Typische Muster bei der Implementierung Domänen-spezifischer Sprachen mit Attributgrammatiken*

Christian Berg und Wolf Zimmermann

[1] christian.berg@informatik.uni-halle.de
[2] wolf.zimmermann@informatik.uni-halle.de
Institut für Informatik
Martin-Luther-Universität Halle-Wittenberg

Zusammenfassung. Diese Arbeit präsentiert typische Muster auf Attributgrammatiken und Definitionstabellen. Die präsentierten Muster nutzen, im Gegensatz zu vielen bestehenden Erweiterungen von Attributgrammatiken, Typen und Baumstruktur aus. Analog den Entwurfsmustern Objekt-orientierter Sprachen werden die vorgestellten Muster auf Laufzeiteinfluss, Speichereinfluss und Einsatzgebiet untersucht.

1 Einleitung

Programmiersprachen sowie Domänen-spezifische Sprachen(DSLs[1]) lassen sich mit Attributgrammatiken entwickeln[18,21]. Bei der Entwicklung von Sprachen mit Attributgrammatiken werden ähnliche Attributierungsregeln verwendet. Aufgrund dieser *typischen Muster* existieren eine Reihe von Erweiterungen und Bibliotheken für die Vereinfachung der Spezifikation unter Verwendung von Attributgrammatiken wie bspw. [8,11,25,23,10,16,17,2].

Werden Definitionstabellen und geordnete Attributgrammatiken[12] genutzt, so ist die Auswahl an typischen Mustern eine andere, als dies bspw. bei der Verwendung von Higher-Order Attributgrammatiken der Fall wäre. Letztere erlauben keine Funktionen mit Seiteneffekten und betrachten, im Gegensatz zu geordneten Attributgrammatiken, Attribute nicht als Vor- und Nachbedingungen[11,23]. Analog Entwurfsmustern[9] stellen wir eine Auswahl der typischen Muster in geordneten Attributgrammatiken vor und evaluieren diese bzgl. Laufzeiteinfluss, Speicherverbrauch und Anwendungsfällen.

In Abschnitt 2 stellen wir nochmals kurz Attributgrammatiken und die von uns verwendete Syntax vor. Abschnitt 3 behandelt bestehende und von uns formulierte Muster, welche in Abschnitt 4 evaluiert werden. Ein Überblick über verwandte Arbeiten folgt in Abschnitt 5. Wir geben eine Zusammenfassung und einen Ausblick in Abschnitt 6.

*Diese Arbeit wurde im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Forschungsprojekts „ELSY“ (Nr. 16M3202D) erstellt.

[1] engl: domain specific languages

⟨*Program*⟩ ::= ⟨*Decls*⟩

⟨*Decls*⟩ ::= ⟨*Decls*⟩ ⟨*Decl*⟩
| ε

⟨*Decl*⟩ ::= ⟨*VariableDecl*⟩

⟨*VariableDecl*⟩ ::= ⟨*VarDef*⟩ ⟨*Expression*⟩

⟨*Decl*⟩ ::= ⟨*Eval*⟩

⟨*Eval*⟩ ::= ⟨*Expression*⟩

⟨*Expression*⟩ ::= ⟨*Expression*⟩ '+' ⟨*Expression*⟩
| ⟨*Expression*⟩ '-' ⟨*Expression*⟩
| ⟨*VarReference*⟩
| ⟨*Constant*⟩

⟨*VarReference*⟩ ::= identifier

⟨*VarDef*⟩ ::= identifier

⟨*Constant*⟩ ::= number

Abb. 1 – Abstrakte Syntax einer Sprache zur Auswertung von Ausdrücken; Nichtterminale groß beginnend, Terminale klein beginnend

2 Grundlagen

Wir folgen in unserer Präsentation den Darstellungen aus [18], [12] sowie [11]. Eine Kontext-freie Grammatik $\mathsf{G} \triangleq (N, T, P, Z)$ mit Nichtterminalen N, Terminalen T, ausgezeichneter Wurzel $Z \in N$ und Produktionen $p \in P$ definiert eine Sprache $L(\mathsf{G})$. Wird durch G eine Sprache $L(\mathsf{G})$ definiert, die eine gültige Folge von Grundsymbolen angibt, dann wird G als *konkrete* Syntax bezeichnet. Definiert G hingegen Baumaufbaukonstruktoren heißt G *abstrakte* Syntax. Produktionen werden durch BNF oder EBNF dargestellt und sind eindeutig identifizierbar. Ein Beispiel einer abstrakten Syntax für eine sehr einfache Ausdruckssprache findet sich in Abbildung 1.

Ein Symbol $\mathtt{Y}$ ($\in (N \cup T)$) heißt **ableitbar** aus $\mathtt{X}$, geschrieben als $\mathtt{X} \Rightarrow \mathtt{Y}$, genau dann, wenn eine Produktion mit linker Seite $\mathtt{X}$ existiert, bei der auf der rechten Seite das Symbol $\mathtt{Y}$ vorkommt; formal: $\exists p \in P, p : \mathtt{X} ::= u\ \mathtt{Y}\ \mathtt{v}$ für $u, v \in (N \cup T)^*$. Der reflexiv-transitive Abschluss einer Ableitung wird mit $\overset{*}{\Rightarrow}$ notiert. Üblicherweise werden u und v von uns nicht weiter beachtet und daher auch nicht aufgeführt.

