Typische Muster bei der DSL-Implementierung unter Verwendung von geordneten Attributgrammatiken

Dissertation

zur Erlangung des
Doktorgrades der Naturwissenschaften (Dr. rer. nat.)

der
Naturwissenschaftlichen Fakultät III
Agrar- und Ernährungswissenschaften, Geowissenschaften und Informatik

der Martin-Luther-Universität
Halle-Wittenberg

vorgelegt

von Herrn Christian Berg
geb. am 7. September 1984 in Halle / Saale

Gutachter: Prof. Dr. Wolf Zimmermann
Gutachter: Prof. Dr. Uwe Aßmann

Verteidigungsdatum: 28.02.2019

Inhaltsverzeichnis

Beispielverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Zusammenfassung

Für die Spezifikation der Sprachsemantik gibt es verschiedene Lösungsansätze, die jeweils nur eine der folgenden Eigenschaften bieten: entweder die Sprachspezifikation ist kompakt oder der generierte Übersetzer ist performant bzw. schnell. Sollen schnelle Übersetzer mit kompakter Spezifikation entwickelt werden, gibt es dafür bisher noch keine Lösung. Diese Arbeit stellt eine Methode vor, die dies leistet, indem von Attributgrammatiken – eine der Standardlösungen aus dem Übersetzerbau – abstrahiert wird. Diese Abstraktion wird formal definiert und es wird bewiesen, dass wichtige Eigenschaften, wie Zerlegbarkeit und die Fähigkeit der Komposition, eingehalten werden und anhand einer Reihe von Beispielen gezeigt, dass die Spezifikation wesentlich kompakter als Attributgrammatiken sind und der resultierende Übersetzer ähnlich schnell arbeitet.

Kapitel 1.

Einleitung

1.1. Hintergrund und Motivation

Domänen-spezifische Sprachen (engl. domain specific languages, DSLs), können in vielen Bereichen eingesetzt werden – Beispiele finden sich u. a. in [29, 107] sowie Visionen für solche Sprachen in [31, 105].

Eine DSL kann in einer integrierten Entwicklungsumgebung (engl. integrated development environment, IDE) eingebunden werden. Durch projektierte Bearbeitung (engl. projectional editing) können verschiedene Endanwender auf denselben Programmen innerhalb einer DSL arbeiten. Ebenso können damit mehrere Bearbeitungsformen – bspw. in Form von Tabellen oder Grafiken – genutzt werden. Zur Realisierung solcher alternativen Bearbeitungsformen kann klassische Codegenerierung – z. B. zur Generierung einer Tabelle – eingesetzt werden. Bei der Rückpropagation der, in der Tabelle gemachten, Änderungen bleibt die Frage offen, ob das Resultat konsistent ist. Ursprünglich für relationale Datenbanken als „View-Update-Problem“ in [14] formuliert, existiert heute mit bidirektionalen Transformationen eine Verallgemeinerung dieses Problems. Die Verallgemeinerung lautet dann also: ausgehend von einer Abstraktion einer Datenquelle und Änderungen an dieser Abstraktion, wie können diese wieder mit den originalen Quellen vereinigt werden, sodass ein konsistentes System entsteht [37, 96]. Abbildung 1.1 stellt diese Fragestellung etwas vereinfacht dar, da die wichtigen Kriterien, die eine Sicherstellung der Konsistenz zur Folge haben sollen, nicht beachtet werden. Die Überprüfung der Konsistenz sowie die Generierung können mit Übersetzern und Codegenerierung erfolgen.

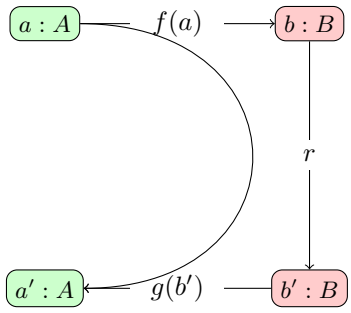


Abbildung 1.1. – Bidirektionale Transformationen $f\colon A \to B$ und $g\colon B \to A$, sowie Änderung c für Daten a und b vom Typ A und B, respektive

Für die Nutzung einer DSL in einer IDE müssen diese Transformationen und Prüfungen ggf. *sehr schnell* ablaufen, damit z. B. während der Bearbeitung Warnungen und Fehler zurückgemeldet werden können. Allgemein ist die Geschwindigkeit von aus einer IDE aufgerufenen Werkzeugen, die der IDE Informationen liefern, wichtig. So nimmt das menschliche Gehirn bereits ab einer Latenz von 200ms diese wahr [94].

Werden DSLs mit bestimmten Werkzeugen erstellt, geschieht die Integration in eine IDE automatisch [45].

Die Implementierung von DSLs und auch Programmiersprachen kann mit den Mitteln des Übersetzerbaus erfolgen. Bei DSLs ist die Modell-basierte Entwicklung momentan vorherrschend – siehe dazu [10, 31, 88, 98]. Eine Gegenüberstellung beider Ansätze wurde von Zimmermann in [121] vorgestellt. Während Geschwindigkeit und Formalismus vorherrschend im Bereich des Übersetzerbaus große Wichtigkeit haben, sind im Bereich der Modell-basierten Entwicklung Editor-Generierung und Modell-Änderungen wichtig.

```
1  -- Namensanalyse
2  rule Root ::= Expr
3  attr Expr.envIn = []
4
5  rule Expr ::= VarDef Expr Expr
6  attr Expr2.envIn = Expr1.envIn
7       Expr3.envIn = ((VarDef.name, Expr2.value):Expr1.envIn)
8  cond VarDef.name ∉ Expr1.envIn ⇒ error "Already defined " ++ VarDef.name
9
10 rule Factor ::= VarRef
11 attr Factor.value = Factor.envIn[VarRef.name]
12 cond VarRef.name ∉ Factor.envIn ⇒ error "Unknown Variable " ++ VarRef.name
13
14 -- Konstantenberechnung
15 rule Root ::= Expr
16 attr report "Output value = " ++ Expr.value
17
18 rule Term ::= Term Factor
19 attr Term1.value = Term2.value * Factor.value
20
21 rule Expr ::= Expr Term
22 attr Expr1.value = Expr2.value + Term.value
23
24 rule Expr ::= VarDef Expr Expr
25 attr Expr1.value = Expr3.value
26
27 -- Identitätsausgabe
28 rule Root ::= Expr
29 attr report "Identity Code = " ++ Expr.output
30
31 rule Expr ::= VarDef Expr Expr
32 attr Expr1.output = "let " ++ VarDef.name ++ " = " ++ Expr2.output ++
33                     " in " ++ Expr3.output
34
35 rule Term ::= Term Factor
36 attr Term1.output = Term2.output ++ " * " ++ Factor.output
37
38 rule Expr ::= Expr Term
39 attr Expr1.output = Expr2.output ++ " + " ++ Term.output
```

Dieses Beispiel zeigt die Attributierung einer einfachen Ausdruckssprache mit zusätzlichen Bezeichnern, denen mittels **let** (**let** *name* = ⟨*Expression*⟩ **in** ⟨*Expression*⟩) *Werte zugewiesen werden können und im zweiten aufgeführten Ausdruck verwendet werden können.*
In den ersten 12 *Zeilen des Beispiels wird der semantisch relevante Teil der Namensanalyse präsentiert, indem eine Art Definitionstabelle in Haskell-artiger Syntax leer initialisiert wird (Zeile* 3*). Diese wird mit den Bezeichnern, denen mittels* **let***-Ausdrücken Werte zugewiesen werden (Zeile* 7*), gefüllt. Die Werte aus dieser Definitionstabelle werden wiederum bei der Berechnung weiter verwendet (Zeile* 11*). Semantisch geprüft werden Definition und Benutzung (Zeilen* 8 *und* 12*). Im folgenden Abschnitt von Zeile* 14 *bis* 26 *wird eine Art Konstantenfaltung durchgeführt und das Ergebnis dieser ausgegeben. Eine der Codegenerierung angelehnte Identitätsausgabe sowie deren Berechnung erfolgt in den letzten Zeilen des Beispiels.*

Beispiel 1.1 – Semantisch relevanter Teil der Attributierungsregeln zur Ausgabegenerierung, Namensanalyse und Wertberechnung für eine Ausdrucksgrammatik

Zwischen Modell-basierter Entwicklung und den Methoden des Übersetzerbaus bestehen konzeptionelle Ähnlichkeiten [121].

DSLs sind u. a. aufgrund der Möglichkeit Domänen-spezifischer semantischer Prüfungen interessant. Prüfungen in Modell-basierten DSLs lassen sich einerseits mit Modelltransformation bzw. Codegenerierung in einer generell-anwendbaren Programmiersprache realisieren oder auch durch Verwendung von OCL (engl. Object Constraint Language), wie in [61], im Meta-Modell. Das Meta-Modell ist dabei die, ggf. grafische Beschreibung zum Aufbau von Modellen – ähnlich der (abstrakten) Syntax einer Programmiersprache. Von der Prüfung semantischer Eigenschaften mittels manuell geschriebener Prüfungen ist abzuraten. Diese werden bei Änderungen des Meta-Modells überschrieben oder müssen manuell gewartet werden. Letzteres widerspricht jedoch der Idee Modell-basierter Entwicklung. Andererseits ist auch von der Verwendung von OCL abzuraten, da in [16, 17] gezeigt wurde, dass semantische Prüfungen mittels OCL um Größenordnungen langsamer sind als alternative Ansätze. OCL ist ebenfalls Bestandteil von QVT (engl. Query/View/Transformations), einem im Modell-basierten Umfeld einsetzbarer Standard zur Implementierung von Sprachsemantik mit Modell-Transformationen. OCL ist ein Bestandteil von QVT, sodass bei Verwendung entsprechender Ausdrücke Ergebnisse wie in [16, 17] erwartet werden kön-

nen. In [110] beschreiben die Szárnyas et. al. einen Benchmark, bei dem viele Werkzeuge und Ansätze miteinander verglichen werden. Auch in [110] wird für eine Implementierung für OCL (Eclipse OCL) gezeigt, dass ab bestimmten Größen der Modellinstanzen keine weiteren Ergebnisse (Zeitüberschreitung oder Speicherbegrenzung) verfügbar sind und diese Implementierung je nach Szenario und Operation(en) im Vergleich mit alternativen Werkzeugen ebenfalls um Größenordnungen langsamer ist. In [20] wird präsentiert, wie aus OCL-Ausdrücken automatisch Anfragen auf Graphen generiert werden können. In [110] wurde festgestellt, dass auf solchen Graphanfrage-Sprachen basierende Systeme im langsamsten Drittel der Werkzeuge anzutreffen sind. QVT, wie auch das aktuell entwickelte QVTd (QVT declarative) enthalten OCL, insbesondere EMF-OCL bzw. Eclipse OCL, sodass auch hier bei der Verwendung entsprechender Ausdrücke von Laufzeiten, die mit denen aus [17, 110] vergleichbar sind, ausgegangen werden kann.

Im Bereich des Übersetzerbaus können für Konsistenzprüfungen, wie z. B. Namens- und Typanalyse, Attributgrammatiken eingesetzt werden [74]. Attributgrammatiken wurden von Knuth in [82] eingeführt und erweitern die (Kontext-unabhängige) Beschreibung des Aufbaus von Programmen um (Kontext-abhängige) „Attribute“ zur Bestimmung semantisch relevanter Eigenschaften. Attributgrammatiken haben den Vorteil, dass daraus effiziente Übersetzer generiert werden können (siehe u. a. [38]). Allerdings müssen Attributgrammatiken selbst auf Konsistenz und Vollständigkeit sowie weitere Eigenschaften geprüft werden, was mitunter ein schwieriges Problem (siehe [68]) sein kann. Für die praktische Anwendbarkeit von Attributgrammatiken werden daher Einschränkungen gemacht, andererseits existieren mit Attributen höherer Ordnung in HOAGs[1] – Attribute die Teile des Programms selbst widerum repräsentieren können – oder Referenzattributen in Referenzattributgrammatiken (RAG[2]) nützliche Erweiterungen klassischer Attributgrammatiken. Referenzattribute sind Verweise auf die Berechnung von Attributen, die nicht im unmittelbaren Kontext definiert sind.

Der Umfang des vollständigen Quelltext zu Beispiel 1.1 gemeinsam mit der Definition der benötigten Attribute und Definition der Grammatik umfasst über 110 Zeilen (siehe Anhang E). Davon sind, im Gegensatz zum hier gezeigten Beispiel, die Mehrheit reine Kopieranweisungen. Werden die in [101] und [76] vorgestellten Paradigmen angewandt, verringert sich der Quellumfang in Beispiel 1.1 auf knapp unter 100 Zeilen. Die Differenz beträgt in diesem Fall weniger als 20 Zeilen.

Eine Reihe sehr einfacher Formen von Attributgrammatiken werden in [93] vorgestellt. Für die heutige Anwendung spielen diese Attributgrammatiken[3] keine Rolle mehr. Eine in der Praxis gut anwendbare Klasse von Attributgrammatiken (siehe dazu u.a. [76, 78]) sind die von Kastens in [75] vorgestellten geordneten Attributgrammatiken. Einen Überblick über die wichtigen Formen von Attributgrammatiken geben [38, 117] sowie [2].

Ein wesentliches Problem von Attributgrammatiken wurde in [86] von Koskimies wie folgt ausgedrückt:

> *„The concept of an attribute grammar is too primitive to be nothing but a basic framework, the ‚machine language‘ of language implementation“*

Attributgrammatiken seien demnach mit einer Assemblersprache vergleichbar. Beispiel 1.1 veranschaulicht diese Eigenschaft sehr deutlich und wird in ähnlicher Form u. a. auch in [50] und [114] verwendet. Eine genauere Beschreibung von Attributgrammatiken und der hier gewählten Darstellung erfolgt in Kapitel 3 (insb. Abschnitt 3.2). Ein Domänen-spezifisches Beispiel ist dagegen Beispiel 1.2 aus [17].

Während in Beispiel 1.1 für Ausdrücke Werte und zu generierender Quelltext berechnet wird und im dazugehörigen Quelltext nur semantisch relevante Informationen dargestellt sind, besteht das Beispiel aus Beispiel 1.2 ausschließlich aus Kopieranweisungen.

