Mehrpässige Compilation mit finalen Coalgebren

Peter Padawitz
Universität Dortmund

9. November 2007

太极

Inhalt

- Von Grammatiken zu Algebren
- Attributierte Grammatiken sind Algebren
- Das tai chi der Modellierung: Algebren und Coalgebren
- Mehrpässige Übersetzung
- Schlusswort

Von Grammatiken zu Algebren

Beispiel Java-Grammatik

```
{fact = 1; while (x > 0) {fact = fact*x; x = x-1;}}
```

```
Block   --> {Seq}
Seq     --> empty | Command Seq
Command --> ; | String = IntE ; | Block | if (BoolE) Block |
            if (BoolE) Block else Block | while (BoolE) Block
IntE    --> Int | String | (IntE) | IntE - IntE | Sum | Prod
Sum     --> IntE | IntE + Sum
Prod    --> IntE | IntE * Prod
BoolE   --> true | false | IntE > IntE | not BoolE
```

Definition Sei $G = (N, T, P, start)$ eine CF-Grammatik. Die mehrsortige Signatur

$$\Sigma(G) = (N, F)$$

heißt **abstrakte Syntax** von G, falls

- es für jede Regel $p = (A \to w_0 A_1 w_1 \dots A_n w_n)$ von G mit $w_i \in T^*$ und $A_i \in N$ ein – **Konstruktor** genanntes – Funktionssymbol $f_p : A_1 \dots A_n \to A$ in F gibt und F keine weiteren Symbole enthält.

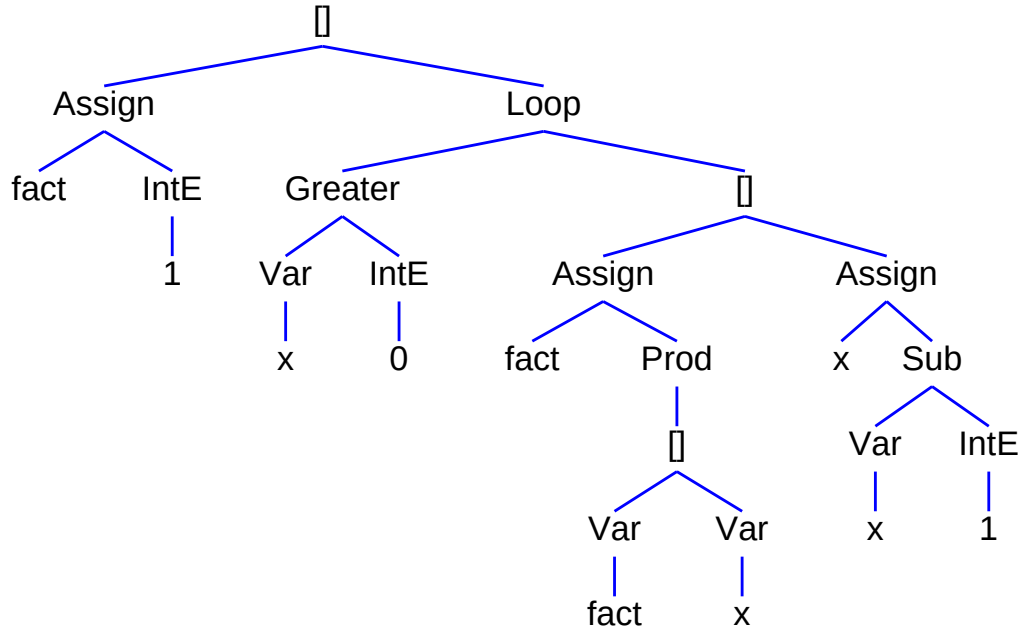
$\Sigma(G)$-Grundterme nennt man üblicherweise **Syntaxbäume von** G.

Beispiel Java-Signatur *JavaSig*

$$
\begin{aligned}
Sorten &= \{\ block, command, intE, boolE\} \\
Operationen &= \{\ mkBlock : [command] \to block, \\
&\quad [] :\to [command], \\
&\quad \_ : \_ : command \times [command] \to [command], \\
&\quad skip :\to command, \\
&\quad assign : String \times intE \to command, \\
&\quad cond : boolE \times block \times block \to command, \\
&\quad loop : boolE \times block \to command, \\
&\quad mkIntE : Int \to intE, \\
&\quad var : String \to intE, \\
&\quad sub : intE \to intE \to intE, \\
&\quad sum : [intE] \to intE, \\
&\quad prod : [intE] \to intE, \\
&\quad [] :\to [intE], \\
&\quad \_ : \_ : intE \times [intE] \to [intE], \\
&\quad mkBoolE : Bool \to boolE, \\
&\quad greater : boolE \times boolE \to boolE, \\
&\quad not : boolE \to boolE\}
\end{aligned}
$$

Beispiel *JavaSig* als Haskell-Datentyp ...

```
type Block   = [Command]
data Command = Skip | Assign String IntE | Cond BoolE Block Block |
               Loop BoolE Block
data IntE    = IntE Int | Var String | Sub IntE IntE | Sum [IntE] |
               Prod [IntE]
data BoolE   = BoolE Bool | Greater IntE IntE | Not BoolE
```



Beispiel ... und Haskell-Typklasse

```
class JavaOpns block command intE boolE where
                  mkBlock :: [command] -> block
                  skip    :: command
                  assign  :: (String,intE) -> command
                  cond    :: (boolE,block,block) -> command
                  loop    :: (boolE,block) -> command
                  mkIntE  :: Int -> intE
                  var     :: String -> intE
                  sum_    :: [intE] -> intE
                  prod    :: [intE] -> intE
                  sub     :: (intE,intE) -> intE
                  mkBoolE :: Bool -> boolE
                  greater :: (boolE,boolE) -> boolE
                  not_    :: boolE -> boolE
```