Für eine abstrakte Syntax $\mathsf{G} \triangleq (N, T, P, Z)$ lässt sich nun eine Attributgrammatik $AG \triangleq (\mathsf{G}, A, R, B)$ definieren, wobei A die Menge aller Attribute für alle Terminale und Nichtterminale der abstrakten Syntax G sind. Für ein Symbol $\mathtt{X} \in N$ mit Attribut $a \in A(\mathtt{X})$ schreiben wir $\mathtt{X}.a$. Für eine Produktion $p \in P$, Symbole $\mathtt{X}_i \in (N \cup T), i \in [1, n]$ und Nichtterminal $\mathtt{X}_0$ schreiben wir p als $p : \mathtt{X}_0$ $\mathtt{::=}\ \mathtt{X}_1\ \cdots\ \mathtt{X}_n$. Solch einer Produktion ist eine Attributierungsregel $r \in R(p)$ zugeordnet; für eine beliebige Funktion f und Symbole der Produktion mit Indizes $j, k, l \in [0, n]$ und beliebige, den Symbolen zugeordnete Attribute $a, b, \ldots, u \in A$, hat r die Form $\mathtt{X}_j.a \leftarrow f(\mathtt{X}_k.b, \ldots, \mathtt{X}_l.u)$. Analog exisiteren Produktionen zugeordnete Berechnungen B. Ein solches Attribut $\mathtt{X}_j.a$ heißt **definiert** in p; definierte Attribute für ein Symbol auf der linken Seite einer Produktion heißen **synthetisiert**, definierte Attribute für Symbole der rechten Seite einer Produktion **ererbt**. Sind bei Berechnungen oder Attributierungsregeln Infixoperationen üblich, so verwenden wir ebenfalls Infixoperationen, die Identitätsfunktion wird von uns nicht mit aufgeführt.

```
1  declare_prop val :: Int ← 0
2
3  rule Expression ::= Constant
4  attr Expression.val ← str_to_int(Constant.sym)
5
6  rule Expression ::= VarReference
7  attr Expression.val ← VarReference.key:val
8
9  rule Expression_1 ::= Expression_2 '-' Expression_3
10 attr Expression_1.val ←Expression_2.val - Expression_3.val
11
12 rule Expression_1 ::= Expression_2 '+' Expression_3
13 attr Expression_1.val ←Expression_2.val + Expression_3.val
14
15 rule VariableDecl ::= VarDef Expression
16 attr VarDef.key:val ← Expression.val
17    VariableDecl.names_chn ← Expression.names_chn >= VarDef.key:val
18
19 chain names_chn
20 symbol Program
21 attr head.names_chn ← ∅
22
23 symbol VarDef
24 attr ↑bind ← bind_in_env(↓names_chn, ↑sym)
25    ↑key ← keyof(↑bind)
26    ↑has_err ← ↑sym ∉ ↓names_chn
27
28    cond ↑sym ∉ ↓names_chn
29      => error "Already declared: " ++ ↑sym
30
31 symbol VarReference
32 attr ↑bind ← lookup(↓names_chn, ↑sym)
33    ↑key ← keyof(↑bind)
34    ↑has_err ← ↑sym ∈ ↓names
35
36    cond ↑sym ∈ ↓names_chn
37      => error "Unknown symbol " ++ ↑sym
38
39 rule Eval ::= Expression
40 attr Eval.val ← Expression.val
41
42 ...
43
44 symbol Program
45 attr cond ↑has_err =>
46      error "Cannot evaluate program."
47    print(↑val)
48
49 bind_in_env :: Env => String => Bind
50 keyof :: Bind => Key
51 print :: Int => ()
52 str_to_int :: String => Int
```

Quelltext 1 – Ausschnitt der Implementierung der Ausdruckssprache mit abstrakter Syntax aus Abb. 1

Wir benutzen in der Praxis geordnete Attributgrammatiken, daher werden Attributierungsregeln und Berechnungen als Vor- und Nachbedingungen aufgefasst[11,12]. Für geordnete Attributgrammatiken lassen sich effiziente Evaluatoren generieren[12]. Eine detaillierte Vorstellung von Attributgrammatiken mit verschiedenen Unterarten findet sich in [3]. Für die Präsentation in dieser Arbeit reicht folgende Definition geordneter Attributgrammatiken aus:

Definition 1. *Eine Attributgrammatik* $AG \triangleq (\mathsf{G}, A, R, B)$ *mit abstrakter Syntax* $\mathsf{G} \triangleq (N, T, P, Z)$ *heißt* ***geordnet*** *wenn für jedes Symbol* $\mathtt{X} \in (N \cup T)$ *und damit assoziierten Attributen* $A(\mathtt{X})$ *eine Halbordnung existiert, sodass in jedem Kontext von* $\mathtt{X}$ *die Auswertereihenfolge der Attribute in diesem Kontext diese Halbordnung enthält[12].*

Eine wesentliche Erweiterung, die für die Verwendung typischer Muster notwendig ist, findet sich mit Symbolen und Klassen in Attributgrammatiken.

Definition 2. *Sei* $AG \triangleq (\mathsf{G}, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $\mathsf{G} \triangleq (T, N, P, Z)$*, Nichtterminalen* $\mathtt{X}_0 \in N, \mathtt{X}_i \in (N \cup T), i \in [1, n], n \in \mathbb{N}$*. Für alle Produktionen* $p, q_i \in P$ *der Form* $p : \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$*,* $q_i : \mathtt{X}_i ::= v_i$*, wobei* $v_i \in (T \cup N)^*$*, den Attributen* $a, b \in A(\mathtt{X}_i)$ *sowie beliebigen Ausdrücken* e_0, e_1 *existieren folgende Regelerzeugungsmuster:*

Symbole (u.a. [11,15,19]):

```
1  symbol X1
2  attr ↓a ← 10
3       ↑b ← f(this.a)
```

entspricht

```
1  rule q1: X1 ::= v
2  attr X1.b ←f(X1.a)
3
4  rule p: X0 ::= X1 ··· Xn
5  attr X1.a ←10
```

Klassen (u.a. [11,15,19]):

```
1  class symbol V
2  attr ↓a ← 10
3       ↑b ← 100
4  symbol X1 ←V
```

entspricht

```
1  symbol X1
2  attr ↓a ← 10
3       ↑b ← 100
```

Head und Tail (ebenfalls [11,15,19]):

```
1  symbol X0
2  attr head.a ← e0
3       ↑b ← tail.b
```

entspricht

```
1  rule p: X0 ::= X1 ··· Xn
2  attr X1.a ←e0
3       X0.b ←Xn.b
```

Attributierungsregeln werden, abhängig vom Attributtyp (ererbt, synthetisiert), in Symbole bzw. Produktionen übernommen. Wird in einer Produktion ein Attribut eines Symbols bereits definiert, für das eine Symbol-Attributierungsregel existiert, so wird diese nicht übernommen. Analoges gilt für Symbole und Klassensymbole. Zuerst werden Klassensymbol-Attributierungsregeln in Symbole übernommen, dann Symbol-Attributierungsregeln in Produktionen. Wird in mehreren Klassensymbolen ein Attribut definiert und erbt ein Symbol von mehreren dieser Klassensymbole, dann ist die attributierte Grammatik konsistent, genau dann wenn alle vererbenden Klassensymbole dieselbe Attributierungsregel für das definierte Attribut angeben.

