Zusammenfassung Übau

Richard Stewing

8. September 2017

Inhaltsverzeichnis

1 Weg zum generischen Compiler

- kontextfreie Sprache → Parser → Termalgebra → generische Faltung → Algebra

2 Algebraische Modellierung

2.1 Grundlagen

Seien A, B Mengen, f: A → B, C $\subseteq A, D \subseteq B, n > 0$.

- $\emptyset$: leere Menge
- 1: $\{\epsilon\}$
- 2: $\{0,1\}$
- [n]: $\{1,..,n\}$
- $\Delta_A^n =_{def} \{(a_1,..,a_n) \in A^n \mid \forall\ 1 \leq i < n: a_i = a_{i+1}\}$
- $f(C) =_{def} \{f(a) \mid a \in C\} = P(f)(C)$
- $img(f) =_{def} f(A)$
- $f^{-1}(D) =_{def} \{a \in A \mid f(a) \in D\}$
- $ker(f) =_{def} \{(a,a') \in A \times A \mid f(a) = f(a')\}$
- $graph(f) =_{def} \{(a, f(a)) \in A \times B \mid a \in A\}$
- $f|_C$: $C \to B = c \to f(c)$
- $f[b/a]$: $A \to B = c \to$ if $c = a$ then b else $f(c)$
- χ: $P(A) \to 2^A = C \to \lambda$ a. if $a \in C$ then 1 else 0
- f ist surjektiv $\Leftrightarrow_{def} img(f) = B$

- f ist injektiv $\Leftrightarrow_{\mathrm{def}}$ $\ker(f) = \Delta^2_A$
- f ist bijektiv $\Leftrightarrow_{\mathrm{def}}$ $\exists$ g: B $\rightarrow$ A: g . f = id_A f . g = id_B
- A $\cong$ B $\Leftrightarrow_{\mathrm{def}}$ $\exists$ f: A $\rightarrow$ B und f ist bijektiv

2.2 Mehrsortige Mengen

- Sei S eine Menge. Ein Tuple $A = (A_s)_{s \in S}$ ist dann eine S-indizierte(sortige) Menge.
- Seien $A = (A_s)_{s \in S}$ und $B = (B_s)_{s \in S}$ S-sortige Mengen. f: A $\rightarrow$ B ist eine S-sortige Menge $(f_s)_{s \in S}$, so dass $\forall$ s $\in$ S. f_s:$A_s \rightarrow B_s$.
- Seien n $>$ 0. Eine n-stellige S-sortige Relation auf A ist eine S-sortige Menge $R = (R_s)_{s \in S}$ mit $R_s \subseteq A^n_s$ für alle s$\in$ S.
- Im Fall n = 1, heißt R S-sortige Teilmenge von A genannt.
- kartesisches Produkt einer I-sortigen Menge: $\times_{i \in I} A_i = \{f{:}I \rightarrow \cup_{i \in I} A_i \mid \forall\, i \in I : f(i) \in A_i\}$
- Gibt es eine Menge A mit $A_i = A$ für alle i$\in$ I, dann stimmt $\times_{i \in I} A_i$ offenbar mit A^I überein. Im Fall I = [n] für ein n $>$ 1 schreibt man auch $A_1 \times \ldots \times A_n$ anstelle von $\times_{i \in I} A_i$ und A^n snatelle von A^I.
- disjunkte Vereinigung: $\uplus_{i \in I} A_i = \cup_{i \in I}(A_i \times \{i\})$
- Im Fall I = [n] für ein n $>$ 1 schreibt man auch $A_1 + .. + A_n$ anstelle von $\uplus_{i \in I} A_i$.
- $A^+ =_{\mathrm{def}} \cup_{n>0} A^n$
- $1 =_{\mathrm{def}} \{\epsilon\}$
- $A^* =_{\mathrm{def}} 1 \cup A^+$
- $\cdot$: $A^* \times A^* \rightarrow A^*$ und $\cdot$: $P(A^*) \times P(A^*) \rightarrow P(A^*)$ ist die Konkatenationen von Strings

2.3 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**.

- Jede Menge mit den universellen Eigenschaften des kartesischen Produkts nennt man *Produkt von A*
- Jede Menge mit den universellen Eigenschaften der disjunkten Vereinigung nennt man *Summe von A*

2.3.1 Produkt

Sei $\mathrm{A} = (\mathrm{A_i})_{\mathrm{i \in I}}$ ein Mengentuple, P eine Menge und $\pi = (\pi_\mathrm{i}\colon \mathrm{P} \to \mathrm{A_i})_{\mathrm{i \in I}}$ ein Funktionstuple. Das Paar (P, π) heißt **Produkt von A**, wenn es für alle Tuple $(\mathrm{f_i}\colon \mathrm{B} \to \mathrm{A_i})_{\mathrm{i \in I}}$ genau eine Funktion f: B $\to$ P gibt derart, dass für alle i$\in$ I Folgendes gilt:

$$\pi_\mathrm{i} \,.\, \mathrm{f} = \mathrm{f_i}$$

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

Demnach gilt:

$$(\forall \ \mathrm{i} \in \mathrm{I} : \pi_\mathrm{i} \,.\, \mathrm{f} = \pi_\mathrm{i} \bigcirc g) \Rightarrow \mathrm{f} = \mathrm{g}$$

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

Die Projektion und Produktextensionen für $\times_{\mathrm{i \in I}}\mathrm{A_i}$ sind wie folgt definiert:

- Für alle i$\in$ I und f$\in$ $\times_{\mathrm{i \in I}}\mathrm{A_i}$, $\pi_\mathrm{i}(\mathrm{f}) =_{\mathrm{def}} \mathrm{f(i)}$
- Für alle $(\mathrm{f_i}\colon \mathrm{B} \to \mathrm{A_i})_{\mathrm{i \in I}}$, b$\in$ B und i $\in$ I, $\langle f_\mathrm{i} \rangle_{\mathrm{i \in I}}(\mathrm{b})(\mathrm{i}) =_{\mathrm{def}} \mathrm{f_i(b)}$.

1. Satz 2.2 Produkte sind *bis auf Isomorphie* eindeutig:
   - Seien (P, π) und (P', π') Produkte von A. Dann sind P und P' isomorph.
   - Seien (P, π) ein Produkt von A, P' eine Menge und h: P' $\to$ P eine bijektive Abbildung. Dann ist (P', π') mit $\pi' = (\pi \,.\, \mathrm{h})_{\mathrm{i \in I}}$ ebenfalls ein Produkt von A.

Die Funktion

$$\Pi_{i \in I} f_i =_{def} \langle f_i \,.\, \pi_i \rangle : P \rightarrow P'$$

heißt **Produkt von f**.

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

$$f^I =_{def} \Pi_{i \in I}\, f_i,$$
$$f_1 \times .. \times f_n =_{def} \Pi_{i \in [n]}\, f_i.$$

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

$$\langle f_i \rangle_{i \in I} \,.\, f = \langle f_i \,.\, f \rangle_{i \in I},$$
$$\pi_k \,.\, \Pi_{i \in I} g_i = g_k\, \pi_k,$$
$$(\Pi_{i \in I} h_i) \,.\, \langle f_i \rangle_{i \in I} = \langle h_i \,.\, f_i \rangle_{i \in I}.$$

2.3.2 Summe

Vom Produkt kommt man zur Summe, indem man alle Funktionspfeile umdreht: Sei A $= (A_i)_{i \in I}$ ein Mengentuple, S eine Menge und $\iota = (\iota_i\colon A_i \rightarrow S)_{i \in I}$ ein Funktionstuple. Das Paar (S, ι) heißt **Summe** oder **Coprodukt von A**, wenn es für alle Tuple $(f_i\colon A_i \rightarrow B)_{i \in I}$ genau eine Funktion f: S $\rightarrow$ B gibt mit

$$f \,.\, \iota_i = f_i$$

für alle i $\in$ I.

