligned}$$

$$base^{regNDA} : BL \to regNDA_{reg}$$

$$base^{regNDA}(\emptyset)(\delta,s,s',next) = (\delta,next)$$

$$\forall\, B \in BL \setminus \{\emptyset\} : base^{regNDA}(B)(\delta,s,s',next) = (addTo(\delta)(s)(B)(s'),next)$$

$addTo(\delta)(s)(x)(s')$ 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 $Acc(BL)$-Algebra mit Zustandsmenge $\mathbb{Z}^*$ (siehe 2.8):

```
accNDA :: NDA -> (Acc BL [Int], [Int])
accNDA nda = (Acc {delta = \qs -> epsHull . deltaP qs, beta = elem 1},
              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.

10.4* 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 *Bool* bzw. *Int*.

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
```

10.5 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

$$Store = String \to \mathbb{Z}$$

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 Jstate = ([Int],Store,Int)
```

Die Bedeutung der einzelnen Befehle ergibt sich aus ihrer Verarbeitung durch folgenden Interpreter:

```
executeCom :: StackCom -> Jstate -> Jstate
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 9.5
              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] -> Jstate -> Jstate
execute cs state@(_,_,n) = if n >= length cs then state
                           else execute cs $ executeCom (cs!!n) state
```

`execute` führt Befehls*folgen* 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_       = \c c' lab -> let code = c lab
                                                 in code++c' (lab+length code),
                       embed      = id,
                       block      = id,
                       assign     = \x e lab -> e lab++[Save x,Pop],
```

```
cond     = \e c c' lab
               -> let code = e lab
                      lab1 = lab+length code+1
                      code1 = c lab1
                      lab2 = lab1+length code1+1
                      code2 = c' lab2
                      exit = lab2+length code2
                  in code++JumpF lab2:code1++Jump exit:code2,
cond1    = \e c lab -> let code = e lab
                           lab' = lab+length code+1
                           code' = c lab'
                           exit = lab'+length code'
                       in code++JumpF exit:code',
loop     = \e c lab -> let code = e lab
                           lab' = lab+length code+1
                           code' = c lab'
                           exit = lab'+length code'+1
                       in code++JumpF exit:code'++[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]

      apply2 :: StackCom -> LCom -> LCom -> LCom
      apply2 op e e' lab = code++e' (lab+length code)++[op]
                           where code = e lab
```

Beispiel 10.6 (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" 7` (siehe Java.hs) in ein Zielprogramm vom Typ `[StackCom]` und schreibt dieses in die Datei `javatarget` ab. Es lautet wie folgt:

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

11 JavaLight+ = JavaLight + I/O + Deklarationen + Prozeduren

(siehe Java2.hs)

11.1 Assemblersprache mit I/O und Kelleradressierung

Die Variablenbelegung $store : String \to \mathbb{Z}$ 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 übersetzenden Blocks bzw. Funktionsaufrufs (s.u.),
- der schon in Abschnitt 10.5 benutzte **Befehlszähler** `pc` (*program counter*),
- der Ein/Ausgabestrom `io`, auf den Lese- bzw. Schreibbefehle zugreifen.

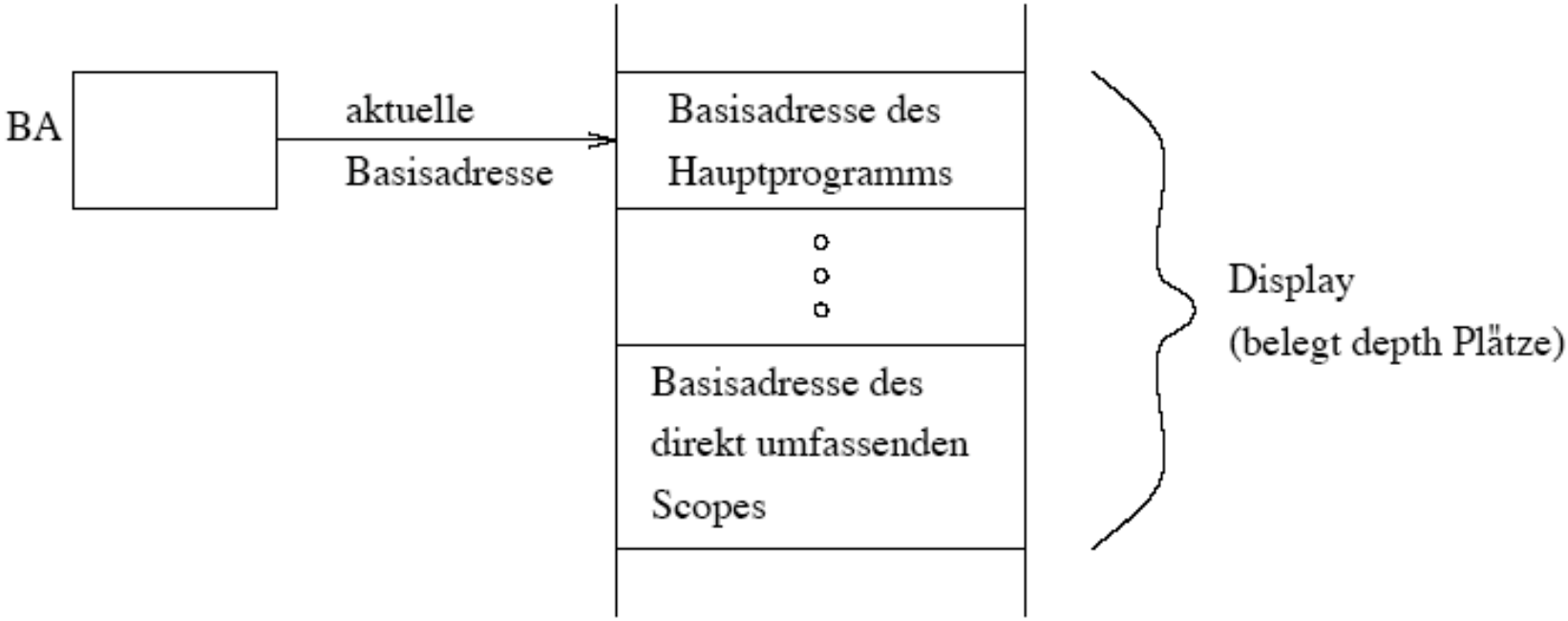
Der entsprechende Datentyp lautet daher wie folgt:

```
data Jstate = Jstate {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 (in`Dex`ed) Adressen der Form `Dex(adr)(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 `baseAdr` berechnet die jeweils aktuelle (symbolische) Basisadresse:

```
baseAdr :: Int -> Int -> SymAdr
baseAdr declDep dep = if declDep == dep then BA else Dex BA declDep
```

Der Compiler ruft **baseAdr** 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:

```
absAdr,contents :: Jstate -> SymAdr -> Int
absAdr _ (Con i)           = i
absAdr state BA            = ba state
absAdr state STP           = stp state
absAdr state TOP           = length $ stack state
absAdr state (Dex BA i)    = ba state+i
```

```
absAdr state (Dex STP i)    = stp state+i
absAdr state (Dex TOP i)    = length (stack state)+i
absAdr state (Dex adr i)    = contents state adr+i
contents state (Dex adr i) = s!!(k-i)
                               where (s,k) = stackPos state adr
contents state adr          = absAdr state adr

stackPos :: Jstate -> SymAdr -> ([Int],Int)
stackPos state adr = (s,length s-1-contents state adr)
                     where s = stack state

updState :: Jstate -> SymAdr -> Int -> Jstate
updState state BA x           = state {ba = x}
updState state STP x          = state {stp = x}
updState state (Dex adr i) x = state {stack = updList s (k-i) x}
                                 where (s,k) = stackPos state 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 SymAdr | Push SymAdr | Pop | Save SymAdr |
                Move SymAdr SymAdr | Add | Sub | Mul | Div | Or_ |
                And_ | Inv | Cmp String | Jump SymAdr | JumpF Int |
                Read SymAdr | Write
```

Analog zu Abschnitt 10.5 ist die Bedeutung der Befehle durch eine Transitionsfunktion *executeCom* gegeben – die jetzt auch die Sprungbefehle verarbeitet:

```
executeCom :: StackCom -> Jstate -> Jstate
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 :: Jstate -> (Int -> Int -> Int) -> Jstate
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] -> Jstate -> Jstate
execute cs state = if curr >= length cs then state
                   else execute cs $ executeCom (cs!!curr) state
                   where curr = pc state
```

11.2 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 Prozedur*variable*. Sie hat den Typ `ForFun(t)`. `t` ist hier der Typ der Prozedur*ergebnisse*. Demgegenüber bezeichnet `Fun(t,lab)` den Typ einer Prozedur*konstanten* 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.

$$
\begin{array}{lcl}
\mathit{Command} & \to & \mathit{read}\ \mathit{String};\ \mid\ \mathit{write}\ \mathit{ExpSemi}\ \mid\ \{\mathit{Commands}\}\ \mid \\
 & & \mathit{TypeDesc}\ \mathit{String};\ \mid\ \mathit{TypeDesc}\ \mathit{String}\ \mathit{Formals}\ \{\mathit{Commands}\}\ \mid \\
 & & \mathit{String} = \mathit{ExpSemi}\ \mid\ \mathit{String}\ \mathit{Actuals} \\
\mathit{Formals} & \to & ()\ \mid\ (\mathit{Formals}' \\
\mathit{Formals}' & \to & \mathit{Formal})\ \mid\ \mathit{Formal},\ \mathit{Formals}' \\
\mathit{ExpSemi} & \to & \mathit{Sum};\ \mid\ \mathit{Disjunct}; \\
\mathit{ExpBrac} & \to & \mathit{Sum})\ \mid\ \mathit{Disjunct}) \\
\mathit{ExpComm} & \to & \mathit{Sum},\ \mid\ \mathit{Disjunct}, \\
\mathit{Factor} & \to & \mathit{String}\ \mathit{Actuals} \\
\mathit{Literal} & \to & \mathit{String}\ \mid\ \mathit{String}\ \mathit{Actuals} \\
\mathit{Actuals} & \to & ()\ \mid\ (\mathit{Actuals}' \\
\mathit{Actuals}' & \to & \mathit{ExpBrac}\ \mid\ \mathit{ExpComm}\ \mathit{Actuals}'
\end{array}
$$

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}
```

11.3 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 $store : String \to \mathbb{Z}$ 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 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 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

      apply1 :: String -> ExpStack -> ExpStack
      apply1 op e lab st dep = (fst (e lab st dep)++[op],INT)

      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`	*Speichern des Stacktops im Register STP*
`pushDisplay dep BA`	*Kellern des Displays des umfassenden Blocks und dessen Adresse*
`Move STP BA`	*Der Inhalt von STP wird zur neuen Basisadresse*
`code`	*Zielcode für die Kommandofolge des Blocks*
`replicate (local-dep') Pop`	*Entkellern der lokalen Variablen des Blocks*
`Save BA`	*Speichern der alten Basisadresse in BA*
`replicate dep' Pop`	*Entkellern des Displays*

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 = baseAdr declDep dep
```

Zielcodeerläuterung:

`code`	*Zielcode für den Ausdruck e, der mit einem Befehl zur Kellerung des aktuellen Wertes von e schließt*
`Save $ Dex ba adrx`	*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*

gespeicherten Relativadresse `adrx` *von* x *ergibt*

`Pop` *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++JumpF lab2:code1++Jump (Con exit):
                               code2,st2,adr2)
                             where (code,_) = e lab st dep
                                   lab1 = lab+length code+1
                                   (code1,st1,adr1) = c lab1 st dep adr
                                   lab2 = lab1+length code1+1
                                   (code2,st2,adr2) = c' lab2 st1 dep adr1
                                   exit = lab2+length code2

cond1 :: ExpStack -> ComStack -> ComStack
cond1 e c lab st dep adr = (code++JumpF exit:code',st',adr')
                           where (code,_) = e lab st dep
                                 lab' = lab+length code+1
                                 (code',st',adr') = c lab' st dep adr
                                 exit = lab'+length code'

loop :: ExpStack -> ComStack -> ComStack
loop e c lab st dep adr = (code++JumpF exit:code'++[Jump $ Con lab],st',adr')
                          where (code,_) = e lab st dep
                                lab' = lab+length code+1
```

```
                                    (code',st',adr') = c lab' st dep adr
                                    exit = lab'+length code'+1
```

Übersetzung eines Lesebefehls

```
read_ :: String -> ComStack
read_ x _ st dep adr = ([Read $ Dex ba adrx],st,adr)
                       where (_,declDep,adrx) = st x
                             ba = baseAdr 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`	*Reservierung eines Kellerplatzes für den Wert eines Aufrufs von f*
`Jump $ Con exit`	*Sprung hinter den Zielcode*

Der Code zwischen *codelab* und *exit* wird erst bei der Ausführung des Zielcodes eines Aufrufs von *f* abgearbeitet (siehe `applyFun`):

`Move TOP BA`	*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 f ausgeführt wird (siehe* `applyProc`*).*
`pushDisplay dep STP`	*Kellern des Displays des umfassenden Prozedurrumpfs und dessen Adresse*
`parcode++[body]`	*erweiterter Prozedurrumpf (siehe* `formals`*)*
`replicate local Pop`	*Entkellern der lokalen Variablen und des Displays des Aufrufs von f*
`Jump retlab`	*Sprung zur Rücksprungadresse, die bei der Ausführung des Zielcodes des Aufruf von f vor dem Sprung zur Adresse des Codes von f gekellert wurde, so dass sie am Ende von dessen Ausführung wieder oben im Keller steht*

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')    (1)
                                         where adr' = adr-1

              f (cs,st,adr) (FunPar x@('@':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 attributierten 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 = baseAdr declDep dep
```

Zielcodeerläuterung:

Im Fall (1) ist x ein aktueller Parameter eines Aufrufs $e = g(e_1, \ldots, e_n)$ 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` *Kellern der Basisadresse* `ba` *von* x

`Push $ Con codelab`	*Kellern der Codeadresse* `codelab` *von x*
`PushA $ Dex ba adr`	*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*

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 in 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 = baseAdr 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`	*Zielcode für die Aufrufparameter*
`Push BA`	*Kellern der aktuellen Basisadresse*
`Move ba STP`	*Speichern der Basisadresse von f im Register STP*
`Push $ Con retlab`	*Kellern der Rücksprungadresse* `retlab`
`Jump codelab`	*Sprung zur Codeadresse* `codelab` *von f, die von* `fundecl` *berechnet wurde*

An dieser Stelle wird bei der Ausführung des Zielcodes zunächst *code*(*f*) abgearbeitet (siehe `fundecl`). Dann geht es weiter mit:

`Pop`	*Entkellern der Rücksprungadresse* `retlab`
`Save BA`	*Speichern der alten Basisadresse in BA*
`Pop`	*Entkellern der alten Basisadresse*
`replicate parLg Pop`	*Entkellern der Aufrufparameter*
`Push result`	*Kellern des Aufrufwertes*

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.