Beispiel 1.2 stellt wieder nur einen Ausschnitt aus der vollständigen Attributgrammatik dar und besteht ebenfalls aus insgesamt über 100 Zeilen Quelltext (siehe ebenfalls Anhang E). Die Tatsache, dass in Attributgrammatiken viele semantisch wenig relevante Kopieranweisungen notwendig sind, ist ein bekanntes Problem [76, 101]. Werden die existierenden Möglichkeiten wie sie aus [76, 101] auf Beispiel 1.2

[1] HOAG ≙ engl. Higher Order Attribute Grammar – Attributgrammatik höherer Ordnung.
[2] RAG ≙ engl. Reference Attribute Grammar
[3] S- und L-Attributgrammatiken in der Literatur.

```
1  rule Decls ::= Decls Decl
2  attr Decls2.declsIn ←Decls1.declsIn
3       Decl.declsIn ← Decls2.declsOut
4       Decls1.declsOut ←Decl.declsOut
5       Decls2.env ←Decls1.env
6       Decl.env ← Decls1.env
7
8  rule Decl ::= RqDecl
9  attr RqDecl.declsIn ← Decl.declsIn
10      Decl.declsOut ← RqDecl.declsOut
11      RqDecl.env ← Decl.env
12
13 rule RqDecl ::= RqDefId RqReferences
14 attr RqDefId.declsIn ← RqDecl.declsIn
15      RqDecl.declsOut ← RqDefId.declsOut
16      RqReferences.env ← RqDecl.env
17      RqDefId.env ← RqDecl.env
18
19 rule Decls ::= ε
20 attr Decls.declsOut ← Decls.declsIn
21
22 rule RqReferences ::= RqReferences RqReference
23 attr RqReferences2.env ←RqReferences1.env
24      RqReference.env ← RqReferences1.env
25
26 -- ....
```

Beispiel 1.2 – Ausschnitt aus einer Attributgrammatik zur Namensanalyse von Anforderungen bestehend nur aus Kopieranweisungen.

angewendet, so enthält der resultierende Quelltext immer noch circa 40 Zeilen Quelltext. Bei der Verwendung von Referenzattributen oder auch Attributen höherer Ordnung, wie sie in [56] und [108, 116] vorgestellt werden, müssen dennoch die Attribute, die bei einer grundlegenden Namensanalyse notwendig sind zwischen den Knoten hin und her kopiert werden, damit die Definition an der Stelle der Referenz als Attribut überhaupt vorhanden ist – somit bieten diese für die Verwendung als Definitionstabelle keine unmittelbare Verbesserung.

In [28] wird gezeigt, wie sich Referenzattributgrammatiken für semantische Prüfungen im Meta-Modell nutzen lassen. Damit kann die häufig geforderte Integration in eine Entwicklungsumgebung realisiert werden. Boyland hat in [26] jedoch gezeigt, dass es nicht entscheidbar ist, ob eine gegebene Referenzattributgrammatik für alle Eingabeprogramme berechenbar ist – frühestens zur Laufzeit des Übersetzers ist klar, ob die Semantik eines gegebenen Programms mit dieser Referenzattributgrammatik bestimmbar ist. Diese Eigenschaft kann nicht nur zu längerer Laufzeit verglichen mit statisch bestimmbarer Berechnungsstrategie führen, sondern bedeutet, dass der generierte Übersetzer zur Übersetzungszeit eines potentiell gültigen Programms abstürzt. Dieses Problem besteht bei geordneten Attributgrammatiken so nicht. Zugleich existiert mit dem in [74] vorgestellten Ansatz eine Möglichkeit die durch Referenzattribute ausgedrückten semantischen Analysen in geordneten Attributgrammatiken unter Verwendung einer Definitionstabelle auszudrücken. Die Referenzattribute können dann in Form von Spalten der Definitionstabelle hinzugefügt werden.

Eine Besonderheit von geordneten Attributgrammatiken, deren Berechnungsstrategie statisch bestimmt wird, ist die Möglichkeit Definitionstabellen effizient zu nutzen, sodass Referenzattribute gar nicht notwendig sind und die damit einhergehenden Probleme nicht auftreten.

Wird der Quellumfang der Spezifikationen von abstrakteren Methoden, wie z. B. OCL, mit Attributgrammatiken verglichen so zeigt sich, dass die abstrakteren Methoden, wie zu erwarten, wesentlich kompakter sind. Andererseits ist die Laufzeit dieser Methoden Größenordnungen schlechter [16, 17]. Für entsprechend umfangreiche Sprachsemantik oder große Modelle werden diese abstrakten Methoden schnell unpraktikabel.

1.2. Problemstellung und Wissenschaftlicher Beitrag

Wie in Abschnitt 1.1 dargelegt ist für die Anwendung von DSLs und auch Programmiersprachen, insbesondere bei der Integration in einer Entwicklungsumgebung, die Performanz des Übersetzers (oder Analysewergzeugs) wichtig.

Der Einsatz von Attributgrammatiken erlaubt die Spezifikation schneller Werkzeuge und mit bidirektionalen Transformationen besteht auch eine formale Grundlage zur Integration solcher Werkzeuge in einer Entwicklungsumgebung.

Wenngleich zur Reduktion der Geschwätzigkeit von Attributgrammatiken Ingenieurslösungen existieren, wie z. B. in [73, 76, 101, 113] vorgestellt, so wird in den vorgestellten Arbeiten nicht untersucht, wie viel Abstraktion dadurch gewonnen und wieviel Geschwindigkeit verloren wird. Eine wissenschaftliche Betrachtung der vorgestellten Lösungen erfolgte in anderen Arbeiten ebenfalls noch nicht.

Die These dieser Arbeit lautet daher:

These 1. Es existiert ein, formal definierbarer, Mechanismus zur Spezifikation der Semantik einer Sprache auf Basis von Attributgrammatiken, der kompakter als klassische Attributgrammatiken ist, die Möglichkeit der Komposition zur Wiederverwendung bietet und für den erstellte Übersetzer nicht weniger performant sind, als ohne Verwendung dieses Mechanismus.

Implizit in dieser These enthalten ist, dass die wesentlichen Eigenschaften von Attributgrammatiken erhalten bleiben. Die Notwendigkeit der formalen Definition ergibt sich aus der Notwendigkeit der Beweisbarkeit von Performanzaussagen. Die Struktur einer Attributgrammatik hat Auswirkungen darauf, ob diese überhaupt berechenbar ist. Zum Erkennen – *vor* Ausführung eines generierten Evaluators – ob eine Attributgrammatik bei gegebener Eingabe überhaupt berechenbar ist, existieren verschiedene prüfbare Eigenschaften. Eine Eigenschaft die, für die Prüfung *vor* Generierung, notwendig ist, ist ob die Attributgrammatik „zerlegbar“ ist. Ist eine Grammatik „geordnet“, dann ist diese auch zerlegbar. Insbesondere ist aus der Literatur klar, dass effiziente Übersetzer für geordnete Attributgrammatiken generiert werden können [16, 17, 76, 77]. Zur Überprüfung des *Abstraktionsgrad* kommt der Quellumfang zum Einsatz. Wenngleich mit Auslagerung in Bibliotheksfunktionalität oder Module dies nicht immer übereinstimmen muss, so ist dies doch im Gegensatz zu Abstraktionsgrad objektiv messbar und prüfbar.

Die Notwendigkeit der formalen Definition der Abstraktion ist allein dadurch gegeben, dass nur damit wirkliche Untersuchungen und allgemein gültige Aussagen z. B. bzgl. Laufzeitverhalten oder Berechenbarkeit getroffen werden können.

Die These ist erfolgreich überprüft, wenn gezeigt werden konnte, dass ein System mit ähnlichem Laufzeitverhalten wie Attributgrammatiken gefunden wurde für die die Entscheidung ob diese geordnet (bzw. zerlegbar) ebenfalls (vor Ausführung eines generierten Übersetzers) prüfbar ist und der Quelltext vor Anwendung des Mechanismus (s. o.) geringer ist, als äquivalente Attributgrammatiken. Letzteres wird im Folgenden auch als Abstraktion bzw. Abstraktionsgrad bezeichnet. Im Sinne dieser Arbeit hat ein Mechanismus auf Basis von Attributgrammatiken einen höheren Abstraktionsgrad als ein anderer Mechanismus, wenn weniger Attributierungsregeln enthalten sind und dennoch alle semantisch relevanten Prüfungen umgesetzt werden können.

1.3. Lösungsansatz

Attributgrammatiken sind prinzipiell gut geeignet um die Sprachsemantik zu spezifizieren. Mit geordneten Attributgrammatiken existiert eine Klasse von Attributgrammatiken für die in kurzer Generierungszeit performante Übersetzer erstellt werden können. Darüber hinaus liegt für Attributgrammatiken

selbst bereits eine formale Definition vor, sodass darauf aufbauend eine formal definierte Abstraktion möglich ist.

Diese formal definierte Abstraktion wird in dieser Arbeit als Muster bezeichnet. Das Vorkommen von Mustern ist u. a. aus Softwaretechnik und Architektur bekannt [6, 70]. Ausgehend von einer Reihe von Basismustern und Musterkomposition sowie daraus resultierender Muster, wird gezeigt, dass sich viele typische Probleme der Sprachsemantik von DSLs (und damit indirekt auch von Programmiersprachen) lösen lassen. Der Beweis, dass diese Muster angewandt auf Attributgrammatiken wesentliche Eigenschaften, wie Zerlegbarkeit und Ordnungseigenschaft erhalten, zeigt ebenso, dass die Berechenbarkeit vor Ausführung eines generierten Übersetzers gegeben ist.

Anhand einer Reihe von Beispielen aus der Literatur wird gezeigt, dass einerseits die Sprachspezifikation kompakter wird und andererseits, dass die resultierenden Übersetzer ähnliche Performanz bieten wie direkt mittels Attributgrammatiken entwickelte Übersetzer.

Durch Ermittlung geordneter Attributgrammatiken aus Mustern als resultierender Attributgrammatik und den Vergleich mit der auf Mustern basierenden Variante lassen sich Kompaktheit und Performanz überprüfen.

Neben eli [52] existiert noch der Utrechter Attributgrammatik Übersetzer (engl. Utrecht University Attribute Grammar Compiler, UUAGC), siehe auch [109], der geordnete Attributgrammatiken unterstützt. Im Gegensatz zu UUAGC integriert eli bereits das Einlesen von Dateien und den Aufbau eines abstrakten Syntaxbaums, daher wird für die Implementierung von Mustern eli herangezogen.

Auf Basis der Sprachsemantik mit Mustern wird mittels eines Generators eine geordnete Attributgrammatik erzeugt, die mit dem Werkzeugsystem eli [52] in fertige Programme übersetzt werden kann. Abbildung 1.2 zeigt den Werkzeugpfad, des im Rahmen dieser Arbeit entwickelten Werkzeugsystems.

Für die Problemauswahl kommt auch die Eingangs motivierte Integration einer DSL in eine Entwicklungsumgebung in Betracht. Die Probleme bei solch einer bidirektionalen Transformation beruhen unter anderem auf den üblichen Problemen, die unter statischer Analyse zusammengefasst werden können. Darüber hinaus kommt ebenso das Erhalten eines konsistenten Dokuments bei der Änderung eines Teildokuments hinzu. Gegenstand dieser Arbeit ist jedoch nicht die Analyse bidirektionaler Transformationen zum Erhalt bidirektionaler Konsistenzeigenschaften. In dieser Arbeit wird sich somit auf die Codegenerierung beschränkt.

Gerade bei der Verwendung einer Definitionstabelle können die Abhängigkeiten der Berechnung zu Attributgrammatiken führen, die nicht mehr geordnet sind. Da jedoch geordnete Attributgrammatiken aufgrund der Geschwindkeitsvorteile verwendet werden, sind drei Eigenschaften zu betrachten: geordnet, zerlegungserhaltend und ordnungserhaltend.

Für diese Eigenschaften ist zu zeigen, dass die Lösung dieser Arbeit, diese Eigenschaften nicht verletzt.

Zusammenfassend ist der Ansatz dieser Arbeit also wie folgt:

1. Nachweis der formalen Definition von Mustern mit dem Nachweis obiger Eigenschaften und Bestimmung der Basismuster zum Aufbau weiterer Muster;

2. Präsentation einiger bekannter typischer Muster die aus diesen Basismustern aufgebaut sind sowie

3. die Vorstellung abstrakterer bzw. kompakterer Beispiele als reine (geordnete) Attributgrammatiken ermöglichen auf Basis der aus der Literatur bekannten Sprachen.

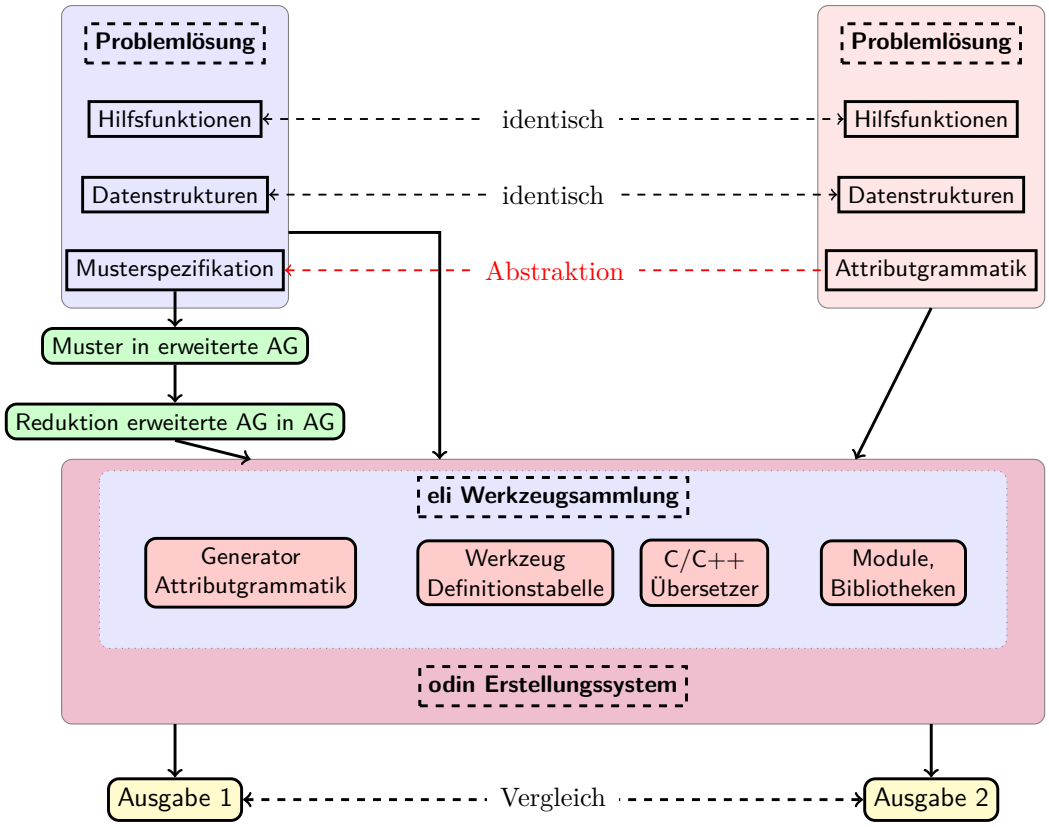


Abbildung 1.2. – Aufbau zum Nachweis der Existenz und Kompaktheit einer bzgl. Attributgrammatiken abstrakteren Darstellung zur Spezifikation der Sprachsemantik sowie Performanz der daraus generierten Programme. Linke Seite: Muster-basierte Lösung, rechte Seite: klassischer Lösungsansatz über reine geordnete Attributgrammatiken (AGs).