Beispiel ... und mit generischen Auswertungsfunktionen

```
eval_Block :: Block -> block
eval_Block cs = mkBlock cs
eval_Command :: Command -> command
eval_Command Skip              = skip
eval_Command (Assign x e)      = assign (x,eval_IntE e)
eval_Command (Cond be cs cs') = cond (eval_BoolE be,
                                      eval_Command cs,
                                      eval_Command cs')
eval_Command (Loop be cs)      = loop (eval_BoolE be,eval_Command cs)
eval_IntE :: IntE -> intE
eval_IntE (IntE i)   = mkIntE i
eval_IntE (Var x)    = var x
eval_IntE (Sub e e') = sub (eval_IntE e,eval_IntE e')
eval_IntE (Sum es)   = sum_ (map eval_IntE es)
eval_IntE (Prod es)  = prod (map eval_IntE es)
eval_BoolE :: BoolE -> boolE
eval_BoolE (BoolE b)       = mkBoolE b
eval_BoolE (Greater e e') = greater (eval_IntE e,eval_IntE e')
eval_BoolE (Not be)        = not_ (eval_BoolE be)
```

Beispiel Ein Interpreter als *JavaSig*-Algebra

```
instance Sigma (State -> State) (State -> State)
               (State -> Int) (State -> Bool) where
    mkBlock = foldl (flip (.)) id
    skip = id
    assign (x,f) st y = if x == y then f st else st y
    cond (f,g,h) st = if f st then g st else st
    loop (f,g) = cond (f,loop (f,g) . g,id)
    intE i _ = i
    var x st = st x
    sub (f,g) st = f st - g st
    sum_ fs st = foldl (+) 0 [f st | f <- fs]
    prod fs st = foldl (*) 1 [f st | f <- fs]
    boolE b _ = b
    greater (f,g) st = f st > g st
    not_ f st = not (f st)
```

Beispiel Ein Pretty-Printer als *JavaSig*-Algebra

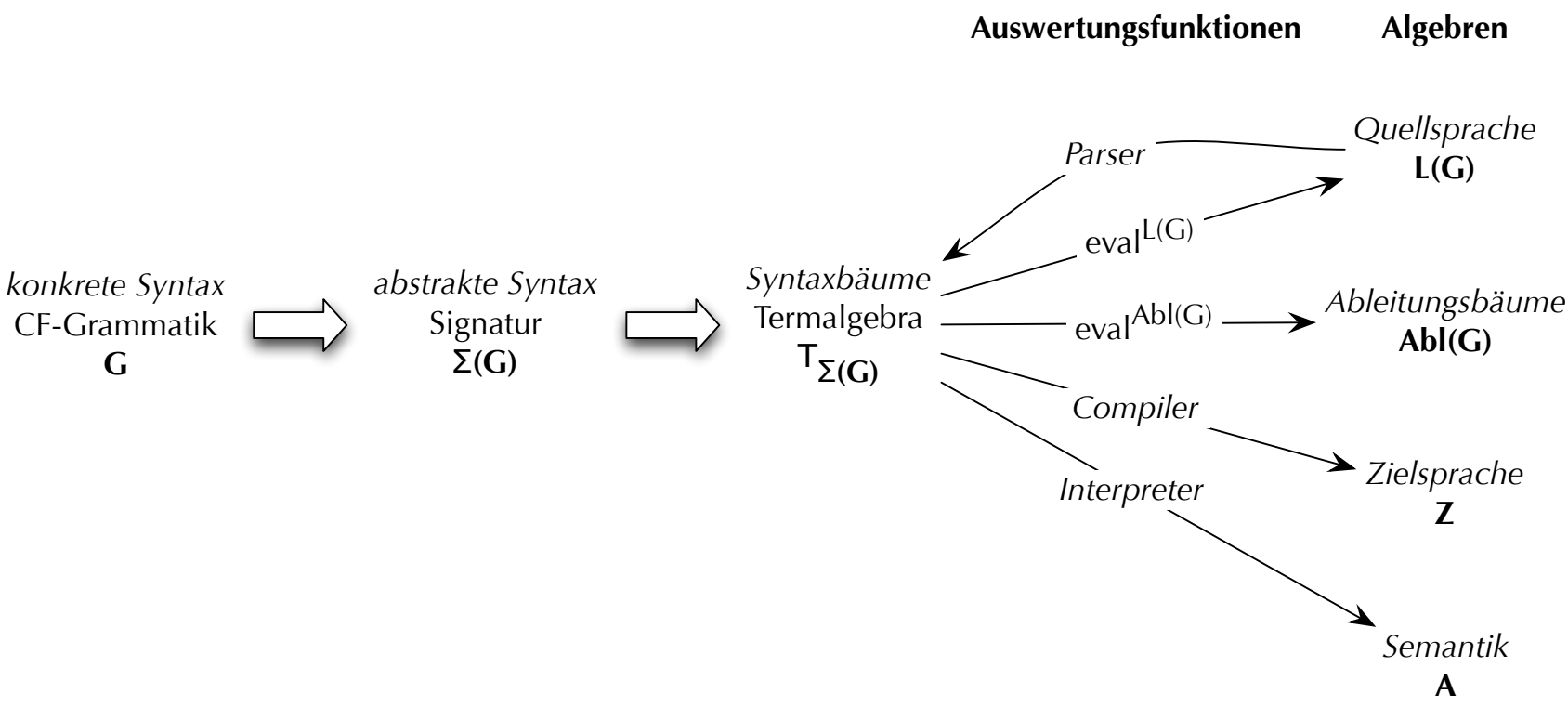
```
[Assign "fact" (IntE 1),
 Loop (Greater (Var "x")
               (IntE 0))
      [Assign "fact" (Prod[(Var "fact"),
                           (Var "x")]),
       Assign "x" (Sub (Var "x")
                       (IntE 1))]]
```

```
instance Sigma (Bool -> Int -> String) (Bool -> Int -> String)
               (Bool -> Int -> String) (Bool -> Int -> String) where
  mkBlock fs b n = append b n (str fs)
      where str []     = "[]"
            str [f]    = '[':f True (n+1)++"]"
            str (f:fs) = '[':f True (n+1)++',':
                              concatMap str' (init fs)++
                              (last fs) False (n+1)++"]"
                         where str' f = f False (n+1)++','
  skip b n = append b n "Skip"
  assign (x,f) b n = append b n str
      where str = "Assign "++show x++' ':f True (n+10+length x) e
  cond (f,g,h) b n = append b n str
      where str = "Cond "++f True (n+5) be++g False (n+5)++
                                              h False (n+5)
  loop (f,g) b n = append b n str
      where str = "Loop "++f True (n+5) be++g False (n+5)
```

```
initE i b n = append b n ("(IntE "++show i++")")
var x b n = append b n ("(Var "++show x++")")
sub (f,g) b n = append b n ("(Sub "++f True (n+5)++
                                        g False (n+5)++")")
sum_ (f:fs) b n = append b n str
    where str = "(Sum["++f True (n+5)++,:
                          concatMap str' (init fs)++
                          (last fs) False (n+5)++"])"
                where str' f = f False (n+5)++','
prod (f:fs) b n = append b n str
    where str = "(Prod["++f True (n+6)++,:
                           concatMap str' (init fs)++
                           (last fs) False (n+6)++"])"
                where str' f = f False (n+6)++','
boolE b' b n = append b n ("(BoolE "++show b++")")
greater (f,g) b n = append b n ("(Greater "++f True (n+9)++
                                              g False (n+9)++")")
not_ f b n = append b n ("(Not "++showBoolE True (n+5) be++")")

append b n str = if b then str else '\n':replicate n ' '++str
```