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

Demnach gilt:

$$(\forall\ i \in I : f \,.\, \iota_i = g \,.\, \iota_i) \Rightarrow f = g$$

1. Satz 2.3 Summen sind *bis auf Isomorphie* eindeutig:
   - Seien (S, ι) und (S', ι') Summen von A. Dann sind P und P' isomorph.

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

Sei (S, ι) eine Summe von $(A_i)_{i\in I}$, ein (S', ι') eine Summe von $(B_i)_{i\in I}$ und $f = (f_i\colon A_i \rightarrow B_i)_{i\in I}$ Die Funktion

$$\coprod_{i\in I} f_i =_{def} [\iota_i' \,.\, f_i]\colon S \rightarrow S'$$

heißt Summe von f.

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

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

2.4 Typen und Signaturen

Sei S eine Menge von - **Sorten** genannten - Symbolen. Die Klasse $T_p(S)$ der **polynomialen Typen über** S:

- $S \subseteq T_p(S)$
- Jede nichtleere Menge ist ein polynomialer Typ
- Für alle nichtleeren Mengen I und $(e_i)_{i\in I} \in T_p(S)^I$, $\Pi_{i\in I}\, e_i$, $\coprod_{i\in I} e_i \in T_p(S)$
- Ein Typ der Form $\Pi_{i\in I} e_i$ heißt **I-stelliger Produkttyp** mit den **Faktoren** e_i, $i\in I$.
- Ein Typ der Form $\coprod_{i\in I} e_i$ heißt **I-stelliger Summentyp** mit den **Summanden** e_i, $i\in I$.

Für alle n>0, e_1,..,e_n, $e\in T_p(S)$ und nichtleere Mengen I,

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

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

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

Für alle $e \in T_p(S)$ und $a \in A_e$ nennen wir e den Typen von a.

Eine S-sortige Funktion h: $A \to B$ wird wie folgt zur $T_p(S)$-sortigen Menge für alle nichtleeren Mengen I und $(e_i)_{i \in} \in T_p(S)^I$.

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

Für alle S-sortigen Mengen A, S-sortige Funktionen h: $A \to B$, $n>0$, nichtleeren Menge I und $e_1,..e_n$, $e \in T_p(S)$ gilt Folgendes:

$$\begin{aligned} A_{e_1 \;..\; e_n} &= A_{e_1} \times .. \times A_{e_n}, \\ A_{e_1 + .. + e_n} &= A_{e_1} + .. + A_{e_n}, \\ A_{e^I} &= A_e^I, \\ A_{e^n} &= A_e^n, \\ A_{e^+} &= A_e^+, \\ A_{e^*} &= A_e^*, \\ h_{e_1 \;..\; e_n} &= h_{e_1} \times .. \times h_{e_n}, \\ h_{e_1 + .. + e_n} &= h_{e_1} + .. + h_{e_n}, \\ h_{e^I} &= h_e^I, \\ h_{e^n} &= h_e^{[n]}, \\ h_{e^+} &= h_e^+, \\ h_{e^*} &= h_e^*. \end{aligned}$$

2.5 Signaturen

- Eine **Signatur** $\Sigma = (S, F)$ besteht aus einer Menge S von Sorten wie oben sowie einer Menge F typisierter Funktionssymbole f: $e \to e'$ mit $e, e' \in T_p(S)$, den Operationen von Σ.

- obs(Σ): die Menge der **beobachtbaren Typen** (observable types) von Σ, das sind alle nichtleeren Mengenm die in Typen von Operationen von Σ vorkommen.

- $\forall f \in F$: f: $e \to e'$, dom(f) = e und ran(f) = e'

- Σ heißt **Gentzen-Signatur**, falls $\forall$ f $\in$ F Mengen I, J existieren sodass dom(f) ein I-stelliger Produkttyp und ran(f) ein J-stelliger Summentyp ist.
- Konsrtuktoren dienen der Synthese von Elementen einer S-sortigen Menge, Destruktoren liefern Werkzeuge zu ihrer Analyse.
- Abstrakte Syntax einer CFG ist eine konstruktive Signaturen
- Parser, Interpreter und Compiler beruhen auf Automatenmodelle, die destruktive Signaturen interpretieren

2.5.1 Konstruktive Signaturen

1. Mon ($\Rightarrow$ Unmarkierte binäre Bäume)
   - S = {mon}
   - F = {one: 1 $\rightarrow$ mon, mul: mon $\times$ mon $\rightarrow$ mon}
2. Nat ($\Rightarrow$ N)
   - S = {nat}
   - F = {zero: 1 $\rightarrow$ nat, succ: nat $\rightarrow$ nat}
3. Dyn(I, X) ($\Rightarrow$ I $\times$ X^*)
   - S = {list}
   - F = {nil: I $\rightarrow$ list. cons: X $\times$ list $\rightarrow$ list}
4. List(X) $=_{\text{def}}$ Dyn(1, X) ($\Rightarrow$ X^*)
5. Bintree(X) ($\Rightarrow$ binäre Bäume endlicher Tiefe mit Knotenmarkierungen aus X)
   - S = {btree}
   - F = {empty: 1 $\rightarrow$ btree, bjoin: btree $\times$ X $\times$ btree $\rightarrow$ btree}
6. Tree(X) ($\Rightarrow$ Bäume endlicher Tiefe und endlichen Knotenausgrads mit Knotenmarkierungen aus X)
   - S = {tree, trees}
   - F = {join: X $\times$ trees $\rightarrow$ tree, nil: 1 $\rightarrow$ trees, cons: tree $\times$ trees $\rightarrow$ trees}

7. Reg(BL) ($\Rightarrow$ reguläre Ausdrücke über BL)

    - S = {reg}
    - F = {par: reg $\times$ reg $\rightarrow$ reg, seq: reg $\times$ reg $\rightarrow$ reg, iter: reg $\rightarrow$ reg, base: BL $\rightarrow$ reg}

8. CCS(Act) ($\Rightarrow$ Calculus of Communicating Systems)

    - S = {proc}
    - F = { pre: Act $\times$ proc $\rightarrow$ proc, cho: proc $\times$ proc $\rightarrow$ proc, par: proc $\times$ proc $\rightarrow$ proc, res: proc $\times$ Act $\rightarrow$ proc, rel: proc $\times$ $\mathrm{Act}^{\mathrm{Act}}$ $\rightarrow$ proc}

2.5.2 Destruktive Signaturen

1. coNat ($\Rightarrow$ $\mathrm{N} \cup \{\infty\}$)

    - S = {nat}
    - F = {pred: nat $\rightarrow$ 1 + nat}

2. coList(X) ($\Rightarrow$ $\mathrm{X}^* \cup \mathrm{X}^{\mathrm{N}}$ (coList(1) $\cong$ coNat))

    - S = {list, X $\times$ list}
    - F = {split: list $\rightarrow$ 1+ X $\times$ list, π_1: X $\times$ list $\rightarrow$ X, π_2: X $\times$ list $\rightarrow$ list}

3. Stream(X) $=_{\mathrm{def}}$ DAut(1, X) ($\Rightarrow$ X^{N})

    - S = {list}
    - F = {head: list $\rightarrow$ X, tail: list $\rightarrow$ list}

4. coBintree(X) ($\Rightarrow$ binäre Bäume beliebiger Tiefe mit Knotenmarkierungen aus X)

    - S = {btree, btree $\times$ X $\times$ btree}
    - F = {split: btree $\rightarrow$ 1 + btree $\times$ X $\times$ btree, π_1: btree $\times$ X $\times$ btree $\rightarrow$ btree, π_2: btree $\times$ X $\times$ btree $\rightarrow$ X, π_3: btree $\times$ X $\times$ btree $\rightarrow$ btree}