1.4. Aufbau der Arbeit

Kapitel 2 gibt anhand einer Reihe von Beispielen und Problemstellungen sowie deren Lösung in Attributgrammatiken einen Überblick über verwandte Arbeiten. Grammatiken, Attributgrammatiken und geordnete Attributgrammatiken werden in Kapitel 3 eingeführt. Die in dieser Arbeit zur Verbesserung des Abstraktionsgrad verwendeten Muster werden in Kapitel 4 vorgestellt. Ausgehend von der Definition von Mustern im Allgemeinen und deren Aufbau aus Termen(siehe Abschnitt 4.1 und Abschnitt 4.2) werden eine Reihe von Basismustern (Abschnitt 4.5) und die Komposition von Mustern vorgestellt. Aufbauend auf der Definition von Mustern wird in wird in Abschnitt 4.3 und Abschnitt 4.4 gezeigt wie Muster auf eine Attributgrammatik angewendet werden und dass diese Anwendung auf einer geordneten Attributgrammatik ordnungserhaltend ist – das Resultat ist wieder eine geordnete Attributgrammatik. Aufbauend auf den Ergebnissen aus Kapitel 4 werden die typischen Muster zur Lösung typischer Probleme bei der Entwicklung Domänen-spezifischer- und Programmiersprachen in Kapitel 5 vorgestellt. In Kapitel 6 werden mehrere Fallstudien zur Nutzung von Mustern auf geordneten Attributgrammatiken durchgeführt. Abgeschlossen wird die Arbeit mit einer Diskussion der Ergebnisse, einer Zusammenfassung und der Präsentation offener Fragestellungen in Kapitel 7

Kapitel 2.

Verwandte Arbeiten

Einen Überblick über fast 80 DSLs aus den Bereichen Software Engineering, Telekommunikation oder auch Simulationen wird in [39] gegeben. Ähnlich wie in [99] gibt auch [39] einen kurzen Abriss über mögliche Arten der Implementierung von DSLs. Weiterhin existieren Übersichtsarbeiten mit unterschiedlichen Fragestellungen und Klassifizierungen, wie bspw. [8, 85, 97].

Neben der Einbettung in eine klassische Programmiersprache bzw. die Nutzung als Bibliothek – sogenannte eingebettete DSLs (eDSLs), werden Präprozessoren oder Methoden des Übersetzerbaus in [39] vorgeschlagen. Standardwerke wie [2, 117] geben einen Überblick darüber, wie Übersetzer implementiert werden können.

Im Folgenden wird ein Überblick über eine Reihe von DSLs und deren Implementierungen gegeben. Falls notwendig wird die abstrakte Syntax präsentiert, anhand derer die Semantik der Sprachen intuitiv erklärt wird. Darauf folgend werden Muster, Transformation von Programmen und Modellen, Attributgrammatiken sowie deren Komposition betrachtet.

2.1. Implementierung Domänen-spezifischer Sprachen und Programmiersprachen

Eine Sprache zur Beschreibung des Verhaltens intelligenter Gebäude wird in [3, 4] und [5] vorgestellt. Ausgehend von der in [3] vorgestellten Bibliothek, die es erlaubt in intelligenten Gebäuden die mit einem Webservice zur Verfügung gestellten Schnittstellen, aus einer einfachen imperativen Sprache aufzurufen, wird in [3, 4] eine DSL vorgestellt. Mit der DSL werden Arbeitsflüsse beschrieben, die nach BPEL4WS (siehe [7]) übersetzt werden. Dabei werden vernetzte, intelligente Geräte und Ereignisse verknüpft, und mittels einer Beschreibung von, bei Ereignissen durchzuführenden Anweisungen, kombiniert. In den Arbeiten [3, 4, 5] wird die Sprachsemantik bisher nicht genauer beschrieben. Es ist davon auszugehen, dass auf Basis der in [4] vorgestellten Bibliothek eine Hierarchie der möglichen intelligenten Gegenstände zur Verfügung steht und die Namen dieser Gegenstände bekannt sind. Ausgehend von der intuitiven Sprachdefinition aus [5] und der Information, dass BPEL4WS generiert werden soll, kann erkannt werden, welche Problemstellungen bei der Generierung gelöst werden müssen. Beispiel 2.1 zeigt ein Beispiel dieser Sprache. Beispiel 2.1 ist eine gekürzte Variante, des in [5] präsentierten Sprachbeispiels.

Anhand von Beispiel 2.1 lassen sich eine Reihe von Konsistenzkriterien präsentieren. Da ein Programm in der Sprache von [5] immer eine geordnete Folge von Deklarationen, dabei dann Subscriptions, Einstellungen, Hauptschleife und dann Ereignissen ist, wird davon ausgegangen, dass die Verwendung von Namen vor deren Deklaration nicht gestattet wird. Wie in Beispiel 2.1 aufgeführt, sind die verwendeten Typen und die damit einhergehenden Aktionen, bspw. bei der Behandlung von Ereignissen, von Relevanz. Implizit lässt sich die dafür notwendige Typanalyse mittels Namensanalysen realisieren. Darüber hinaus ist zu prüfen, ob in Schleifen überhaupt auf eine Liste zugegriffen wird und ob der Typ der Liste identisch mit dem Typ der Iterationsvariablen ist.

```
1  Process EnergySavingScenario {
2    List heaters = discover(Heater);
3    List lamps = discover(Lamp);
4    List room = discover(Room);
5    Shower shower = discover(Shower, "shower_C");
6    subscribe TemperatureEvent;
7    subscribe AccessEvent;
8    /// ...
9    sequence {
10     while(true) {
11       onEvent(AccessEvent, Map input1);
12       if(input1.get("homeState") == "vacant") {
13         for(Lamp lamp : lamps) { action(lamp, "switchOff"); }
14         for(Heater heater : heaters) { action(heater, "switchOff"); }
15   }}}
16   /// ...
17   onEvent(TemperatureEvent, Map input5) {
18     if( input1 .get("temperature") > hotTemp) {
19       for(Heater h : heaters) {
20         if(h.getLocation().get("locationID") == location) {
21           action(h, "switchoff");
22   }}}}
23   /// ...
24 }
```

Eine intelligente Gebäudesteuerung beschreibt mehrere Prozesse. Jeder Prozess, wie das in Zeile 1 *beginnende „*`EnergySavingScenario`*“, besteht aus einer Folge von Instanziierungen (Zeilen* 2-5*) von Variablen fest vorgegebener Typen. Die Typen der Variablen können Listen (***`List`***) oder verschiedene mit Sensoren und Aktuatoren verbundene Objekte wie Heizungen, Lampen, Räume und Duschen (***`Heater`***, ***`Lamp`***, ***`Room`*** und ***`Shower`***) sein. Auf diese Folge von Deklarationen folgt eine Reihe von Zuweisungen an diese Variablen, welche im Quelltext nicht aufgeführt ist und durch Zeile* 8 *nur angedeutet werden soll.*
*Nach solch einer Liste folgt immer eine sogenannte „Anweisungssequenz“ (****`sequence`****), welche im Beispiel als Hauptschleife dient. Nach der Definition dieser Hauptschleife folgt immer eine Behandlung verschiedener Ereignisse, im Beispiel in Zeile* 17 *aufgeführt.*
Aus den Erklärungen zu [4] und [5] geht hervor, dass den Typen unterschiedliche Aktionen zugeordnet sind. Ereignissen wiederum sind verschiedene Werte zugeordnet, sodass einerseits in Zeile 18 *des Beispiels ein Fehlerhafter Zugriff statt findet. Dabei wird entweder auf das gar nicht definierte Element* `input1` *zugegriffen (definiert in der Hauptschleife* ***`sequence`****) oder es wird auf ein für den Typ „*`AccessEvent`*“ undefiniertes Attribut* `temperature` *zugegriffen.*

Beispiel 2.1 – Ausschnitt eines Beispielprogramms der DSL aus [5] und dieser Quelle entnommen; Fehler in Zeile 18 (rot unterlegt) ebenfalls in [5].

Weiterhin wird aus den einzelnen Segmenten zum Behandeln von Ereignissen und für die Variablendefinition BPEL4WS Quellcode generiert. Der entsprechende BPEL4WS Quelltext hat ca. fünfmal so viele Zeilen wie ein in der DSL aus [5] entwickeltes Programm.

Die Programmierung der Signalverarbeitung mit einer DSL ist Gegenstand der Arbeit [106]. Stewart et. al. stellen die Sprache „Ziria“ vor, die es erlaubt die Rekonfiguration und den Kontrollfluss eines physikalischen Protokolls umzusetzen. Neben der eigentlichen DSL wird in [106] auch ein Algorithmus vorgestellt, der es erlaubt auf Basis der in der DSL spezifizierten Anwendung eine Optimierung bzgl. der Datenbreiten und dem Fließbandablauf durchzuführen. Prinzipiell ist die Sprache aus [106] angelehnt an Datenflussbeschreibungen und funktional-reaktive Programmierung. Letzteres ist auch Gegenstand der Arbeit [118] und präsentiert eine Reihe ähnlicher Sprachen im Kontext funktional-reaktiver Programmierung. Grundsätzlich ist diesen Arbeiten gemein, dass, wie in Beispiel 1.1, eine Namensanalyse für **`let`**-Ausdrücke erfolgen muss. Darüber hinaus werden vordefinierte Operatoren genutzt, bspw. in [106] zum Verbinden mehrerer Datenquellen. Da viele Operationen auf einzelnen Bits definiert sind, wird durch Ziria – die in [106] vorgestellte Sprache – dies analysiert und automatisch in Operationen auf Tabellen übersetzt. Generiert wird Quellcode in der Programmiersprache C. Zwar ist davon auszugehen, dass Ziria kompakter ist als per Hand entwickelter Code, doch wird kein Vergleich aufgeführt. Stattdessen wird in [106] die Performanz der generierten Programme betrachtet. Beispiel 2.2 präsentiert eine mögliche Beschreibung eines Ziria-Programms.

```
let comp Decode(h : struct HeaderInfo) =
  DemapLimit(0) >>>
    if h.modulation == M_BPSK then
      DemapBPSK() >>> DeinterleaveBPSK()
    else if h.modulation == MQPSK then
      DemapQPSK() >>> DeinterleaveQPSK
    else ... -- QAM16, QAM64 cases
  >>> Viterbi(h.coding, h.len*8 + 8)
  >>> scrambler()
let comp receiver () =
  seq { det <- detectSTS()
      ; params <- LTS(det.shift)
      ; DataSymbol(det.shift) >>>
        FFT() >>>
        ChannelQualization(params) >>>
        PilotTrack() >>>
        GetData() >>>
        ReceiveBits() }
```

Beispiel 2.2 – Beispielprogramm in der Sprache Ziria, entnommen [106]; beschreibt WiFi 8.02.11a/g Empfang (Ausschnitt)

In [106] enthält der aus Beispiel 2.2 generierte C-Code etwa doppelt so viele Zeilen. Dabei nicht mit eingerechnet sind die Tabellen, da Bitoperationen auf Tabellenzugriffe für bessere Performanz zurückgeführt werden.

Die Programmierung von Industrierobotern in einer interaktiven Umgebung und die dazugehörige DSL, die das Verhalten der Industrieroboter mittels Zustandsmaschinen modelliert, ist Gegenstand von [29]. Die Zustandsmaschinen werden rekursiv durch andere Zustandsmaschinen beschrieben: Zustände können durch Zustandsmaschinen definiert werden. Zustände selbst wiederum haben eine Reihe möglicher Ereignisse, die beim Betreten und Verlassen ausgelöst werden. Diese Ereignisse selbst werden durch objektorientierte Programme implementiert. Beispiel 2.3 zeigt einen Ausschnitt der in [29] vorgestellten DSL mit der ein Industrieroboter so programmiert wird, dass dieser solange einer schwarzen Linie auf dem Boden folgt, bis ein Hindernis erreicht wird. Nach einer Drehung soll dieses Verhalten wieder ausgeführt werden.

```
(var lightlim := [128])(var maxlook := [100])(var forward := [0.2])
(var search := [0.2])(var back := [-0.2])(var turn := [1])
(machine follower
  (state moving
    (state running [robot move: [:msg|msg linear x:forward]]))
  (on outofline moving -> looking tlooking)
  (on intheline looking -> moving tmoving)
  (event outofline [robot light data > lightdim + 10])
  (event intheline [robot light data < lightdim - 10])
  (state looking
    (machine lookalgo
      (var time := [maxlook])
      (state lookleft
        (running [robot move: [:msg| msg angular z: search]]))
      (state returnleft
        (running [robot move: [:msg| msg angular z: search * -1]])))
// ....
(spawn follower looking)
```

Beispiel 2.3 – Steuerung eines Industrieroboters, der einer schwarzen Linie bis zu einem Hindernis folgen soll und nach einer Drehung dieses Verhalten weiter zeigen soll; entnommen aus [29] und in der DSL aus [29] gezeigt.

Wie auch bei den bisherigen Beispielen ist die Grundlage eine Namensanalyse, die keine Benutzung vor Definition erlaubt. Es kann also kein `follower` mittels **`spawn`** erzeugt werden, bevor dieser nicht definiert ist. Aus Sicht der Wiederverwendbarkeit wäre das Erlauben von benannten Zustandsmaschinen innerhalb der rekursiven Definition von Zuständen nützlich. Andererseits müsste dafür geprüft werden,

dass keine zyklische Definition eingeführt wird. Neben einer Namensanalyse wird auch in [29] Quelltext in einer anderen Sprache heraus generiert. Darüber hinaus jedoch werden diese Teil-Programme interpretiert indem die beschriebene Zustandsmaschine die Programme aufruft. Aufgrund der Anforderung der (Pseudo)-Echtzeitfähigkeit ist die Performanz des Interpreters und der Codegenerierung wichtig.

Wenngleich der Fokus der vorliegenden Arbeit doch auf textuellen Sprachen, die menschenlesbar sind, liegt, so beschreiben Hess et. al. in [62] eine DSL, mit der sich die Benutzerschnittstelle innerhalb eines Automobils beschreiben lässt. Diese DSL ist als Schema für XML gegeben, sodass die Generierung visueller Editoren aus diesem Schema möglich ist. Eine Prüfung, die über Namen oder Typen hinausgeht, wird in [62] nicht vorgestellt. Analoges ist der Fall bei dem in [43] vorgestellten Austauschformat ReqIF zur Beschreibung von Anforderungen. Die Beschreibung von Anforderungen ist auch Gegenstand von [17], welches das Beispiel 1.2 in ähnlicher Form ebenfalls enthält. In [17] werden die aus [101] und [76] bekannten Methoden zur Verringerung des Quellumfang eingesetzt. Im Gegensatz zu den bisher präsentierten Arbeiten wird in [17] die Benutzung vor Definition erlaubt.