Für beliebige Symbole S stehen demnach **`head`** und **`tail`** für das erste bzw. letzte Symbol auf der rechten Seite aller Produktionen mit linker Seite S.

```
1  symbol X
2  attr ↑a ← 10
3  -- Propagation von X.a als Attribut b mit dem Wert von X.a
4  propagate X.a as b
5  -- Inklusion des Attributs
6  symbol Y
7  attr ↓c ← include X.a
8  -- Kettenattribut
9  chain a
```

Quelltext 2 – Erweiterung von Attributgrammatiken mit Inklusion, Propagation, Kettenberechnung und Beträgen, `X` $\overset{*}{\Rightarrow}$ `Y`, Wurzel `Z`[15,7]

Ausgehend von Definition 2 lassen sich Inklusion, Ketten und Unterbaumzugriffe definieren. Eine umfangreiche Behandlung dieser findet sich u.a. in [11,2] sowie [7]. Quelltext 2 zeigt enige dieser existierenden Erweiterungen, wobei der Zugriff auf Attribute in einem höheren Kontext über **`include`** geschieht, **`propagate`** `X.`a **`as`** b analog der Inklusion ein Attribut b mit dem Wert von `X.`a in jedem von `X` aus ableitbaren Nichtterminal definiert und **`chain`** für ein Attribut a eine vorgefertigte Auswertereihenfolge definiert, bei der zuerst von links nach rechts in die Tiefe zur Berechnung abgestiegen wird[15,14]. Diese Auswertereihenfolge kann auch optimiert werden – es ist nicht notwendig in Teilbäume abzusteigen, in denen weder lesender noch schreibender Attributzugriff statt findet, d.h. weder Vor- und Nachbedinungen durch das Kettenattribut in diesem Teilbaum definiert werden[14].

Kommt ein Nichtterminal auf der linken Seite nur in Form von Kettenproduktionen, d.h. Produktionen mit genau einem Symbol auf der rechten Seite der Produktion, vor, dann werden die Kettenattributierungen zum Holen eines synthetisierten Attributs von uns nicht mit aufgeschrieben (Lösungsansätze dafür werden u.a. in [22,15,14] vorgestellt). Wie in [10] und [11] beschrieben, können diese Attributierungen aus einer Spezifikation heraus generiert werden. Bei Nichtterminalen aus denen nur Terminale direkt ableitbar sind, wird auf den Wert, bzw. die Zeichenfolge, des Terminalsymbols mit dem Attribut `sym` von diesem Nichtterminal aus zugegriffen.

Die Typisierung der Funktionen, Attribute, Bedingungen und Ausdrücken folgt typischerweise der Programmiersprache, für die ein Evaluator generiert wird oder in der der Evaluator interpretiert wird. Für die Präsentation in dieser Arbeit wird ein an Haskell angelehntes Typsystem ohne Klassen, Monaden oder parametrischer Datentypen, aber traditioneller mathematischer Funktionsapplikation, genutzt. Funktionen mit Seiteneffekt werden von uns statt mit Monaden mit dem Typ `()` deklariert. Dieser „Typ“ kann nur als letzter Typ, oder im Falle von Funktionen, die Speicher allozieren, als erster Argumenttyp vorkommen. Typen werden, abzüglich eben genannter, wie in Haskell definiert. Wir verwenden die Listendarstellung Haskells zur Beschreibung allgemeiner mehrelementiger Datentypen, bspw. Listen, Arrays oder Mengen.

Einhergehend mit den Konventionen von Haskell werden Terminale, Nichtterminale, Typen und Typkonstruktoren bei uns am Anfang groß geschrieben; Variablen, Funktionsnamen und Attribute hingegen werden am Anfang klein geschrieben. Funktionen werden bei uns wie in Haskell typisiert, wenngleich wir Currying in der Praxis nicht einsetzen.

Die üblichen Datentypen wie Bool, Int, String und Typen zur Code-Generierung und Namensanalyse sind vordefiniert. Zum Zugriff auf die Definitionstabelle dient der Typ Key, zum Aufbau des Namensraums werden Env und Bind genutzt.

Die Definitionstabelle, als Bestandteil der Namensanalyse, kann automatisch aus Spezifikationen generiert werden[25]. Wir beschreiben eine Eigenschaft n, die in der Definitionstabelle abgelegt werden soll, mit

```
declare_prop n :: τ ←e
```

oder

```
declare_prop n :: τ
```

wobei τ ein beliebiger Typ ist und e ein beliebiger Ausdruck. Letzterer gibt an, welcher Wert angenommen werden soll, wenn ein Eintrag nicht in der Definitionstabelle vorhanden ist. Das Setzen eines Wertes in der Definitionstabelle erfolgt über ein Attribut vom Typ Key gefolgt von einem Doppelpunkt und dem Namen der zu setzenden Eigenschaft. Soweit möglich, verzichten wir auf die Präsentation der Typen, wenn diese aufgrund der Verwendung hergeleitet werden können oder für das Verständnis bzw. das vorgestellte Muster nicht von Relevanz sind.

Zusätzliche Abhängigkeiten, die in geordneten Attributgrammatiken (siehe [12]) notwendig sein können, werden von uns mit

```
symbol X
attr ↑gotB ← true >= constituent Y.gotB
```

definiert, wobei >= für die zusätzliche Abhängigkeit einer Attributierung steht und **constituent** alle von X ableitbaren Y „aufsammelt“. Bei der Definition eines Wertes kommt in unserer Präsentation **constituent** nur vor, wenn der Wert irrelevant ist und dadurch nur eine Abhängigkeit geschaffen werden soll. Für eine detaillierte Einführung zu **constituent** wird auf [15] und [11] verwiesen.

Quelltext 1 zeigt einen Ausschnitt zur Implementierung der Ausdruckssprache aus Abbildung 1 unter Verwendung der bisher eingeführten Notation und Muster.