Von Grammatiken zu Algebren



Attributierte Grammatiken sind Algebren

Sei $G = (N, T, P, start)$ eine CF-Grammatik. Zu jedem $s \in N$ gibt es **vererbte Attribute** $s.v_1, ..., s.v_m$ und **abgeleitete Attribute** $s.a_1, ..., s.a_n$.

Jedes $p \in P$ wird mit Attributzuweisungen versehen:

$$p \ : \ s \longrightarrow \ldots s_1 \ldots s_k \ldots \qquad \begin{array}{l} s_1.v_{11} := e_{11} \ \ldots \ s_1.v_{1n_1} := e_{1n_1} \\ \vdots \\ s_k.v_{k1} := e_{k1} \ \ldots \ s_k.v_{kn_k} := e_{kn_k} \\ s.a_1 := e_1 \ \ldots \ s.a_n := e_n \end{array}$$

Die Ausdrücke e_{ij} können vererbte Attribute von $s, s_1, \ldots s_k$ enthalten.

Was bedeutet das?

- Als **Programmvariablen** sind vererbte und abgeleitete Attribute semantisch (Komponenten von) **Funktionen**, die Werten vererbter Attribute Werte abgeleiteter Attribute zuordnen:

$$Dom(s.v_1) \times \ldots \times Dom(s.v_m) \;\rightarrow\; Dom(s.a_1) \times \ldots \times Dom(s.a_n)$$

- Demzufolge sind **Zuweisungen** an Attribute **Updates** von Funktionen.
- Diese Updates definieren eine Interpretation der Syntaxbäume von G, also eine $\Sigma(G)$**-Algebra** A.
- Die Trägermengen von A sind die o.g. Funktionen:

$$A_s \;=\; [Dom(s.v_1) \times \ldots \times Dom(s.v_m) \;\rightarrow\; Dom(s.a_1) \times \ldots \times Dom(s.a_n)]$$

- Z.B. interpretiert die obige attributierte Regel den Konstruktor

$$f_p : s_1 \times \ldots \times s_k \rightarrow s \;\in\; \Sigma(G)$$

wie folgt: Für alle $1 \leq i \leq k$ sei $g_i \in A_{s_i}$.

$$f_p^A(g_1, \ldots, g_k)(x_1, \ldots, x_m) = (e_1, \ldots, e_n)$$
$$\text{where } (y_{11}, \ldots, y_{1n_1}) = g_1(e_{11}, \ldots, e_{1n_1})$$
$$\vdots$$
$$(y_{k1}, \ldots, y_{kn_k}) = g_k(e_{k1}, \ldots, e_{kn_k})$$

$\rightsquigarrow$ **Eine attributierte Grammatik mit zugrundeliegender CF-Grammatik G ist eine $\Sigma(G)$-Algebra.**

Mehrpässige Übersetzung

$$f_p^A(g_1, \ldots, g_k)(x_1, \ldots, x_m) = (e_1, \ldots, e_n)$$
$$\text{where } (y_{11}, \ldots, y_{1n_1}) = g_1(e_{11}, \ldots, e_{1n_1})$$
$$\vdots$$
$$(y_{k1}, \ldots, y_{kn_k}) = g_k(e_{k1}, \ldots, e_{kn_k})$$

Die (logischen) Variablen $x_1, \ldots, x_m$ und $y_{i1}, \ldots, y_{in_i}$ stehen für Werte abgeleiteter Attribute von s bzw. vererbter Attribute von s_i.