Eine Reihe von Eigenschaften, die bei der Anforderungsbeschreibung für Produktlinien zu prüfen sind, stellen Lauenroth und Pohl in [91] vor. Eine DSL, die Anforderungen im Umfeld von Produktlinien beschreibbar macht, kann diese prüfen. Neben den zu prüfenden Eigenschaften werden in [91] auch ein möglicher Algorithmus zur Umsetzung dieser Prüfung vorgeschlagen. Eine Produktlinie besteht aus einer Menge von Komponenten, die einander bedingen oder auch ausschließen können. In [32] wird beschrieben, wie diese Bedingungen als Erfüllbarkeitsproblem (engl. satisfiability, SAT) umgesetzt werden. Damit einher geht die potentiell exponentielle Laufzeit für exakte Lösungen zur Frage, ob mit den gegebenen Teilen ein, die Anforderungen erfüllbares, Produkt erstellt werden kann. In [32] kommt dafür ein externer SAT-Löser zum Einsatz.

Sehr häufig zu prüfen ist, dass eine Beschreibung keine Zyklen enthält, bspw. beschreiben [56] und [36] zwei Arten dies zu prüfen. [56] stellt Referenzattributgrammatiken vor und wie durch die Referenzen ein ähnlicher Ausdruck wie der in [36] vorgestellte OCL-Ausdruck (engl. object constraint language, OCL) genutzt werden kann um Zyklen zu finden. In [17] wird dieser Ausdruck ebenfalls genutzt um die Performanz von OCL mit geordneten Attributgrammatiken zu vergleichen.

Eine weitere Arbeit, bei der eine DSL implementiert wird, ist [83]. Kokash et. al. präsentieren darin eine textuelle Sprache, deren Ziel es ist eine gemeinsame „Basis“ in Form eines Modells darzustellen. Dieses Modell soll bei der Entwicklung von Medikamenten angewendet werden, dies ist jedoch nur das motivierende Beispiel der Arbeit. Die in [83] vorgestellten Konzepte sind auch auf andere, ähnliche Domänen anwendbar. Wie ReqIF (siehe [43]) ist die vorgestellte Sprache im wesentlichen Austauschformat, geht aber darüber hinaus indem die in den Werkzeugen der Medikamentenentwicklung verwendeten (chemischen und biologischen) Modelle generiert werden.

Ähnlich zu [83] ist die Sprache aus [49] im Bereich der Medizin angesiedelt. In [49] beschreiben Florence et. al. eine DSL zur Beschreibung der Verschreibung von Medikamenten im Krankenhaus. Die Beschreibungssprache aus [49] ist vergleichbar mit der Programmierung der Roboterbeispiele, da ebenfalls Zustandsmaschinen beschrieben werden [49].

Prinzipiell über die Problematik des Datenaustauschs hinaus gehen Sprachen wie WOOL (engl. Worfkflow Language) oder YAWL (engl. yawl yet another workflow language), die in [65] und [1] präsentiert werden. In [1] wird ausgehend von Petri-Netzen und häufig gefundenen Arbeitsabläufen (engl. workflows), die schwer als Petri-Netz formulierbar sind, eine Notation eingeführt und deren Abbildung auf Petri-Netze gezeigt. Etwas näher an der Entwicklung einer DSL ist die Sprache aus [65]. In [65] wird eine Beschreibungssprache vorgestellt, die, ähnlich der in [29] vorgestellten, die rekursive Definition von Abläufen erlaubt. Während in [29] Industrieroboter gesteuert werden sollen, ist Gegenstand der Arbeit [65] die Steuerung von Arbeitsabläufen. Ähnlich odin oder make (siehe [33] und [48]) wird die Transformation von Ein- und Ausgaben und die dazugehörigen Abhängigkeiten beschrieben. Es ist üblich, dass diese Abhängigkeiten azyklisch sein müssen [33, 48]. Der Unterschied zwischen diesen Werkzeugen sind die möglichen Datentypen. Während bei make mittels Abhängigkeiten und Dateien Befehle ausgeführt werden, wird bei odin eine Ableitung eines Datei- oder Ordner-basierten Ziels angefordert und die Befehlsfolge automa-

tisch bestimmt. In [65] wird diese Befehlsfolge bzw. die Verknüpfung programmiert und entsprechender imperativer Code der die Werkzeuge aufruft und die Daten transformiert heraus generiert.

```
1  data AreaC = Rectangle Float Float
2             | Circle Float
3             | Square Float
4
5  getArea (Rectangle a b) = a * b
6  getArea (Circle a) = a * a * pi
7  getArea (Square s) = s * s
```

```
9  main = do
10   let x = getArea (Rectangle 10 60)
11   let y = getArea (Square 5)
12   let z = getArea (Circle 8)
13   let t1 = "x = " ++ (show x)
14   let t2 = t1 ++ " y = " ++ (show y)
15   putStrLn $ t2 ++ " z = " ++ (show z)
```

a) Haskell-Variante

```
1  #define PI 3.14159265359
2  #include <iostream>
3  using namespace std;
4
5  class AreaC {
6  public:
7    AreaC(){};
8    virtual ~AreaC() {};
9
10   virtual float getArea() const {
11     return 0.0;
12   }
13 };
14
15 class Rectangle : public AreaC {
16   float w;
17   float h;
18 public:
19   Rectangle()
20     : AreaC(), w(0.0f), h(0.0f) {}
21   Rectangle(float w, float h)
22     : AreaC(), w(w), h(h) {}
23
24   ~Rectangle() {};
25   float getArea() const {
26     return w * h;
27   }
28 };
```

```
30 class Circle : public AreaC {
31   /// ....
32   float getArea() const {
33     return r * r * PI;
34   }
35 };
36
37 class Square : public AreaC {
38   /// ....
39   float getArea() const {
40     return s * s;
41   }
42 };
43
44 int main(int argc, char** argv) {
45   AreaC *x = new Rectangle(10, 60);
46   AreaC *y = new Square(5);
47   AreaC *z = new Circle(8);
48   cout << " x = " << x->getArea()
49        << " y = " << y->getArea()
50        << " z = " << z->getArea()
51        << endl;
52 }
53
54
55
56
57
```

b) Variante in C++11 (Ausschnitt)

Beispiel 2.4 – Gegenüberstellung eines funktionalen Programms und eines objektorienterten Programms zur Bestimmung der Fläche verschiedener Datentypen unter Beibehaltung der Semantik. Zeilen 8 in Teilbeispiel 2.4a und Zeilen 29, 53 bis 58 der Variante unter 2.4b Leerzeilen. Zur Generierung von Binärcode sind ebenfalls Namens- und Typanalyse in beiden Sprachen notwendig.

In [65] wird neben der Syntax der Sprache auch vorgestellt, welche Eigenschaften die Arbeitsabläufe einzuhalten haben. Die Sprache selbst beschreibt Blöcke und Verbindungen zwischen diesen Blöcken, wobei die Verbindungen identische Typen haben müssen. Typen können definiert werden, sodass auch komplexe Typen und Aggregationen erlaubt sind. Nicht nur müssen die Verbindungen identischen Typ haben, auch dürfen nur Ausgänge mit Eingängen verknüpft werden. Darüber ob die Arbeitsabläufe zyklisch sein dürfen, macht Hulette in [65] keine Aussage.

Werkzeuge, wie die in [33, 48], werden häufig genutzt um verschiedene Programmiersprachen zu integrieren. In anderen Arbeiten vorgestellte Anwendungen, bspw. den in [1, 65] präsentierten, werden genutzt um wissenschaftliche Arbeitsabläufe zu programmieren.

In den bisher präsentierten Analysen und bei den Fähigkeiten der Codegenerierung verhalten sich die Beispiele vergleichsweise einfach. Jedoch existieren mit DSLs für die Beschreibung von Ontologien oder der Generierung von DSLs aus solchen Beschreibungen (siehe dazu z.B. [102]) sowie klassischen Pro-

grammiersprachen Beispiele mit sehr komplexen Analysen. Beispielsweise müssen für objektorientierte Sprachen mit Vererbung und dynamischen Untertypen zur Laufzeit nicht nur die Typbeziehung und die verfügbaren Bezeichner bekannt sein und auch zur Laufzeit des generierten Programms auf Basis der Typhierarchie die entsprechende Funktion aufgerufen werden. In [41] gibt Ducournau einen Überblick darüber wie dieser generierte Quelltext prinzipiell aussehen muss bei unterschiedlichen Fähigkeiten der objektorientierten Sprache und welche Alternativen existieren.

Ebenfalls zum Einsatz bei der Implementierung von Programmiersprachen kommt die automatische Typanpassung, bspw. wie in [53] vorgestellte Anpassung von Zeichenketten an ganze Zahlen, wenn diese eine Zahl darstellen. Auch wird in [53] beschrieben, wie die Maschinenregister zu nutzen sind, sodass Datentypen im Speicher verwaltet werden können. Ähnlich wie in [117] muss die Größe der Typen und bei der Kombination in Verbünden die Gesamtgröße bestimmt werden. Auch an dieser Stelle ist die Prüfung auf Zyklen erforderlich – zyklisch definierte Datentypen ohne Zeiger sind nicht realisierbar. Um diesen Zyklentest durchführen zu können, ist in der Regel eine Vorbedingung, dass Aliase der Typen aufgelöst werden.

Die Implementierung funktionaler Sprachen ist Gegenstand von [9, 103]. Wie in Beispiel 1.1 muss dafür eine Namensanalyse erfolgen. Viele Konzepte, die bei der Implementierung funktionaler Sprachen eingesetzt werden, unterscheiden sich kaum von denen bei imperativen Sprachen: neben einer Namensanalyse sind auch Typanalyse und Codegenerierung durchzuführen. Beispiel 2.4 stellt eine funktionale und imperative Darstellung eines einfachen Programms gegenüber. Zur Generierung entsprechenden Binärcodes ist neben der Typhierarchie und den entsprechenden Namen ebenso die Fähigkeit der Codegenerierung von Relevanz. Wenngleich die Aussage, dass in allgemeinen Programmiersprachen Namens- und Typanalysen sowie Codegenerierung benötigt werden bekannt ist, so zeigt Beispiel 2.4, dass die grundlegende Semantik dieser Analysen, auch bei sehr unterschiedlichen Sprachen, ähnlich sein kann.

Die Entwicklung von in einer allgemeinen Programmiersprache eingebetteten DSLs, sogenannten EDSLs (engl. embedded domain specific language, EDSL) ist Gegenstand von [66]. Prinzipiell werden wie in anderen Ansätzen die Operatoren überladen, sodass Ausdrücken neue Eigenschaften und eine neue Interpretation zugeordnet werden können.

Ein verwandtes Konzept wird in [87] verwendet um OCL als eine Art eingebetteter DSL in EMF Modellen verwenden zu können, stößt aber ebenso auf Performanzprobleme, wie [16] und [17]. In [87] wird die Prüfung der OCL-Ausdrücke durch Überladen von Operatoren erreicht, indem in dem überladenen Operator eine Bibliothek aufgerufen wird.

Auch mit Makros lassen sich solche eDSLs umsetzen. Bei rein textuellen Makros – siehe z. B. den `C` Präprozessor – werden dann keine semantischen Eigenschaften geprüft. In [119] wird eine eDSL auf Basis syntaktischer Makros vorgestellt, bei der nur ein Codegenerator, der auf einem abstrakten Syntaxbaum arbeitet, programmiert wird. Diese Arbeit geht damit bereits über einfache Makros hinaus. Wird solch ein Ansatz verfolgt, bei dem Makros syntaktische und semantische Informationen verwenden, so werden die Konzepte, wie diese im Bereich des Übersetzerbaus – Baumtransformation, Attributierung und Codegenerierung – bekannt sind, verwendet.

Einen Überblick darüber wie allgemein nutzbare Programmiersprachen implementiert werden können, geben [2] und [117]. Neben manueller Programmierung kommen häufig Attributgrammatiken und, für Teilprobleme, spezielle Sprachen zum Einsatz. Wie die Beispiele aus diesem Abschnitt mit Attributgrammatiken umgesetzt werden können, ist Gegenstand von Kapitel 6 und wird konzeptuell auch im folgenden Abschnitt betrachtet.

2.2. Attributgrammatiken und Sprachkomposition

Nach der Einführung von Attributgrammatiken durch Knuth in [82] wurde festgestellt, dass die Bestimmung ob aus einer Attributgrammatik ein Übersetzer generiert werden kann und dieser für jede mögliche Eingabe Ergebnisse berechnen kann exponentielle Laufzeit erfordert [68]. Verschiedene Möglichkeiten

diese exponentielle Laufzeit auszuschließen wurden vorgestellt. Vordefinierte Berechnungsstrategien, wie [69, 93], oder unabhängig vom abstrakten Syntaxbaum bestimmbare Berechnungsstrategien, wie [75], oder *erst* zur Laufzeit des Übersetzers fixierte Berechnungsstrategien, wie [81], sind dafür eine mögliche Lösung.

Ein Ansatz zur dynamischen Erweiterung der Sprachsemantik mit Attributgrammatiken wird von Hedin in [59] vorgestellt. Der Ansatz dieser Arbeit basiert darauf, den Zugriff auf berechnete Attribute über Schnittstellen zur Verfügung zu stellen. Bibliotheken, die diese Schnittstellen verwenden, können dann nachgeladen werden. Diese Technologie wird in JastAdd umgesetzt [57]. JastAdd ist ein Werkzeug zur Spezifikation von Referenzattributgrammatiken, welches um Baumersetzung und aspektorientierte Programmierung erweitert wurde [44, 60]. Baumersetzung wie in [44] ist nicht mit Attributen höherer Ordnung zu verwechseln. Letztere erlauben eine Baumtransformation nur für wohldefinierte Symbole an ausgezeichneten Produktionen. Das Verfahren aus [44] stellt hingegen die Definition solcher Baumtransformationen dem Endanwender zur Verfügung und ist damit praktisch mit einer Transformationssprache wie TXL (siehe [34]) vergleichbar. Der Zusammenhang zu Attributgrammatiken wird indirekt durch Verwendung von JastAdd zur Definition der abstrakten Syntax – dem dann zu transformierenden Baum – und der Definition von Attributberechnungen, die jeweils vor und nach der Baumtransformation durchgeführt werden, hergestellt.

Attribute höherer Ordnung sind eine Möglichkeit mehrere Zwischensprachen nacheinander zu attributieren und verallgemeinern damit die Methode aus [50]. In [50] werden Attributgrammatiken an bestimmten Stellen „zusammengesteckt“. Ursprünglich vorgestellt wurden Attribute höherer Ordnung in [108, 116] als ausgezeichnete Attribute, die Teilbäume repräsentieren. An speziellen Knoten im Baum können diese dann eingehangen werden. Im Gegensatz zu beliebigen Transformationen, wie diese mit den in [34] und [115] vorgestellten Werkzeugen möglich sind, werden bei Attributen höherer Ordnung neue Teilbäume in einen bestehenden Baum eingehangen. Dies ermöglicht die Verwendung von Attributen höherer Ordnung auch mit geordneten Attributgrammatiken, was mit der in [44] vorgestellten Methode nicht der Fall ist.