3 Typische Muster und deren Äquivalenzen

Bei der Vorstellung der Muster betrachten wir eine abstrakte Syntax definiert durch eine Grammatik $\mathsf{G} \triangleq (T, N, P, Z)$ mit attributierter Grammatik $AG \triangleq (\mathsf{G}, A, R, B)$. Für die folgende Präsentation sei folgende Notation eingeführt: für $i \in [0, n], j \in [0, m], m, n \in \mathbb{N}$ sind als Nichtterminale $\mathtt{A}, \mathtt{B}, \mathtt{X}_i \in N$ sowie als

Terminale $E_j \in T$ üblich. Mit kleinen Buchstaben werden Attribute $a, b, \cdots, z \in A$ notiert. Wir fordern üblicherweise, dass $Z \overset{*}{\Rightarrow} \mathtt{A}$, sowie für alle $i \in [0, n], n \in \mathbb{N} : \mathtt{A} \overset{*}{\Rightarrow} \mathtt{X_i}$ gilt[2]. Werden in den aufgelösten Mustern Attribute verwendet, die bei der Definition des Musters nicht verwendet werden, so sind dies neue, bisher nicht benutzte oder definierte, Attribute. Werden bei den folgenden Mustern keine Quellen in der Definition angeben, so konnten dazu von uns bisher keine Quellen gefunden werden, die das Muster so oder in ähnlicher Form beschreiben.

3.1 Direkte Muster, Basismuster und Kombinationen

Ein häufig auftretendes Muster, bspw. bei der *Hashwert-Berechnung* oder *Codegenerierung*, ist die Bestimmung eines Attributs auf Basis eines anderen Attributs.

Definition 3. *(**Attributabbildung**)*

```
a is f_x(X_0.b)
```

entspricht

```
symbol X_0
attr ↑a ← f_x(this.b)
```

für ein $\mathtt{X_0}$ *für das b definiert wird und einem neuen Attribut a. Wird* $\mathtt{X_0}$*. vor b weggelassen, so gilt diese Abbildung für jedes Symbol* $\mathtt{X_i}$*, für das b definiert wird.*

Diese *Attributabbildung* findet sich z.B. in Quelltext 1 bei der Berechnung von `VarDef.key` aus dem Attribut `bind`. Der zweite Teil der Definition erlaubt zusätzlich die Berechnung von `VarReference.key` ohne weitere Spezifikation. Es wird eine Vereinfachung der Attributabbildung für mehrere Symbole der abstrakten Syntax ermöglicht. In anderen Fällen kann man die Attributabbildung heranziehen um andere Muster zu beschreiben. Aufgrund der einfacheren Präsentation und der Möglichkeit der verlustfreien Abbildung von Baumstrukturen auf Listen und wieder zurück[24] betrachten wir bei den Mustern diese als Liste von Attributwerten. Unter dieser Betrachtung beschreibt Def. 3 die Funktion `map` $:: (\tau_a \to \tau_b) \to [\tau_a] \to [\tau_b]$ bzw. den einzelnen Schritt von `map`.

Ebenfalls häufig bei der *Codegenerierung* oder *Typisierung* anzutreffen sind einfache Beiträge. Diese werden genutzt um eine Faltung über Attribute von Knoten des abstrakten Syntaxbaumes durchzuführen.

Definition 4. *(**einfache Beiträge**, u.a. [7,2,11])*

```
contribute X_1.a, ···, X_n.o
  to A.b ←e using ⊕
```

entspricht

```
chain b
symbol A
attr head.b ← e

symbol X_1
attr ↑b ← tail.b ⊕ this.a
···
symbol X_n
attr ↑b ← tail.b ⊕ this.o
```

für eine binäre Operation $\oplus : \tau \to \tau \to \tau$ *und einen Initialisierungsausdruck e, der auch durch eine Konstante c repräsentiert werden kann.*

[2] Damit gilt auch $Z \overset{*}{\Rightarrow} \mathtt{X}_i$ für $i \in [0, n], n \in \mathbb{N}$.

Für einen Typ τ und eine binäre Operation $\oplus::\tau \to \tau \to \tau$ sowie eine Konstante c muss $(\tau, \oplus, c)$ ein Monoid sein, d.h. c ist das neutrale Element von τ bzgl. der Operation $\oplus$, welche assoziativ sein muss. Bei der Betrachtung als Liste von Attributen sind einfache Beiträge der funktionalen Programmierung mit `fold` nicht fern. Allerdings unterscheidet sich bei einfachen Beiträgen die Signatur mit `fold` :: $(\tau \to \tau \to \tau) \to \tau \to [\tau] \to \tau$ geringfügig von der bekannten Signatur funktionaler Programmierung.

Ist die Operation $\oplus$ nicht assoziativ können Umsortierungen und andere Aktionen beim Aufbau der abstrakten Syntax (siehe dazu auch [10]) zu unerwarteten Ergebnissen führen. Die Auswertung folgt der Baumstruktur des abstrakten Syntaxbaums.

Einfache Beiträge und Attributabbildungen können genutzt werden um das *Aufsammeln* von Attributen in ein anderes Attribut oder gar einen Definitionstabelleneintrag zu erreichen[14]. Dies ist nicht zu verwechseln mit *Collection*-Attributen aus [7,2], denn diese sind bereits in Beiträgen (Def. 4) gebündelt. Das Aufsammeln von Attributen kann in der Codegenerierung und auch Typisierung genutzt werden um Funktionsparameter oder auch Parametertypen aufzusammeln.

Definition 5. ***(Aufsammeln**, u.a. in [2,7])*

```
1  collect X_1.b,···,X_n.b in X_0.a
```

entspricht

```
1  X_1.b' is (X_1.b:[])
2  ...
3  X_m.b' is (X_m.b:[])
4  contribute X_1.b', ··· X_m.b'
5    to X.a ← [] using ++
```

wobei $(:)\colon \tau_Y \to [\tau_Y] \to [\tau_Y]$ *ein Listenkonstruktor,* $[]$ *die leere Liste darstellt,* $++\colon [\tau_Y] \to [\tau_Y] \to [\tau_Y]$ *die Listenkonkatenation und* τ_Y *der Typ der Attribute* $\mathtt{X}_i.b$ *ist; die aufgesammelten Attribute müssen denselben Typ haben.*

Definition 5 beschreibt somit eine Parametrierung einfacher Beiträge um Attribute aufzusammeln, wobei eine wesentlich kompaktere Darstellung erreicht wird, als dies mit Beiträgen selbst erreichbar ist. Nicht nur Funktionsparameter und Parametertypen lassen sich dadurch aufsammeln, sondern ebenso Konstruktoren oder Alternativen. Das Aufsammeln beschreibt also die Anwendung der Faltung um die implizite Liste der Baumstruktur explizit in einem Attribut abzulegen.