Vorkommen der letzteren in den Ausdrücken e_{ij} können **zyklische Abhängigkeiten** beschreiben. Diese lassen sich oft durch eine **Zerlegung** der gesamten Attributmenge in r Mengen

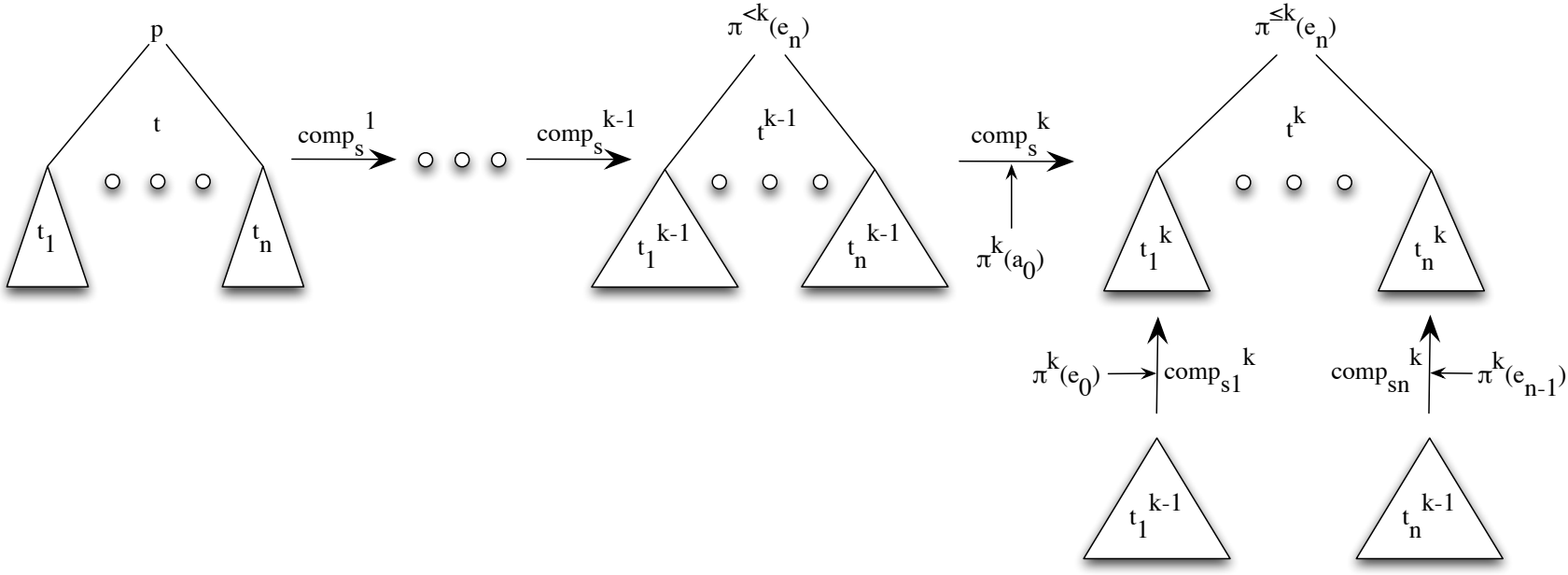
$$At^1, \ldots, At^r$$

und eine entsprechende Zerlegung der Algebra A in r Algebren (**Pässe**)

$$A^1, \ldots, A^r$$

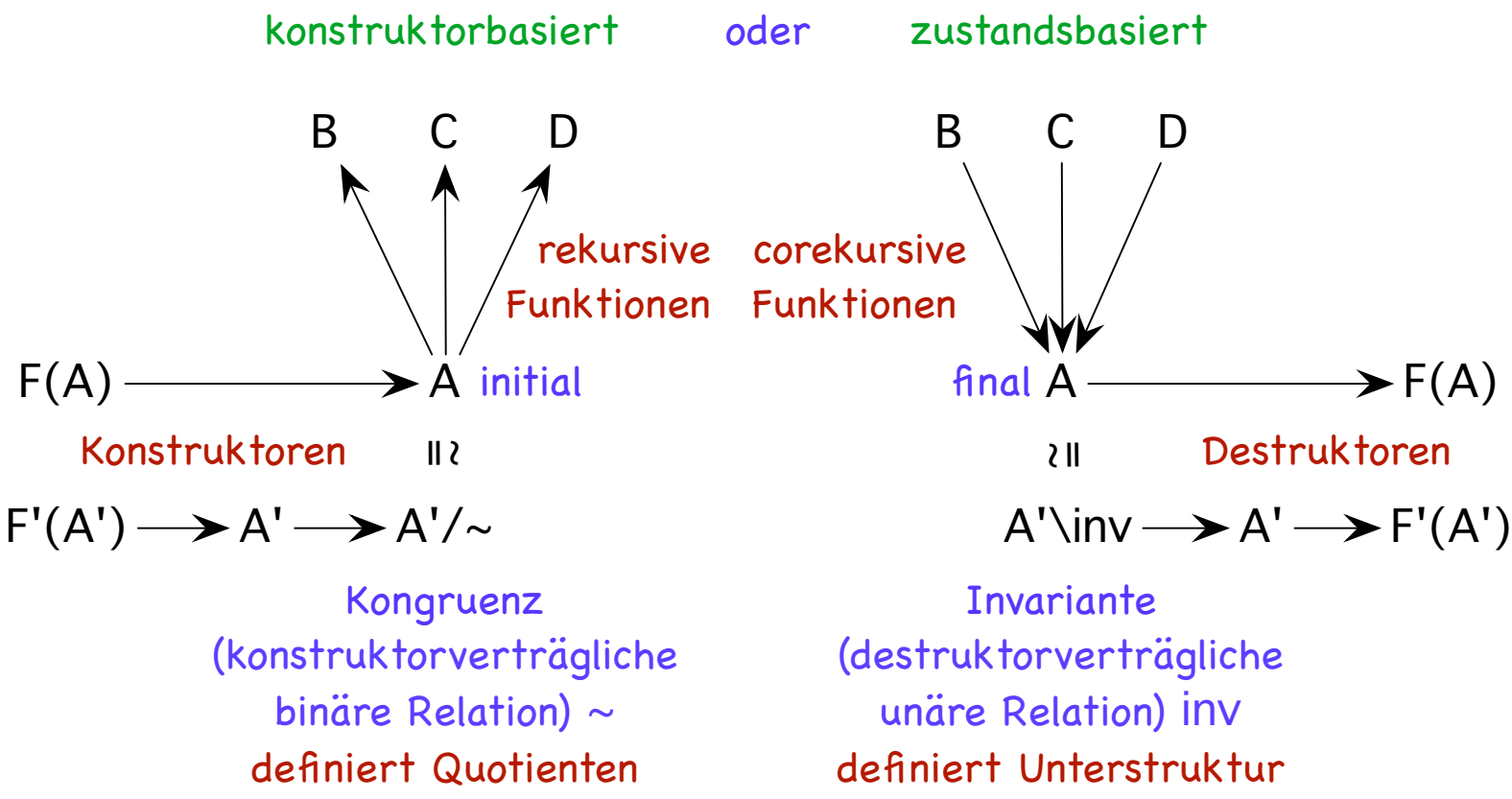
auflösen.