Die modularisierte Entwicklung von Attributgrammatiken durch, bspw. textuelles, Vereinigen mehrer Attributgrammatiken ist u. a. Gegenstand der Arbeiten [13, 24, 42, 47, 100] oder auch [114]. Das Verfahren der syntaktischen Verknüpfung, welches ähnlich Makros oder dem in [73] vorgestellten Verfahren funktionert, ist Gegenstand von [24, 47]. Die Komposition von Attributgrammatiken wird erstmals von Ganzinger und Giegerich in [50] vorgestellt. Ganzinger und Giegerich zeigen, wie unterschiedliche Zwischensprachen schrittweise mittels Attributgrammatik spezifiziert und generiert werden können. Ziel ist somit einen Übersetzer mit mehreren Phasen, auf Basis von Attributgrammatiken, zu beschreiben. Farrow, Marlowe und Yellin verwenden einen sehr ähnlichen Ansatz in [47] um einen Modulansatz auf Basis dieser Verkettung von Attributgrammatiken vorzustellen. Aufbauen darauf entwickelt Boyland in [24] eine praktisch anwendbare Beschreibungssprache für Attributgrammatiken, die diese Art der Komposition mit Referenzattributen verknüpft und wie diese Komposition implementiert werden kann. Detailliert stellt Boyland die Sprache und dazugehörige Implementierung in [25] vor. Ein wesentliches Merkmal des Verfahrens ist einerseits die Notwendigkeit verknüpfender Attributierungsregeln, wie bspw. in [47] oder die Notwendigkeit ausgezeichneter „Ausgabeattribute“. Letzteres bedeutet lediglich, dass ein Attribut ausgewählt wird, welches die Schnittstelle zwischen zwei Attributgrammatiken bildet.

In [73] wird unter Verwendung von Vererbung die Nutzung anpassbarer Bibliotheken beschrieben. In [74] wird ein darauf aufbauendes Modul vorgestellt, welches durch zwei Parameter instanziiert werden kann. Diese Instanziierung erfolgt durch Kopieren der Bibliothek und Ersetzung höchstens zweier Bezeichner an vordefinierten Stellen durch die Instanzparameter. Typische Szenarien der Namensanalyse und deren Realisierung mit Attributgrammatiken wird in [74] vorgestellt. Eine Alternative besteht darin die Namensanalyse statt mit Attributgrammatiken durch dafür geschaffene DSLs implementieren. DSLs für Namensanalyse sind unter anderem Gegenstand von [84] und [89].

Ähnlich Attributen höherer Ordnung beschreiben Saraiva und Swierstra in [104] eine Generalisierung des Prinzips, welches dort zur Anwendung kommt: bestimmte Symbole bzw. Produktionen (somit Nichtterminale und Terminale) der abstrakten Syntax der Attributgrammatik weisen vorerst „Lücken“ auf. Bei Besuch dieser Lücken wird die dazugehörige Attributauswertung aufgerufen und ggf. dynamisch nachgeladen. Im Gegensatz zu den Ansätzen aus [25, 47] ist kein manuell geschriebener Integrationscode

```
1  autocopy env
2
3  collect Prog.errors, Dcl.errors, Dcls.errors, Type.errors,
4          Stmt.errors, Stmts.errors, Expr.errors
5  in Root.errors using ++
6
7  rule Prog ::= Decls
8  attr Decls.env = []
9       Prog.code = "#include <stdio.h>\n" ++ Decls.code
10
11 rule Expr ::= Id Exprs -- function call
12 attr Expr.code ← Id.code ++ "(" ++ Exprs.code ++ ")"
13      Expr.errors ←
14        if Id.type = funcType then Exprs.errors
15        else Exprs.errors ++ ["Not a function " ++ Id.name]
16
17 rule logical_or_expr: Expr ::= Expr Expr
18 attr Expr1.code ←Expr2.code ++ " || " ++ Expr3.code
19      Expr1.errors ←Expr2.errors ++ Expr3.errors ++
20       if Expr2.type ≠ Expr3.type ||
21          Expr2.type ≠ boolType then ["Logical or requires boolean types"]
22       else []
23 -- ...
```

Beispiel 2.5 – Auszug aus dem Attributierungsbeispiel aus [112] in der Notation dieser Arbeit zur Namens- und Typanalyse einer imperativen Minisprache ohne Deklaration der Attribute und der kontextfreien Grammatik.

notwendig, wenngleich die Nichtterminale der „Lücken“ höchstens ein ererbtes Attribut haben dürfen. Damit, und, dass die ausgezeichneten Symbole – Elemente, die in diese Lücken eingesetzt werden dürfen – nicht in der eigentlichen Attributgrammatik verwendet werden dürfen, werden eine Reihe von Eigenschaften sichergestellt. Diese Eigenschaften sind, wie die Eigenschaften der vorliegenden Arbeit, notwendig für die Überprüfung ob die Generierung von Übersetzern aus solch einer Spezifikation überhaupt möglich ist.

Für das System „Silver“ ist ein Verfahren zur Beschreibung von Attributierungsregeln durch „Weiterleitung“ umgesetzt [112, 113]. Weiterleitung beschreibt, dass die Spezifikation einer Attributierungsregel für eine Produktion nicht zu dieser Produktion spezifiziert wird, sondern auf eine andere Produktion weitergeleitet wird. Konzeptuell ist die ähnlich der „Vererbung“ aus [73]: Berechnungen für Symbole (unabhängig von der Produktion) können durch „erben“ der Berechnungen aus anderen Symbolen oder – angelehnt an objektorientierte Programmierung – Klassensymbolen, erreicht werden. Weiterleitung, wie diese in [112] kurz eingeführt wurde, benötigt zur Realisierung Attribute höherer Ordnung oder Referenzattribute mit Baumersetzung – Teilbäume werden in andere Teilbäume transformiert um die, für diese Teilbäume definierten, Attributierungsregeln verwenden zu können. Für einige der für Silver vorgestellten Beispiele lassen sich äquivalente Ergebnisse durch die Bildung von Äquivalenzklassen und Abbildung der konkreten auf eine einfachere, abstrakte Syntax erreichen. Dieses Verfahren wurde in [72] vorgestellt. Eine Anwendung ist bspw. die Transformation der, in Beispiel 1.1 verwendeten, **`let`**-Ausdrücke auf **`where`**-Ausdrücke durch Umsortierung der Argumente der konkreten Syntax in der abstrakten Syntax. Details zu diesem Verfahren und anderen Anwendungsfällen finden sich in [72].

In [112] werden von Wyk et. al. ebenso weitere Erweiterungen von Attributgrammatiken vorgestellt, die den Abstraktionsgrad der Spezifikationen erhöhen. Präsentiert werden die Beispiele anhand einer imperativen Mini-Sprache, die nach C übersetzt wird. In Beispiel 2.5 wird ein Ausschnitt aus den Attributierungsregeln, die [112] entnommen sind, in der Notation der vorliegenden Arbeit präsentiert. Neben den in [25, 60] und [95] beschriebenen mengenwertigen-Attributen (engl. collection attributes) und der bereits erwähnten Weiterleitung unterstützt Silver das automatische Kopieren ererbter Attribute.

Das Kopieren ererbter Attribute wird auch in anderen Werkzeugen, zum Teil mit unterschiedlicher Syntax unterstützt. In [76] und [101] wird statt des in Beispiel 2.5 verwendeten **`autocopy`** bzw. dem in [17] verwendeten **`propagate`**-Mechanismus der Zugriff auf Attribute, die im Baum entfernt liegen, mittels **`including`** und **`constituents`** erreicht. Darüber hinaus wird in [76] auch nochmal auf die Verwendung von, der aus [73] bekannten, Vererbung eingegangen. Unter Ausnutzung aller in [76] und [101] vorgestellten Methoden lässt sich Beispiel 1.1 in Beispiel 2.6 überführen.

```
1  symbol AAARoot
2  attr head.env ← []
3
4  symbol VarRef
5  attr this.value ← ↓env[this.name]
6
7  rule Factor ::= VarRef
8  cond VarRef.name ∈ Factor.env  ⇒ error "Unknown Reference " ++ VarRef.name
9
10 rule Expr ::= VarDef Expr Expr
11 attr Expr3.env ← ((VarDef.name, Expr2.value):Expr1.env)
12 cond VarDef.name ∉ Expr1.env  ⇒ error "Already defined " ++ VarDef.name
13
14 class symbol VCalc
15 attr this.value ← constituent (VarRef.value, Int.sym, VCalc.value)
16                  with (Int, +, id, 0) shield VCalc
17
18 symbol Expr inherits VCalc
19 symbol Term inherits VCalc
20 symbol Factor inherits VCalc
21
22 rule <rExprMul> Term ::= Term Factor
23 attr Term1.value ←Term2.value * Factor.value
24
25 rule <rLetExpr> Expr ::= VarDef Expr Expr
26 attr Expr1.value ←Expr3.value
27
28 symbol AAARoot
29 attr report "Value ← " ++ (constituent VCalc.value shield VCalc)
30
31 class symbol OCalc
32 attr this.output ← constituent (VarRef.output, Int.output, OCalc.output)
33                           with (String, ++, id, "") shield OCalc
34 symbol Expr inherits OCalc
35 symbol Term inherits OCalc
36 symbol Factor inherits OCalc
37
38 rule Term ::= Term Factor
39 attr Term1.output ←Term2.output ++ " * " ++ Factor.output
40 rule Expr ::= VarDef Expr Expr
41 attr Expr1.output ←"let " ++ VarDef.name ++ " = " ++ Expr2.output ++ " in " ++ Expr3.output
42 rule Expr ::= Expr Term
43 attr Expr1.output ←Expr2.output ++ " + " ++ Term.output
44
45 symbol VarRef
46 attr this.output ← "" ++ this.name
47
48 symbol Int
49 attr this.output ← "" ++ this.value
50
51 symbol AAARoot
52 attr report "output ← " ++ (constituent OCalc.output shield OCalc)
```

Beispiel 2.6 – Aus Beispiel 1.1 hergeleitete Attributgrammatik unter Ausnutzung der in [76] und [101] vorgestellten Paradigmen.

Hervorzuheben an Beispiel 2.6 ist, dass dieses Beispiel die Semantik vollständig definiert, was bei Beispiel 1.1 nicht der Fall war. Zusammen mit der Definition der Grammatik und der Attribute sind immer noch etwa 100 Zeilen notwendig. Für einen Vergleich reicht jedoch die reine Definition der Attributierung, da davon ausgegangen werden kann, dass bei gleicher Semantik ähnlich viele Attribute und damit ähnlich viele Attributdefinitionen notwendig sind. In dieser Arbeit werden mit **`this`** Attribute referenziert, für die die Richtung des Attributs – ererbt oder synthetisiert – automatisch herzuleiten ist.

In Beispiel 2.6 kommen darüber hinaus Kettenattribute zum Einsatz. Während die Berechnung von Kettenattributen, außer an Stellen der expliziten Definition, automatisch inferiert wird, sind die **`thread`**-Attribute, die in [54] und [109] verwendet werden, nur eine kürzere Definition zweier Attribute. Eine weitere Möglichkeit der Attributierung stellt Farnum in [46] vor. Dabei werden Attributierungsregeln für Produktionen mittels Mustererkennung (engl. pattern matching) inferiert. Eine ähnliche Methodik kann auch in Silver angewendet werden.

Einen weiteren Ansatz zur Reduktion der Attributspezifikation, die ebenfalls Mustererkennung verwendet, stellen Kats, Sloane und Visser in [80] vor. Mit dem Werkzeug aus [80] wird die Attributierung mit

```
1  decorator down(a) =
2    if a.defined then
3      a
4    else
5      id.parent.down(a)
6    end
7
8  decorator down at-root(a) =
9    if not(id.parent) then
10     a
11   else
12     fail
13   end
```

Beispiel 2.7 – Ausschnitt aus [80] zur Beschreibung der der Propagation eines Attributs nach unten, wie dies in [101, 76] als **`including`** umgesetzt ist.

einem Termersetzungssystem programmiert. Damit kann bspw. die Propagation von Attributen nach unten mittels dem in Beispiel 2.7 vorgestellten Beispiel programmiert werden.

Das Verfahren aus [80] benutzt im Hintergrund Stratego und übersetzt die Attributgrammatik in ein Stratego-Programm. Dieses resultierende Programm ist mindestens dreimal langsamer als eine vergleichbare Implementierung mit Referenzattributgrammatiken [80]. In [80] wird das von Bird in [22] vorgestellte Beispiel „repmin“ verwendet. Dieses Beispiel behandelt die Propagation des Minimalwerts für alle Elemente eines Baums zurück zu den Elementen. Zusätzlich sollen diese Werte in den Knoten abgelegt sein. Dies ist mit Attributgrammatiken ohne Termersetzung nicht realisierbar. Mindestens Attribute höherer Ordnung werden dafür benötigt.

2.3. Zusammenfassung

Wie in Abschnitt 2.1 beschrieben existieren viele Sprachen und DSLs mit ganz unterschiedlicher konkreter Syntax. Weiterhin unterscheiden sich die vorgestellten Sprachen ganz erheblich in den generierten Artefakten. Während aus den Programmen der in [3, 4, 5] vorgestellte(n) Sprache(n) zur intelligenten Gebäudesteuerung BPEL4WS erzeugt wird, wird aus einem Programm in der Sprache Ziria ein Programm in C generiert.

Gleichwohl ist den Sprachen aus Abschnitt 2.1 gemein, dass eine Form der Namensanalyse, sowie ggf. Typanalysen, sowie eine Codegenerierung notwendig sind.

In Abschnitt 2.2 wurden Attributgrammatiken und verschiedene Erweiterungen darauf vorgestellt und Methoden der Komposition von Teil-Sprachen oder Sprachkonzepten gegenüber gestellt. Dabei wurden eine Reihe von Methoden der Komposition vorgestellt, für die jedoch *keine* formale Betrachtung erfolgt ist oder die Einschränkungen bezüglich Kompositionalität sehr groß sind. Während diese Ansätze im wesentlichen syntaktischer Natur sind, wie bspw. [13, 24, 42, 47, 73, 100] existieren Ansätze bei denen Programmcode oder auch Attributgrammatiken als Zwischenschritt generiert werden. Diese Ansätze, wie in [34, 80, 115] oder auch [79, 84, 89] vorgestellt bzw. verwendet, sind nützlich bei der Implementierung von Sprachkonzepten. Dennoch erfolgt auch hier, wie in den syntaktischen Ansätzen, weder eine Betrachtung des gewonnenen Abstraktionsgrad noch eine Betrachtung zu den formalen Eigenschaften. Letztendlich wurden in Abschnitt 2.2 unterschiedliche Arten von Attributgrammatiken oder Erweiterungen von Attributgrammatiken vorgestellt. Für die wichtige Klasse der geordneten Attributgrammatiken existieren Attribute höherer Ordnung mit denen sich eine Reihe der bereits genannten Ansätze (bspw. [13, 24, 42, 47]) auf ähnliche Art umsetzen lassen. Ebenso wurde darauf eingegangen, dass einige der vorgestellten Erweiterungen, wie bspw. Weiterleitung in Attributgrammatiken, in geordneten Attributgrammatiken bereits realisierbar sind.