Ein wiederkehrendes Muster, bspw. *bei* der *Lebendigkeitsanalyse* oder der Analyse *allgemeiner Abhängigkeitsbeziehungen*, ist das „runterreichen, zusammenfassen, aufsammeln und ablegen“. Beispielhaft für dieses Muster, da es im Rahmen der Analyse von Pumpensystemen (siehe [1]) häufig vorkommt ist die Beschreibung von Abhängigkeitstypen.

Definition 6. ***(Abhängigkeitsaufbau)** Seien* τ_1 *und* τ_2 *Typen*

```
1  deptype of (τ_1, τ_2) is
2    A.a => X_0.b in B.c
```

entspricht

```
1  propagate A.a as a'
2  X_0.b' is ((X_0.a', X_0.b):[])
3  contribute X_0.b'
4    to A.sub_c ← [] using ++
5  contribute A.sub_c
6    to B.c ← [] using ++
```

wobei $B \overset{*}{\Rightarrow} A$ *und der Typ von* `A.a` *bzw.* $\texttt{X}_0\texttt{.b}$ τ_1 *bzw* τ_2 *ist.*

Der Abhängigkeitsaufbau wird ausschließlich durch Rückführung auf andere Muster erreicht.

Soll die Sortierung von Elementen bestimmt werden, Listen durchnummieriert werden oder ganz allgemein etwas unter Definition weiterer Attribute bestimmt werden, so nutzen wir dafür komplexe Beiträge. Ein komplexer Beitrag erweitert einen einfachen Beitrag um Zwischenergebnisse und zusätzlicher Verallgemeinerung.

Definition 7. *(**komplexer Beitrag**) Für Attribute und Definitionstabelleneigenschaften* $\texttt{x}_i$*,* $i \in [1, n]$*, Ausdrücke* $\texttt{e}_j$ *für* $j \in [1, p], p \in \mathbb{N}$ *ist*

```
1  contribute X1.a, ···, Xn.o
2    to A.c ←e
3  via x1 ←e1,
4      ···
5      xo ←eo,
6      chain ← ep
```

äquivalent mit

```
1  chain c
2  symbol A
3  attr head.c ← e
4
5  symbol X1
6  attr ↑xo ←eo
7      ···
8      ↑x1 ←e1
9      ↑c ←ep
10       >= (t.xo, ···, t.x1)
11 ···
12 symbol Xn
13 attr ↑xo ←eo
14     ···
15     ↑x1 ←e1
16     ↑c ←ep
17       >= (t.xo, ···, t.x1)
```

Bei komplexen Beiträgen erfolgt der Beitrag erst nach Berechnung aller **via**-Attribute. Diese **via**-Attribute können als Seiteneffekte während der Faltung betrachtet werden. Folgende Annahme muss in weiteren Arbeiten überprüft werden.

Hypothese 1. (**Ordnungserhaltend**): Für eine gegebene geordnete Attributgrammatik ist diese durch Hinzufügen der Expansion der Definitionen 4 und 7 weiterhin geordnet (Def. 1).

Durch komplexe Beiträge kann der Quelltext wesentlich knapper dargestellt werden, jedoch ist Hauptanwendungsfall die Umsetzung anderer Muster und Implementierung von Modulen.

Ein solches Muster ist die Akkumulation von Werten und stellt eine Parametrierung komplexer Beiträge dar. Für ein Symbol bestimmt die *Präfixsumme* eine Akkumulation eines Attributs in der Reihenfolge der Eingabe und legt Zwischenergebnisse in einem Attribut ab. Diese werden als Liste in einem Attribut zur Verfügung gestellt.

Definition 8. *(**Präfixsumme**) Seien die Attribute* $\texttt{X}_1\texttt{.a}, \cdots \texttt{X}_n\texttt{.j}$ *und eine (assoziative) binäre Operation* $\oplus :: \tau \to \tau \to \tau$ *dann sind*

```
1  scan X1.a, ···, Xn.j
2    to A.x ← c with y using ⊕
```

äquivalent

```
1  contribute X1.a, ···, Xn.j
2    to A.chn ← c
3    via y ←chain ⊕ tribute
4    chain ← chain ⊕ tribute
5  collect X1.y,···,Xn.y
6    in A.x
```

wobei **`tribute`** *die Referenz auf den aktuellen Beitrag ist.*

Für Präfixsummen bilden $(\tau, \oplus, c)$ wieder einen Monoid.

Wie bereits beschrieben kommt es ebenfalls häufig vor Indizes zu bestimmen. Da dies bei uns sehr häufig vorkommt, bspw. bei der Codegenerierung für die Zugriffsauswahl bei Feldern und Verbünden, existiert folgendes Muster.

Definition 9. ***(Indexbestimmung)***

```
1  count X0, ···, Xn
2    from A in s start with e
```

entspricht

```
1  symbol X0
2  attr ↑trib ← 1
3
4  ···
5
6  symbol Xn
7  attr ↑trib ← 1
8  scan X0.trib, ··· Xn.trib
9    to A.cnt ← e
10   with s using +
```

wobei `cnt`, `trib` *und* s *neue Attribute sind und* e *ein Initialisierungswert.*

Neben der beschriebenen Äquivalenz zur Indexbestimmung könnten auch komplexe Beiträge ohne Umweg herangezogen werden.

Komplexe Beiträge sind somit vielseitig nutzbar. Ein weiteres Beispiel dafür zeigt sich darin, dass diese genutzt werden können um die Namensanalyse, ähnlich wie [16], umzusetzen. Im Gegensatz zu der Variante aus [16] ist die folgende Definition nicht über Bibliothekscode und Vererbung oder reiner Textersetzung umgesetzt. Bei der Namensanalyse kommt es sehr oft vor, dass bereits ein umfangreicher Teil der Definitionstabelle gefüllt werden kann.