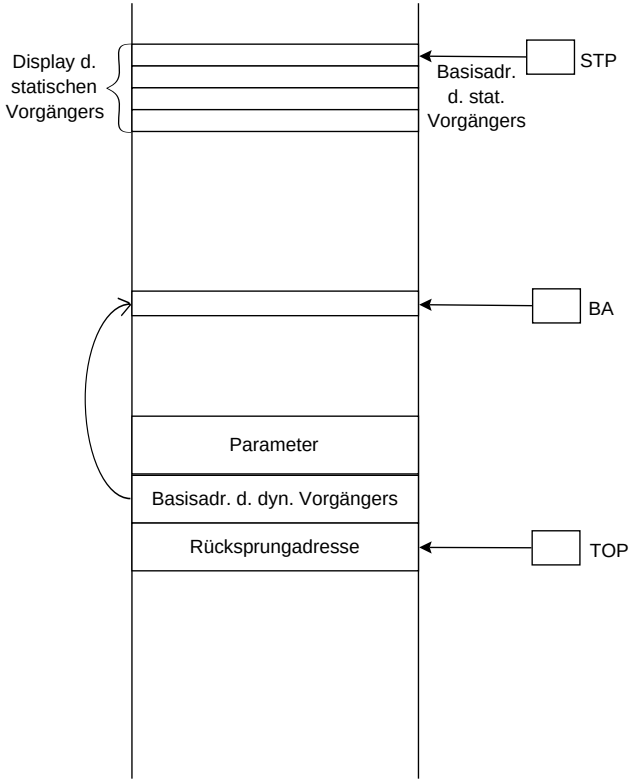
Damit `Push result` analog zum Fall (1) nicht die Resultatadresse von h, sondern den dort abgelegten Wert kellert, muss sie vorher derefenziert werden. Deshalb wird `result` im Fall (2) auf

`Dex (Dex ba $ adr+2) 0`

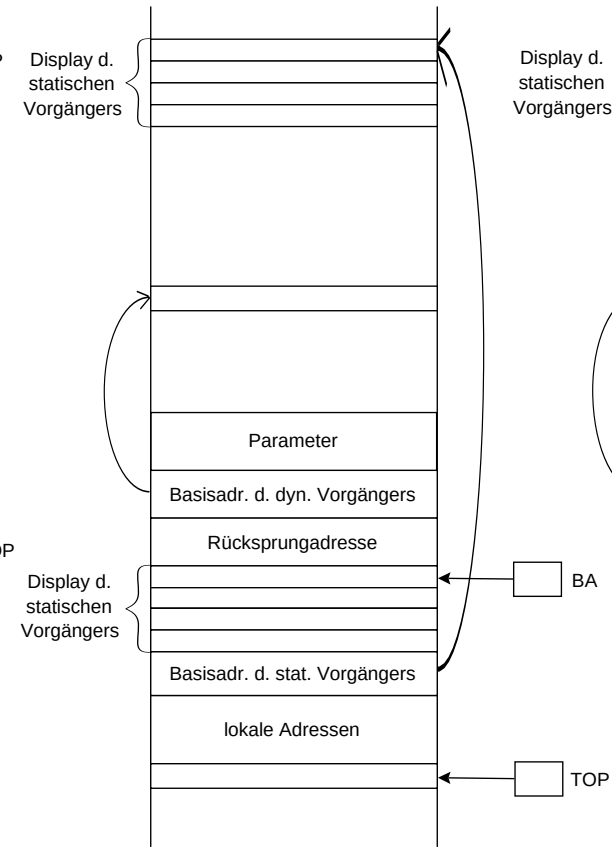
gesetzt.

Den Aufruf einer Prozedur p ohne Rückgabewert (genauer gesagt: mit Rückgabewert vom Typ `UNIT`) betten wir in eine Zuweisung an p ein:

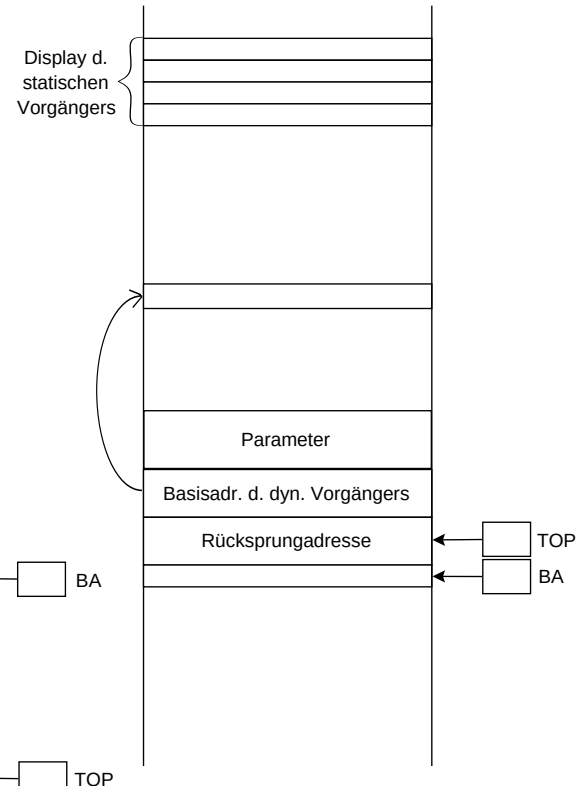
```
applyProc :: String -> ActsStack -> ComStack
applyProc f p = assign p . applyFun p
```



Kellerzustand beim Sprung zur Codeadresse



Kellerzustand während der Ausführung des Funktionsrumpfes



Kellerzustand beim Rücksprung

```
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 [30], Kapitel 5, wird auch die Übersetzung von Feldern und Records behandelt.

Grundlagen der Kompilation funktionaler Sprachen liefert [30], Kapitel 7.

Die Übersetzung objektorientierter Sprachen ist Thema von [48], Kapitel 5.

Beispiel 11.4 (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<2
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)                jump to g
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))         end f
68: Push (Con 0)
69: Jump (Con 79)
70: Move TOP BA                 begin g
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
```

Sei $G = (S, Z, BT, R)$ eine CFG und $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$-Algebra.

Wir kommen zurück auf die in Kapitel 12 behandelte Form

$$A_{v_1} \times \ldots \times A_{v_m} \;\rightarrow\; A_{a_1} \times \ldots \times A_{a_n}$$

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 $c : s_1 \times \ldots \times s_k \rightarrow s$ ein Konstruktor von $\Sigma(G)$. Das Schema einer Interpretation $c^A : A^k \rightarrow A$ von c in A offenbar wie folgt:

$$c^A(f_1, \ldots, f_k)(x_1, \ldots, x_m) = (t_1, \ldots, t_n) \;\text{ where }\; \begin{array}{l} (x_{11}, \ldots, x_{1n}) = f_1(t_{11}, \ldots, t_{1m}) \\ \vdots \\ (x_{k1}, \ldots, x_{kn}) = f_k(t_{k1}, \ldots, t_{km}) \end{array} \qquad (1)$$

Hier sind $f_1, \ldots, f_k, x_1, \ldots, x_m, x_{i1}, \ldots, x_{1n}, \ldots, x_{k1}, \ldots, x_{kn}$ paarweise verschiedene Variablen und $e_1, \ldots, e_n, x_{i1}, \ldots, t_{1m}, \ldots, t_{k1}, \ldots, t_{km}$ $\Sigma(G)$-Terme.

Ist $s \to w_0 s_1 w_1 \ldots s_k w_k$ die Grammatikregel, aus der c hervorgeht, dann wird (1) oft als Liste von Zuweisungen an die Attribute der Sorten $s, s_1, \ldots, s_k$ geschrieben:

$$s.a_1 := t_1 \;\ldots\; s.a_n := t_n \qquad s_1.v_1 := t_{11} \;\ldots\; s_1.v_m := t_{1m}$$
$$\ldots$$
$$s_k.v_1 := t_{k1} \;\ldots\; s_k.v_m := t_{km}$$

c^A ist genau dann wohldefiniert, wenn für alle $f_1, \ldots, f_k, x_1, \ldots, x_m$ (1) eine Lösung in A hat, d.h., wenn es eine Belegung g der Variablen $x_{i1}, \ldots, x_{1n}, \ldots, x_{k1}, \ldots, x_{kn}$ in A gibt mit

$$(g(x_{i1}), \ldots, g(x_{in})) = f_1(g^*(t_{i1}), \ldots, g^*(t_{im}))$$

für alle $1 \leq i \leq k$. Hinreichend für die Lösbarkeit ist die Zyklenfreiheit 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 $At = \{v_1, \ldots, v_m, a_1, \ldots, a_n\}$ aller Attribute in r Teilmengen $At_1, \ldots, At_r$ sowie jedes (funktionale) Argument f_i von c^A, $1 \leq i \leq k$, in r Teilfunktionen $f_i^1, \ldots, f_i^r$ 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:

$$\begin{aligned} c^A(f_1, \ldots, f_k)(x_{01}, \ldots, x_{0m}) = (t_{(k+1)1}, \ldots, t_{(k+1)n}) \\ \text{where} \quad (x_{11}, \ldots, x_{1n}) = f_1(t_{11}, \ldots, t_{1m}) \\ \vdots \\ (x_{k1}, \ldots, x_{kn}) = f_k(t_{k1}, \ldots, t_{km}) \end{aligned} \tag{2}$$

Für jeden Konstruktor c und jedes Attributpaar (at, at') ist der **Abhängigkeitsgraph**

$$depgraph(c)(at, at') \subseteq \{0, \ldots, k\} \times \{1, \ldots, k+1\}$$

für c und (at, at') wie folgt definiert:

$$\exists\, 1 \le i \le m,\ 1 \le j \le n : at = v_i \wedge at' = a_j \qquad \textit{(at vererbt, at' abgeleitet)}$$
$$\Rightarrow \quad depgraph(c)(at, at') = \begin{cases} \{(0, k+1)\} & \text{falls } x_{0i} \text{ in } t_{(k+1)j} \text{ vorkommt,} \\ \emptyset & \text{sonst,} \end{cases}$$
$$\exists\, 1 \le i, j \le m : at = v_i \wedge at' = v_j \qquad \textit{(at und at' vererbt)}$$
$$\Rightarrow \quad depgraph(c)(at, at') = \{(0, s) \mid 0 \le s \le k,\ x_{0i} \text{ kommt in } t_{sj} \text{ vor}\},$$
$$\exists\, 1 \le i \le n,\ 1 \le j \le m : at = a_i \wedge at' = v_j \qquad \textit{(at abgeleitet, at' vererbt)}$$
$$\Rightarrow \quad depgraph(c)(at, at') = \{(r, s) \mid 0 \le r, s \le k,\ x_{ri} \text{ kommt in } t_{sj} \text{ vor}\},$$
$$\exists\, 1 \le i, j \le n : at = a_i \wedge at' = a_j \qquad \textit{(at und at' abgeleitet)}$$
$$\Rightarrow \quad depgraph(c)(at, at') = \{(r, k+1) \mid 0 \le r \le k,\ x_{ri} \text{ kommt in } t_{(k+1)j} \text{ vor}\}.$$

(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 $At_1, \ldots, At_r$ 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:

$$p < q \ \vee \ (p = q \wedge \forall\, (i,j) \in depgraph(c)(at, at') : i < j). \tag{3}$$

Für alle $1 \leq p \leq r$ und $1 \leq i \leq k$ sei

$$\{v_{i_{p1}}, \ldots, v_{i_{pmp}}\} = \{v_1, \ldots, v_m\} \cap At_p, \quad \{a_{j_{p1}}, \ldots, a_{j_{pnp}}\} = \{a_1, \ldots, a_n\} \cap At_p,$$

$$\begin{array}{rcl} \pi_p : A_{v_1} \times \ldots \times A_{v_m} & \to & A_{v_{i_{p1}}} \times \ldots \times A_{v_{ipmp}} \\ (x_1, \ldots, x_m) & \mapsto & (x_{i_{p1}}, \ldots, x_{i_{pmp}}), \\ \pi'_p : A_{a_1} \times \ldots \times A_{a_n} & \to & A_{a_{j_{p1}}} \times \ldots \times A_{a_{jpnp}} \\ (x_1, \ldots, x_n) & \mapsto & (x_{j_{p1}}, \ldots, x_{j_{pnp}}) \end{array}$$

und $f_i^p : A_{v_{i_1}} \times \ldots \times A_{v_{imp}} \to A_{a_{j_1}} \times \ldots \times A_{a_{jnp}}$ die eindeutige Funktion, die das folgende Diagramm kommutativ macht:

$$\begin{array}{ccc} A_{v_1} \times \ldots \times A_{v_m} & \xrightarrow{\pi_p} & A_{v_{i_{p1}}} \times \ldots \times A_{v_{ipmp}} \\ \downarrow f_i & & \downarrow f_i^p \\ A_{a_1} \times \ldots \times A_{a_n} & \xrightarrow{\pi'_p} & A_{a_{j_{p1}}} \times \ldots \times A_{a_{jpnp}} \end{array}$$

Die zu (2) äquivalente Definition von c^A mit r Pässen lautet wie folgt:

$$
\begin{aligned}
c^A(f_1,\ldots,f_k)(x_{01},\ldots,x_{0m}) \;=\; & (t_{(k+1)1},\ldots,t_{(k+1)n}) \\
& \text{where}\quad (x_{1j_{11}},\ldots,x_{1j_{1n_1}}) = f_1^1(t_{1i_{11}},\ldots,t_{1i_{1m_1}}) \\
& \qquad\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Pass 1} \\
& \qquad (x_{kj_{11}},\ldots,x_{kj_{1n_1}}) = f_k^1(t_{ki_{11}},\ldots,t_{ki_{1m_1}}) \\
& \qquad\vdots \\
& \qquad\vdots \\
& \qquad (x_{1j_{r1}},\ldots,x_{1j_{rn_r}}) = f_1^r(t_{1i_{r1}},\ldots,t_{1i_{rm_r}}) \\
& \qquad\vdots \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{Pass r} \\
& \qquad (x_{kj_{r1}},\ldots,x_{kj_{rn_r}}) = f_k^r(t_{ki_{r1}},\ldots,t_{ki_{rm_r}})
\end{aligned}
$$