5. Infbintree(X) ($\Rightarrow$ binäre Bäume unendlicher Tiefe mit Knotenmarkierungen aus X)

    - S = {btree}

- F = {root: btree $\to$ X, left, right:btree $\to$ btree}

6. DAut(X,Y) ($\Rightarrow$ Y^{X^*} = Verhalten det. Moore-Automaten mit Eingabemenge X und Ausgabemenge Y)
   - S = {state, state^X}
   - F = {δ: state $\to$ state^X, β: state $\to$ Y} $\cup$ {π_x: state^X $\to$ state | x$\in$ X}
7. Acc(X) $=_{def}$ DAut(X,2) ($\Rightarrow$ P(X^*) = Wortsprache über X)
8. Proctree(Act) ($\Rightarrow$ Prozessbäume, deren Kanten mit Aktionen markiert sind)
   - S = {tree} $\cup$ {$(\mathrm{Act} \times \mathrm{tree})^n$ | n > 0}
   - F = {δ: tree $\to$ $(\mathrm{Act} \times \mathrm{tree})^*$} $\cup$ {π_n: $(\mathrm{Act} \times \mathrm{tree})^n$ $\to$ Act $\times$ tree | n > 0} $\cup$ {π_1: Act $\times$ tree $\to$ Act, π_2: Act $\times$ tree $\to$ tree}

2.6 Algebren

Sei Σ = (S, F) eine Signatur. Eine Σ- **Algebra** A = (A, Op) besteht aus einer S-sortigen Menge A und einer F-sortigen Menge

$$\mathrm{Op} = (f^A\colon A_e \to A_{e'})_{f\colon e \to e' \in F}$$

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

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

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

gilt. Ist h bijektiv, dann heißt h Σ-Isomorphismus und A und B sind Σ-isomorph. h induziert die Bildalgebra h(A):

- Für alle e$\in$ T_p(S), $h(A)_e =_{def} h_e(A_e)$
- Für alle f: e$\to$ e' $\in$ F und a $\in$ A_e, $f^{h(A)}(h(a)) =_{def} f^B(h(a))$

2.6.1 Beispiele

1. Nat-Algebra
   - $zero^N$: $1 \to N$, $succ^N$: $N \to N$
   - $zreo^N(\epsilon) = 0$, $succ^N(n) = n + 1$
2. Word(X) (eine Mon-Algebra)
   - $one^{Word(X)}$: $1 \to X^*$, $mul^{Word(X)}$: $X^* \times X^* \to X^*$
   - $one^{Word(X)}(\epsilon) = \epsilon$, $mult^{Word(X)}(u, v) = uv$

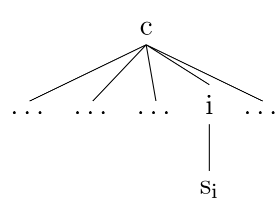
2.7 Terme und Coterme

2.7.1 Terme

Sei $\Sigma = (S,C)$ eine konstruktive Signatur, $X = \cup\, obs(\Sigma)$ und V eine S-sortige Menge von "Variablen". Die Menge $CT_\Sigma(V)$ der Σ-Terme über V ist die größte $(S \cup obs(\Sigma))$-sortige Menge M det. Bäume über $(X, C \cup X \cup V)$ mit folgenden Eigenschaften:

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

1. Fall
   b
   $b \in B$, $B \in obs(\Sigma)$
2. Fall
   x
   x ist von s, $s \in S$, $x \in V_s$
3. Fall

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

   - Die Elemente von $CT_\Sigma =_{def} CT_\Sigma(\lambda\ x.\ \emptyset)$ heißen Σ-Grundterme.

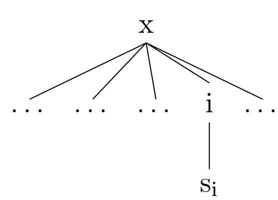
2.7.2 Coterme

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

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

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

1. Fall
   b
   $b \in B$, $B \in obs(\Sigma)$

2. Fall
   

   $x \in V_s$ $c: s \to \Pi_{i \in I}\ s_i \in D$

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

2.8 *Bool*-Algebra

Die Menge 2 ist Trägermenge der REG(BL)-Algebra *Bool*.

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

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

2.9 Termfaltung

- $\Sigma = (S, C)$
- V eine S-sortige Menge

- $\mathcal{A}$ = (A, Op) eine Σ-Algebra
- g: V → A eine **Variablenbelegung** (*valuation*)
- g^* intuitiv definiert.

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

Folglich ist $fold^A$ der einzige Σ-Homomorphismus von T_Σ nach A

2.10 Zustandsentfaltung

- Σ = (S, D)
- V eine S-sortige Menge
- $\mathcal{A}$ = (A, Op) eine Σ-Algebra
- g: A → V eine Färbung
- $g^\#$ intuitiv definiert

Offenbar hängt die Einschränkung von $g^\#$ auf Grundterme nicht von g ab. Sie wird **Zustandsentfaltung** genannt und mit $unfold^A$ bezeichnet.

3 Rechnen mit Algebren

3.1 Unteralgebra

Bezüglich der Operationen geschlossene Untermenge.

3.2 Substitution

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

3.3 Termäquivalenz

$\forall$ t,t' ∈ E, g ∈ A^V : $g^*(t) = g^*(t')$

3.4 Normalformen

Werden für jede Signatur selbst definiert und können durch das verwenden definierter Gleichungen erzeugt Werden

4 Kontextfreie Grammatiken (CFGs)

4.1 Definition

G = (S, BS, R) mit

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

4.2 Die JavaLight Grammatik

4.2.1 R

- Commands → Command Commands | Command
- Command → {Commands} | String = Sum; |
  if Disjunct Command else Command |
  if Disjunct Command | while Disjunct Command
- Sum → Sum + Prod | Sum - Prod | Prod
- Prod → Prod * Factor | Prod/Factor | Factor
- Factor → Z | String | (Sum)
- Disjunct → Conjunct || Disjunct | Conjunct
- Conjunct → Literal && Conjunct | Literal
- Literal → !Literal | Sum Rel Sum | 2 | (Disjunct)

4.2.2 BS

- String (alle Zeichenfolgen außer Elementen anderer Basismengen von JavaLight)
- Z
- Rel (nicht näher spezifizierter binärer Relationen auf Z

4.3 Beispiel Programm

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

4.4 Linksrekursive Grammatiken

Sei G = (S, BS, R) und X = ∪ BS.

X ist die Menge der Eingabesymbole, die Compiler für G verarbeiten müssen.

4.4.1 Einschub Ableitungsrelation

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

4.4.2 Definition

- G heißt **linksrekursiv**, falls es eine **linkrekursive** Ableitung $s \to_G sv$ gibt.
- G heißt LL-kompilierbar, falls es eine partielle Ordnung $\leq$ auf S gibt mit $s' \leq s$ für alle Ableitungen $sv \to_G s'w$

1. Umgangssprachlich Man hat eine Regel, sodass die Sorte auf der linken Seite auf der rechten Seite ganz links vorkommt.

4.4.3 Beispiel

Z.B. REG ist LL-kompilierbar, JavaLight jedoch <u>nicht</u>.

4.4.4 Verfahren zur Elemenierung von Linksrekursion

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

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

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

1. Beispiel JavaLight

   - Commands $\to$ Command Commands | Command
   - Command $\to$ {Commands} | String = Sum; |
     if Disjunct Command else Command |
     if Disjunct Command | while Disjunct Command
   - Sum $\to$ Prod Sumsect
   - Sumsect $\to$ + Prod Sumsect | - Prod Sumsect | ϵ
   - Prod $\to$ Factor Prodsect
   - Prodsect $\to$ * Factor Prodsect | / Factor Prodsect | ϵ
   - Factor $\to$ Z | String | (Sum)
   - Disjunct $\to$ Conjunct || Disjunct | Conjunct
   - Conjunct $\to$ Literal && Conjunct | Literal
   - Literal $\to$!Literal | Sum Rel Sum | 2 | (Disjunct)

4.5 Abstrakte Syntax

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

Die folgende Funktion typ: $(S \cup BS)^* \to T_p(S)$ streicht alle Elemente von Z(G) aus Wörtern über $S \cup BS$ und überführt diese in die durch sie bezeichneten Produkttypen:

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

Dann ist $\Sigma(G) = (S, BS, \{f_{s \to w}\text{:typ}(w) \to s \mid s \to w \in R\})$
$\Sigma(G)$-Grundterme werden Syntaxbäume von G genannt.