Definition 10. ***(Namensanalyse)***

```
1  use_before_def X1.sym, X2.sym
2    in A.n props: x1 ⊖ C.a,
3                  ···
4                  xo ⊖ D.y
5    with error unknown,
6         error unique
```

entspricht

```
1  contribute X1.sym to A.n
2  via x1 ←X1.c_a
3    ···
4    xo ←X1.d_y
5    bind ← nbnd(chain,tribute)
6    key ← keyof(bind)
7    chain ← chain
8  propagage A.n as env
9
10 symbol X2
11 attr cond ↑sym ∈ this.env
12   => error "not defined"
13
14 symbol X1
15 attr cond ↑sym ∉ ↓n
16   => error "already defined"
```

wobei $\mathtt{C}, \mathtt{D} \in (N \cup T)$, $\mathtt{x}_i$, $i \in [1, n], n \in \mathbb{N}$ *Attribute und Definitionstabelleneigenschaften,* $\mathtt{X}_1 \overset{*}{\Rightarrow} \mathtt{D}$ *und* $\mathtt{X}_1 \overset{*}{\Rightarrow} \mathtt{C}$. *Seien* τ *der Typ eines Attributs* y *eines aus* $\mathtt{X}_1$ *ableitbaren Symbols* $\mathtt{X}_i$, *analog* $\mathtt{D}.y$ *und* $\mathtt{C}.a$, *entspricht* **`props:`** $\mathtt{x}_a \ominus \mathtt{X}_i.y$

```
collect X_i.y in X_1.c_a
```

bzw.

```
contribute X_i.y
  to X_1.c_a ←c using r
```

wenn der Typ von $\mathtt{X}_1.\mathtt{x}_a$ $[\tau]$ *ist (linke Seite), respektive in allen anderen Fällen; wobei* (τ, r, c) *einen Monoid bilden. Die Attribute* `bind` *und* `key` *sind neue, noch nicht verwendete Attribute zum Zugriff auf die Definitionstabelle. Dazugehörige Funktionen erstellen einen Eintrag im Namensraum (*`nbnd`*) und machen diesen als Definitionstabelleneintrag (*`keyof`*) verfügbar.*

Eine weitere zu prüfende Eigenschaft ist das Verhalten der Kombination dieser Muster miteinander. Seien A_m, A_n und A_e die Attribute eines Musters, wobei A_m die referenzierten Attribute eines Muster darstellt, A_n die neuen Attribute bei der Expansion eines Musters und $A_e \triangleq A_m \cup A_n$ die expandierten Attribute eines Musters sind und $A_m \cap A_n = \emptyset$ gilt. Weiterhin seien zwei beliebige Muster m_1 und m_2, wobei $A_{n_{m_1}} \cap A_{n_{m_2}} = \emptyset$ und eine geordnete Attributgrammatik $AG \triangleq (\mathsf{G}, A, R, B)$. Folgende Annahme muss von uns noch bewiesen werden:

Hypothese 2. (**Abgeschlossenheit der Kombination**) Für zwei beliebige Muster m_1 und m_2 und expandierter Varianten $M_1 \triangleq R_{m_1} \cup B_{m_1}$, $M_2 \triangleq R_{m_2} \cup B_{m_2}$ ist $AG' \triangleq (\mathsf{G}, A \cup A_{n_{m_1}} \cup A_{n_{m_2}}, R \cup R_{m_1} \cup R_{m_2}, B \cup B_{m_1} \cup B_{m_2})$ eine geordnete Attributgrammatik.

3.2 Muster der Definitionstabelle

Bei Verwendung der Definitionstabelle unter Beachtung der Ordnungseigenschaft geordneter Attributgrammatiken treten weitere Muster auf. Zum Befüllen und dann Abfragen, bspw. bei der Alias- oder Typanalyse, kommt folgendes Muster zum Einsatz:

Definition 11. ***(Speichern und Laden)****:*

```
store_load X_0.x_1,···,X_n.x_n
  in A.a with A.k:p :: τ
via Y_1.b_1 ←e_1
    ···
    Y_m.b_m ←e_m
```

bzw.

```
declare_prop p :: τ
symbol S
attr ↑gotStA_k_p ← true
     >= constituent A.k:p
symbol Y_1
attr this.b_1 ←e_1
     >= include S.gotStA_k_p
...
symbol Y_m
attr this.b_m ←e_m
     >= include S.gotStA_k_p
symbol A
attr this.a ← A.k:p
     >= include S.gotStA_k_p
```

wobei $\mathtt{Y}_j \in (N \cup T), j \in [1, m], m \in \mathbb{N}$*;* $S \triangleq Z$*, und die* $\mathtt{X}_i.\mathtt{x}_i, i \in [0, n]$ *analog Def. 10 über Aufsammeln oder Beitrag in die Definitionstabelleneigenschaft* $\mathtt{A.k} : \mathtt{p}$ *hinzugenommen wird. Die* **`via`***-Attribute müssen nicht aufgeführt werden.*

Das Muster zum Speichern und Laden stellt also sicher, dass, bevor eine Spalte der Definitionstabelle geladen wird, der Eintrag in dieser Spalte vorhanden ist.

Für viele der Muster aus Abschnitt 3.1 existieren Varianten, die die gewonnenen Informationen in die Definitionstabelle überführen. Diese Varianten unterscheiden sich nur marginal – durch Verwendung von zu speichernden Eigenschaften statt Attributen – von den ursprünglichen Mustern. Wir verzichten daher auf einer Präsentation dieser Muster.

3.3 Weitere Muster

Viele Muster lassen sich durch Bibliotheken umsetzen. Beispiele dafür sind die Namensanalyse oder die Analyse der Gültigkeitsbereichze von Bezeichnern (Scoping)[16]. Andere Muster basieren auf der der Generierung von Konstruktoren, die in der Attributgrammatik als Makros zur Verfügung gestellt werden können, bspw. Codegenerierung oder Typanalysen[17,25]. Eine genaue Vorstellung dieser Ansätze würde den Rahmen dieser Arbeit sprengen.

Muster, die die Definitionstabelle analysieren oder traversieren existieren ebenso wie Muster, die Einträge in der Definitionstabelle aus bestehenden Einträgen bestimmen. Die Beschreibung der Traversierung der Definitionstabelle kann ebenso kompakt erfolgen, wie OCL dies für andere Mengen-orientierte Datentypen erreicht.

4 Evaluierung der Muster

Anhand des Sprachbeispiels aus Abschnitt 2, insbesondere Quelltext 1, wird vorgestellt, wie sich die Muster bezüglich Speicher- und Laufzeiteinfluss verhalten.

Die Verwendung von Mustern statt der Quellen aus Quelltext 1 zeigt Quelltext 3.