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 einelementigen 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
```

13 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 instanziiert wie z.B. `Int` oder `Bool`. Typvariablen zweiter Ordnung werden durch Typen zweiter Ordnung instanziiert 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
```

Leserfunktor

```
instance Functor ((->) state) where
                 fmap f h = f . h
```

Schreiberfunktor

```
instance Functor ((,) state) where
                 fmap f (st,a) = (st,f a)
```

```
newtype State state a = State {runS :: state -> (a,state)}

newtype StateT state m a = StateT {runST :: state -> m (a,state)}
```

Zustandsfunktoren

```
instance Functor (State state) where
         fmap f (State h) = State $ (\(a,st) -> (f a,st)) . h

instance Monad m => Functor (StateT state m) where
         fmap f (StateT h) = StateT $ (>>= \(a,st) -> return (f a,st))
```

Anforderungen an die Instanzen der Typklasse `Functor`:

Für alle Mengen a, b, c, $f : a \to b$ und $g : b \to c$,

```
fmap id       =  id
fmap (f . g)  =  fmap f . fmap g
```

Monaden

`Monad` ist eine Unterklasse von `Functor`:

```
class Functor m => Monad m where
      return :: a -> m a                    Einheit η
      (>>=)  :: m a -> (a -> m b) -> m b     bind-Operatoren
      (>>)   :: m a -> m b -> m b
      fail   :: String -> m a               Wert im Fall eines Matchfehlers
      m >> m' = m >>= const m'
```

```
instance Monad Id where return = Id                    Identitätsmonade
                        Id a >>= f = f a
```

```
instance Monad [ ] where return a = [a]                Listenmonade
                         (>>=) = flip concatMap
                         fail _ = []
```

```
instance Monad Maybe where return = Just               Ausnahmemonaden
                           Just a >>= f = f a
                           e >>= _      = e
                           fail _ = Nothing
```

```
instance Monad (Either e) where return = Right
                                Right a >>= f = f a
                                e >>= _       = e
```

```
instance Monad ((->) state) where return = const
                                  (h >>= f) st = f (h st) st
```

Lesermonade

```
class Monoid a where
      mempty :: a; mappend :: a -> a -> a
```

Schreibermonade

Anforderungen an die Instanzen von `Monoid`:

```
(a `mappend` b) `mappend` c  =  a `mappend` (b `mappend` c)
mempty `mappend` a           =  a
a `mappend` mempty           =  a
```

```
instance Monoid state => Monad ((,) state) where
         return a = (mempty,a)
         (st,a) >>= f = (st `mappend` st',b) where (st',b) = f a
```

```
instance Monad (State state) where
         return a = State $ \st -> (a,st)
         State h >>= f = State $ (\(a,st) -> runS (f a) st) . h

instance Monad m => Monad (StateT state m) where
         return a = StateT $ \st -> return (a,st)
         StateT h >>= f = StateT $ (>>= \(a,st) -> runST (f a) st) . h
```

Hier komponiert der bind-Operator `>>=` zwei Zustandstransformationen (h und dann f) sequentiell. Dabei liefert die von der ersten erzeugte Ausgabe die Eingabe der zweiten.

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
```

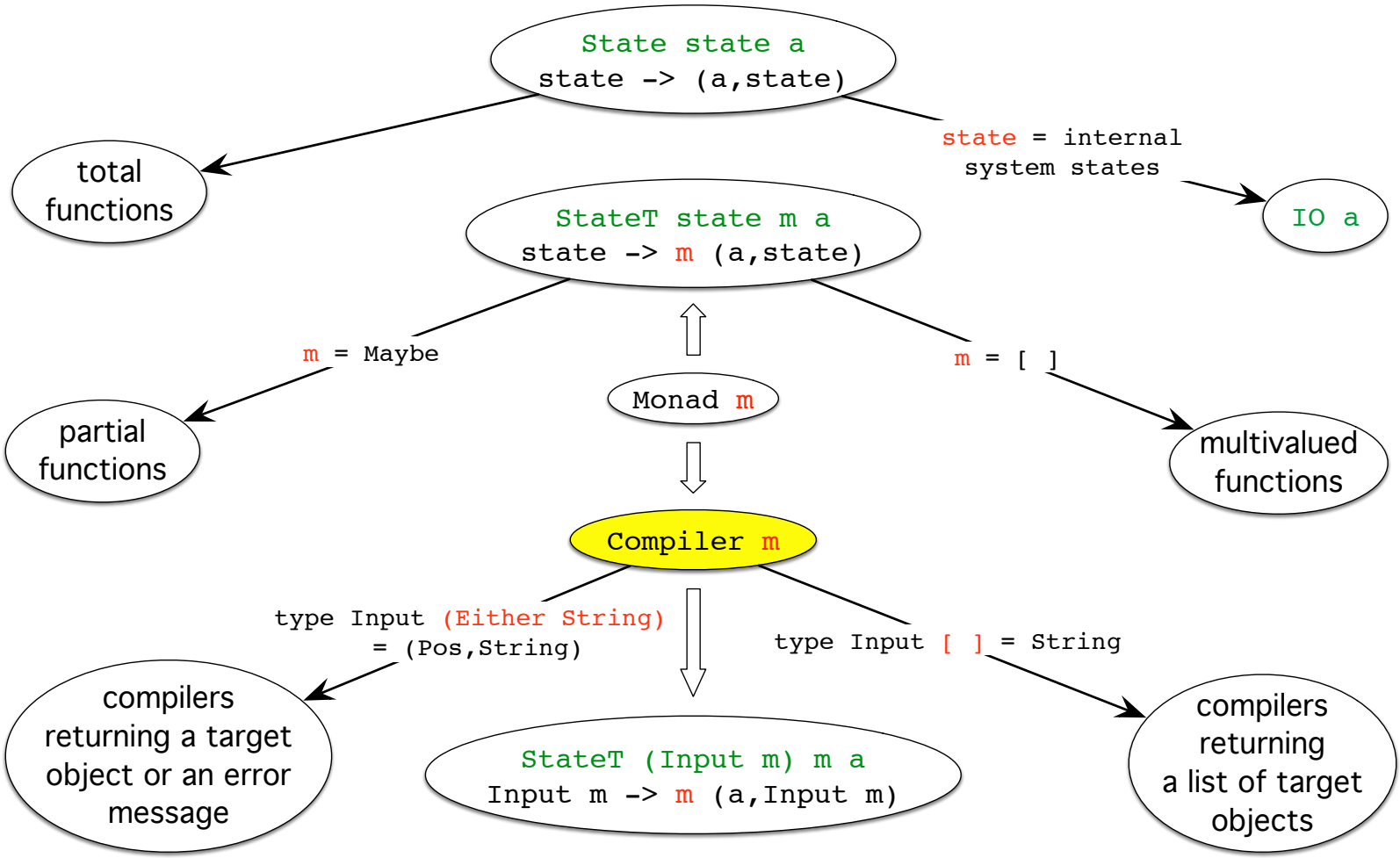
liftM2 liftet eine zweistellige Funktion $f : A \to (B \to C)$ zu einer Funktion des Typs $M(A) \to (M(B) \to M(C))$:

```
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.

14 Monadische Compiler



Sei $G = (S, Z, BT, R)$ eine LL-kompilierbare CFG, $X = Z \cup \bigcup BT$ und M eine Compilermonade (siehe Kapitel 5). Ein LL-Compiler

$$compile_G \;=\; (compile_G^A : X^* \to M(A))_{A \in Alg_{\Sigma(G)}},$$

genauer gesagt: die von $compile_G$ verwendeten Transitionsfunktionen

$$trans_s^{\mathcal{A}} : X^* \to M(A_s \times X^*), \qquad s \in S \cup Z \cup BT,\ A \in Alg_{\Sigma(G)}$$

(siehe Kapitel 5) ließen sich als Zustandsmonaden vom Typ *TransM(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 Eingabetyp auf *String* zu beschränken ist dann nicht mehr adäquat.

Deshalb implementieren wir M als Instanz folgender Unterklasse von *Monad*:

```
class Monad m => Compiler m where
                 type Input m :: *
                 errmsg :: Input m -> m a
                 empty  :: Input m -> m Bool
                 ht     :: Input m -> m (Char, Input m)
                                        (erstes Eingabezeichen, Resteingabe)
                 plus   :: m a -> m a -> m a
```

Compiler verlangt einen von der Monade m abhängigen Eingabetyp erster Ordnung.

Eine Typklasse mit Typkomponenten wird auch **Typfamilie** genannt.

errmsg behandelt fehlerhafte Eingaben. *empty* prüft, ob die Eingabe leer ist. Wenn nicht, dann liefert *ht* das erste Zeichen der Eingabe und die Resteingabe. *plus* wir durch die parallele Komposition $\oplus$ von M instanziiert (siehe Kapitel 5).

Die Compiler selbst werden dann als Bilder von

```
type Trans m = StateT (Input m) m
```

implementiert.

Zwei Instanzen der Typklasse `Compiler`

Für mehrere korrekte Ausgaben, aber nur eine mögliche Fehlermeldung:

```
instance Compiler [ ] where
         type Input [ ] = String
         errmsg _ = []
         empty = return . null
         ht (c:str) = [(c,str)]
         plus = (++)
```

Für höchstens eine korrekte Ausgabe, aber mehrere mögliche Fehlermeldungen:

```
type Pos = (Int,Int)

instance Compiler (Either String) where
         type Input (Either String) = (Pos,String)
         errmsg (pos,_) = Left $ "error at position "++show pos
         empty = return . 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.

14.1 Compilerkombinatoren

```
runC :: Compiler m => Trans m a -> Input m -> m a
runC comp input = do (a,input) <- runST comp input
                     b <- empty input
                     if b 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 m => Trans m a -> Trans m a -> Trans m a
StateT f `cplus` StateT g = StateT $ liftM2 plus f g

csum :: Compiler m => [Trans m a] -> Trans m a
csum = foldr1 cplus

some, many :: Compiler m => Trans m a -> Trans 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 m => Bool -> Trans m ()
cguard b = if b then return () else StateT errmsg
```

Sind (1), (4) und (5) erfüllt, dann gelten die folgenden semantischen Äquivalenzen:

```
do cguard True; m1; ...; mn   ist äquivalent zu   do m1; ...; mn
do cguard False; m1; ...; mn  ist äquivalent zu   mzero
```

14.2 Monadische Scanner

Scanner sind Compiler, die einzelne Symbole erkennen. Der folgende Scanner *sat(f)* erwartet, dass das Zeichen am Anfang des Eingabestrings die Bedingung f erfüllt:

```
sat :: Compiler m => (Char -> Bool) -> Trans m Char
```

```
sat f = StateT $ \input -> do b <- empty input
                              let err = errmsg input
                              p@(c,_) <- if b then err else ht input
                              if f c then return p else err

char :: Compiler m => Char -> Trans m Char
char chr = sat (== chr)

nchar :: Compiler m => String -> Trans 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 m => Trans 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 m => String -> Trans m String
string  = mapM char
```

Die folgenden Scanner erkennen Elemente von Standardtypen und übersetzen sie in entsprechende Haskell-Typen:

```
bool :: Compiler m => Trans m Bool
bool = csum [do string "True"; return True,
             do string "False"; return False]


nat,int :: Compiler m => Trans m Int
nat = do ds <- some digit; return $ read ds
int = csum [nat,
            do char '-'; n <- nat; return $ -n]
```

Kommas trennen die Argumente von *csum*.

```
identifier :: Compiler m => Trans m String
identifier = liftM2 (:) letter $ many
                                 $ nchar "(){}<>=!&|+-*/^.,:; \n\t"
```

```
relation :: Compiler m => Trans m String
relation  = csum $ map string $ words "<= >= < > == !="
```

token(*comp*) erlaubt vor und hinter dem von *comp* erkannten String Leerzeichen, Zeilenumbrüche oder Tabulatoren:

```
token :: Compiler m => Trans m a -> Trans m 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
```

14.3 Monadische LL-Compiler

Sei $G = (S, Z, BT, R)$ eine LL-kompilierbare CFG, $X = Z \cup \bigcup BT$, $G' = (S', Z, BT, R')$ die daraus gebildete nicht-linksrekursive CFG, M eine Compilermonade, $\mathcal{A} = (A, Op)$ eine $\Sigma(G)$-Algebra und $s \in S \cup Z \cup BT$.

Wir implementieren

$$compile^{\mathcal{A}}_{G,s} : X^* \to M(A_s)$$

(siehe Kapitel 6) wie folgt: Sei $S = \{s_1, \ldots, s_k\}$.

```
compile_G :: Compiler m => Alg s1...sk -> Input m -> m s
compile_G = runC . trans_s
```

Compiler für $B \in Z \cup BT$ werden vorausgesetzt:

```
trans_B :: Compiler m => Alg s1...sk -> Trans m B
```

Seien $s \in S'$ und $r_1, \ldots, r_m$ die Regeln von R' mit linker Seite s, $r_1 = (s \to e_1 \ldots e_n)$ und $\{i_1, \ldots, i_k\} = \{1 \leq i \leq n \mid e_i \in S' \cup BT\}$.