4.5.1 Beispiel JavaLight

- S = {Commands, Command, Sum, Prod, Factor, Disjunct, Conjunct, Literal}
- I = {Z, String, Rel, 2}

- F = {
  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: Z → 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: Rel × Sum × Sum → Literal,
  embedB: 2 → Literal,
  encloseD: Disjunct → Literal}

4.5.2 JavaLight' (entlinksrekursiv)

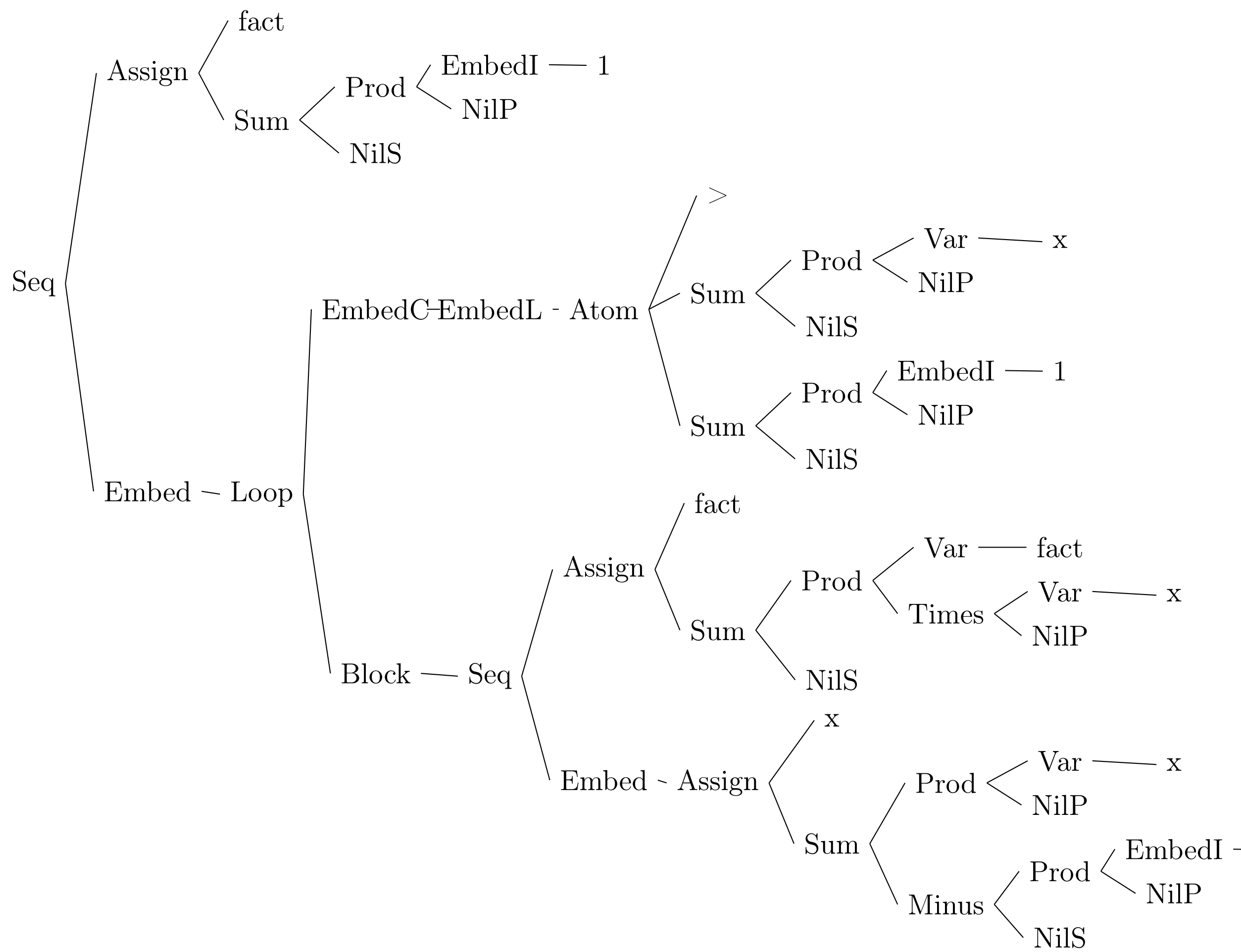
- wird um die Sorten Sumsect und Prodsect erweitert und um die Konstruktoren

- sum: Prod × Sumsect → Sum,
  plus, minus: Prod × Sumsect → Sumsect,
  nilS: 1 → Sumsect,
  prod: Factor × Prodsect → Prod,
  times, div: Factor × Prodsect → Prodsect,
  nilP: 1 → Prodsect

4.5.3 Syntaxbaum Beispiel

Beispiel Programm:

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



4.6 Definition derec(G)

- Für alle $s \in S \cup BS \setminus Z(G)$, $derec(G)_s = T_{\Sigma(G'),s}$
- Für alle $s \in S \setminus recs(S)$ und $s \rightarrow v \in R$ und $t \in T_{\Sigma(G'),typ(v)}$, $f^{derec(G)}_{s \rightarrow v}(t)$ $= f_{s \rightarrow v}(t)$
- Für alle $s \rightarrow v \in nonrecs(R)$, $s \rightarrow sw \in R$, $t \in T_{\Sigma(G'),typ(v)}$, $t' \in T_{\Sigma(G'),s'}$ und $u \in T_{\Sigma(G'),\ typ(w)}$
  $f^{derec(G)}_{s \rightarrow v}(t) = f_{s \rightarrow vs'}(t,\ f_{s' \rightarrow \epsilon})$
  $f^{derec(G)}_{s \rightarrow sw}(f_{s \rightarrow vs'}(t,t'),u) = f_{s \rightarrow vs'}(t,\ f_{s' \rightarrow ws'}(u,\ t'))$

Mit derec(G) kann man eine Syntaxbaum in G in einen Syntaxbaum der nicht linksrekursiven Grammatik G' umwandeln

4.7 Wort- und Ableitungsbaumalgebra

Neben $T_{\Sigma(g)}$ lassen sich auch die Menge der Wörter über X und die Menge der Ableitungsbäume von G zu Σ(G)-Algebren erweitern.

4.7.1 Wortalgebra

$\mathrm{fold}^{\mathrm{Word(G)}}$ bildet Terme auf die entsprechenden Wörter der Sprache.

4.7.2 Ableitungsbaumalgebra

Bildet auf einen Baum ab, der auch die Wörter darstellt (inklusive der Terminale aus Z(G))

4.8 Zustandsmodell von JavaLight

- Sei Store = String → Z (Variablenbelegung)
- Wir bilden eine Σ(JavaLight)-Algebra

4.8.1 Sorten

- $\mathrm{A}_{\mathrm{Commands}} = \mathrm{A}_{\mathrm{Command}} = \mathrm{Store} \rightarrow \mathrm{Store}$
- $\mathrm{A}_{\mathrm{Sum}} = \mathrm{A}_{\mathrm{Factor}} = \mathrm{A}_{\mathrm{Prod}} = \mathrm{Store} \rightarrow \mathrm{Z}$
- $\mathrm{A}_{\mathrm{Disjunkt}} = \mathrm{A}_{\mathrm{Conjunct}} = \mathrm{A}_{\mathrm{Literal}} = \mathrm{Store} \rightarrow 2$

4.8.2 Operationen

Für alle f,g: Store → Store, x ∈ Store, e: Store ∈ Z, st ∈ Store
und p: Store → 2.