Sei $1 \leq i \leq r$. Im Gegensatz zu A haben die (funktionalen) Trägermengen von A^i einen weiteren Parameter, nämlich Syntaxbäume, deren Knoten mit allen vor dem i-ten Pass berechneten Attributwerten markiert sind, denn die – in A^i durchgeführte – Berechnung von At^i-Werten im i-ten Pass muss auf jene Attributwerte zurückgreifen!

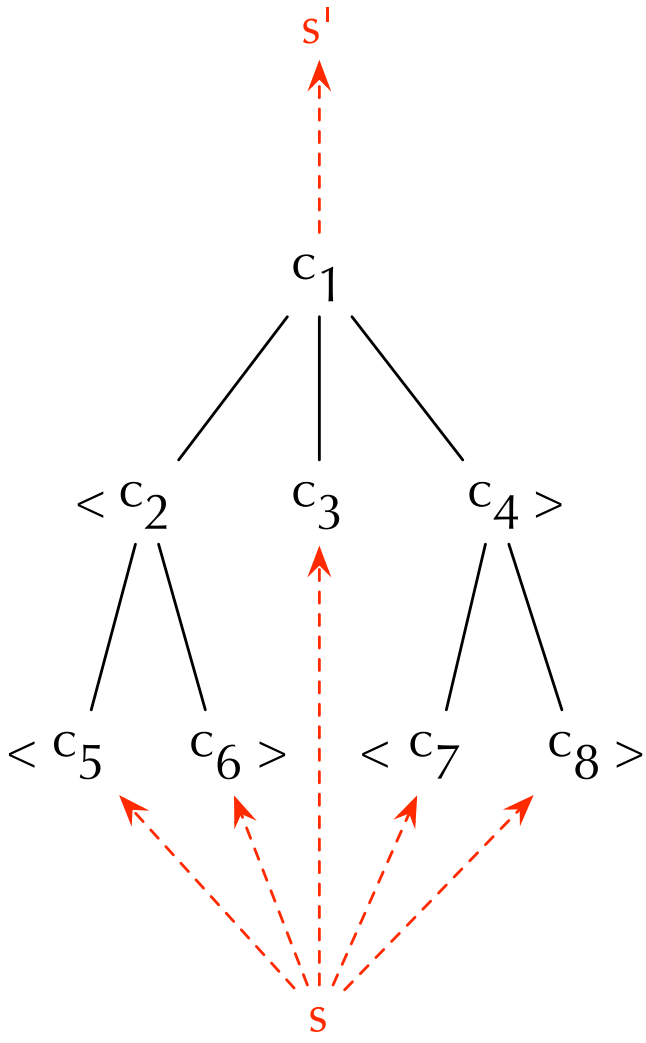


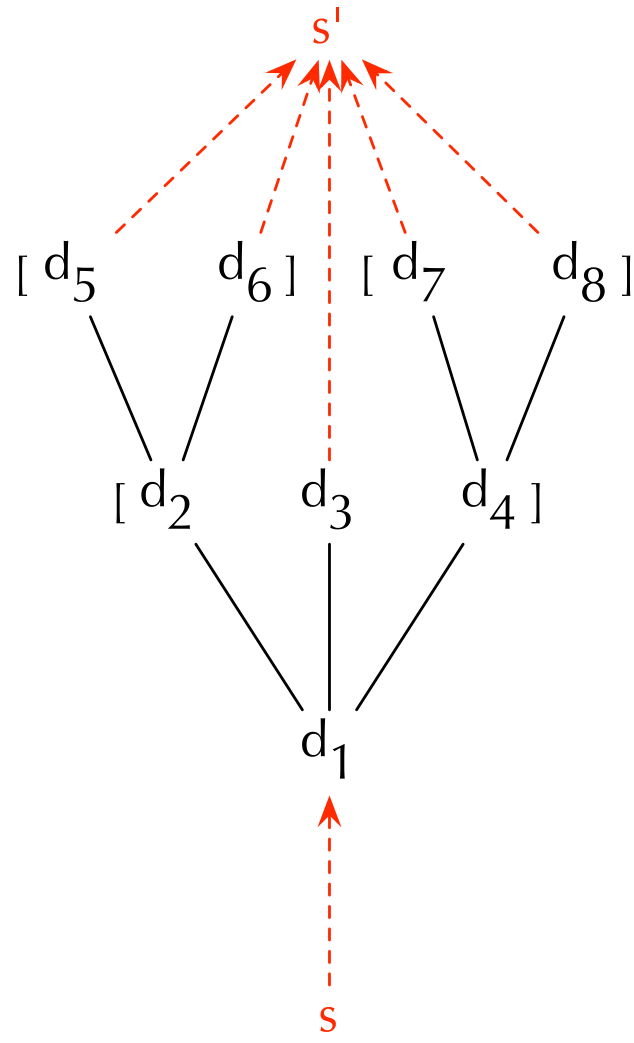
Attributierte Syntaxbäume bilden eine (finale) $\Sigma(G)$-Coalgebra.