Letztendlich bleibt zusammenfassend zu sagen, dass, werden Attributgrammatiken verwendet, üblicherweise geordnete Attributgrammatiken in der Lage sind, die Sprachsemantik zu formulieren und dabei performant sind. Darüber hinaus sind deren Eigenschaften gut verstanden, für Referenzattributgrammatiken hingegen ist es nicht entscheidbar, ob überhaupt für eine Sprachsemantik ein Übersetzer generiert werden kann [26].

Kapitel 3.

Grundlagen

In den folgenden Abschnitten werden die notwendigen Grundbegriffe vorgestellt. Die Arbeit folgt in der Präsentation im wesentlichen [117] und den Arbeiten von Kastens [75, 76] sowie Knuth [82] für formale Sprachen und Attributgrammatiken.

Die verwendeten Notationen können auch Anhang A entnommen werden.

3.1. Grammatik und abstrakte Syntaxbäume

Grundlage dieser Arbeit sind kontextfreie Grammatiken. Kontextfreie Grammatiken beschreiben den kontextfreien Anteil einer formalen Sprache, wie bspw. Programmiersprachen. Einen Überblick über die möglichen Techniken und etwaige Einschränkungen kontextfreier Grammatiken um schnelleres Erkennen kontextfreier Sprachanteile zu erreichen, bieten [2] und [117]. Eine kontextfreie Grammatik ist wie folgt definierbar:

Definition 3.1. Ein 4-Tupel $G \triangleq (N, T, P, Z)$ mit

- einer endlichen Menge Nichtterminalen N,
- einer endlichen Menge Terminalen T und
- Produktionen $P \subseteq N \times \Sigma^*$ mit
- $Z \in N$

heißt **kontextfreie Grammatik**, wobei Z ein ausgezeichnetes Nichtterminal und als *Startsymbol* bezeichnet wird.

Es ist üblich zu fordern, dass das Startsymbol nicht auf der rechten Seite einer Produktion vorkommt, d.h. $P \subseteq N \times (\Sigma \setminus \{Z\})^*$. Diese Bedingung lässt sich leicht durch Hinzufügen eines neuen Nichtterminals Z' mit einer Regel $\langle Z' \rangle ::= \langle Z \rangle$ erreichen.

Anmerkung: Kontextfreie Grammatiken werden im Übersetzerbau genutzt um die konkrete und abstrakte Syntax zu spezifizieren. Die konkrete Syntax beschreibt dabei den Aufbau der Struktur von Sprachkonstrukten – aus imperativen Sprachen sind Anweisungen, Schleifen und Ausdrücke bekannt. Diese Sprachkonstrukte sind unter anderem aus Schlüsselworten, Bezeichnern und anderen Terminalen bzw. Terminalfolgen[1] aufgebaut. In der abstrakten Syntax werden Schlüsselworte und andere semantisch irrelevante Informationen der konkreten Syntax nicht mehr aufgeführt. Neben Schlüsselworten sind auch Operatorpräzedenzen nicht mehr notwendig. Darüber hinaus ist es möglich verschiedene Sprachkonzepte der konkreten Syntax – ein häufig genutztes Beispiel sind verschiedene Schleifenarten – auf

[1] Im Übersetzerbau wird der Begriff Grundsymbolfolge zur Beschreibung solcher Ströme aus denen auf Basis der konkreten Syntax ein Syntax- bzw. Parsebaum aufgebaut wird, verwendet.

ein Sprachkonzept der abstrakten Syntax abgebildet werden. Für mehr Details der verschiedenen Konzepte zum Aufbau formaler Sprachen können [2] und [117] entnommen werden. In dieser Arbeit wird die abstrakte Syntax verwendet, da diese bereits alle zur Beschreibung einer Sprachsemantik relevanten Teile enthält. Ist für die Präsentation von Beispielen die Verwendung von Schlüsselworten notwendig, sind in den Beispielen die Schlüsselworte der abstrakten Syntax hinzugefügt ohne von obiger Diskussion abzuweichen. Eine separate Darstellung konkreter und abstrakter Syntax sowie deren Abbildung, bspw. unter Verwendung der Ansätze aus [72] würden keine zusätzliche Erkenntnis liefern.

Nichtterminale einer kontextfreien Grammatik $G \triangleq (N, T, P, Z)$ $X_i \in N, i \in \{1, \cdots, n\}$ werden für eine abstrakte Syntax üblicherweise als $\langle X_i \rangle$ dargestellt, bei Terminalen wird auf die Begrenzungssymbole verzichtet. Trennungssymbol für die linke und rechte Seite einer Produktion ist in der Regel ::= Terminale und Nichtterminale der konkreten Syntax werden am Anfang klein geschrieben. Ein Beispiel für eine abstrakte Syntax findet sich in Abbildung 3.1. Ebenfalls in dieser Arbeit üblich ist das Verzichten auf Begrenzungssymbole in der abstrakten Syntax. In diesem Fall sind Nichtterminal mit großem Anfangsbuchstaben und Terminalsymbole klein geschrieben.

$$
\begin{array}{lll}
\langle \mathit{Description} \rangle & ::= & \langle \mathit{Declarations} \rangle \\
\langle \mathit{Declarations} \rangle & ::= & \langle \mathit{Declarations} \rangle \ \langle \mathit{Declaration} \rangle \\
 & \mid & \varepsilon \\
\langle \mathit{Declaration} \rangle & ::= & \langle \mathit{RootStat} \rangle \\
 & \mid & \langle \mathit{RqDefinition} \rangle \\
\langle \mathit{RqDefinition} \rangle & ::= & \langle \mathit{RqDefId} \rangle \ \langle \mathit{Dependencies} \rangle \\
\langle \mathit{RootStat} \rangle & ::= & \langle \mathit{UseId} \rangle \\
\langle \mathit{RqDefId} \rangle & ::= & \mathrm{ID} \\
\langle \mathit{Dependencies} \rangle & ::= & \langle \mathit{Dependencies} \rangle \ \langle \mathit{Dependency} \rangle \\
 & \mid & \varepsilon \\
\langle \mathit{Dependency} \rangle & ::= & \langle \mathit{UseId} \rangle \\
\langle \mathit{UseId} \rangle & ::= & \mathrm{ID}
\end{array}
$$

Abbildung 3.1. – Abstrakte Syntax einer Sprache zur Beschreibung von Anforderungen, Terminale ohne Begrenzungssymbole

Es gibt verschiedene Formen eine kontextfreie Grammatik darzustellen – neben der hier verwendeten Repräsentation über Backus-Naur-Form [12] (BNF) existieren auch eine erweiterte Backus-Naur-Form (EBNF) sowie Syntaxdiagramme (siehe z. B. [15, 117, 120]). In dieser Arbeit wird für die Darstellung (meist) BNF genutzt, welches sich aus EBNF herstellen lässt [117].

Die Darstellung von Produktionen über (E)BNF ist prägnant und hilft bei der Definition der Ableitungsrelationen:

Definition 3.2. Für eine kontextfreie Grammatik $G \triangleq (N, T, P, Z)$ mit Produktion $p \in P$ mit $p : \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ für $\mathtt{X}_0 \in N$, $\mathtt{X}_i \in \Sigma$ und $0 < i < n$, $n \in \mathbb{N}$ dann heißt $\mathtt{X}_i$ **herleitbar** aus $\mathtt{X}_0$ bzgl. der Produktion p.

Die Relation $\overset{*}{\rightsquigarrow}$ bezeichnet den reflexiv-transitiven Abschluss der Relation direkt ableitbarer Symbole, $\overset{+}{\rightsquigarrow}$ den transitiven Abschluss dieser Relation. Ist $\mathtt{X} \overset{+}{\rightsquigarrow} \mathtt{Y}$ so heißt $\mathtt{Y}$ **Nachfahr** von $\mathtt{X}$ und $\mathtt{X}$ **Vorfahr** für $\mathtt{Y}$ mit $\mathtt{X} \in N$ und $\mathtt{Y} \in \Sigma$.

In klassischen Werken, wie [64, 117], wird der Begriff der Ableitungsrelation bzw. Ableitung zur Ableitung von Worten, d. h. einer Sequenz von Terminalsymbolen, und nicht von einzelnen Terminalen und Nichtterminalen verwendet. In der vorliegenden Arbeit ist jedoch das Verhältnis der Ableitungen in der

Folge von Relevanz. In dieser *Ableitbarkeitsrelation für Symbole aus Symbolen* ist wichtig ob aus einem Symbol wieder dieses Symbol ableitbar ist, also ob `X` Vorfahr und Nachfahr von sich selbst ist.

Für das Beispiel aus Abbildung 3.1 ist also einerseits ⟨*Declarations*⟩ aus ⟨*Description*⟩ ableitbar, d.h. ⟨*Description*⟩ ⇒ ⟨*Declarations*⟩, andererseits gelten unter anderem auch:

$$\begin{aligned} \texttt{Declarations} &\overset{*}{\rightsquigarrow} \texttt{Dependency} \\ \texttt{Declaration} &\overset{*}{\rightsquigarrow} \texttt{Declaration} \\ \texttt{Declaration} &\overset{+}{\rightsquigarrow} \texttt{UseId} \end{aligned}$$

Folgende Definition wird in der Beschreibung des Aufbaus kontextfreier Grammatiken benötigt.

Definition 3.3. Sei $G \triangleq (N, T, P, Z)$ eine kontextfreie Grammatik mit Symbolen $\mathtt{X}_0 \in N, \mathtt{X}_i \in \Sigma$ für $1 < i \leq n$ mit $n \in \mathbb{N}$. Für eine Produktion $p \in P, p : \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n$ ist $|\mathtt{X}_i|_p$ die Anzahl der Vorkommen des Symbols $\mathtt{X}_i$ auf der rechten Seite der Produktion p. Die Anzahl aller Vorkommen des Symbols $\mathtt{X}_i$ in der Produktion p ist definiert als

$$\lceil \mathtt{X}_i \rceil_p = \begin{cases} |\mathtt{X}_i| + 1 \text{ falls } \mathtt{X}_i = \mathtt{X}_0 \\ |\mathtt{X}_i| \text{ sonst} \end{cases}$$

Tabelle 3.1 gibt die unterschiedlichen Werte der Anzahl nach Definition 3.3 für die ersten Produktionen der abstrakten Syntax aus der Grammatik aus Abbildung 3.1 an.

Produktion	**Vorkommen**
$p : \langle\texttt{Description}\rangle ::= \langle\texttt{Declarations}\rangle$	$\lvert\texttt{Description}\rvert_p = 0$ $\lvert\texttt{Declarations}\rvert_p = 1$ $\lceil\texttt{Description}\rceil_p = 1$ $\lceil\texttt{Declarations}\rceil_p = 1$
$q : \langle\texttt{Declarations}\rangle ::= \langle\texttt{Declarations}\rangle\ \langle\texttt{Declaration}\rangle$	$\lvert\texttt{Declarations}\rvert_q = 1$ $\lvert\texttt{Declaration}\rvert_q = 1$ $\lceil\texttt{Declarations}\rceil_q = 2$ $\lceil\texttt{Declaration}\rceil_q = 1$
$r : \langle\texttt{Declarations}\rangle ::= \varepsilon$	$\lvert\texttt{Declarations}\rvert_r = 1$ $\lceil\texttt{Declarations}\rceil_r = 1$

Tabelle 3.1. – Anzahl der Symbolvorkommen nach Definition 3.3 für die drei ersten Produktionen aus der Grammatik aus Abbildung 3.1.

Eine mögliche visuelle Darstellung des abstrakten Syntaxbaums für eine Eingabe zur Grammatik aus Abbildung 3.1 zeigt Abbildung 3.2, dabei werden Terminale durch ovale Blätter dargestellt wohingegen Nichtterminale und ε-Ableitungen durch eckige Knoten dargestellt werden. Der abstrakte Syntaxbaum repräsentiert die Konstruktion der abstrakten Syntax für eine Eingabe, d.h. Knoten sind Instanzen für Nichtterminale der abstrakten Syntax; in Terminalsymbolen wird der Inhalt der Terminalsymbole angegeben.

Eine Definition eines abstrakten Syntaxbaums findet sich in Definition 3.4. Dafür notwendige Definitionen, wie Graphen und geordneten Bäumen, finden sich in Anhang B.2.

Definition 3.4. (siehe [117]) Sei $G \triangleq (N, T, P, Z)$ und $w \in L(G)$. Sei $AST = (E, K)$ ein markierter, geordneter Baum mit Wurzel k_0, $k_1, \cdots, k_n$ für $n \in \mathbb{N}$ und $n > 0$ unmittelbare Nachfolger von k_0 sowie Markierungsfunktion $f\colon E \to M$, dann heißt AST **abstrakter Syntaxbaum** des Wortes w bzgl. G genau dann, wenn folgende Bedingungen gelten:

1. $M \subseteq \Sigma \cup \{\varepsilon\}$
2. $f(k_0) = Z$

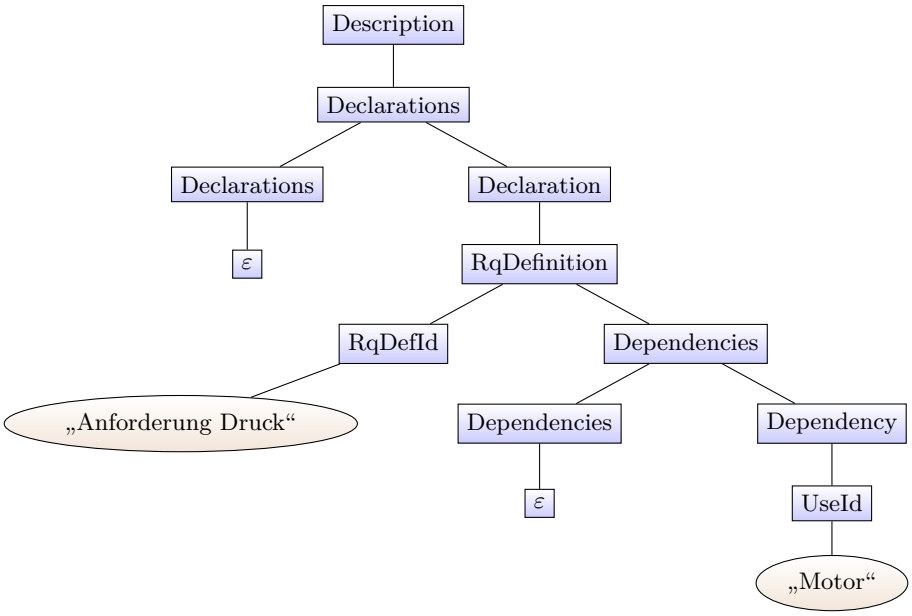


Abbildung 3.2. – Möglicher Abstrakter Syntaxbaums zur abstrakten Syntax aus der Grammatik aus Abbildung 3.1. Nichtterminal-Knoten sind eckig, Terminalknoten oval mit Wert des Terminals als Inhalt dargestellt.