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

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

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

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

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

5 Parser und Compiler für CFGs

$\mathrm{T}_{\Sigma(\mathrm{G})} \xrightarrow{fold^Z} Z \xrightarrow{evaluate} Mach$

$T_{\Sigma(\mathrm{G})} \xrightarrow{fold^S} S \xrightarrow{encode} Mach$

- S: Sem, die ebenfalls als Σ(G)-Algebra gegebene Semantik der Quellsprache L(G)
- Mach, eine in der Regel unabhängig von Σ(G) definiertes Modell der Zielsprache, meist in der Form einer abstrakten Maschine
- evaluate, ein Interpreter der die Zielsprache in der abstrakten Maschine Mach ausführt
- encode, eine Funktion, die Sem auf Mach abbildet und die gewünschte Arbeitsweise des Compilers auf der semantischen Ebene ausführt

5.1 Definition Parser

Parser für G: eine S-sortige Funktion

$$\text{parse}_G\colon X^* \to M(T_{\Sigma(G)})$$

die entweder einen Synatxbaum für das Eingabewort erzeugt oder eine Fehlermeldung zurück gibt. (Syntaxbaum und Fehlermeldung sind abhänig von der Monade M)

5.2 Funktoren und Monaden

5.2.1 Definition Kategorie

Eine Kategorie K besteht aus

- einer - ebenfalls mit K bezeichneten- Klasse von K-Objekten
- für alle A.B $\in$ K einer Menge K(A,B) von K-Morphismen
- einer assoziativen Komposition

$$\begin{gathered} .\colon K(A,B) \times K(B,C) \to K(A,C) \\ (f,g) \to g\ .\ f \end{gathered}$$

- einer Identität $id_A \in K(A,A)$, die bezüglich . neutral ist

5.2.2 Defintion Funktor

Seien K und L Kategorien. Ein Funktor F: K $\to$ L ist eine Funktion(müssen das wirklich Funktionen sein?), die jedem K-Objekt ein L-Objekt und jedem K-Morphismus f: A $\to$ B eine L-Morphismus F(f): F(A) $\to$ F(B) zuordnet, sowie folgende Gleichungen erfüllt:

- Für alle K-Objekte A, $F(id_A) = id_{F(A)}$
- Für alle K-Morphismen f: A $\to$ B und g: B $\to$ C, F(g . f) = F(g) . F(f)
- Funktoren lassen sich wie Funktionen zu neuen Funktoren komponieren.

1. Beispiele

    - Sei B $\in$ L. Der konstante Funktor const(B): K $\to$ L ordnet jedem K-Objekt B und jedem K-Morphismus die Identität auf B zu
    - Der Diagonalfunktor $\Delta_K\colon K \to K^2$ ordnet jedem K-Objekt A das Paar (A,A) und jedem K-Morphismus f das Paar (f,f) zu

- Produktfunktor, _ $\times$ _ : $\mathrm{Set}^2 \to$ 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 $=_{\mathrm{def}}$ λ(a,c).(f(a),g(c)) zu
- Ausnahmefunktor _ + E : Set $\to$ Set ordnet jeder Menge A die Menge A + E zu und jeder Funktion f: A $\to$ B die Funktion

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

5.2.3 Definition Natürliche Transformation

Seien F, G: K $\to$ L Funktoren. Eine natürliche Transformation τ: F $\to$ G ordnet jedem K-Objekt A einen L-Morphismus τ_{A}: F(A) $\to$ G(A) derart, dass für alle K-Morphismen f:A $\to$ B das gilt:

- F(A) $\to^{\tau_{\mathrm{A}}}$ G(A)
- F(A) $\to^{\mathrm{F(f)}}$ F(B)
- F(B) $\to^{\tau_{\mathrm{B}}}$ G(B)
- G(A) $\to^{\mathrm{G(f)}}$ G(B)

5.2.4 Definition Monade

Ein Funktor M: K $\to$ K heißt Monade, wenn es zwei natürliche Transformationen η: Id_{K} $\to$ M (Einheit) und μ: M . M $\to$ M (Multiplikation) gibt, die für alle A $\in$ K folgednes gelten lassen:

- Seite 154 Folienscript

1. Beispiel Ausnahmefunktor

   - η_{A}: A $\to$ A + E
     η_{A}(a) = (a,1)
   - μ_{A}: (A + E) + E
     μ_{A}(((a,1),1)) = (a,1)
     μ_{A}(((e,2),1)) = (e,2)
     μ_{A}((e,2)) = (e,2)

2. Der bind-Operator Seien A und B Mengen

$$»=: M(A) \times (A \to M(B)) \to M(B)$$
$$(»= f) = \mu_B \,.\, M(f)$$

3. Intuitive Erklärung einer Monade

   Intuitiv stellt man sich ein monadisches Object $m \in M(A)$ als Berechnung vor, die eine -evt leere- Menge von Werten in A erzeugt. Ein Ausdruck der Form $»= f$ wird dann wie folgt ausgewertet: Die von m berechneten Werte $a \in A$ werden als Eingabe an die Berechnung von f übergeben und von f(a) verarbeitet.

4. Es ergeben sich ein paar Eigenschaften

   - $m »= \eta_A = m$
   - $\eta_A(a) »= f = f(a)$
   - $(m »= f) »= g = m »= \lambda$ a. $f(a) »= g$
   - $M(h)(m) = m »= \mu_B \,.\, h$
   - $\eta_A(m') = m' »= id_M(A)$

5. Plusmonade Plusmonaden haben zusätzlich eine parallele **Komposition**

$$\oplus: M \times M =_{def} \_ \times \_ \,.\, \Delta \,.\, M \to M$$

5.2.5 Compilermonade

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

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

MMenge der Ausnahmewerte“

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

$$(m \oplus m') \oplus m'' = m \oplus (m' \oplus m'')$$
$$M(h)(e) = e$$
$$M(h)(m \oplus m') = M(h)(m) \oplus M(h)(m')$$
$$set_A(m \oplus m') = set_A(m) \cup set_A(m')$$
$$set_A(\eta_A(a)) = \{a\}$$
$$set_B(m »= f) = \cup\{set_B(f(a)) \mid a \in set_A(m)\}$$

5.2.6 Monadenbasierte Parser und Compiler

- Sei G = (S, BS, R) LL-kompilierbar
- G' die daraus gebildete nicht-linksrekursive Grammatik
- $X = \cup$ BS
- $\Sigma(G) = (S, F)$
- $M : Set^S \to Set^S$

Ein Compiler für G in die als $\Sigma(G)$-Algebra A formulierte Zielsprache einen Parser für G mit der Faltung in A der vom Parser erzeugten Syntaxbaume komponieren. Daher passen wir die Übersetzungsfunktion ein bisschen an:

$$\mathrm{compile}_G^A = X^* \to^{\mathrm{compile}_G^{T_{\Sigma(G)}}} M(T_{\Sigma(G)}) \to^{M(\mathrm{fold}_A)} M(A)$$

Ist G linksrekursiv terminiert das evt nicht, daher fordern wir in dem Fall:

$$\mathrm{compile}_G^A = X^* \to^{\mathrm{parse}_G^{T_{\Sigma(G)}}} M(T_{\Sigma(G')}) \to^{M(\mathrm{fold}_{\mathrm{derec}(A)})} M(A)$$

Es gilt die Verträglichkeit mit Σ-Homomorphismen (eingebettet in die Monade natürlich)

Gilt 5.2.6, dann folgt aus 5:

- $X^* \to^{\mathrm{compile}_G^A} M(A) \to^{M(\mathrm{evaluate})} M(\mathrm{Mach})$
- $X^* \to^{\mathrm{compile}_G^S} M(S) \to^{M(\mathrm{encode})} M(\mathrm{Mach})$

6 LL Kompiler

Das erste L steht für die Verarbeitung von **Links** nach Rechts. Das zweite L steht für das bilden einer **Linksableitung**.