```
trans_s :: Compiler m => Alg s1...sk -> Trans m s
trans_s alg = csum [try_r1,...,try_rm] where
              try_r1 = do a1 <- trans_e1 alg
                          ...
                          an <- trans_en alg
                          return $ f_r1 (derec alg) a_i1 ... a_ik
              ...
```

Beispiel

Aus der abstrakten Syntax der Grammatik SAB von Beispiel 4.5 ergibt sich die folgende Haskell-Implementierung der Klasse aller $\Sigma(\mathrm{SAB})$-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

$$r_1 = S \to aB,\ r_2 = S \to bA,\quad r_3 = S \to \epsilon,$$
$$r_4 = A \to aS\quad r_5 = A \to bAA,\ r_6 = B \to bS,\ r_7 = B \to aBB$$

von SAB die folgende Haskell-Version der Transitionsfunktionen des LL-Compilers von Beispiel 6.1 für die Sorten von SAB:

```
transS :: Compiler m => SAB s a b -> Trans m s
transS alg = csum [do char 'a'; c <- transB alg; return $ f_1 alg c,
                   do char 'b'; c <- transA alg; return $ f_2 alg c,
                   return $ f_3 alg]
```

```
transA :: Compiler m => SAB s a b -> Trans m a
transA alg = csum [do char 'a'; c <- transS alg; return $ f_4 alg c,
                   do char 'b'; c <- transA alg; d <- transA alg
                      return $ f_5 alg c d]

transB :: Compiler m => SAB s a b -> Trans m b
transB alg = csum [do char 'b'; c <- transS alg; return $ f_6 alg c,
                   do char 'a'; c <- transB alg; d <- transB alg
                      return $ f_7 alg c d]
```

`char'a'` und `char'b'` sind die Transitionsfunktionen für das Terminal a bzw. b.

In Compiler.hs steht dieser Compiler zusammen mit der Zielalgebra *SABcount* von Beispiel 4.5. Für $m = Maybe$ und alle $w \in \{a, b\}^*$ und $i, j \in \mathbb{N}$ gilt

$$compile_{SAB}(SABcount)(w) \;=\; Just(i, j)$$

genau dann, wenn i und j die Anzahl der Vorkommen von a bzw. b in w ist. ❑

14.4 Generischer JavaLight-Compiler (siehe Abschnitt 9.2 und Java.hs)

```
compJava :: Compiler m => JavaLight s1 s2 s3 s4 s5 s6 s7 s8 -> Trans m s1
compJava alg = commands where
               alg' = derec alg
               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 <- sum_
                                     tchar ';'; return $ assign alg x e]
               sum_ = do e <- prod; f <- sumsect; return $ sum' alg' e f
               sumsect = csum [do op <- csum $ map tchar "+-"
                                  e <- prod; f <- sumsect
                                  return $ if op == '+' then plus' alg' e f
                                                        else minus' alg' e f,
                               return $ nilS alg']
               prod = 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 <- sum_; 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 <- sum_; rel <- trelation; e' <- sum_
                   return $ atom alg rel e e',
                do tchar '('; e <- disjunctC; tchar ')'
                   return $ encloseD alg e]
```

Der ensprechende Compiler für JavaLight+ (siehe Kapitel 13) steht hier.

14.5 Korrektheit und Testumgebung des JavaLight-Compilers

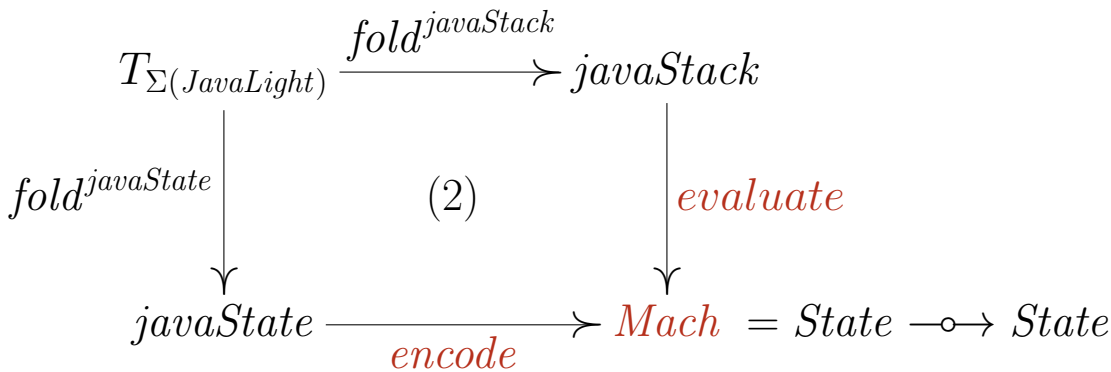
Sei $Store = String \to \mathbb{Z}$, $State = \mathbb{Z}^* \times Store$,

$$\begin{aligned} comS &= \{Commands, Command\}, \\ expS &= \{Sum, Prod, Factor\}, \\ bexpS &= \{Disjunct, Conjunct, Literal\}, \end{aligned}$$

und $javaState = (A, Op)$ und $javaStack = (B, Op')$ wie im Abschnitt 9.5 bzw. 10.5 definiert. Dann gilt für alle Sorten s (der linksrekursiven Version) von JavaLight:

$$\begin{aligned} A_s &= \begin{cases} Store \rightarrow\!\!\!\!\circ\; Store & \text{falls } s \in comS, \\ Store \to \mathbb{Z} & \text{falls } s \in expS, \\ Store \to 2 & \text{falls } s \in bexpS, \end{cases} \\ B_s &= \mathbb{Z} \to StackCom^*. \end{aligned}$$

Formal ist die semantische Korrektheit von *compJava* durch die Kommutativität folgender Instanz des Diagramm (1) am Anfang von Kapitel 5 gegeben:



Folgende Definitionen von *encode* bzw. *evaluate* machen (2) kommutiativ:

Für alle $f \in A_s$, $g \in B_s$, $stack \in \mathbb{Z}^*$ und $store \in Store$,

$$encode_s(f)(stack, store) = \begin{cases} (stack, f(store)) & \text{falls } s \in comS \\ & \text{und } f(store) \text{ definiert ist,} \\ (f(store) : stack, store) & \text{falls } s \in expS \cup bexpS \\ & \text{und } f(store) \text{ definiert ist,} \\ \text{undefiniert} & \text{sonst,} \end{cases}$$

$$evaluate_s(g)(stack, store) = \langle \pi_1, \pi_2 \rangle (execute(g(0))(stack, store, 0))$$

(2) beschreibt u.a. folgende Zusammenhänge zwischen $compJava(javaStack)$ und dem Interpreter *evaluate* der Zielsprache:

- 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.

Laut Kapitel kommutiert (2), wenn *Mach* eine JavaLight-Algebra ist und *encode* und *evaluate* JavaLight-homomorph sind. Dieses Ziel wird in Abschnitt 17.2 verfolgt.

Eine Testumgebung für Übersetzungen des JavaLight-Compilers in die JavaLight-Algebren der Kapitel 9 und 10 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 JavaLight+-Algebra *javaStackP* von Kapitel 11 liefert die Funktion *javaToStack(file)* von Java2.hs, die ein Quellprogramm vom Typ *commands* aus der Datei *file* und nach *javaStackP* überführt.

14.6 Generischer XMLstore-Compiler (siehe Compiler.hs)