Das konkrete Speicher- und Laufzeitverhalten der Muster ist abhängig von der konkreten Ausprägung des Musters und der verwendeten Hilfsfunktionen sowie Algorithmen. Darüber hinaus ist auch die verwendete Grammatik sowie die Eingabe an den generierten Evaluator oder Übersetzer wichtig.

Wir beschränken uns daher darauf nur auf einige Sonderfälle einzugehen. Eine Propagation erzeugt, wenn das Attribut aufgrund von Optimierungen eine globale Variable ist (siehe dazu [13]), ein neues Attribut. Das Erlauben von Seiteneffekten könnte andernfalls zu Änderungen von Werten zwischen Start der Propagation und Auswertung des propagierten Attributs an anderen Stellen führen. Weiterhin erzeugen **`via`**-Attribute neue Attribute bzw. Definitionstabellenspalten.

Die Laufzeit der Muster ist selten höher als ein kompletter Durchlauf des abstrakten Syntaxbaums. Eine Ausnahme bilden die auf komplexen Beiträgen aufbauenden Muster sowie Muster der Definitionstabelle, welche einen zweiten Besuch eines Knotens nach sich ziehen. Diese Betrachtung schließt jedoch nicht aus, dass es keine zusätzlichen Besuche eines Knotens gibt, wenn andere Attributierungsregeln vorhanden sind.

```
1  declare_prop val :: Int ← 0
2  val is str_to_int(Constant.sym)
3  def_before_use VarDef.sym, VarReference.sym in Program.names_chn
4    with error unknown, error unique
5
6  symbol VarDef
7  attr ↑has_err ← ↑sym ∉ ↓names_chn
8
9  symbol VarReference
10 attr ↑has_err ← ↑sym ∈ ↓names
11
12 ...
13
14 deptype of (Key, Key) is VarDef.key =>VarReference.key in Program.deps
15 count VarDef from Program in defs
16
17 symbol Program
18 attr ↑has_err ← cyclic(↑deps) || Program.defs > 1
19    cond ↑has_err =>
20      error "Cannot evaluate program."
21    print(↑val)
```

Quelltext 3 – Ausschnitt der Implementierung der Ausdruckssprache mit abstrakter Syntax aus Abb. 1

5 Verwandte Arbeiten

Attributgrammatiken wurden von Knuth in [18] eingeführt. Wesentliche Erweiterungen der klassischen Attributgrammatiken finden sich in geordneten Attributgrammatiken (Ordered Attribute Grammars) von Kastens beschrieben in [12], wiederbeschreibbaren (bzw. umschreibbaren) Referenz-Attributgrammatiken von Hedin[4] und Higher-Order Attributgrammatiken[23].

Kastens zeigte in [12], dass es für die sehr mächtige Menge geordneter Attributgrammatiken immer möglich ist eine effiziente Berechnungsstrategie zu bestimmen und einen Evaluator zu generieren, der für alle Eingaben diese Berechnungen durchführt. Die Übersetzerbau-Werkzeugsammlung eli[6] nutzt OAGs zur Generierung vollständiger Übersetzer. Eli unterstützt gleichzeitig bereits eine Reihe von Erweiterungen klassischer Attributgrammatiken wie Vererbung und parametrierbare Module. Für eli wurde ebenfalls der „abstrakte Datentyp zur Namensanalyse“ in [16] vorgestellt.

In [11] werden syntaktische Methoden zur Modularisierung von Attributgrammatiken vorgestellt. Module lassen sich durch Parameter individualisieren und werden instanziiert.

DSLs, die auf dem Erkennen typischer Muster in Attributgrammatiken basieren und Konstruktoren definieren, finden sich in [17]. Weitere Beispiele, wie Verwendung der Definitionstabelle oder Codeerzeugung, werden in [25] vorgestellt.

Erstmals wurde eine Variante des contribution-Musters in [2] vorgestellt. Im Gegensatz zu unserer Variante sind keine in einem Symbol lokalen *via*-Attribute vorgesehen. Auf Boylands Arbeiten aufbauend wird u.a. in [8,4,7,20] eine Variante vorgestellt, die Feld-orientiert arbeitet. Für skalare Datentypen sind Hilfsklassen zu schreiben[20].

Einen Überblick über Attributgrammatiken liefern [3] sowie [22]. Letzteres liefert darüber hinaus einen Überblick zur Verringerung des Aufwands bei der Spezifikation mit Attributgrammatiken.

6 Zusammenfassung

In dieser Arbeit wurden eine Reihe typischer Muster in geordneten Attributgrammtiken vorgestellt. Es wurde gezeigt, wie auf Basis dieser Muster weitere Muster aufgebaut werden können. Die von uns vorgestellten Muster sind kompakter und bieten einen höheren Abstraktionsgrad als die entsprechenden Lösungen, die mit rein Modul-basierten Lösungen wie [11] möglich wären. Für die präsentierten Muster konnten Anwendungsfälle vorgestellt werden. Ebenso wurde auf Speichereinfluss und Laufzeiteinfluss der Muster eingegangen, wenngleich eine genauere Untersuchung und dazugehörige Vorstellung notwendig ist.

Hervorzuheben sind Beiträge als Muster, die in vielen Szenarien eingesetzt werden können und auch als Basis für andere Muster von Nutzen sind. Beiträge, wie sie von Hedin (siehe u.a. [7,4]) verwendet werden, gehen auf die globalen „Collection"-Attribute Boylands (siehe [2]) zurück und verhalten sich ähnlich dem **`constituent`**-Konstrukt aus LIGA und GAG (siehe u.a. [15]) in der Wurzel. Im Gegensatz zu den eben beschriebenen Ansätzen erlauben komplexe Beiträge auch die Einführung oder Verwendung zusätzlicher Attribute ohne direkten Beitrag zu einem Attribut. Damit gehen diese von uns vorgestellten Muster weiter als bisherige Ansätze.

Ebenfalls in [2] erkannt wurden Muster zum Aufsammeln von Attributen, jedoch keine verbesserte Abstraktion als Beiträge dafür gefunden, die bei Boyland noch als Attribute der Wurzel interpretiert werden können. In der bei uns vorliegenden Form bieten sich mehr Einsatzmöglichkeiten.