- G = (S, BS, R)
- G' = (S', BS, R')
- $X = \cup$ BS

- M eine Compilermonade
- errmsg: $X^* \to E$
- A = (A, Op) eine Σ(G)-Algebra
- $compile_G$ heißt LL-Compiler, wenn $(\mathrm{compile}^A_{G,s}\colon X^* \to M(A_s))_{s\in S}$
- $\mathrm{compile}^A_{G,s}(w) = \mathrm{trans}^A_s(w) \gg= \lambda(a, w).$ if $w = \epsilon$ then $\eta_A(a)$ else errmsg(w), wobei für alle $s \in S' \cup BS$
  $\mathrm{trans}^A_s : X^* \to M(A_s \times X^*)$ wie folgt definiert ist:

6.1 Fall 1: s ∈ BS. Für alle x ∈ X und w ∈ X*

$$\mathrm{trans}^A_s(xw) = \text{if } x \in s \text{ then } \eta_{A\ \ X^*}(x,w) \text{ else errmsg}(xw)$$
$$\mathrm{trans}^A_s(\epsilon) = \mathrm{errmsg}(\epsilon)$$

6.2 Fall 2: s ∈ S'. Für alle w ∈ X*,

$$\mathrm{trans}^A_s(w) = \oplus_{r=(s \to e)\in R}\ \mathrm{try}^A_r(w)$$

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

$$\mathrm{try}^A_r(w) = \mathrm{trans}^A_{e_1} \gg= \lambda(a_1,w_1).\ \ldots\ldots\ \mathrm{trans}^A_{e_n}(w_{n-1}) \gg= \lambda(a_n, w_n).\eta_{A\ \ X^*}(f^{derec(A)}_r(a_{i_1},..,a_{i_k}),w_n)$$

wobei $\{i_1,..,i_k\} = \{1 \le i \le n \mid e_i \in S' \cup BS \setminus Z(G)\}$

Ein Beispiel ist auf Seite 170 im Folienscript. Das JavaLight+ Beispiel beginnt auf Seite 172

7 LR Kompiler

- Konstruieren **Rechtsreduktion** (⇒ Rechtsableitung)
- Entspricht dem *mit dem Blättern beginnender Aufbau eines Syntaxbaumes*, daher **Bottom-up** Compiler
- LR-Compiler sind auf LR(k)-Grammatiken beschränkt
- Sei G = (S, BS, R) eine CFG und X = ∪ BS
- Vorraussetzung: *start* ∈ S und *start* kommt in keiner Regel auf der rechten Seite vor

7.1 LR(k) Grammatiken

Eine Grammatik ist eine LR(k) Grammatik, wenn das vorrauslesen von k Symbolen reicht, damit man entscheiden kann ob ein weiteres Zeichen gelesen werden muss oder eine Reduktion durchgeführt werden muss (und wenn ja welche). Außerdem müssen die BS disjunkt sein.

7.1.1 Definition first- und follow- Wormengen

Sei $k > 0$, $\alpha \in (S \cup BS)^*$ und $s \in S$

$$\mathrm{first}_k(\alpha) = \{\beta \in BS^k \mid \exists\, \gamma \in BS^*\colon \alpha \to_G^* \beta\gamma\} \cup \{\beta \in BS^{<k} \mid \alpha \to_G^* \beta\}$$
$$\mathrm{follow}_k(s) = \{\beta \in BS^k \mid \exists\, \alpha, \gamma \in BS^*\colon \mathrm{start} \to_G^* \alpha\ s\ \beta\gamma\} \cup \{\alpha \in BS^{<k} \mid \mathrm{start} \to_G^* \alpha\ s\ \beta\}\ \mathrm{first}(\alpha) = \mathrm{first}_1(\alpha)\ \mathrm{follow}(s) = \mathrm{follow}_1(s)$$

Sei recog_1 ein Erkenner für die Sprache (Definition auf Seite 190)

7.2 LR-Automat

Zustandsmenge:

- $Q_G = \{\mathrm{state}(\phi) \mid \phi \in (S \cup BS)^*\}$

partielle Transitionsfunktion (auch goto-Tabelle genannt)

- $\delta\colon Q_G \to Q_G^{S \cup BS}$
  $\mathrm{state}(\phi) = \lambda(s).\mathrm{state}(\phi\ s)$

7.2.1 Simultane induktive Definition von Q_G und δ_G

$$\mathrm{start} \to \alpha \in R \Rightarrow (\mathrm{start}, \epsilon, \alpha, \epsilon) \in q_0$$
$$(s, \alpha, s', \beta, u) \in q \wedge s' \to \gamma \in R \wedge v \in \mathrm{first}(\beta\ u) \Rightarrow (s', \epsilon, \gamma, v) \in q$$
$$(s, \alpha, s'\beta, u) \in q,\ s' \in S \cup BS \Rightarrow (s, \alpha\ s', \beta, u) \in \delta_G(q)(s')$$

- $h\colon (S \cup BS)^* \to Q_G^*$
- $\mathrm{state}(\epsilon) =_{\mathrm{def}} \lambda(\epsilon).\ q_0$
- $\lambda(s_1,..,s_n).\ (\mathrm{state}(s_1 \ldots s_n), \mathrm{state}(s_1 \ldots s_{n-1}),.., \mathrm{state}(s_1), \mathrm{state}(\epsilon))$

Wir definieren $\mathrm{recog}_2\colon Q_G^* \to 2^{X^*}$
Für alle $q, q_i \in Q_G$, $qs \in Q_G^*$, $x \in X$ und $w \in X^*$,

- $\mathrm{recog}_2(q{:}qs)(xw) = \mathrm{recog}_2(\delta_G(q)(x){:}q{:}qs)(w)$
  falls $\exists\ (s, \alpha, \beta, \epsilon) \in q : \beta \neq \epsilon,\ x \in \mathrm{first}(\beta) \cap Z(G)$

- $\text{recog}_2(\text{q:qs})(\text{xw}) = \text{recog}_2(\delta_G(\text{q})(\text{B})\text{:q:qs})(\text{w})$
  falls $\exists$ (s, α, β, ϵ) $\in$ q, B $\in$ BS:
  $\beta \neq \epsilon$, x $\in$ B $\in$ first(β)

- $\text{recog}_2(\text{q}_1\text{: \ldots :q}_{|\alpha|}\text{:q:qs})(\text{w}) = \text{recog}_2(\delta_G(\text{q})(\text{s})\text{:q:qs})(\text{w})$
  falls $\exists$ (s, α, ϵ u) $\in$ q_1:s $\neq$ start,
  w $= \epsilon =$ u $\vee$ head(w) $=$ u $\in$ Z(G) $\vee$
  head(w) $\in$ u $\in$ BS

- $\text{recog}_2(\text{q:qs})(\epsilon) = 1$ falls $\exists\, \varphi$: (start, φ, ϵ, ϵ) $\in$ q

- $\text{recog}_2(\text{qs})(\text{w}) = 0$ sonst

Offenbar gilt $\text{recog}_1 = \text{recog}_2 \bigcirc h$.
Zur optimieren verwenden wir eine Aktionstabelle

- act_G: $Q_G \times (\text{BS} \cup 1) \to \text{R} \cup \{\text{shift, error}\}$

Für alle u $\in$ BS $\cup$,

- act_G(q, u) $=$ shift falls $\exists$ (s, α, β, ϵ) $\in$ q: $\beta \neq \epsilon$, u $\in$ first(β),
  s $\to \alpha$ falls $\exists$ (s, α, ϵ u) $\in$ q,
  error sonst

Dann erhält man eine Kompaktedefinition für recog_2:
Für alle q, $q_i \in Q_G$, qs $\in Q_G^*$, x $\in$ X und w $\in X^*$,