```
compXML :: Compiler m => XMLstore s1 s2 s3 s4 s5 s6 s7 s8 s9 -> Trans m s1

compXML alg = storeC where
              storeC    = do tstring "<store>"
                             csum [do stck <- stock'; return $ store alg stck,
                                      do ords <- ordersC; stck <- stock'
                                         return $ storeO 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 $ embedO 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 m => Trans m String
text = do strs <- some $ token $ some $ nchar "< \n\t"
          return $ unwords strs
```

Rekursive 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

$$E = \{dc(x_1, \ldots, x_{n_c}) = t_{d,c} \mid c : e_1 \times \cdots \times e_{n_c} \to s \in C,\ d : s \to e \in D\}$$

von Σ-Gleichungen (siehe Kapitel 3) mit folgenden Eigenschaften heißt **rekursives Ψ-Gleichungssystem**:

- Für alle $d \in D$ und $c \in C$, $\mathit{free}(t_{d,c}) \subseteq \{x_1, \ldots, x_{n_c}\}$.
- 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 $C\Sigma = (S, C)$ eine konstruktive Signatur, 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 $\{c_1 : e_1 \to s, \ldots, c_n : e_n \to s\} = \{c : e \to s' \in C \mid s' = s\}$. Die Summenextension $[c_1^A, \ldots, c_n^A]$ ist bijektiv. Es gibt also eine Funktion

$$d_s^A : A_s \to A_{e_1} + \cdots + A_{e_n}$$

mit $[c_1^A, \ldots, c_n^A] \circ d_s^A = id_{A_s}$ und $d_s^A \circ [c_1^A, \ldots, c_n^A] = id_{A_{e_1}+\cdots+A_{e_n}}$.

(2) Sei $\{d_1 : s_1 \to e_1, \ldots, d_1 : s_n \to e_n\} = \{d : s' \to e \in D \mid s' = s\}$. Die Produktextension $\langle d_1^A, \ldots, d_n^A \rangle$ ist bijektiv. Es gibt also eine Funktion

$$c_s^A : A_{e_1} \times \ldots \times A_{e_n} \to A_s$$

mit $c_s^A \circ \langle d_1^A, \ldots, d_n^A \rangle = id_{A_s}$ und $\langle d_1^A, \ldots, d_n^A \rangle \circ c_s^A = id_{A_{e_1} \times \ldots \times A_{e_n}}$. ❑

Satz 15.1 Sei C_2 leer und A eine initiale $C\Sigma$-Algebra. Dann hat E genau eine induktive Lösung in A.

Beweis. Siehe [35], Theorem INDSOL. ❏

Beispiel 15.2 ($C\Sigma = Reg(BL)$; siehe 2.4 und 2.5) Sei $X = \bigcup BL$,

$$\begin{aligned} D\Sigma \;=\; & (\{reg\}, \{2, X\}, \{max, * : 2 \times 2 \to 2\} \cup \{\_ \in B : X \to 2 \mid B \in BL\}, \\ & \{\delta : reg \to reg^X,\ \beta : reg \to 2\}) \end{aligned}$$

und $\Psi = (Reg(BL), D\Sigma)$. Dann bildet die Menge BRE der Brzozowski-Gleichungen von Abschnitt 3.6 ein rekursives Ψ-Gleichungssystem, das in der initialen $Reg(BL)$-Algebra $T_{Reg(BL)}$ die in Abschnitt 2.10 durch $Bro(BL)$ definierte induktive Lösung A hat.

Nach Satz 15.1 ist A die einzige induktive Lösung von BRE in $T_{Reg(BL)}$. ❏

Weitere Induktionsverfahren, mit denen Eigenschaften kleinster Relationen bewiesen werden und die nicht nur auf initiale Algebren 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 15.3 Sei A eine finale $D\Sigma$-Algebra. Dann hat E genau eine coinduktive Lösung in A, die zur Σ-Algebra erweiterte Termalgebra $T_{C\Sigma}$ erfüllt E und es gilt

$$unfold^{T_{C\Sigma}} = fold^{A}.$$

Beweis. Siehe [36], Theorem COINDSOL or [35], Theorem RECFUN3. ❑

Beispiel 15.4

Sei $\Psi = (Reg(BL), D\Sigma)$ die Bisignatur von Beispiel 15.2, $\Sigma = Reg(BL) \cup D\Sigma$ und A die Σ-Algebra mit $A|_{Reg(BL)} = Lang(X)$ und $A|_{Acc(X)} = \mathcal{P}(X)$.

A erfüllt das rekursive Ψ-Gleichungssystem BRE von Abschnitt 3.6 und ist daher nach Satz 15.3 die einzige coinduktive Lösung von BRE in $\mathcal{P}(X)$.

Da $\chi : \mathcal{P}(X^*) \to 2^{X^*}$ (siehe 2.10) Σ-homomorph ist, wird BRE auch von der Bildalgebra $\chi(A)$ erfüllt. Nach Satz 15.3 ist $\chi(A)$ die einzige coinduktive Lösung von BRE in $\chi(\mathcal{P}(X)) = Beh(X, 2)$ und gelten die Gleichungen

$$fold^{Lang(X)} = unfold^{Bro(BL)} : Bro(BL) \to \mathcal{P}(X),$$
$$fold^{regB} = unfold^{Bro(BL)} : Bro(BL) \to Beh(X, 2).$$

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. ❑

Entscheidende Argumente im Beweis von Satz 15.3 entstammen dem Beweis von [41], Thm. 3.1, und [42], Thm. A.1, wo rekursive $Stream(X)$-Gleichungen für Stromkonstruktoren untersucht werden (siehe auch [10], Anhänge A.5 and A.6).

Analoge Ergebnisse erhält man für weitere destruktive Signaturen wie z.B. unendliche Binärbäume ([43], Thm. 2), nichtdeterministische Transitionssysteme [3], Mealy-Automaten [9], nebenläufige Prozesse [13, 39, 14] und formale Potenzreihen ([41], 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:

$$\frac{}{base(B) \xrightarrow{\delta} \lambda x.ite(x \in B, base(\{\epsilon\}), base(\emptyset))}$$

$$\frac{t \xrightarrow{\delta} t',\ u \xrightarrow{\delta} u'}{par(t,u) \xrightarrow{\delta} \lambda x.par(t'(x), u'(x))} \qquad \frac{t \xrightarrow{\delta} t',\ u \xrightarrow{\delta} u'}{seq(t,u) \xrightarrow{\delta} \lambda x.par(seq(t'(x), u), ite(\beta(t), u'(x), base(\emptyset)))}$$

$$\frac{t \xrightarrow{\delta} t'}{iter(t) \xrightarrow{\delta} \lambda x.seq(t'(x), iter(t))} \qquad \frac{}{base(B) \xrightarrow{\beta} ite(B = 1, 1, 0)}$$

$$\frac{t \xrightarrow{\beta} m,\ u \xrightarrow{\beta} n}{par(t,u) \xrightarrow{\beta} max\{m,n\}} \qquad \frac{t \xrightarrow{\beta} m,\ u \xrightarrow{\beta} n}{seq(t,u) \xrightarrow{\beta} m * n} \qquad \frac{}{iter(t) \xrightarrow{\beta} 0}$$

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. [44, 16, 11, 14]).

15.5 Cotermbasierte Erkenner regulärer Sprachen

Unter Verwendung der Haskell-Darstellung von Cotermen als Elemente des rekursiven Datentyps *StateC(x)(y)* (siehe 8.2) machen wir $DT_{Acc(X)}$ zur $Reg(BL)$-Algebra *regA* und zwar so, dass *regA* mit $\xi(\chi(Lang(X))$ übereinstimmt, wobei $\xi : Beh(X,Y) \to DT_\Sigma$ den $DAut(X,Y)$-Isomorphismus von $Beh(X,Y)$ nach DT_Σ bezeichnet (siehe Beispiel 2.15).

Für alle $B \in BL$, $f, g : X \to DT_{Acc(X)}$, $b, c \in 2$ und $t \in DT_{Acc(X)}$,

$$
\begin{aligned}
&base^{regA}(B) = StateC(\lambda x.\mathit{if}\ x \in B\ \mathit{then}\ base^{regA}(1)\ \mathit{else}\ base^{regA}(\emptyset))\\
&\qquad\qquad\qquad\qquad (\mathit{if}\ B = 1\ \mathit{then}\ 1\ \mathit{else}\ 0),\\
&par^{regA}(StateC(f)(b), StateC(g)(c)) = StateC(\lambda x.par^{regA}(f(x), g(x)))(max\{b,c\}),\\
&seq^{regA}(StateC(f)(1), StateC(g)(c)) = StateC(\lambda x.par^{regA}(seq^{regA}(f(x), StateC(g)(c)),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad g(x)))(c),\\
&seq^{regA}(StateC(f)(0), t) = StateC(\lambda x.seq^{regA}(f(x), t))(0),\\
&iter^{regA}(StateC(f)(b)) = StateC(\lambda x.seq^{regA}(f(x), iter^{regA}(StateC(f)(b))))(1).
\end{aligned}
$$

Sei $\Psi = (Reg(BL), D\Sigma)$ die Bisignatur von Beispiel 15.2, $\Sigma = Reg(BL) \cup D\Sigma$ 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 Abschnitt 3.6) von A erfüllt wird und $\chi : \mathcal{P}(X^*) \to 2^{X^*}$ (siehe 2.10) und ξ Σ-homomorph sind, wird das rekursive Ψ-Gleichungssystem BRE von Abschnitt 3.6 BRE auch von der Bildalgebra $\xi(\chi(A))$ erfüllt.

Nach Satz 15.3 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

$$fold^{regA} = unfold^{Bro(BL)} : Bro(BL) \to DT_{Acc(X)}.$$

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

$$unfold^{Bro(BL)} = \xi \circ \chi \circ unfold^{Bro(BL)} = \xi \circ \chi \circ fold^{Lang(X)}. \tag{3}$$

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.

16 * Iterative Gleichungen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur und V eine endliche S-sortige Menge von "Variablen". Eine S-sortige Funktion

$$E : V \to T_\Sigma(V)$$

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.

$g : V \to A$ **löst** E **in** $\mathcal{A}$, wenn $g^* \circ E = g$ gilt.

Lemma 16.1 ([35], Lemma ITER (iii).)

Sei $reduce : T_\Sigma(V) \to T_\Sigma(V)$ eine Funktion mit

$$g^* \circ reduce = g^* \tag{1}$$

für alle $g : V \to A$. Dann gilt

$$g^* \circ (reduce \circ E^*)^n = g^* \tag{2}$$

für alle Lösungen $g : V \to A$ von E in A und $n \in \mathbb{N}$.

16.2 Das iterative Gleichungssystem einer CFG

Sei $G = (S, Z, BT, R)$ eine CFG und für alle $s \in S$ sei

$$right_s = \{w \in (S \cup Z \cup BT)^* \mid s \to w \in R\}.$$

G induziert das iterative $Reg(BL)$-Gleichungssystem

$$E_G : S \to T_{Reg(BL)}(S),$$

das jede Sorte $s \in S$ auf den regulären Ausdruck abbildet, der sich wie folgt aus $right_s$ ergibt:

$$E_G(s) = \begin{cases} \overline{w} & \text{falls } right_s = \{w\}, \\ par(\overline{w_1}, \ldots, \overline{w_k}) & \text{falls } right_s = \{w_1, \ldots, w_k\},\ k > 1, \end{cases}$$

wobei für alle $n > 1$, $e, e_1, \ldots, e_n \in S \cup Z \cup BT$ und $t_1, \ldots, t_n \in T_{Reg(BL)}(S)$,

$$\begin{array}{rcl} \overline{e} & = & e, \\ \overline{e_1 \ldots e_n} & = & seq(e_1, \ldots, e_n), \\ par(t_1, \ldots, t_n) \text{ für } par(t_1, par(t_2, \ldots, par(t_{n-1}, t_n) \ldots)) & & \text{steht}, \\ seq(t_1, \ldots, t_n) \text{ für } seq(t_1, seq(t_2, \ldots, seq(t_{n-1}, t_n) \ldots)) & & \text{steht}. \end{array}$$

Wir nennen E_G das **Gleichungssystem von** G.

Satz 16.3 (Fixpunktsatz für CFGs) Sei $X = Z \cup \bigcup BT$.

(i) Die Funktion $sol_G : S \to Lang(X)$ mit $sol_G(s) = L(G)_s$ für alle $s \in S$ löst E_G in $Lang(X)$.

Sei $g : S \to \mathcal{P}(X^*)$ 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 $g : V \to Lang(X)$, $s \in S$, $\{r_1, \ldots, r_k\} = right_s$, $1 \leq i \leq k$, $r_i = (s \to w_i)$ und

$$dom(f_{r_i}) = e_{i1} \times \ldots \times e_{in_i}.$$

Dann gibt es $s_1, \ldots, s_n \in S \cup Z \cup BT$ mit $e_1 \ldots e_n = w_i$ und

$$g^*(\overline{w_i}) = g^*(\overline{e_1 \ldots e_n}) = g^*(seq(\overline{e_1}, \ldots, \overline{e_n})) = seq^{Lang(X)}(g^*(\overline{e_1}), \ldots, g^*(\overline{e_n}))$$
$$= g^*(\overline{e_1}) \cdot \ldots \cdot g^*(\overline{e_n}) = f_{r_i}^{Word(G)}(g^*(\overline{e_{i1}}) \cdot \ldots \cdot g^*(\overline{e_{in_i}})). \qquad (1)$$

Beweis von (i).

$$
\begin{aligned}
&sol_G(s) = L(G)_s = fold_s^{Word(G)}(T_{\Sigma(G),s}) \\
&= \bigcup_{i=1}^{k} \{fold_s^{Word(G)}(f_{r_i}(t)) \mid t \in T_{\Sigma(G),e_{i1}\times\ldots\times e_{in_i}}\} \\
&= \bigcup_{i=1}^{k} \{f_{r_i}^{Word(G)}(fold_{e_{i1}\times\ldots\times e_{in_i}}^{Word(G)}(t)) \mid t \in T_{\Sigma(G),e_{i1}\times\ldots\times e_{in_i}}\} \\
&= \bigcup_{i=1}^{k} f_{r_i}^{Word(G)}(fold_{e_{i1}\times\ldots\times e_{in_i}}^{Word(G)}(T_{\Sigma(G),e_{i1}\times\ldots\times e_{in_i}})) = \bigcup_{i=1}^{k} f_{r_i}^{Word(G)}(L(G)_{e_{i1}\times\ldots\times e_{in_i}}) \\
&= \bigcup_{i=1}^{k} f_{r_i}^{Word(G)}(L(G)_{e_{i1}} \cdot \ldots \cdot L(G)_{e_{in_i}}) \\
&= \bigcup_{i=1}^{k} f_{r_i}^{Word(G)}(sol_G^*(\overline{e_{i1}}) \cdot \ldots \cdot sol_G^*(\overline{e_{in_i}})) \\
&\overset{(1)}{=} \bigcup_{i=1}^{k} sol_G^*(\overline{w_i}) = par^{Lang(X)}(sol_G^*(\overline{w_1}), \ldots, sol_G^*(\overline{w_k})) \\
&= sol_G^*(par(\overline{w_1}, \ldots, \overline{w_k})) = sol_G^*(E_G(s)).
\end{aligned}
$$

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 $1 \leq i \leq k$,

$$f_{r_i}^{Word(G)}(Sol_{e_{i1}} \cdot \ldots \cdot Sol_{e_{in_i}}) \subseteq Sol_s. \tag{2}$$

Nach Definition von Sol gilt $Sol_{e_{ij}} = g^*(\overline{e_{ij}})$ für alle $1 \leq j \leq n_i$.

Beweis von (2).

$$
\begin{aligned}
& f_{r_i}^{Word(G)}(Sol_{e_{i1}} \cdot \dots \cdot Sol_{e_{in_i}}) = f_{r_i}^{Word(G)}(g^*(\overline{e_{i1}}) \cdot \dots \cdot g^*(\overline{e_{in_i}})) \\
& \overset{(1)}{=} g^*(\overline{w_i}) \subseteq \bigcup_{i=1}^{k} g^*(\overline{w_i}) = par^{Lang(X)}(g^*(\overline{w_1}), \dots, g^*(\overline{w_k})) = g^*(par(\overline{w_1}, \dots, \overline{w_k})) \\
& \overset{Def.\ E_G}{=} g^*(E_G(s)) = g(s) = Sol_s.
\end{aligned}
$$

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

$$sol_G(s) = L(G)_s \subseteq Sol_s = g(s). \qquad \square$$

16.4 Beispiele Sei $X = \{a, b\}$.

1. Sei $G = (\{A, B\}, X, \emptyset, \{A \to BA,\ A \to a,\ B \to b\})$. 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:

$$g(A) = \{b\}^* \cdot \{a\}, \qquad g(B) = \{b\}.$$

2. Sei $G = \text{SAB}$ (siehe Beispiel 4.5). E_G ist wie folgt definiert:

$$\begin{aligned} E_G(S) &= par(seq(\overline{a}, B), seq(\overline{b}, A), eps) \\ E_G(A) &= par(seq(\overline{a}, S), seq(\overline{b}, A, A)), \\ E_G(B) &= par(seq(\overline{b}, S), seq(\overline{a}, B, B)). \end{aligned}$$

Die einzige Lösung g von E_G in $Lang(X)$ lautet:

$$\begin{aligned} g(S) &= \{w \in X^* \mid \#a(w) = \#b(w)\}, \\ g(A) &= \{w \in X^* \mid \#a(w) = \#b(w) + 1\}, \\ g(B) &= \{w \in X^* \mid \#a(w) = \#b(w) - 1\}. \end{aligned}$$

Andererseits ist g mit $g(S) = g(A) = g(B) = X^+$ keine Lösung von E_G in $Lang(X)$, weil a einerseits zu $g(S)$ gehört, aber nicht zu

$$g^*(E_G(S)) = g^*(par(seq(\overline{a}, B), seq(\overline{b}, A))) = (\{a\} \cdot g(B)) \cup (\{b\} \cdot g(A)).$$

Beide Grammatiken sind nicht linksrekursiv. Deshalb haben ihre Gleichungssysteme nach Satz 16.8 (s.u.) jeweils genau eine Lösung in $Lang(X)$. ❏

16.5 Von Erkennern regulärer Sprachen zu Erkennern kontextfreier Sprachen

Sei $\Sigma = (S, F)$ eine konstruktive Signatur und V eine S-sortige Menge.

$$\Sigma_V \ =_{def} \ (S, F \cup \{in_s : V_s \to s \mid s \in S\}).$$

Lemma 16.6

$\sigma_V : V \to T_{\Sigma_V}$ 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

$$(in^A)^* = fold^A \circ \sigma_V^* : T_\Sigma(V) \to A,$$

wobei $in^A = (in_s^A : V_s \to A_s)_{s \in S}$.