3. $Z \to f(k_1) \cdots f(k_n) \in P$

4. ist $f(k_i) \in T$ oder $n = 1$ und $f(k_i) = \varepsilon$ so ist k_i ein Blatt

5. ist $f(k_i) \in N$ dann ist k_i Wurzel eines abstrakten Syntaxbaums bezüglich der Grammatik $G' = (T, N, P, f(k_i))$ für ein Wort $w' \in L(G')$

Dabei heißt k_i ***i*-ter Nachfahre** von k_0 und für jedes k_i heißt k_0 **Vorfahr** bzgl. des abstrakten Syntaxbaums AST. Für einen Knoten k heißt $f(k)$ **Typ des Knotens** k falls $f(k) \neq \varepsilon$. Statt Nachfahr oder Vorfahr werden diese auch als *Kindknoten* und *Elternknoten* bezeichnet.

Zur Bestimmung der eigentlichen Sprachsemantik werden kontextfreie Grammatiken um Attribute erweitert. Diese Erweiterung heißt Attributgrammatik.

3.2. Attributgrammatiken

Die Idee, durch Erweiterung der abstrakten Syntax um Attribute die Semantik einer formalen Sprache zu definieren, geht auf Knuth zurück, der dies in [82] vorgestellt hatte.

Definition 3.5. Eine **attributierte Grammatik** ist ein Tupel $AG \triangleq (G, A, R, B)$, wobei

- $G \triangleq (N, T, P, Z)$ eine kontextfreie Grammatik ist, die eine abstrakte Syntax definiert,
- $A \triangleq \biguplus_{X \in \Sigma} A_X$ eine endliche Menge von **Attributen**,
- $R \triangleq \biguplus_{p \in P} R_p$ eine endliche Menge **Attributierungsregeln** der Form $a_0 \leftarrow f(a_1, \cdots, a_k)$, wobei f eine Funktion ist und
- $B \triangleq \biguplus_{p \in P} B_p$ eine endliche Menge von **Bedingungen** der Form $\varphi(a_1, \cdots, a_k)$, wobei φ ein Prädikat ist.

Für alle $a_0 \leftarrow f(a_1, \cdots, a_k) \in R_p$ einer Produktion $p \in P, p\colon \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ muss gelten: $a_i \in A_{\mathtt{X}_0} \cup \cdots A_{\mathtt{X}_n}, i = 0, \cdots, k$ und für alle $\varphi(a_1, \cdots, a_k) \in B_p$ für eine Produktion $p \in P, p\colon \mathtt{X}_0 ::= \mathtt{X}_1 \cdots \mathtt{X}_n$ muss gelten: $a_i \in A_{\mathtt{X}_0} \cup \cdots A_{\mathtt{X}_n}, i = 1, \cdots, k$.

Wie in der Literatur üblich, werden in dieser Arbeit ggf. Infixnotation, bzw. Ausdrücke sowie das Auslassen verschiedener Funktionen verwendet. Typen von Funktionen werden in dieser Arbeit wie in Haskell dargestellt. Beispielsweise ist `map`$\colon (a \to b) \to [a] \to [b]$ die Funktion, die eine Liste von Elementen des Typs a in eine Liste von Elementen vom Typ b überführt und dafür eine im ersten Argument befindliche Funktion von a nach b verwendet. Wird kein Argument verwendet aber eine Unterscheidung in Funktion und Variable ist notwendig, so wird der Typ () als einziger Argumenttyp verwendet. Der Rückgabetyp ist üblicherweise das letzte Typargument. Eine formale Definition von Listen findet sich in Anhang B.1. Muss eine Funktion f ausgewertet vorliegen, wird dafür $\dot{f}$ geschrieben. Gleiche Produktionen, die unterschieden werden müssen, werden in dieser Arbeit jeweils unterschiedliche Bezeichner – Label – vorangestellt.

Die Attribute einer attributierten Grammatik werden in folgende Arten unterteilt:

Definition 3.6. Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Seien $u, v, w \in \Sigma^*$ und e ein beliebiger Ausdruck, dann heißt ein Attribut $a \in A$

ererbt gdw. $a \in A_{\mathtt{X}}$ ist und für eine Produktion $p \in P$ mit $p\colon \mathtt{Y} ::= u\mathtt{X}v$ die Attributierungsregel $\mathtt{X}.a \leftarrow e \in R_p$ ist und

synthetisiert gdw. $a \in A_{\mathtt{X}}$ ist für eine Produktion $p \in P$ mit $p\colon \mathtt{X} ::= w$ eine Attributierungsregel $\mathtt{X}.a \leftarrow e \in R_p$ ist.

Die Einteilung in synthetisierte und ererbte Attribute ist notwendig zur Bestimmung einer Auswertereihenfolge. Zuvor muss jedoch sichergestellt werden, dass die Attributgrammatik vollständig ist und für jeden abstrakten Syntaxbaum eine Auswertereihenfolge existiert. Synthetisierte Attribute wurden von Irons in [67] vorgestellt. Knuth definierte davon ausgehend in [82] Attributgrammatiken, welche darüber hinaus ererbte Attribute enthalten.

Abbildung 3.3 zeigt einen Ausschnitt einer attributierten Grammatik zur Namensanalyse und stellt einen Ausschnitt aus Beispiel 1.2 dar. Die abstrakte Syntax von Beispiel 1.2 befindet sich in Abbildung 3.1

```
1  rule Description ::= Declarations
2  attr Declarations.names ← ∅
3       Declarations.env ← Declarations.names
4
5  rule RootStat ::= UseId
6  attr UseId.env ← RootStat.env
```

Abbildung 3.3. – Ausschnitt einer attributierten Grammatik zur abstrakten Syntax aus Abbildung 3.1, siehe auch Beispiel 1.2.

Mit dem Schlüsselwort **`rule`** werden Produktionen spezifiziert und über **`attr`** werden diese attributiert. Das zu berechnende Attribut befindet sich auf der linken Seite des Pfeils (←), die Berechnungsfunktion – in diesem Fall die Identität, die weggelassen wird – befindet sich auf der rechten Seite des Pfeils. Kommt ein Nichtterminal mehrfach in einer Produktion vor, dann wird für den Zugriff auf ein Attribut eines dieser Nichtterminale ein Index benutzt, der bei 1 beginnt. Ein Ausschnitt aus dem Beispiel 1.2 in folgendem Abbildung 3.4, zeigt dies.

Eine Attributgrammatik ist vollständig, wenn für jedes Attribut in allen Kontexten in denen es vorkommt eine Attributierungsregel existiert. Darüber hinaus ist eine Attributgrammatik konsistent, wenn höchstens eine solche Regel existiert. Formal:

Definition 3.7. Eine Attributgrammatik $AG \triangleq (G, A, R, B)$ heißt **konsistent** genau dann, wenn für alle abstrakten Syntaxbäume AST folgende Bedingungen erfüllt sind:

```
1  rule Decls ::= Decls Decl
2  attr Decls₂.declsIn ←Decls₁.declsIn
3       Decl.declsIn ← Decls₂.declsOut
4       Decls₁.declsOut ←Decl.declsOut
5       Decls₂.env ←Decls₁.env
6       Decl.env ← Decls₁.env
```

Abbildung 3.4. – Veranschaulichung des Zugriffs auf Attribute bei mehreren (teils identischen) Nichtterminalen in derselben Produktion aus Beispiel 1.2.

- für jedes Attribut $a \in A$ in jedem Knoten k höchstens eine definierende Regel $\mathsf{X}.a \leftarrow e \in R$ existiert und
- der $Typ(k) = \mathsf{X}$ ist.

Die Attributgrammatik heißt **vollständig**, wenn in jedem Knoten k mindestens eine Regel $\mathsf{X}.a \leftarrow e \in R$ existiert.

Die Überprüfung dieser Kriterien kann statisch erfolgen, indem nur geprüft wird, ob es innerhalb derselben Produktion unterschiedliche Attributierungsregeln gibt. Die Prüfung der Konsistenz einer Attributgrammatik wird u. a. in [82, 117] beschrieben.

Sind zu einem Terminal X synthetisierte Attribute zu definieren, dann wird ggf. in der Darstellung eine Produktion $\mathsf{X} ::= \varepsilon$ attributiert.

Auf Basis von Definition 3.5 und Definition 3.6 lassen sich auch Attribute und insbesondere deren Abhängigkeiten in Syntaxbäumen darstellen. Die Abhängigkeiten zwischen Attributen der abstrakten Syntax – ein definiertes Attribut hängt von den in der Definition auf der rechten Seite des Pfeils aufgeführten Attibute ab – können leicht auf Abhängigkeiten im abstrakten Syntaxbaum abgebildet werden.

Die Regeln einer Attributgrammatik lassen sich für eine Produktion wie in Abbildung 3.5 darstellen. Im weiteren Verlauf der Arbeit ist die konkrete Berechnungsvorschrift von untergeordneter Relevanz. Statt der konkreten Berechnung werden im Folgenden hauptsächlich Abhängigkeiten betrachtet.

Die Abbildung 3.5 stellt jedoch nicht die Berechnungen im abstrakten Syntaxbaum dar, sondern die Berechnungen der Attributgrammatik, die unabhängig von der konkreten Eingabe spezifiziert sind. Über den abstrakten Syntaxbaum mit Knoten, deren Typ den Nichtterminalen (und Terminalen) der Grammatik der abstrakten Syntax entspricht, lassen sich diese Berechnungen auf Berechnungen im abstrakten Syntaxbaum abbilden. Abbildung 3.6 stellt die Berechnungen in einem abstrakten Syntaxbaum für den Ausschnitt aus Abbildung 3.4 dar.

Anhand von Abbildung 3.6 ist erkennbar, dass die Berechnung der Attribute nicht nur von der Spezifikation der Attributgrammatik, sondern ebenso von der Eingabe abhängig ist. Anhand des abstrakten Syntaxbaum kann bestimmt werden, in welcher Reihenfolge Attribute auszuwerten sind. Um ein Attribut berechnen zu können, ist notwendig, alle „gelesenen“ Attribute vorher zu berechnen.

Definition 3.8. Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und einem abstrakten Syntaxbaum AST von G mit Knoten $K_0, \cdots, K_n$ heißt eine Liste $[K_{i_0}.a_0, \cdots, K_{i_m}.a_m]$ aller Attribute aller Knoten von AST **berechenbare Reihenfolge** genau dann, wenn jedes Attribut jedes Knotens genau einmal in der Liste vorkommt und für jedes Attribut $K_{i_j}.a_j$ eine Definition $K_{i_j} \leftarrow f(K_{i_{j_1}}.a_{j_1}, \cdots, K_{i_{j_q}}.a_{j_q})$ der entsprechenden Produktion p der Attributgrammatik existiert, sodass $j_1, \cdots, j_q < j$ ist.

Implizit ist in der Definition der berechenbaren Reihenfolge bereits der Begriff der Abhängigkeit von Attributen enthalten:

Definition 3.9. Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für jede Produktion p: $\mathsf{X}_0 ::= \mathsf{X}_1 \cdots \mathsf{X}_n \in P$ mit Attributierungsregel $\mathsf{X}_j.b \leftarrow f(\cdots \mathsf{X}_i.a \cdots)$

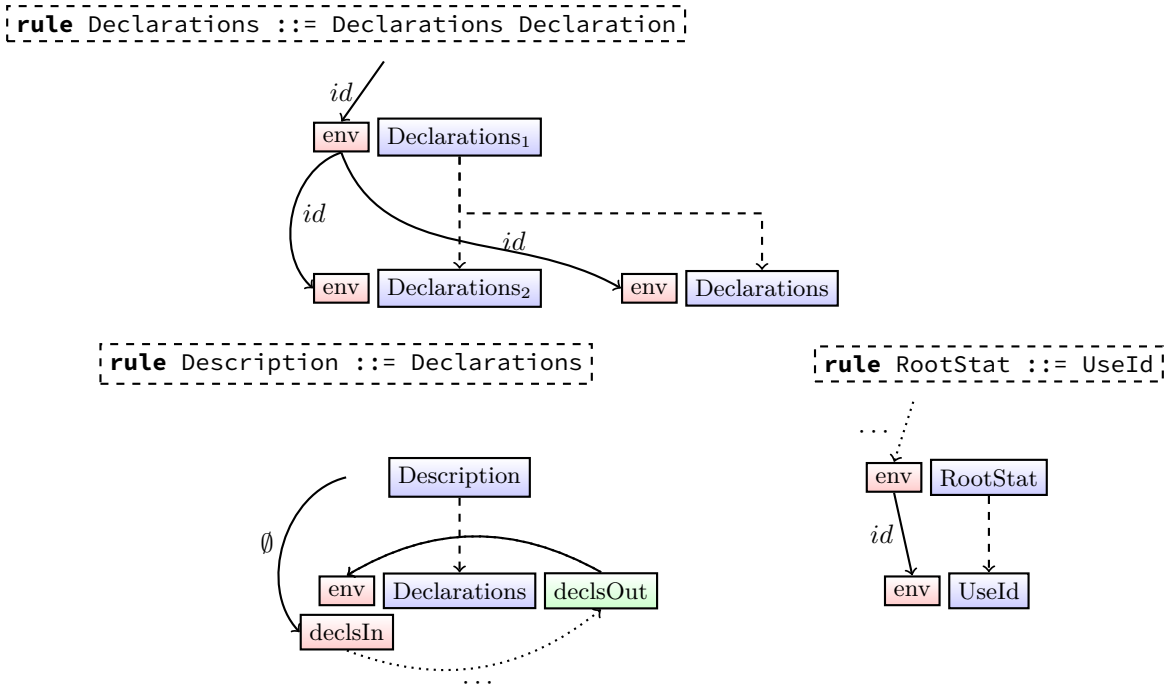


Abbildung 3.5. – Darstellung der Regeln aus Abbildung 3.4 und Abbildung 3.3. Symbole blau, synthetisierte Attribute eines Symbols grün rechts neben Symbol, ererbte Attribute eines Symbols in rot links neben dem Symbol. Berechnungen außerhalb der Produktion als gepunktete Linien, Berechnungen der Produktion mit Berechnungsvorschrift an durchgezogener Kante, pot. Ableitungsbaum der Produktion mit gestrichelten Linien.

heißt $\mathtt{X}_i.a \to \mathtt{X}_j.b \subseteq A \times A$ **lokale Abhängigkeit der Produktion** p. Für eine Produktion $p \in P$ heißt $DG_p \triangleq (A_p, DDP_p)$ mit $A_p \triangleq A_{\mathtt{X}_0} \cup \cdots \cup A_{\mathtt{X}_n}$ und $DDP_p \triangleq \{\mathtt{X}_i.a \to \mathtt{X}_j.b : \mathtt{X}_j.b \leftarrow f(\cdots \mathtt{X}_i.a \cdots) \in R_p\}$ **lokaler Abhängigkeitsgraph zu** p.

Der lokale Abhängigkeitsgraph einer Produktion ist somit der Graph, der aus allen lokalen Abhängigkeiten der Produktion und den darin involvierten Attributen besteht.

Definition 3.10. Eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ heißt **lokal azyklisch** genau dann, wenn für alle Produktionen $p \in P$ die Graphen DG_p nach Definition 3.9 azyklisch sind.