- $\text{recog}_2(\text{q:qs})(\text{xw}) = \text{recog}_2(\delta_G(\text{q})(\text{x})\text{:q:qs})(\text{w})$
  falls x $\in$ Z(G), act_G(q, x) $=$ shift

- $\text{recog}_2(\text{q:qs})(\text{xw}) = \text{recog}_2(\delta_G(\text{q})(\text{B})\text{:q:qs})(\text{w})$
  falls $\exists$ B $\in$ BS : x $\in$ B, act_G(q, B) $=$ shift

- $\text{recog}_2(\text{q}_1\text{:\ldots:q}_{|\alpha|}\text{:q:qs})(\text{w}) = \text{recog}_2(\delta_G(\text{q})(\text{s})\text{:q:qs})(\text{w})$
  falls $\exists$ u $\in$ BS $\cup$ 1: $\text{act}_G(q_1, \text{u}) = \text{s} \to \alpha$,
  w $= \epsilon =$ u $\vee$ head(w) $=$ u $\in$ Z(G) $\vee$
  head(w) $\in$ u $\in$ BS

- $\text{recog}_2(\text{q:qs})(\epsilon) = 1$ falls $\text{act}_G(\text{q}, \epsilon) = \text{start} \to \alpha$

- $\text{recog}_2(\text{qs})(\text{w}) = 0$ sonst

Beispiel für SAB2 auf Seite 197 im Skript.

7.2.2 Formulierung der Korrektheit von compile_G

- Für alle $w \in X^*$ und $t \in T_{\Sigma(G),\, start}$,
  $\text{compile}_G^A(q_0, \epsilon)(w) = \eta(\text{fold}^A(t)) \Rightarrow \text{fold}^{Word(G)}(t) = w$,
  $\text{compile}_G^A(q_0, \epsilon)(w) = \text{error}(w) \Rightarrow w \notin L(G)_{start}$

7.2.3 Beispiel SAB2; Seite 203

7.2.4 Beispiel yacc; Seite 204

7.2.5 Sprachklassen-Hierarchie

- CFG ⊂ LR(k) ⊂ LL(k)

- LR(k) ⊂ LALR(k) ⊂ SLR(k)

- LL(k)-Grammatiken sind kurz gesagt diejenigen nicht-linksrekursiven CFGs, deren LL-Parser ohne Backtracking auskommen.

- Für jeden Grammatiktyp T bedeutet die Formulierung **L ist eine T-Sprache** lediglich, dass eine T-Grammatik *existiert* die L erzeugt

1. Kommentar zu LL-Compiler

   Während der LL-Compiler von Kapitel 6 - nach Beseitigung von Linksrekursion - jede kontexfreie Grammatik verarbeitet, selbst dann, wenn sie mehrdeutig ist, zeigt die obige Grafikm dass die Forderung, dabei ohne Backtracking auszukommen, die Klasse der kompilierbaren Sprache 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 BA), 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 Yacc.

8 Haskell: Typen und Funktionen

Ich hoffe ihr seid fit in Haskell

9 Haskell: Listen

10 Haskell: Datentypen und Typklassen

11 Algebren in Haskell

Sei Σ = (S, F) eine Signatur, obs(Σ) = $\{x_1, \ldots, x_k\}$, S = $\{s_1, \ldots, s_m\}$ und F = $\{f_1: e_1 \to e'_1 \,..\, f_n: e_n \to e'_n\}$.
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$(x_1)\ldots(x_k)$ repräsentieren die Trägermengen einzelner Algebren.

11.1 Beispiel für Nat

```
--natT implementiert T_Nat
data Nat nat = Nat {zero :: nat, succ :: nat -> nat}

natT :: Nat Int
natT = Nat {zero = 0, succ = (+1)}

--listT implementiert T_list(X)
data List x list = List {nil :: list, cons :: x -> list -> list}

listT :: List x [x]
listT = List {nil = [], cons = (:)}

--Beispiel foldList

foldList :: List x list -> [x] -> list
foldList alg [] = nil alg
foldList alg (x:s) = cons alg x $ foldList alg s
```

Weitere Beispiele ab Seite 252.

11.2 Datentypen der JavaLight-Algebren

```
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}

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}

derec :: JavaLight s1 s2 sum prod factor s3 s4 s5 -> SumProd sum (sum -> sum) prod (pro
derec alg = SumProd {sum' = \a g -> g $ sum_ alg a,
     plus' = \a g x -> g $ plus alg x a,
     minus' = \a g x -> g $ minus alg x a,
     nilS = id,
```

```
     prod' = \a g -> g $ prod alg a,
     times' = \a g x -> g $ times alg x a,
     div' = \a g x -> g $ div_ alg x a,
     nilP = id}
```

11.3 Die Termalgebra von JavaLight

```
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


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}

javaWord :: JavaLight String String String String String String String
javaWord = JavaLight {seq_ = (++)
     ,embed = id
     ,block = \cs -> " {" ++ cs ++ "}"
```

```
    ,assign = \x e -> x ++ " = " ++ e ++ "; "
    ,cond = \e c c' -> "if " ++ e ++ c ++ " else" ++ c'
    ,cond1 = \e c -> "if " ++ e ++ c
    ,loop = \e c -> "while " ++ e ++ c
    ,sum_ = id
    ,plus = \e e' -> e ++ '+':e'
    ,minus = \e e' -> e ++ '-':e'
    ,prod = id
    ,times = \e e' -> e ++ '*':e'
    ,div_ = \e e' -> e ++ '/':e'
    ,embedI = show
    ,var = id
    ,encloseS = \e -> '(' : e ++ ")"
    ,disjunct = \e e' -> e ++ " || " ++ e'
    ,embedC = id
    ,conjunct = \e e' -> e ++ " && " ++ e'
    ,embedL = id
    ,not_ = \be -> '!' : be
    ,atom = \rel e e' -> e ++ rel ++ e'
    ,embedB = show
    ,encloseD = \e -> '(' : e ++ ")"}
```

11.4 Zustandsmodell von JavaLight

```
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 B
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
      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 f $ f st else st
```

11.4.1 Interpretation eines JavaLight Programms

prog = fact = 1; while x > 1 {fact = fact*x; x = x-1;}
$\quad \text{compile}^{\text{A}}_{\text{JavaLight}}(\text{prog}) : \text{Store} \to \text{Store}$
$\text{compile}^{\text{A}}_{\text{JavaLight}} = \lambda(\text{store}).\ \lambda(\text{z}).$ if z = x then 0 else if z = fact then store(x)!
else store(z)

11.5 Ableitungsbaumalgebra von JavaLight

```
type TS = Tree String

javaDeri :: JavaLight TS TS TS TS TS TS TS
javaDeri = JavaLight {seq_ = \c c' -> F "Commands" [c,c']
     ,embed = \c -> F "Commands" [c]
     ,assign = \x e -> command [leaf x, leaf "=", e, leaf ";"]
```

```
  ,cond = \e c c’ -> command [leaf "if", e, c, leaf "else", c’]
  ,cond1 = \e c -> command [leaf "if", e, c]
  ,loop = \e c -> command [leaf "while", e, c]
  ,sum_ = \e -> F "Sum" [e]
  ,plus = \e e’ -> F "Sum" [e, e’]
  ,minus = \e e’ -> F "Sum" [e, e’]
  ,prod = \e -> F "Prod" [e]
  ,times = \e e’ -> F "Prod" [e, e’]
  ,div_ = \e e’ -> F "Prod" [e, e’]
  ,embedI = \i -> factor [leaf $ show i]
  ,var = \x -> factor [leaf x]
  ,encloseS = \e -> factor [leaf "(", e, leaf ")"]
  ,disjunct = \e e -> F "Disjunct" [e, leaf, "||", e’]
  ,embedC = \e -> F "Disjunct" [e]
  ,conjunct = \e e’ -> F "Conjunct" [e, leaf "&&", e’]
  ,embedL = \e -> F "Conjunct" [e]
  ,not_ = \be -> literal [leaf "!", be]
  ,atom = \rel e e’ -> literal [e, leaf rel, e’]
  ,embedB = \b -> literal [leaf $ show b]
  ,encloseD = \e -> literal [leaf "(", e, leaf ")"]}
where
  command = F "Command"
  factor = F "Factor"
  literal = F "Literal"
  leaf = flip F []
```