Das tai chi der Modellierung: Algebren und Coalgebren



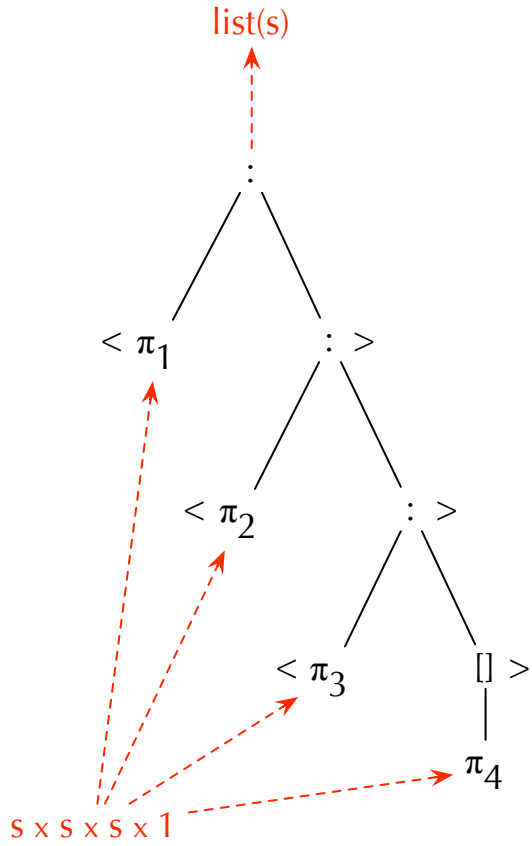
Konstruktoren und Destruktoren als Generatoren bzw. Beobachter

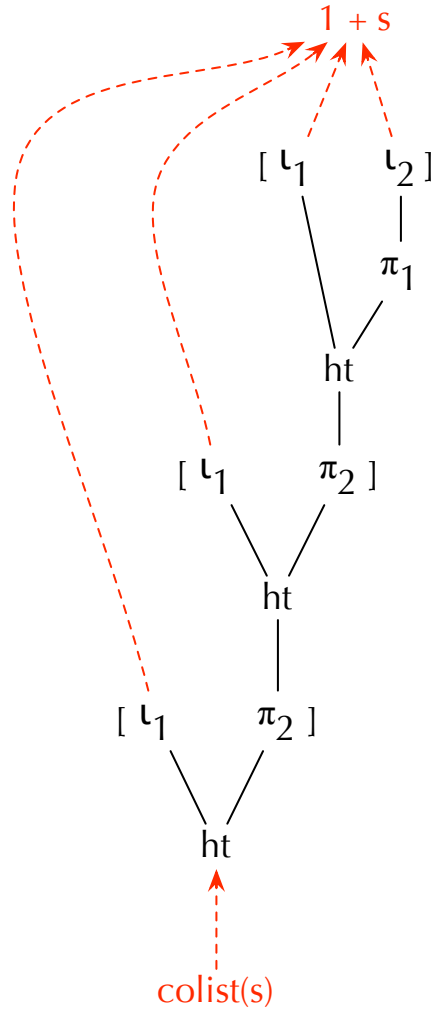




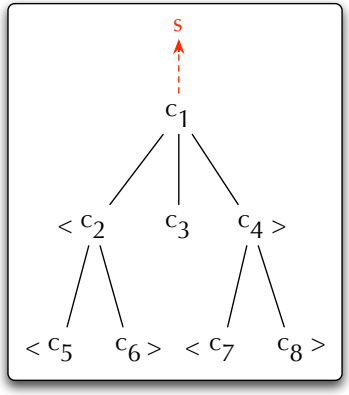
Dreielementige Liste

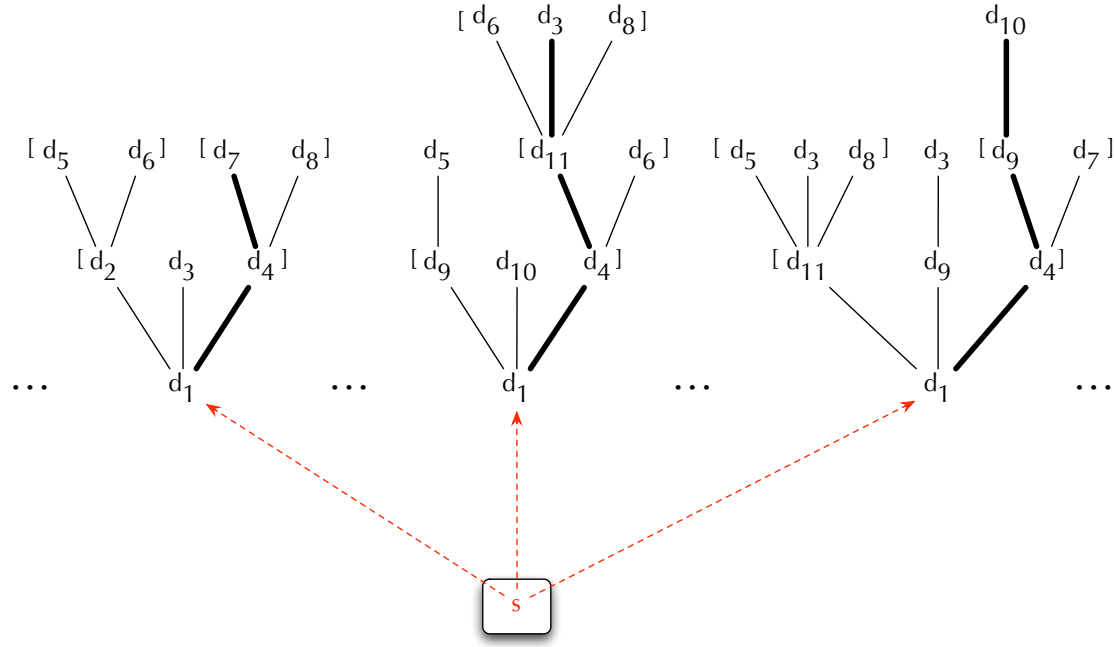
Wie lautet das dritte Element (falls es existiert)?