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 $fold^A : T_{\Sigma_V} \to A$ mit in_s verträglich ist, gilt

$$fold^A(\sigma_V(x)) = fold^A(in_s x) = in_s^A(x).$$

Aus (3) folgt $(in^A)^* = (fold^A \circ \sigma_V)^* \overset{Satz\ 3.7(1)}{=} fold^A \circ \sigma_V^*$. ❏

Sei $G = (S, Z, BT, R)$ eine nicht-linksrekursive CFG und $reduce$ die in Abschnitt 3.5 beschriebene Reduktionsfunktion für reguläre Ausdrücke.

Dann gibt es für alle $s \in S$ $k_s, n_s > 0$, $B_{s,1}, \ldots, B_{s,n_s} \in Z \cup BT$ und $Reg(BL)$-Terme $t_{s,1}, \ldots, t_{s,n_s}$ über S mit

$$(reduce \circ E_G^*)^{k_s}(s) = par(seq(\overline{B_{s,1}}, t_{s,1}), \ldots, seq(\overline{B_{s,n_s}}, t_{s,n_s})) \tag{4}$$

oder

$$(reduce \circ E_G^*)^{k_s}(s) = par(seq(\overline{B_{s,1}}, t_{s,1}), \ldots, seq(\overline{B_{s,n_s}}, t_{s,n_s}), eps). \tag{5}$$

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 $in =_{def} in_{reg} : S \to reg$ als weiteres Operationssymbol.

Sei $D\Sigma$ wie in Beispiel 15.2 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:

$$\begin{aligned} rec(E_G) \;=\; & \{\delta(in(s)) = \lambda x.\sigma_S^*(par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, \\ & \qquad ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))) \mid s \in S\} \cup \\ & \{\beta(in(s)) = 1 \mid s \in S_{eps}\} \cup \\ & \{\beta(in(s)) = 0 \mid s \in S \setminus S_{eps}\}. \end{aligned}$$

Satz 16.7

Sei $X = Z \cup \bigcup BT$, $g : S \to Lang(X)$ 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 $h : V \to A_g$ $h^*(t) = h^*(t')$ gilt.

Sei $h : V \to A_g$. Für alle $s \in S$,

$$h^*(in(s)) = in^{A_g}(s) = g(s) = g^*(s) \overset{Lemma\ 16.1}{=} g^*((reduce \circ E_G^*)^{k_s}(s)) \tag{6}$$

Lemma 16.6 liefert

$$g^* = (in^{A_g})^* = fold^{A_g} \circ \sigma_S^* : T_{Reg(BL)}(S) \to A. \tag{7}$$

Gehört s nicht zu S_{eps}, dann gilt

$$\begin{aligned} &g^*((reduce \circ E_G^*)^{k_s}(s)) = g^*(par(seq(\overline{B_{s,1}}, t_{s,1}), \ldots, seq(\overline{B_{s,n_s}}, t_{s,n_s}))) \\ &= par^{A_g}(seq^{A_g}(\overline{B_{s,1}}^{A_g}, g^*(t_{s,1})), \ldots, seq^{A_g}(\overline{B_{s,n_s}}^{A_g}, g^*(t_{s,n_s}))) \\ &= \bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})), \end{aligned} \tag{8}$$

also

$$
\begin{aligned}
&h^*(\delta(in(s))) = \delta^{A_g}(h^*(in(s))) \overset{(6)}{=} \delta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s))) \\
&\overset{(8)}{=} \delta^{A_g}(\textstyle\bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i}))) = \lambda x.\delta^{A_g}(\bigcup_{i=1}^{n}(B_{s,i} \cdot g^*(t_{s,i})))(x) \\
&\overset{Def.\ \delta^{A_g}}{=} \lambda x.\{w \in X^* \mid xw \in \textstyle\bigcup_{i=1}^{n}(B_{s,i} \cdot g^*(t_{s,i}))\} \\
&= \lambda x.\{w \in X^* \mid x \in B_{s,i},\ w \in g^*(t_{s,i}),\ 1 \leq i \leq n\} \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s}\{w \in X^* \mid x \in B_{s,i},\ w \in g^*(t_{s,i})\} \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s}\{w \in X^* \mid if\ x \in B_{s,i}\ then\ w \in g^*(t_{s,i})\ else\ w \in \emptyset\} \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s}(if\ x \in B_{s,i}\ then\ g^*(t_{s,i})\ else\ \emptyset) \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s}(if\ x \in B_{s,i}\ then\ g^*(t_{s,i})\ else\ base^A(\emptyset)) \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s}(if\ x \in B_{s,i}\ then\ g^*(t_{s,i})\ else\ g^*(base(\emptyset))) \\
&= \lambda x.\textstyle\bigcup_{i=1}^{n_s} g^*(ite(x \in B_{s,i}, t_{s,i}, base(\emptyset))) \\
&= \lambda x.g^*(par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))) \\
&= g^*(\lambda x.par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))) \\
&\overset{(7)}{=} fold^{A_g}(\sigma_S^*(\lambda x.par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset))))) \\
&= h^*(\sigma_S^*(\lambda x.par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))))
\end{aligned}
$$

und

$$h^*(\beta(in(s))) = \beta^{A_g}(h^*(in(s))) \overset{(6)}{=} \beta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s)))$$
$$\overset{(8)}{=} \beta^{A_g}(\bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i}))) \overset{Def.\ \beta^{A_g}}{=} 0 = h^*(0).$$

Gehört s zu S_{eps}, dann gilt

$$\begin{aligned} &g^*((reduce \circ E_G^*)^{k_s}(s)) = g^*(par(seq(\overline{B_{s,1}}, t_{s,1}), \ldots, seq(\overline{B_{s,n_s}}, t_{s,n_s}), eps)) \\ &= par^{A_g}(seq^{A_g}(\overline{B_{s,1}}^{A_g}, g^*(t_{s,1})), \ldots, seq^{A_g}(\overline{B_{s,n_s}}^{A_g}, g^*(t_{s,n_s}), eps^{A_g})) \qquad (9) \\ &= \bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}, \end{aligned}$$

also

$$h^*(\delta(in(s))) = \delta^{A_g}(h^*(in(s))) \overset{(6)}{=} \delta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s)))$$
$$\overset{(9)}{=} \delta^{A_g}(\bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}) = \lambda x.\delta^{A_g}(\bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\})(x)$$
$$\overset{Def.\ \delta^{A_g}}{=} \lambda x.\{w \in X^* \mid xw \in \bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}\}$$
$$= \lambda x.\{w \in X^* \mid xw \in \bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i}))\}$$
$$\overset{wie\ oben}{=} h^*(\sigma_S^*(\lambda x.par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))))$$

und

$$h^*(\beta(in(s))) = \beta^{A_g}A(h^*(in(s))) \overset{(6)}{=} \beta^{A_g}(g^*((reduce \circ E_G^*)^{k_s}(s)))$$
$$\overset{(9)}{=} \beta^{A_g}(\bigcup_{i=1}^{n_s}(B_{s,i} \cdot g^*(t_{s,i})) \cup \{\epsilon\}) \overset{Def.\ \beta^{A_g}}{=} 1 = h^*(1). \qquad \square$$

Satz 16.8 A_{sol_G} ist die einzige coinduktive Lösung von $E = BRE \cup rec(E_G)$ in $\mathcal{P}(X)$.

Proof. Nach Satz 16.3 (i) ist sol_G eine Lösung von E_G in $Lang(X)$. Also wird $rec(E_G)$ nach Satz 16.7 von A_{sol_G} erfüllt. Nach Beispiel 15.4 wird auch BRE von A_{sol_G} erfüllt. Folglich ist A_{sol_G} nach Satz 15.3 die einzige Lösung von E in $\mathcal{P}(X)$. ❏

Da $\chi : \mathcal{P}(X^*) \to 2^{X^*}$ (siehe 2.10) und die Bijektion $\xi : 2^{X^*} \to DT_{Acc(X)}$ 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 15.3 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 16.9 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 16.7 sind A_{sol_G} und A_{sol} coinduktive Lösungen von E in $\mathcal{P}(X)$. Nach Satz 16.8 stimmen sie miteinander überein. Also sind auch sol_G und sol gleich. ❏

$rec(E_G)$ legt folgende Erweiterung des Brzozowski-Automaten $Bro(BL)$ zur $Reg(BL)'$-Algebra $Bro(BL)'$ nahe:

Für alle $s \in S$,

$$\delta^{Bro(BL)'}(in(s)) = \lambda x.\sigma_S^*(par(ite(x \in B_{s,1}, t_{s,1}, base(\emptyset)), \ldots, ite(x \in B_{s,n_s}, t_{s,n_s}, base(\emptyset)))),$$
$$\beta^{Bro(BL)'}(in(s)) = \mathit{if}\ s \in S_{eps}\ \mathit{then}\ 1\ \mathit{else}\ 0.$$

Sei $Lang(X)' = A_{sol_G}|_{Reg(BL)'}$, $regB' = \chi(A_{sol_G})|_{Reg(BL)'}$, $regA' = \xi(\chi(A_{sol_G}))|_{Reg(BL)'}$ und $\Sigma = Reg(BL)' \cup D\Sigma$.

$Bro(BL)'$ stimmt mit der zur Σ-Algebra erweiterten Termalgebra $T_{Reg(BL)'}$ überein (siehe Satz 15.3). Demnach gelten die Gleichungen

$$fold^{Lang(X)'} = unfold^{Bro(BL)'} : Bro(BL)' \to \mathcal{P}(X), \tag{10}$$
$$fold^{regB'} = unfold^{Bro(BL)'} : Bro(BL)' \to Beh(X, 2), \tag{11}$$
$$fold^{regA'} = unfold^{Bro(BL)'} : Bro(BL)' \to DT_{Acc(X)}, \tag{12}$$

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

$$unfold^{Bro(BL)'} = \xi \circ \chi \circ unfold^{Bro(BL)'} = \xi \circ \chi \circ fold^{Lang(X)'}.$$

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 15.4 bzw. Abschnitt 15.5 gezeigt.

Zusätzlich erhalten wir für alle $s \in S$:

$$unfold^{Bro(BL)'}(in(s)) \overset{(10)}{=} fold^{Lang(X)'}(in(s)) = in^{Lang(X)'}(s) = sol_G(s) = L(G)_s. \tag{14}$$

Also erkennt der initiale Automat $(Bro(BL)', in(s))$ die Sprache von G, realisiert also deren charakteristische Funktion:

$$unfold^{Bro(BL)'}(in(s)) = \chi(unfold^{Bro(BL)'}(in(s))) \overset{(14)}{=} \chi(L(G)_s), \tag{15}$$

und den entsprechenden $Acc(X)$-Coterm:

$$unfold^{Bro(BL)'}(in(s)) = \xi(\chi(unfold^{Bro(BL)'}(in(s)))) \overset{(14)}{=} \xi(\chi(L(G)_s)). \tag{16}$$

Daraus folgt

$$in^{regB'}(s) = fold^{regB'}(in(s)) \overset{(11)}{=} unfold^{Bro(BL)'}(in(s)) \overset{(15)}{=} \chi(L(G)_s), \tag{17}$$

$$in^{regA'}(s) = fold^{regA'}(in(s)) \overset{(12)}{=} unfold^{Bro(BL)'}(in(s)) \overset{(16)}{=} \xi(\chi(L(G)_s)). \tag{18}$$

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.

17 * 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 $\bot$ 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**, $A_\bot =_{def} A + \{\bot\}$, einen ω-CPO: Die zugrundeliegende Halbordnung $\leq$ ist

$$\{(a, b) \in A^2 \mid a = \bot \vee a = b\}.$$

Der ω-CPO der partiellen Funktionen von A nach B

Für alle $f, g : A \rightharpoonup B$,

$$f \leq g \quad \Leftrightarrow_{def} \quad def(f) \subseteq def(g) \wedge \forall\, a \in def(f) : f(a) = g(a).$$

Die **nirgends definierte Funktion** $\Omega : A \rightharpoonup B$ mit $def(\Omega) =_{def} \emptyset$ ist das kleinste Element von $A \rightharpoonup B$ bzgl. $\leq$.

Jede ω-Kette $f_0 \leq f_1 \leq f_2 \leq \ldots$ von $A \rightharpoonup\!\!\!\!\circ\; B$ hat das folgende Supremum: Für alle $a \in A$,

$$(\bigsqcup_{i \in \mathbb{N}} f_i)(a) \;=_{def}\; \begin{cases} f_i(a) & \text{falls } \exists\, i \in \mathbb{N} : a \in def(f_i), \\ \text{undefiniert} & \text{sonst.} \end{cases}$$

Ein Produkt $A_1 \times \cdots \times A_n$ 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 $A_1, \ldots, A_n$ bzw. A werden – wie im Abschnitt Kongruenzen und Quotienten beschrieben – zur Halbordnung auf $A_1 \times \cdots \times A_n$ bzw. A^B geliftet.

Das kleinste Element von $A_1 \times \cdots \times A_n$ ist das n-Tupel der kleinsten Elemente von $A_1, \ldots, A_n$. $\lambda x.\bot$ ist das kleinste Element von A^B, wobei $\bot$ das kleinste Element von B ist.

Suprema sind komponenten- bzw. argumentweise definiert: Für alle ω-Ketten
$a_0 = (a_{0,1}, \ldots, a_{0,n}) \leq a_1 = (a_{1,1}, \ldots, a_{1,n}) \leq a_2 \leq \ldots$ von $A_1 \times \cdots \times A_n$,

$$\bigsqcup_{i \in \mathbb{N}} a_i \;=_{def}\; (\bigsqcup_{i \in \mathbb{N}} a_{i,0}, \ldots, \bigsqcup_{i \in \mathbb{N}} a_{i,n}). \tag{2}$$

Für alle ω-Ketten $f_0 \leq f_1 \leq f_2 \leq \ldots$ von B^A und $a \in A$,

$$(\bigsqcup_{i \in \mathbb{N}} f_i)(a) \;=_{def}\; \bigsqcup_{i \in \mathbb{N}} f_i(a). \tag{3}$$

Seien A, B halbgeordnete Mengen. Eine Funktion $f : A \to B$ 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 $\bot_A$ von A auf das kleinste Element $\bot_B$ von B abbildet. Ist A ein Produkt, dann bildet eine strikte Funktion f nicht nur $\bot_A$, sondern jedes Tupel von A, das ein kleinstes Element enthält, auf $\bot_B$ 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 $A \to_c B$ 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 $\bot_{A,s}$ 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^A ω-stetig ist.

Fixpunktsatz von Kleene

Sei A ein ω-CPO und $f : A \to A$ ω-stetig. Da f monoton ist, erhalten wir die ω-Kette

$$\bot \leq f(\bot) \leq f^2(\bot) \leq \ldots.$$

Deren Supremum

$$lfp(f) =_{def} \bigsqcup_{n \in \mathbb{N}} f^n(\bot)$$

ist der (bzgl. $\leq$) kleinste Fixpunkt von f (siehe [35]). ❑

Anwendung auf iterative Gleichungssysteme

Sei $E : V \to T_\Sigma(V)$ ein iteratives Σ-Gleichungssystem (siehe Kapitel 16) 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 $E_\mathcal{A} : A^V \to A^V$ 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 [8], Proposition 4.13, ist $E_{\mathcal{A}}$ ω-stetig. Also folgt aus dem Fixpunktsatz von Kleene, dass

$$lfp(E_{\mathcal{A}}) \;=\; \bigsqcup_{n\in\mathbb{N}} E_{\mathcal{A}}^{n}(\lambda x.\bot_A) \tag{1}$$

der kleinste Fixpunkt von $E_{\mathcal{A}}$ in A ist.

Für alle strikten und ω-stetigen Σ-Homomorphismen $h : \mathcal{A} \to \mathcal{B}$ gilt:

$$h \circ lfp(E_{\mathcal{A}}) \;=\; lfp(E_{\mathcal{B}}(h)) \tag{2}$$

(siehe [35], Lemma ITER (ii).)

Iterative Gleichungssysteme dienen u.a. der Spezifikation partieller Funktionen. Sind z.B. zwei partielle Funktionen f, g gegeben, dann ist die Gleichung $h = g \circ h \circ f$ zunächst nur eine *Bedingung* an eine dritte Funktion h. Die aus der Gleichung gebildete Schrittfunktion

$$\lambda h.(g \circ h \circ f) : (A \multimap B) \to (A \multimap B)$$

ist ω-stetig und hat deshalb nach dem Fixpunktsatz von Kleene einen kleinsten Fixpunkt, der als **denotationelle Semantik** von h bezeichnet wird.

17.1 Schleifensemantik

Sei Σ die abstrakte Syntax von JavaLight (siehe Beispiel 4.6) zusammen mit zwei Konstanten $e : 1 \to \mathit{Disjunct}$ und $c : 1 \to \mathit{Command}$. Sei Sem das Zustandsmodell von JavaLight (siehe 4.9) zusammen mit festen Interpretationen $p \in \mathit{javaState}_{\mathit{Disjunct}}$ und $f \in \mathit{javaState}_{\mathit{Command}}$ von e bzw. c.

Sem ist ω-stetig.

Sei $V = \{x : \mathit{Command}\}$, E das iterative Σ-Gleichungssystem

$$\begin{aligned} E : V &\to T_\Sigma(V) \\ x &\mapsto \mathit{cond}(e, \mathit{block}(\mathit{seq}(c, x)), \mathit{id}) \end{aligned}$$

und E_{Sem} die Schrittfunktion von E (s.o.). Der kleinste Fixpunkt von E_{Sem} liefert uns die Interpretation von $\mathit{loop}(e, c)$ in Sem:

$$\mathit{loop}^{\mathit{Sem}}(p, f) =_{\mathit{def}} \mathit{lfp}(E_{\mathit{Sem}})(x) : \mathit{Store} \multimap \mathit{Store}.$$

Die Haskell-Funktion