11.6 Beispiel XMLstore-Algebren (Seite 265)

12 Attributierte Übersetzung

12.1 Binärdarstellung rationaler Zahlen

12.2 Strings mit Hoch- und Tiefdarstellung

12.3 übersetzung regulärer Ausdrück in erkennenden Automaten

12.4 Darstellung von Termen als hierarchische Listen

12.5 Eine kellerbasierte Zielsprache für JavaLight

Der folgende Datentyp liefert die Befehle einer Assemblersprache, die auf einem Keller vom Typ Z und einem Speicher vom Typ

$$\text{Store} = \text{String} \to \text{Z}$$

operiert. Hierbei betrachten wir die Abstraktion eines realen Speichers.

```
data StackCom = Push Int | Pop | Load String | Save String | Add |
Sub | Mul | Div | Or_ | And_ | Inv | Cmp String | Jump Int |
JumpF Int

type State = ([Int], Store, Int)

executeCom :: StackCom -> State -> State
executeCom com (stack, store, n) =
  case com of Push a -> (a:stack, store, n+1)
      Pop    -> (tail stack, store, n+1)
      Load x -> (store x:stack, store, n+1)
      Save x -> (stack, update store x $ head stack, n+1)
      Add    -> (a+b:s, store, n+1) where a:b:s = stack
      Sub    -> (b-a:s, store, n+1) where a:b:s = stack
      Mul    -> (a*b:s, store, n+1) where a:b:s = stack
      Div    -> (a`div`b: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
      Jump k  -> (stack, store, k)
      JumpF k -> (stack, store, if a == 0 then k else n+1) where a:_ = stack

execute :: [StackCom] -> State -> State
execute cs state@(_,_,n) = if n >= length cs then State
   else execute cs $ executeCom (cs !! n) state
```

Die Trägermengen haben neben dem jeweiligen Zielcode code ein (vererbtes) Attribut, das die Nummer des erten Befehls von code wiedergibt. Dementsprechend interpretiert javaStack alle Sorten von JavaLight durch den Funktionstyp

```
type LCom = Int -> [StackCom]

javaStack :: JavaLight LCom LCom LCom LCom LCom LCom LCom
javaStack = JavaLight {seq_ = seq_
      ,embed = id
      ,block = id
      ,assign = \x e lab -> e lab ++ [Save x, Pop]
      ,cond = \e c c' lab -> let (code, exit) = fork e c 1 lab
 code' = e' exit
     in code ++ Jump (exit + length code') : code'
      ,cond1 = \e c -> fst . fork e c 0
      ,loop = \e c lab -> fst (fork e c 1 lab) ++ [Jump lab]
      ,sum_ = id
      ,plus = apply2 Add
      ,minus = apply2 Sub
      ,prod = id
      ,times = apply2 Mul
      ,div_ = apply2 Div
      ,embedI = \i -> const [Push i]
      ,var = \x -> const [Load x]
      ,encloseS = id
      ,disjunct = apply2 Or_
      ,embedC = id
      ,conjunct = apply2 And_
      ,embedL = id
      ,not_ = apply1 Inv
      ,atom = apply2 . Cmp
```

```
      ,embedB = \b -> const [Push $ if b then 1 else 0]
      ,encloseD = id}
    where apply1 :: StackCom -> LCom -> LCom
  apply1 op e lab = e lab ++ [op]
  seq_ :: LCom -> LCom -> LCom
  seq_ e e' lab = code ++ e' (lab + length code)
    where code = e lab
  apply2 :: StackCom -> LCom -> LCom -> LCom
  apply2 op e e' lab = code ++ e' (lab + length code) ++ [op]
    where code = e lab
  fork :: LCom -> LCom -> Int -> Int -> ([StackCom], Int)
  fork e c n lab = (code ++ JumpF exit : code', exit)
    where code = e lab
  lab' = lab + length code + 1
  code' = c lab'
  exit = lab' + length code' + n
```

13 JavaLight+ = JavaLight + I/O + Deklaration + Prozedure

Ich meine, er hätte gesagt, dass sei für die Prüfung nicht mehr relevant, deshalb habe ich viele Details weg gelassen.

13.1 Assemblersprache mit I/O und Kelleradressierung

Die Variablenbelegung store: String $\rightarrow$ Z mit Zustandsmodell von Abschnitt Assemblerprogramm als JavaLight-Zielalgebra wird ersetzt durch den Keller stack $\in Z^*$, der jetzt nicht nur der schrittweisen Auswertung von Ausdrücken dient, sondern auch der Ablage von Variablenwerten unter vom Compiler berechneten Adressen. Witerer Zustandskomponenten sind:

- der Inhalt des Registers **BA** für die jeweils aktuelle Basisadresse
- der Inhalt des Registers **STP** für die Basisadresse des statischen Vorgängers des jeweils zu überstzenden Blocks bzw. Funktionsaufruf
- der schon Abschnitt 12.5 benutze **Befehlszähler** pc (program counter)
- der Ein/Ausgabestrom io, auf den Lese- bzw Schreibbefehler zugreifen

Der entsprechende Datentyp lautet daher wie folgt

```
data State = State {stack, io :: [Int], ba, stp, pc :: Int}

baseAdr :: Int -> Int -> SymAdr
baseAdr declDep dep = if declDep == dep then BA else Dex BA declDep

-- Die folgenden Funktionen berechnen aus symbolischen Adressen
-- absolute Adressen bzw. Kellerinhalte:

absAdr, contents :: State -> 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 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 :: State -> SymAdr -> ([Int], Int)
stackPos state adr = (s, length s-1-contents state adr)
     where s = stack State

updState :: State -> SymAdr -> Int -> State
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
```

Die Algebra funktioniert ähnlich wie die vorige (für eine Definition, Folien Skript 193)

13.2 Grammatik und abstrakte Syntax von JavaLight+

JavaLight+ enthählt neben den Sorten von JavaLight die Sorten Formals und Actuals für Listen formaler bzw. aktueller Parameter von Prozeduren. Auch die BS von JavaLigth werden übernommen. Hinzu kommt eine für formale Paramter. 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) : Prozedurvariable
- ForFun(t) : Typ einer Prozedurvariable, t ist hier der Typ der Prozedurergebnisse
- Fun(t,lab): Prozedurkonstante, t Ergebnistyp, Codeadresse lab.

Die Grammatik findet sich auf Seite 294 im Skript.
Abstrakte Syntax Seite 295.

13.3 javaStackP-Alpgebra, Seite 297

javaStackP umschließt im Gegensatz zu javaStack bei der Zusammenfassung einer Kommandofolge cs zu einem Block den code von cs mit zusätzlichen Zielcode. (Beispiel S. 302)

13.4 Weitere Lektüre

- Kapitel 5, wird auch die Übersetzung von Feldern und Records behandelt. (?)
- Grundlagen der Kompilation funktionaler Sprachen findet man in Kapitel 7
- Die Übersetzung oo Sprachen sind Thema von [1], Kapitel 5

13.5 Beispiel Programm (S. 314)

14 Mehrpässige Compiler (S. 319)

Wird nicht besprochen in der Veranstaltung

15 Funktoren und Monaden in Haskell

Sind aus FuPro *hoffentlich* bekannt.

[1] DEFINITION NOT FOUND.

16 Induktion, Coinduktion und rekursive Gleihcungen (S. 354)

Sollte auch bekannt sein.

17 Iterative Gleichungen (S. 370)

18 Interpretation in stetigen Algebren (S. 387)

19 Literatur (S. 420)