Konstruktoren und Destruktoren in initialen bzw. finalen Modellen





Beispiele für Destruktoren einer Sorte s

Rote Sorten sind **primitiv**, d.h. haben in allen Modellen dieselbe feste Interpretation.
s und s' sind Zustandssorten.

nat. Zahlen mit ∞	$pred:$	s	$\longrightarrow$	$1 + s$
endl. oder unendl. Listen	$ht:$	s	$\longrightarrow$	$1 + a \times s$
endl. oder unendl. Bäume	$rs:$	s	$\longrightarrow$	$a \times s'$
	$ht:$	s'	$\longrightarrow$	$1 + s \times s'$
det. Automaten	$trans:$	s	$\longrightarrow$	$a \longrightarrow s$
	$out:$	s	$\longrightarrow$	b
Petri – Netze	$trans:$	s	$\longrightarrow$	s
	$place_i:$	s	$\longrightarrow$	a_i
nichtdet. Automaten	$trans:$	s	$\longrightarrow$	$a \longrightarrow \coprod_n s^n$
	$out:$	s	$\longrightarrow$	b

Mengen	$elem:$	s	$\rightarrow$	$a \rightarrow bool$
Multimengen	$card:$	s	$\rightarrow$	$a \rightarrow nat$
gewichtete Mengen	$weight:$	s	$\rightarrow$	$a \rightarrow int$
Funktionen (Felder)	$apply:$	s	$\rightarrow$	$a \rightarrow b$
OO – Klassen	$method_i:$	s_i	$\rightarrow$	s
	$attr_i:$	s	$\rightarrow$	a_i
UML – Klassendiagramme	$assoc_{s,s',i}:$	s	$\rightarrow$	$\coprod_n (s')^k$
	$method_{s,i}:$	s	$\rightarrow$	s
	$attr_{s,i}:$	s	$\rightarrow$	$a_{s,i}$
XML – Schemata	$args_s:$	s	$\rightarrow$	$\coprod_{f:s_1\times\ldots\times s_n\rightarrow s\in F} s_1 \times \ldots \times s_n$
F = Menge von Konstruktoren.	$attr_{s,i}:$	s	$\rightarrow$	$a_{s,i}$
	$link_{s,i}:$	s	$\rightarrow$	s'
attributierte Grammatiken	$args_s:$	s	$\rightarrow$	$\coprod_{f:s_1\times\ldots\times s_n\rightarrow s\in F} s_1 \times \ldots \times s_n$
F = Menge von Konstruktoren.	$attr_{s,i}:$	s	$\rightarrow$	$v \rightarrow a_{s,i}$

Theorem Jeder polynomiale Funktor F hat einen kleinsten Fixpunkt *lfp* und einen größten Fixpunkt *gfp* (d.h. alle Morphismen mit Zielobjekt *lfp* oder Quellobjekt *gfp* sind Isomorphismen), m.a.W. es gibt

eine **initiale F-Algebra** $F(lfp) \to lfp$ und eine **finale F-Coalgebra** $gfp \to F(gfp)$. ❏

Sei $\Sigma = (S, F)$ eine Konstruktorsignatur. Der Funktor

$$F_\Sigma : Set^S \to Set^S$$

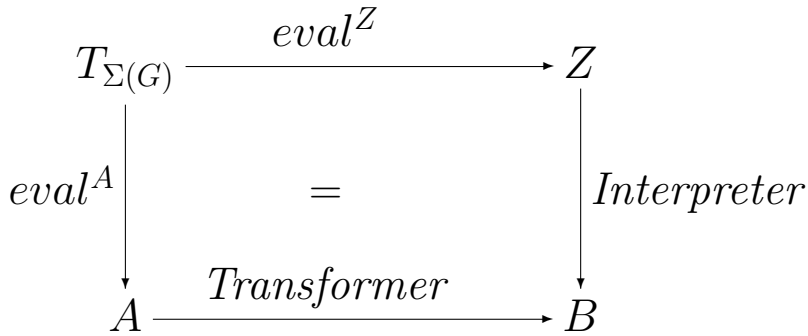
mit

$$F_\Sigma(A)_s \;=\; \coprod_{f:w\to s \,\in\, F} A_w$$

für alle S-sortierten Mengen A und $s \in S$ polynomial.

Die Menge T_Σ der endlichen Σ-Grundterme ist Trägermenge der initialen F_Σ-Algebra (= Σ-Algebra).

Die Initialität vereinfacht z.B. die Compilerverifikation.



Korrektheit eines Compilers $eval^Z$ in eine Zielsprache Z

Die Menge T_Σ^∞ der endlichen oder unendlichen Σ-Grundterme ist Trägermenge der finalen F_Σ-Coalgebra (= Σ-Coalgebra = des_Σ-Algebra).