```
loop :: St Bool -> St Store -> St Store
loop p f = cond p (loop p f . f) id
```

von Abschnitt 9.5 implementiert $\mathit{loop}^{\mathit{Sem}}$. Für alle $st \in \mathit{Store}$ terminiert $\mathit{loop}(p)(f)(st)$ genau dann, wenn $\mathit{loop}^{\mathit{Sem}}(p, f)(st)$ definiert ist. ❑

17.2 Semantik der Assemblersprache $StackCom^*$

Auch jede Befehlsfolge $cs \in StackCom^*$ lässt sich als iteratives Gleichungssystem $eqs(cs)$ darstellen.

Die konstruktive Signatur $\Sigma(StackCom)$, aus deren Operationen die Terme von $eqs(cs)$ gebildet sind, lautet wie folgt:

$$\begin{aligned}
\Sigma(StackCom) &= (\ \{com\}, \{\mathbb{Z}, String\}, F\ ), \\
F &= \{\ push : \mathbb{Z} \to com, \\
&\quad load, save, cmp : String \to com, \\
&\quad pop, add, sub, mul, div, or, and, inv, nix : 1 \to com, \\
&\quad cons, fork : com \times com \to com\ \}.
\end{aligned}$$

$push, load, save, cmp, pop, add, sub, mul, div, or, and, inv$ 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 $eqs : StackCom^* \to Eqs$ übersetzt Assemblerprogramme in solche Gleichungssysteme:

Sei $cs \in StackCom^*$ und $V(cs)$ die Menge der Nummern der Sprungziele von cs, also

$$V(cs) = \{0\} \cup \{n < |cs| \mid Jump(n) \in cs \vee JumpF(n) \in cs\}.$$

$$\begin{aligned} eqs(cs) : V(cs) &\to T_{\Sigma(StackCom)}(V(cs)) \\ n &\mapsto mkTerm(drop(n)(cs)) \end{aligned}$$

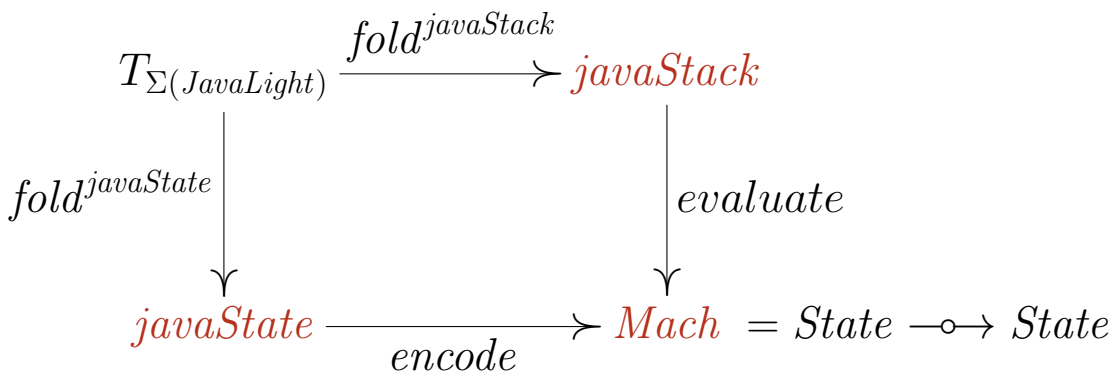
$$\begin{aligned} mkTerm : StackCom^* &\to T_{\Sigma(StackCom)}(V(cs)) \\ \epsilon &\mapsto nix \\ c : cs' &\mapsto \begin{cases} n & \text{falls } c = Jump(n) \wedge n < |cs| \\ nix & \text{falls } c = Jump(n) \wedge n \geq |cs| \\ fork(n, mkTerm(cs')) & \text{falls } c = JumpF(n) \wedge n < |cs| \\ fork(nix, mkTerm(cs')) & \text{falls } c = JumpF(n) \wedge n \geq |cs| \\ cons(c, mkTerm(cs')) & \text{sonst} \end{cases} \end{aligned}$$

Beispiel

Das Assemblerprogramm von Beispiel 10.6 zur Berechnung der Fakultätsfunktion lautet als iteratives $\Sigma(StackCom)$-Gleichungssystem wie folgt:

$$
\begin{aligned}
E : \{0,3\} &\to T_{\Sigma(StackCom)}(\{0,3\}) \\
0 &\mapsto cons(push(1), cons(save(fact), cons(pop, 3))) \\
3 &\mapsto cons(load(x), cons(push(1), cons(cmp(>), fork(nix, \\
&\quad cons(load(fact), cons(load(x), cons(mul, cons(save(fact), \\
&\quad cons(pop, cons(load(x), cons(push(1), cons(sub, cons(save(x), \\
&\quad cons(pop, 3)))))))))))))))
\end{aligned}
$$

In Abschnitt 14.5 haben wir informell gezeigt, dass $compJava(javaStack)$ semantisch korrekt ist, dass also folgendes Diagramm kommutiert:



Mach kann zu einer ω-stetigen $\Sigma(StackCom)$-Algebra erweitert werden:

Für alle $f, g : State \multimap State$, $\otimes \in \{add, sub, mul, div, or, and\}$,
$(stack, store) \in State$, $a \in \mathbb{Z}$ und $x \in String$,

$$push^{Mach}(i)(stack, store) = (a : stack, store),$$

$$load^{Mach}(x)(stack, store) = (store(x) : stack, store),$$

$$save^{Mach}(x)(stack, store) = (stack, update(store)(x)(a)),$$

$$cmp^{Mach}(x)(stack, store) = \begin{cases} (1 : s, store) & \text{falls } \exists\, a, b, s : stack = a : b : s \\ & \qquad \wedge\, evalRel(x)(a)(b), \\ (0 : s, store) & \text{falls } \exists\, a, b, s : stack = a : b : s \\ & \qquad \wedge\, \neg evalRel(x)(a)(b), \\ \text{undefiniert} & \text{sonst}, \end{cases}$$

$$pop^{Mach}(stack, store) = \begin{cases} (s, store) & \text{falls } \exists\, a, s : stack = a : s, \\ \text{undefiniert} & \text{sonst}, \end{cases}$$

$$\otimes^{Mach}(stack, store) = \begin{cases} (op(\otimes)(b, a) : s, store) & \text{falls } \exists\, a, b, s : stack = a : b : s, \\ \text{undefiniert} & \text{sonst}, \end{cases}$$

$$inv^{Mach}(stack, store) = \begin{cases} ((a + 1)\ mod\ 2 : stack, store) & \text{falls } \exists\, a, s : stack = a : s, \\ \text{undefiniert} & \text{sonst}, \end{cases}$$

$$nix^{Mach}(stack, store) = (stack, store), cons^{Mach}(f, g) = g \circ f,$$

$$fork^{Mach}(f, g)(stack, store) = \begin{cases} f(s, store) & \text{falls } \exists\ s : stack = 0 : s, \\ g(s, store) & \text{falls } \exists\ a, s : stack = a : s \wedge a \neq 0, \\ \text{undefiniert} & \text{sonst}, \end{cases}$$

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 10.5: Für alle $cs \in StackCom^*$ und $n \in V(cs)$,

$$execute(cs) \ = \ lfp(eqs(cs)_{Mach}) : V(cs) \to Mach. \tag{4}$$

Im Folgenden wird $Mach$ so zu einer JavaLight-Algebra erweitert, dass $execute$ und $encode$ JavaLight-homomorph werden und aus der Initialität von $T_{\Sigma(JavaLight)}$ die Kommutativität von (1) folgt (siehe Kapitel 5).

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 $f : State \multimap State$,

$$assign^{Mach}(x, f) = pop^{Mach} \circ save^{Mach}(x) \circ f.$$

Für alle $f, g, h : State \multimap State$, $cond^{Mach}(f, g, h) = fork^{Mach}(h, g) \circ f$.

Für alle $f, g : State \multimap State$, $cond1^{Mach}(f, g) = fork^{Mach}(id_{Mach}, g) \circ f$.

Für alle $f, g : State \multimap State$, $loop^{Mach}(f, g) = lfp(E_{Mach})(x)$, wobei

$$\begin{aligned} E : \{x\} &\to T_{\Sigma(StackCom)}(\{x, f, g\}) \\ x &\mapsto cons(f, fork(nix, cons(g, x))) \end{aligned}$$

Für alle $f, g : State \multimap State$,

$$\begin{aligned} sumsect^{Mach}(+, f, g) &= g \circ add^{Mach} \circ f, \\ sumsect^{Mach}(-, f, g) &= g \circ sub^{Mach} \circ f, \\ prodsect^{Mach}(*, f, g) &= g \circ mul^{Mach} \circ f, \\ prodsect^{Mach}(/, f, g) &= g \circ div^{Mach} \circ f. \end{aligned}$$

$embedI^{Mach} = push^{Mach}$.

$var^{Mach} = load^{Mach}$.

Für alle $f : State \dashrightarrow State$, $not^{Mach}(f) = inv^{Mach} \circ f$.

Für alle $x \in String$ und $f, g : State \dashrightarrow State$,

$$atom^{Mach}(f, x, g) = cmp^{Mach}(x) \circ g \circ f.$$

Für alle $b \in 2$, $embedB^{Mach}(b) = push^{Mach}(b)$.

17.3 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.

Seien B, X, ST, C' und V wie in Abschnitt 2.7.

Mengen $CT_\Sigma^\perp(V)$ **geordneter Σ-Term über** V und $T_\Sigma^\perp(V)$ sind ebenso definiert wie $CT_\Sigma(V)$ bzw. $T_\Sigma(V)$, außer dass für alle $s \in S \cup ST$ V_s durch $V_s \cup \{\Omega\}$ ersetzt ist.

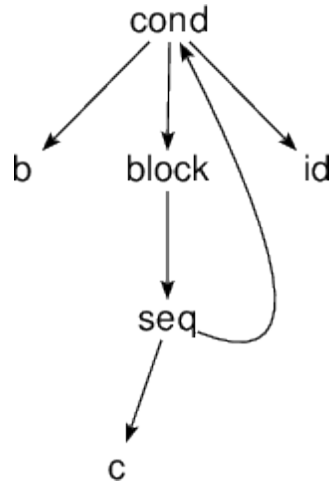
Die Elemente von $CT_\Sigma^\perp =_{def} CT_\Sigma^\perp(\lambda s.\emptyset)$ und $T_\Sigma^\perp =_{def} T_\Sigma^\perp(\lambda s.\emptyset)$ heißen **geordnete Σ-Grundterme**.

Satz 17.3 $CT_\Sigma^\perp$ ist initial in $PAlg_\Sigma$. $CT_\Sigma^\perp$ ist initial in $CAlg_\Sigma$ (siehe Abschnitt Continuous algebras in [35], Kapitel 14).

Satz 17.4 Jedes iterative Σ-Gleichungssystem $E : V \to T_\Sigma(V)$ hat genau eine Lösung in $CT_\Sigma^\perp$ (siehe [35], Kapitel 15).

Beispiel *loop* Sei $\Sigma = \Sigma(JavaLight)$. Die eindeutige Lösung $sol : \{x\} \to CT_\Sigma(\{b, c\})$ des iterativen Σ-Gleichungssystems $E : \{x\} \to T_\Sigma(\{x, b, c\})$ von Abschnitt 17.1 hat folgende Graphdarstellung: Für alle $w \in \mathbb{N}^*$,

$$sol(x)(w) = \begin{cases} cond & \text{falls } w \in (212)^* \\ b & \text{falls } w \in (212)^*1 \\ block & \text{falls } w \in (212)^*2 \\ id & \text{falls } w \in (212)^*3 \\ seq & \text{falls } w \in (212)^*21 \\ c & \text{falls } w \in (212)^*211 \end{cases}$$



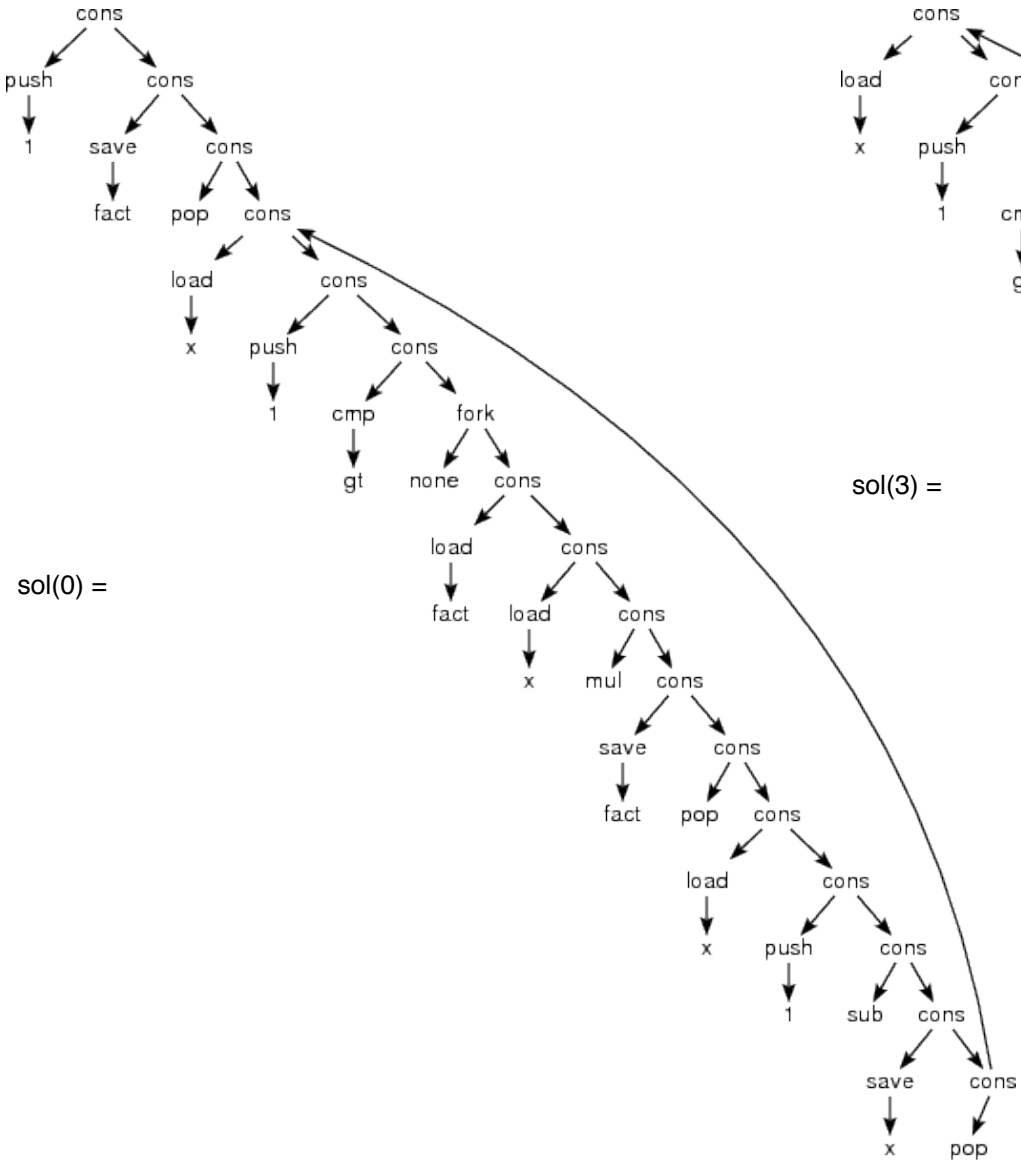
Beispiel *StackCom*

Sei $\Sigma = \Sigma(\mathit{StackCom})$ und cs das Assemblerprogramm von Beispiel 10.6. Die eindeutige Lösung $sol : \{0,3\} \to CT_\Sigma$ des iterativen Σ-Gleichungssystems

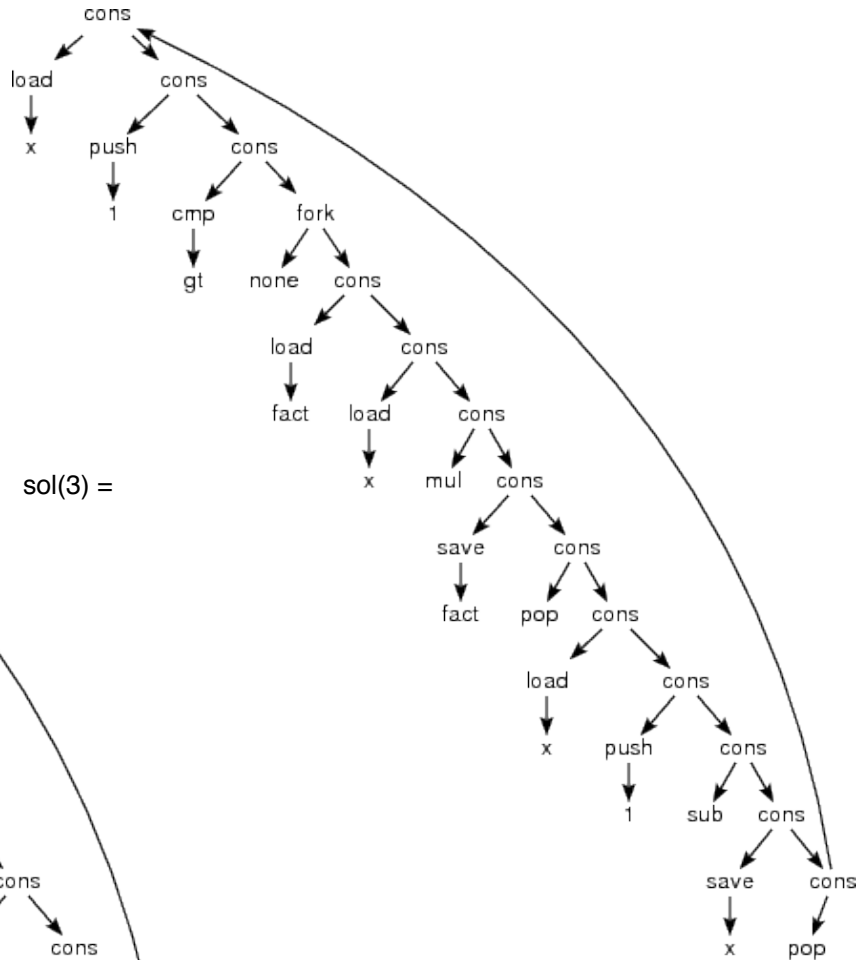
$$eqs(cs) : \{0,3\} \to T_\Sigma(\{0,3\})$$

von Abschnitt 17.2 hat folgende Graphdarstellung:

sol(0) =


cons
push
cons
1
save
cons
fact
pop
cons
load
cons
x
push
cons
1
cmp
fork
gt
none
cons
load
cons
fact
load
cons
x
mul
cons
save