Die von uns vorgestellte Namensanalyse unterscheidet sich von den in [16] und [5] durch Verwendung des Musters mit Beiträgen und damit einhergehend der Möglichkeit die Definitionstabelle sofort mit semantisch relevanten Informationen zu befüllen. Es existieren weitere Eigenschaften bei der Namensanalyse, wie Namensbereiche, auf die wir bisher nicht eingegangen sind. Eine Herausforderung wird es sein, diese in Muster zu fassen, die mit anderen Mustern kombiniert immernoch eine geordnete Attributgrammatik ergeben.

Es existieren viele Erweiterungen, Bibliotheken und Werkzeuge denen gemein ist, dass sie aufgrund des Erkennens typischer Muster entwickelt wurden, bspw. [11,2,25,16,17]. Dennoch wurden diese typischen Muster nicht genutzt um einen höheren Abstraktionsgrad zu gewinnen, der auf die Semantik des Musters eingeht.

In den (älteren) Übersichtsarbeiten [3] und [22] wird nicht auf typische Muster eingegangen – Ausnahme bildet der Zugriff auf entfernte Attribute. Letzteres ist ähnlich der **Propagation** und dem **Aufsammeln**.

Neben den in Abschnitt 3 beschriebenen Mustern existieren weitere Muster, welche insbesondere auf der Verwendung von Graphen oder Datentypen, basieren. Diese weiteren sowie rein auf Produktionen anwendbaren Muster zu

identifizieren und zu analysieren ist das Ziel zukünftiger Forschung. Weiterhin muss ebenfalls noch gezeigt werden, dass die von uns aufgestellten Hypothesen (1 und 2) gültig sind. Wir erwarten von den dazugehörigen Beweisen konkretere Aussagen bzgl. Laufzeiteinfluss und Speichereinfluss als dies in Abschnitt 4 von uns erfolgt ist.

Die von uns gewonnenen Ergebnisse wollen wir in einer DSL zur Verfügung stellen.

Danksagung
Diese Arbeit entstand im Rahmen des vom Bundesministerium für Bildung und Forschung (BMBF) geförderten Forschungsprojekts ELSY (Nr. 16M3202D). Weiterhin danken wir den anonymen Gutachtern früherer Arbeiten, die uns auf die Beiträge in JastAdd aufmerksam gemacht haben.

Literatur

1. Berg, C., Zimmermann, W.: DSL Implementation for Model-based Development of Pumps. In: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change, pp. 391–406. Springer (2014)
2. Boyland, J.T.: Descriptional Composition of Compiler Components. Ph.D. thesis (1996), aAI9722877
3. Deransart, P., Jourdan, M., Lorho, B.: Attribute Grammars: Definitions, Systems and Bibliography. Springer-Verlag (1988)
4. Ekman, T., Hedin, G.: Rewritable Reference Attributed Grammars. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming, Lecture Notes in Computer Science, vol. 3086, pp. 147–171. Springer (2004)
5. Ekman, T., Hedin, G.: Modular Name Analysis for Java Using JastAdd. In: Lämmel, R., Saraiva, J.a., Visser, J. (eds.) Generative and Transformational Techniques in Software Engineering, Lecture Notes in Computer Science, vol. 4143, pp. 422–436. Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/11877028_18
6. Gray, R.W., Levi, S.P., Heuring, V.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM 35(2), 121–130 (1992)
7. Hedin, G.: An Introductory Tutorial on JastAdd Attribute Grammars. In: Fernandes, J.a., Lämmel, R., Visser, J., Saraiva, J.a. (eds.) Generative and Transformational Techniques in Software Engineering III, Lecture Notes in Computer Science, vol. 6491, pp. 166–200. Springer Berlin Heidelberg (2011), http://dx.doi.org/10.1007/978-3-642-18023-1_4
8. Hedin, G.: Reference Attributed Grammars. Informatica (Slovenia) 24(3), 301–317 (2000)
9. Johnson, R., Helm, R., Vlissides, J., Gamma, E.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1995)
10. Kadhim, B., Waite, W.: Maptool — supporting modular syntax development. In: Gyimóthy, T. (ed.) Compiler Construction, Lecture Notes in Computer Science, vol. 1060, pp. 268–280. Springer-Verlag (1996), http://dx.doi.org/10.1007/3-540-61053-7_67
11. Kastens, U., Waite, W.: Modularity and reusability in attribute grammars. Acta Informatica 31(7), 601–627 (1994), http://dx.doi.org/10.1007/BF01177548

12. Kastens, U.: Ordered Attributed Grammars. Acta Informatica 13(3), 229–256 (1980)
13. Kastens, U.: Lifetime analysis for attributes. Acta Informatica 24(6), 633–652 (1987)
14. Kastens, U.: Attribute Grammars as a specification method. In: Alblas, H., Melichar, B. (eds.) Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science, vol. 545, pp. 16–47. Springer Berlin Heidelberg (1991), http://dx.doi.org/10.1007/3-540-54572-7_2
15. Kastens, U., Hutt, B., Zimmermann, E.: GAG, a practical compiler generator, Lecture Notes in Computer Science, vol. 141. Springer-Verlag (1982)
16. Kastens, U., Waite, W.M.: An abstract data type for name analysis. Acta Informatica 28(6), 539–558 (1991)
17. Kastens, U., Waite, W.M.: Reusable specification modules for type analysis. Software: Practice and Experience 39(9), 833–864 (2009), http://dx.doi.org/10.1002/spe.917
18. Knuth, D.E.: Semantics of Context-Free Languages. Mathematical systems theory 2(2), 127–145 (1968)
19. Koskimies, K.: Object-orientation in attribute grammars. In: Attribute Grammars, Applications and Systems. pp. 297–329. Springer-Verlag (1991)
20. Magnusson, E., Ekman, T., Hedin, G.: Extending attribute grammars with collection attributes–evaluation and applications. In: Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on. pp. 69–80 (Sept 2007)
21. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Comput. Surv. 37(4), 316–344 (Dec 2005)
22. Paakki, J.: Attribute grammar paradigms – a high-level methodology in language implementation. ACM Comput. Surv. 27(2), 196–255 (Jun 1995), http://doi.acm.org/10.1145/210376.197409
23. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher Order Attribute Grammars. In: Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation. pp. 131–145. PLDI '89, ACM (1989)
24. Voigtländer, J.: Bidirectionalization for free!(Pearl). ACM SIGPLAN Notices 44(1), 165–176 (2009)
25. Waite, W., Kastens, U., Sloane, A.M.: Generating Software from Specifications. Jones and Bartlett Publishers, Inc., USA (2007)