Was ist des_Σ?

Definition Sei $\Sigma = (S, F)$ eine Konstruktorsignatur. Eine Signatur $\Sigma' = (S, F')$ heißt **Destruktorsignatur über** Σ, falls

- F' für jede Sorte $s \in S$ ein **inverser S-Konstruktor** genanntes Funktionssymbol

$$args_s : s \to \coprod_{f:w \to s \in F} w,$$

  enthält und
- alle weiteren Funktionssymbole S-Destruktoren sind, also einen Typ der Form $s \to s'$ mit $s \in S$ haben.

des_Σ bezeichnet die durch Σ eindeutig bestimmte Destruktorsignatur, die nur inverse S-Konstruktoren enthält.

Da alle Funktionen einer Destruktorsignatur $\Sigma' = (S, F')$ S-Destruktoren sind, gibt es eine finale Σ'-Algebra $Fin(\Sigma')$.

Sind alle Destruktoren $f : s \to s'$ von Σ', die keine inversen S-Konstruktoren sind, **S-Attribute**, d.h. s' ist primitiv, dann

besteht die Trägermenge von $Fin(\Sigma')$ aus allen endlichen oder unendlichen Σ-Grundtermen, deren Knoten mit jeweils einem Tupel

$$a \in \prod_{f:s\to s',\ s'\ primitiv} A_{s'}$$

markiert ist, wobei $A_{s'}$ die feste Interpretation der primitiven Sorte s' ist.

Für alle $t \in Fin(\Sigma')_s$ ist $args_s^{Fin(\Sigma')}(t)$ ist das Tupel von Unterbäumen, das entsteht, wenn man die Wurzel von t streicht.

Die finale $\Sigma(G)$-Coalgebra der attributierten Syntaxbäume

Die Zerlegung $At^1, \ldots, At^r$ der Attributmenge einer attributierten Grammatik G definiert r Destruktorsignaturen $\Sigma^1, \ldots, \Sigma^r$ über Σ: Seien $1 \leq i \leq r$ und $s.a^i_1, \ldots, s.a^i_{n_i}$ die bis zum i-ten Pass abgeleiteten Attribute von s.

$$\Sigma^i \;=\; (S, \{subs_s \mid s \in S\} \cup \{at^i_s : s \to Dom(s.a^i_1) \times \ldots \times Dom(s.a^i_{n_i}) \mid s \in S\}.$$

Die Definition der Σ-Algebren $A^1, \ldots, A^r$ ergibt sich aus der "Definition" von A.
Sei $1 \leq i \leq r$, $s \in S$ und $s.v^i_1, \ldots, s.v^i_{n_i}$ die an den i-ten Pass vererbten Attribute von s.

- $A^i_s \;=\; [Dom(s.v^i_1) \times \ldots \times Dom(s.v^i_{m_i}) \;\rightarrow\; Fin(\Sigma^{i-1})_s \;\rightarrow\; Fin(\Sigma^i)_s]$.

- Sei $f_p : s_1 \times \ldots \times s_k \rightarrow s \;\in\; \Sigma(G)$ und $\pi^i, \pi^{\leq i}$ Projektionen auf Attributwertetupeln: π^i und $\pi^{<i}$ filtern die Werte der an den i-ten Pass vererbten bzw. bis zum i-ten Pass abgeleiteten Attribute heraus. ι_f bettet $Fin(\Sigma^i)_{s_1 \times \ldots \times s_k}$ in die Summe $\coprod_{f \,:\, w \rightarrow s \,\in\, F} Fin(\Sigma^i)_w$ ein. Für alle $1 \leq j \leq k$ sei $g_j \in A^i_{s_j}$.

$$\langle at^i_s, args_s \rangle (f^{A^i}_p(g_1, \ldots, g_k)(\pi^i(x_1, \ldots, x_m))(t^{i-1}) = (\pi^{\leq i}(e_1, \ldots, e_n), \iota_f(t^i_1, \ldots, t^i_k))$$
$$\text{where } t^i_1 = g_1(\pi^i(e_{11}, \ldots, e_{1n_1}))(t^{i-1}_1)$$
$$\vdots$$
$$t^i_k = g_k(\pi^i(e_{k1}, \ldots, e_{kn_k}))(t^{i-1}_k)$$
$$(t^{i-1}_1, \ldots, t^{i-1}_k) = subs_s(t^{i-1})$$

- Die zyklenfreie Interpretation von f_p in A lautet schließlich wie folgt:

$$f^A_p(g_1, \ldots, g_k)(x_1, \ldots, x_m) = at^r_s(t^r)$$
$$\text{where } t^1 = f^{A^1}_p(g_1, \ldots, g_k)(\pi^1(x_1, \ldots, x_m))$$
$$\vdots$$
$$t^r = f^{A^r}_p(g_1, \ldots, g_k)(\pi^r(x_1, \ldots, x_m))(t^{r-1})$$

Schlusswort

- Dialgebraische Spezifikationen benötigen verallgemeinerte Signatur, Term- und Formelbegriffe:
  - (logische) Variablen werden zu Projektionen,
  - Terme zu Morphismen (zwischen polynomialen Typen),
  - Formeln zu Relationen (Lösungsmengen)
    $\leadsto$ globale Semantik, Einbindung von Modal- und Temporallogiken
- Erst die Kombination aus initialen und finalen Algebren liefert eine adäquate Semantik für attributierte Grammatiken, XML-Schemata, Klassendiagramme, etc., in der man auch rechnen und etwas beweisen kann (Simplifikation, Rewriting, Resolution, Narrowing, Induktion, Coinduktion).