cons
fact
pop
cons
load
cons
x
push
cons
1
sub
cons
save
cons
x
pop

sol(3) =


cons
load
cons
x
push
cons
1
cmp
fork
gt
none
cons
load
cons
fact
load
cons
x
mul
cons
save
cons
fact
pop
cons
load
cons
x
push
cons
1
sub
cons
save
cons
x
pop

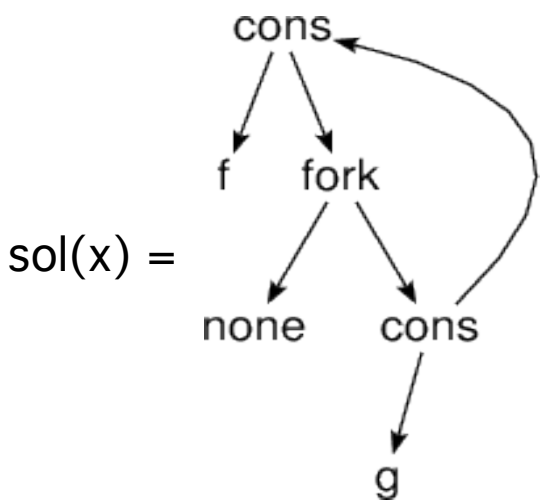
Offenbar repräsentiert jeder Pfad des Σ-Baums $sol(0)$ eine – möglicherweise unendliche – Kommandofolge, die einer Ausführungssequenz von cs entspricht. ❑

Beispiel

Sei $\Sigma = \Sigma(StackCom)$. Die eindeutige Lösung $sol : \{x\} \to CT_\Sigma(\{b, c\})$ des iterativen Σ-Gleichungssystems

$$E : \{x\} \to T_\Sigma(\{x, b, c\}),$$

dessen kleinste Lösung in $Mach$ den Konstruktor $loop$ von $\Sigma(StackCom)$ in $Mach$ interpretiert (siehe 17.2), hat folgende Graphdarstellung:



Sei $cs \in StackCom^*$. Da $fold^{Mach}$ strikt, ω-stetig und Σ-homomorph ist, folgt

$$execute'(cs) \;=\; fold^{Mach} \circ lfp(eqs(cs)_{CT_\Sigma}) : V(cs) \to Mach \tag{7}$$

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 $Mach$. ❏

Die eindeutige Lösung eines iterativen Σ-Gleichungssystems E in CT_Σ lässt sich nicht nur als kleinsten Fixpunkt von E_{CT_Σ} 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:

$$co\Sigma \;=\; (S, \{d_s : s \to \coprod_{f:e \to s \in F} e \mid s \in S\}).$$

Satz 17.5

CT_Σ ist eine finale $co\Sigma$-Algebra (siehe Abschnitt From constructors to destructors in [35], Kapitel 14).

Sei $E : V \to T_\Sigma(V)$ ein iteratives Σ-Gleichungssystem.

E induziert die folgende $co\Sigma$-algebra $T(E)$:

- Für alle $s \in S$, $T(E)_s = T_\Sigma(V)_s$.
- Für alle $c : \prod_{i\in I} e_i \to s \in C$ und $(t_i)_{i\in I} \in \mathsf{X}_{i\in I} T_\Sigma(V)_{e_i}$, $d_s^{T(E)}(c\{i \to t_i\}) = ((t_i)_{i\in I}, c)$.
- Für alle $s \in S$ und $x \in V_s$, $d_s^{T(E)}(x) = d_s^{T(E)}(E(x))$.

(8) $V \overset{inc_V}{\to} T_\Sigma(V) \overset{unfold^{T(E)}}{\to} CT_\Sigma$ löst E in CT_Σ (siehe [35], Kapitel 15).

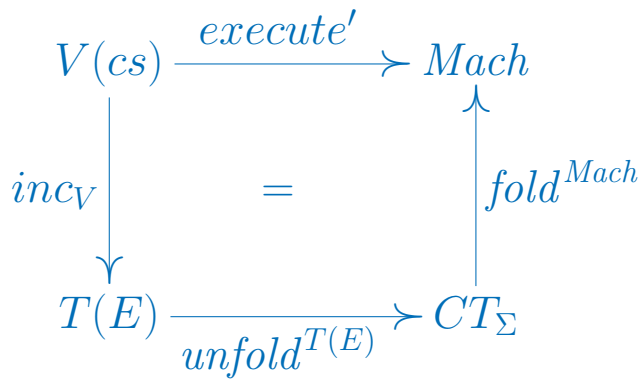
(9) $g : V \to CT_\Sigma$ löst E genau dann in CT_Σ, wenn $g^* : T(E) \to CT_\Sigma$ $co\Sigma$-homomorph ist (siehe [35], Kapitel 15).

Satz 17.6 Jedes iterative Σ-Gleichungssystem $E : V \to T_\Sigma(V)$ hat genau eine Lösung in CT_Σ.

Beweis. Wegen (8) hat E eine Lösung in CT_Σ. Angenommen, $g, h : V \to CT_\Sigma$ lösen E in CT_Σ. Dann sind $g^*, h^* : T(E) \to CT_\Sigma$ wegen (9) $co\Sigma$-homomorph. Da CT_Σ eine finale $co\Sigma$-Algebra ist, gilt $g^* = h^*$, also $g = g^* \circ inc_V = h^* \circ inc_V = h$. ❑

Beispiel

Sei $cs \in StackCom^*$. Aus (7), (8) und Satz 17.4 (oder 17.6) folgt, dass $execute'$ die Entfaltung von $V(cs)$ in CT_Σ mit der Faltung der entstandenen Σ-Terme in $Mach$ verknüpft:



18 Literatur

[1] R. Bird, *Introduction to Functional Programming*, Prentice Hall 1998

[2] L.S. Bobrow, M.A. Arbib, *Discrete Mathematics: Applied Algebra for Computer and information Science*, W.B. Saunders Company 1974

[3] M. Bonsangue, J. Rutten, A. Silva, *An Algebra for Kripke Polynomial Coalgebras*, Proc. 24th LICS (2009) 49-58

[4] J.A. Brzozowski, *Derivatives of regular expressions*, Journal ACM 11 (1964) 481–494

[5] H. Comon et al., *Tree Automata: Techniques and Applications*, Inria 2008

[6] F. Drewes, ed., *Tree Automata*, Course notes, Umeå University, Sweden 2009

[7] J. Gibbons, R. Hinze, *Just do It: Simple Monadic Equational Reasoning*, Proc. 16th ICFP (2011) 2-14 Jeremy Gibbons and Ralf Hinze

[8] J.A. Goguen, J.W. Thatcher, E.G. Wagner, J.B. Wright, *Initial Algebra Semantics and Continuous Algebras*, Journal ACM 24 (1977) 68-95

[9] H.H. Hansen, J. Rutten, *Symbolic Synthesis of Mealy Machines from Arithmetic Bitstream Functions*, Scientific Annals of Computer Science (2010) 97-130

[10] R. Hinze, *Functional Pearl: Streams and Unique Fixed Points*, Proc. 13th ICFP (2008) 189-200

[11] R. Hinze, D.W.H. James, *Proving the Unique-Fixed Point Principle Correct*, Proc. 16th ICFP (2011) 359-371

[12] 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

[13] G. Hutton, *Fold and unfold for program semantics*, Proc. 3rd ICFP (1998) 280-288

[14] B. Jacobs, *Introduction to Coalgebra*, draft, Radboud University Nijmegen 2012

[15] 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

[16] B. Klin, *Bialgebras for structural operational semantics: An introduction*, Theoretical Computer Science 412 (2011) 5043-5069

[17] D. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory 2 (1968) 127-145; Corrections: Math. Systems Theory 5 (1971) 95-96

[18] D. Kozen, *Realization of Coinductive Types*, Proc. Math. Foundations of Prog. Lang. Semantics 27, Carnegie Mellon University, Pittsburgh 2011

[19] C. Kupke, J. Rutten, *On the final coalgebra of automatic sequences*, in: Logic and Program Semantics, Springer LNCS 7230 (2012) 149-164

[20] A. Kurz, *Specifying coalgebras with modal logic*, Theoretical Computer Science 260 (2001) 119–138

[21] P.J. Landin, *The Mechanical Evaluation of Expressions*, Computer Journal 6 (1964) 308–320

[22] W. Martens, F. Neven, Th. Schwentick, *Simple off the shelf abstractions for XML Schema*, SIGMOD Record 36,3 (2007) 15-22

[23] K. Meinke, J.V. Tucker, *Universal Algebra*, in: Handbook of Logic in Computer Science 1 (1992) 189 - 368

[24] E. Moggi, *Notions of Computation and Monads*, Information and Computation 93 (1991) 55-92

[25] F.L. Morris, *Advice on Structuring Compilers and Proving Them Correct*, Proc. ACM POPL (1973) 144-152

[26] 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

[27] R.N. Moll, M.A. Arbib, A.J. Kfoury, *Introduction to Formal Language Theory*, Springer (1988)

[28] P. Padawitz, *Church-Rosser-Eigenschaften von Graphgrammatiken und Anwendungen auf die Semantik von LISP*, Diplomarbeit, TU Berlin 1978

[29] P. Padawitz, *Formale Methoden des Systementwurfs*, TU Dortmund 2015

[30] P. Padawitz, *Übersetzerbau*, TU Dortmund 2015

[31] P. Padawitz, *Modellieren und Implementieren in Haskell*, TU Dortmund 2015

[32] P. Padawitz, *Logik für Informatiker*, TU Dortmund 2015

[33] 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)

[34] P. Padawitz, *From Modal Logic to (Co)Algebraic Reasoning*, TU Dortmund 2013

[35] P. Padawitz, *Fixpoints, Categories, and (Co)Algebraic Modeling*, TU Dortmund

[36] P. Padawitz, *FCAM - old stuff*, TU Dortmund

[37] H. Reichel, *An Algebraic Approach to Regular Sets*, in: K. Futatsugi et al., Goguen Festschrift, Springer LNCS 4060 (2006) 449-458

[38] W.C. Rounds, *Mappings and Grammars on Trees*, Mathematical Systems Theory 4 (1970) 256-287

[39] J. Rutten, *Processes as terms: non-wellfounded models for bisimulation*, Math. Struct. in Comp. Science 15 (1992) 257-275

[40] J. Rutten, *Automata and coinduction (an exercise in coalgebra)*, Proc. CONCUR '98, Springer LNCS 1466 (1998) 194–218

[41] J. Rutten, *Behavioural differential equations: a coinductive calculus of streams, automata, and power series*, Theoretical Computer Science 308 (2003) 1-53

[42] J. Rutten, *A coinductive calculus of streams*, Math. Struct. in Comp. Science 15 (2005) 93-147

[43] A. Silva, J. Rutten, *A coinductive calculus of binary trees*, Information and Computation 208 (2010) 578–593

[44] D. Turi, G. Plotkin, *Towards a Mathematical Operational Semantics*, Proc. LICS 1997, IEEE Computer Society Press (1997) 280–291

[45] 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

[46] 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

[47] Ph. Wadler, *Monads for Functional Programming*, Proc. Advanced Functional Programming, Springer LNCS 925 (1995) 24-52

[48] R. Wilhelm, H. Seidl, *Übersetzerbau - Virtuelle Maschinen*, Springer 2007

19 Index