Definition 3.9 und Definition 3.10 führen zu der Definition wohldefinierter Attributgrammatik – jenen Attributgrammatiken für die eine berechenbare Reihenfolge (siehe Def. 3.8) existiert.

Definition 3.11. Eine konsistente attributierte Grammatik AG heißt **wohldefiniert**, genau dann, wenn für jeden abstrakten Syntaxbaum AST eine berechenbare Reihenfolge existiert.

Lemma 3.1. ([111, 117]) Eine konsistente attributierte Grammatik AG ist genau dann wohldefiniert, wenn sie vollständig ist und für jeden abstrakten Syntaxbaum AST der Abhängigkeitsgraph DT_T azyklisch ist.

Die genaue Definition des Abhängigkeitsgraphen eines abstrakten Syntaxbaums ist analog Definition 3.9, jedoch abhängig vom aufgebauten abstrakten Syntaxbaum und unter Verwendung der Abhängigkeit zwischen Attributen von Knoten. Die genaue Definition kann [117] entnommen werden. In dieser Arbeit ist nur die dahinter liegende Motivation relevant. Während Lemma 3.1 bzw. Definition 3.11 nur während der Ausführungszeit eines Übersetzers angewendet werden können, wird in der vorliegenden Arbeit die statische Bestimmung ähnlicher Eigenschaften untersucht.

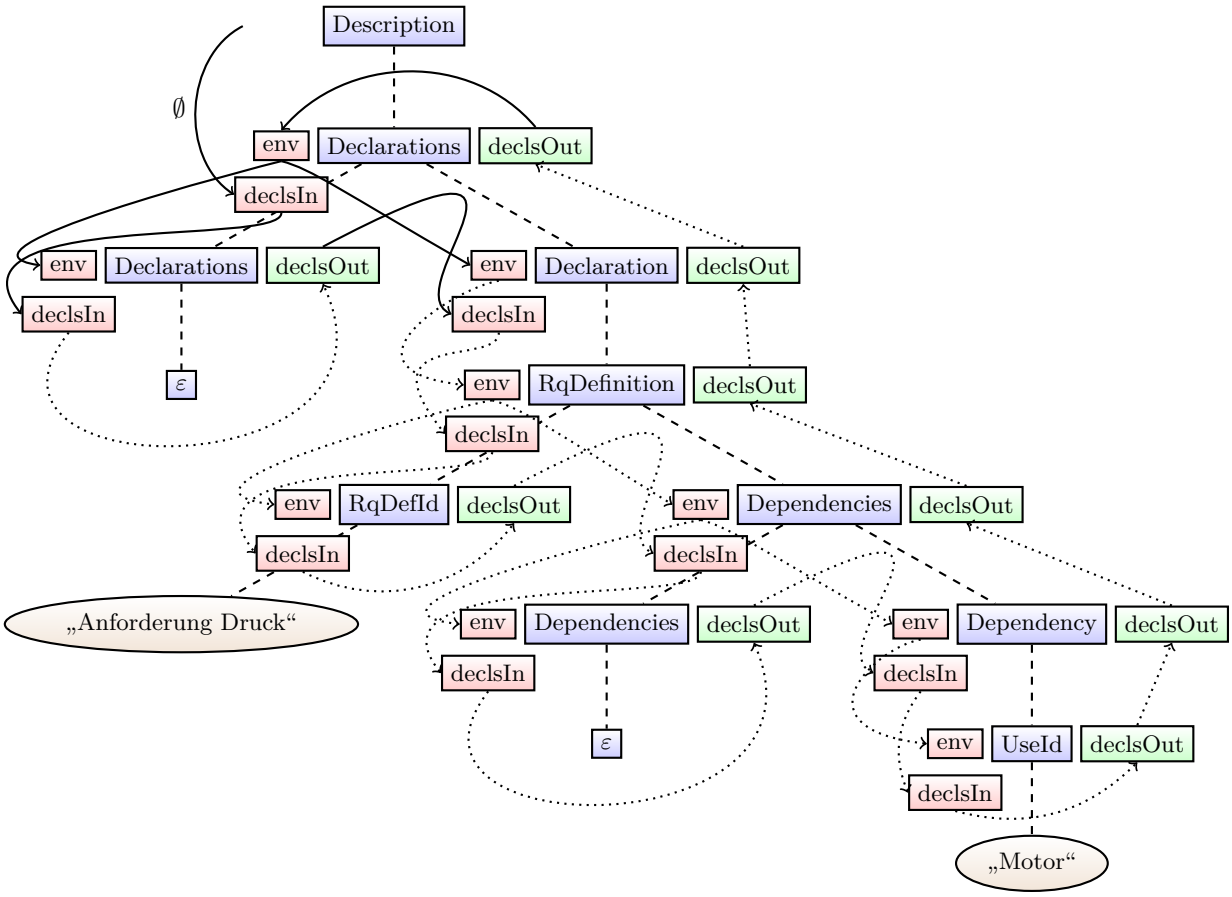


Abbildung 3.6. – Attributberechungen u. a. aus Abbildung 3.3 und Abbildung 3.4 angewandt auf den abstrakten Syntaxbaum aus Abbildung 3.2. Nichtterminale in blau, synthetisierte Attribute eines Symbols rechts vom Symbol in grün, ererbte Attribute eines Symbols links von diesem Symbol in rot. Ableitung eines Nichtterminals gestrichelt. Nicht in den Quelltexten vorkommende Berechnungsregeln gepunktet dargestellt und aus dem vollständigen Quelltext zu Beispiel 1.2 entnommen; andere Berechnungsvorschriften durchgängig mit verwendeter Hilfsfunktion an Kante angegeben. Terminalsymbole oval in Braun mit Inhalt im Knoten. Kanten ohne Angabe zeigen implizit die Hilfsfunktion *id* an.

Durch die Bestimmung in welcher Reihenfolge Attribute auswertbar sind, lassen sich in der Praxis effiziente Evaluatoren und Codegeneratoren für Attributgrammatiken implementieren.

Definition 3.12. Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und allen Symbolen $\mathsf{X} \in \Sigma$ heißt die Partitionierung von $A_\mathsf{X} = A_\mathsf{X}(1) \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$ **zulässige Zerlegung** genau dann, wenn für alle Symbole X gilt $A_\mathsf{X}(i) \subseteq AS_\mathsf{X}$ für $i = m_\mathsf{X}, m_\mathsf{X}-2, \cdots$ und $A_\mathsf{X}(i) \subseteq AI_\mathsf{X}$ für $i = m_\mathsf{X} - 1, m_\mathsf{X} - 3, \cdots$.

Laut Definition 3.12 werden Attribute somit in ererbte und synthetisierte Attribute aufgeteilt, sodass die letzten berechneten Attribute synthetisierte Attribute sind. Hintergrund ist, dass in der Literatur davon ausgegangen wird, dass das letzte Attribut in der Wurzel dem generierten Code eines Übersetzers entspricht.

Definition 3.13. Eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ heißt **zerlegbar** genau dann, wenn sie lokal azyklisch ist und für jedes Symbol $\mathsf{X} \in \Sigma$ eine zulässige Zerlegung $A_\mathsf{X} = A_\mathsf{X}(1) \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$ existiert, sodass für jeden abstrakten Syntaxbaum AST eine berechenbare Reihenfolge existiert so, dass die Attribute des Knoten K mit $Typ(K) = \mathsf{X}$ in der Reihenfolge $A_\mathsf{X}(1), \cdots, A_\mathsf{X}(m_\mathsf{X})$ berechnet werden können.

Definition 3.14. Eine zerlegbare attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit zulässiger Zerlegung $A_{\mathsf{X}} = A_{\mathsf{X}}(1) \uplus \cdots \uplus A_{\mathsf{X}}(m_{\mathsf{X}})$ für alle Symbole $\mathsf{X} \in \Sigma$ heißt **zerlegt**.

In der Arbeit [68] wurde gezeigt, dass die Bestimmung ob eine Attributgrammatik zerlegbar ist, NP-vollständig ist. Verschiedene Forschungen aufgrund dieser Problematik führten zu verschiedenen Klassen von Attributgrammatiken, wie wohldefiniert [111, 117] oder Attributgrammatiken mit vordefinierter Berechnungsreihenfolge wie [69, 93]. Auch heute wird noch an der Problematik der Auswertereihenfolge von Attributen geforscht, wie die beiden neueren Arbeiten [21] und [27] zeigen.

3.3. Geordnete Attributgrammatiken und Definitionstabellen

In [75] beschreibt Kastens eine Methode, wie unabhängig von den abstrakten Syntaxbäumen eine Berechnungsstrategie abgeleitet werden kann, die nicht vordefiniert ist und dennoch für alle abstrakten Syntaxbäumen Berechenbarkeit garantiert. Die von Kastens beschriebenen geordneten Attributgrammatiken (engl. ordered attribute grammars, OAGs) sind Grundlage dieser Arbeit. In weiteren Arbeiten, wie [76, 78], zeigt Kastens, wie mit OAGs die üblichen Analysen im Übersetzerbau implementiert werden können. Die Performanz der von Kastens in [76] und [78] vorgestellten Lösungen ist vergleichbar mit den damals vorherrschenden manuell programmierten Lösungen[76, 78].

$$
\begin{array}{lll}
\langle \mathit{Program} \rangle & ::= & \langle \mathit{Stats} \rangle \\
\langle \mathit{Stats} \rangle & ::= & \langle \mathit{Stats} \rangle \ \langle \mathit{Stat} \rangle \\
 & \mid & \langle \mathit{Stat} \rangle \ \langle \mathit{Stats} \rangle \\
 & \mid & \langle \mathit{Stat} \rangle \\
\langle \mathit{Stat} \rangle & ::= & \langle \mathit{VarStat} \rangle \\
\langle \mathit{VarStat} \rangle & ::= & \text{id id} \\
\langle \mathit{VarStat} \rangle & ::= & \text{id number}
\end{array}
$$

Grammatik 3.1 – Abstrakte Syntax für eine Attributgrammatik die geordnet ist, aber nicht durch eine statische Besuchsstrategie ausgewertet werden kann.

Zur Bestimmung einer Ordnung in der die Attribute ausgewertet werden können, die für alle Eingaben gültig ist und nie in einem abstrakten Syntaxbaum einen Zyklus verursacht, wird die Definition einer Reihe von Mengen auf Basis von Definition 3.9 verwendet:

Definition 3.15. ([81]) Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für alle Produktionen $p \in P$ mit $p : \mathsf{X}_0 ::= \mathsf{X}_1 \cdots \mathsf{X}_n$ heißt $NDDP_p = DDP_p^+ \setminus \{(\mathsf{X}_i.a, \mathsf{X}_j.b) : \mathsf{X}_i.a$ und $\mathsf{X}_j.b$ werden in p definiert $\}$ **normalisierte direkte Abhängigkeit der Produktion** p. Für alle Symbole $\mathsf{X} \in \Sigma$ sind die **induzierten Attributabhängigkeiten** das kleinste System von Mengen $IDS_{\mathsf{X}} \subseteq A_{\mathsf{X}} \times A_{\mathsf{X}}$, $IDP_p \subseteq A \times A$, welches folgende Gleichungen erfüllt:

1. $NDPP_p \subseteq IDP_p$
2. $IDS_{\mathsf{X}} = \{\mathsf{X}.a \to \mathsf{X}.b \colon \exists q \in P$ sodass $\mathsf{X}.a \to \mathsf{X}.b \in IDP_q^+\}$
3. $IDP_p = IDP_p \cup IDS_{\mathsf{X}_0} \cup \cdots \cup IDS_{\mathsf{X}_n}$

IDS sind die **induzierten Abhängigkeiten zwischen Symbolattributen**, IDP sind die **induzierten Abhängigkeiten zwischen Attributvorkommen**.

Die Attributgrammatik AG heißt **absolut azyklisch** genau dann, wenn für alle Nichtterminale $\mathsf{X} \in N$ und alle Produktionen $p \in P$ IDS_{X} und IDP_p azyklisch sind.

Diese Definition legt implizit eine Halbordnung der Attribute fest. Kann in jedem Kontext eines Symbols einer Attributgrammatik für die mit diesem Symbol assoziierten Attribute eine Auswertereihenfolge, d.h. Zerlegung, angegeben werden, die diese Halbordnung enthält, dann heißt diese Attributgrammatik *geordnet*.

Definition 3.16. (siehe [75]) Sei $AG \triangleq (G, A, R, B)$ eine attributierte Grammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für alle Symbole $\mathtt{X} \in \Sigma$ seien

1. $T_{\mathtt{X}}(-1) \triangleq \emptyset$
2. $T_{\mathtt{X}}(0) \triangleq \emptyset$
3. $T_{\mathtt{X}}(2k-1) \triangleq \{a \in AS_{\mathtt{X}}$ so, dass $\forall b \in A_{\mathtt{X}} : \mathtt{X}.a \to \mathtt{X}.b \in IDS_{\mathtt{X}} \Rightarrow \exists j \leq 2k-1$ mit $\mathtt{X}.b \in T_{\mathtt{X}}(j)\}$
4. $T_{\mathtt{X}}(2k) \triangleq \{a \in AI_{\mathtt{X}}$ so, dass $\forall b \in A_{\mathtt{X}} : \mathtt{X}.a \to \mathtt{X}.b \in IDS_{\mathtt{X}} \Rightarrow \exists j \leq 2k$ mit $\mathtt{X}.b \in T_{\mathtt{X}}(j)\}$
5. $A_{\mathtt{X}}(i) \triangleq T_{\mathtt{X}}(m-i+1) \setminus T_{\mathtt{X}}(m-i-1)$ mit $i = 1, \cdots, m$

wobei $m \in \mathbb{N}$ minimal und $T_{\mathtt{X}}(m-1) \cup T_{\mathtt{X}}(m) = A_{\mathtt{X}}$. Die attributierte Grammatik heißt **geordnet** (OAG) genau dann, wenn sie mit den Partitionen $A_{\mathtt{X}} = A_{\mathtt{X}}(1) \uplus \cdots \uplus A_{\mathtt{X}}(m_{\mathtt{X}})$ zerlegt ist und für alle Produktionen $p \in P$ der erweiterte Abhängigkeitsgraph $EDP_p \triangleq IDP_p \cup \{\mathtt{X}.a \to \mathtt{X}.b : \exists h, k \in \mathbb{N}$ mit $\mathtt{X}.a \in A_{\mathtt{X}}(h) \wedge \mathtt{X}.b \in A_{\mathtt{X}}(k) \wedge h < k\}$ azyklisch ist.

Bei der Erstellung der Besuchssequenz wird der erweiterte Abhängigkeitsgraph topologisch sortiert, jedoch so, dass alle innerhalb einer Zerlegungsmenge $A_{\mathtt{X}}(i)$ liegenden Attribute „wie ein Attribut“ behandelt werden. Die Intuitive Idee hinter geordneten Attributgrammatiken ist es, die Attribute eines Symbols unabhängig vom Vorkommen dieses Symbols in ererbte und synthetisierte aufzuteilen. Die Attribute sollen dann so spät wie möglich berechnet werden, jedoch so, dass alle Attribute rechtzeitig berechnet sind.

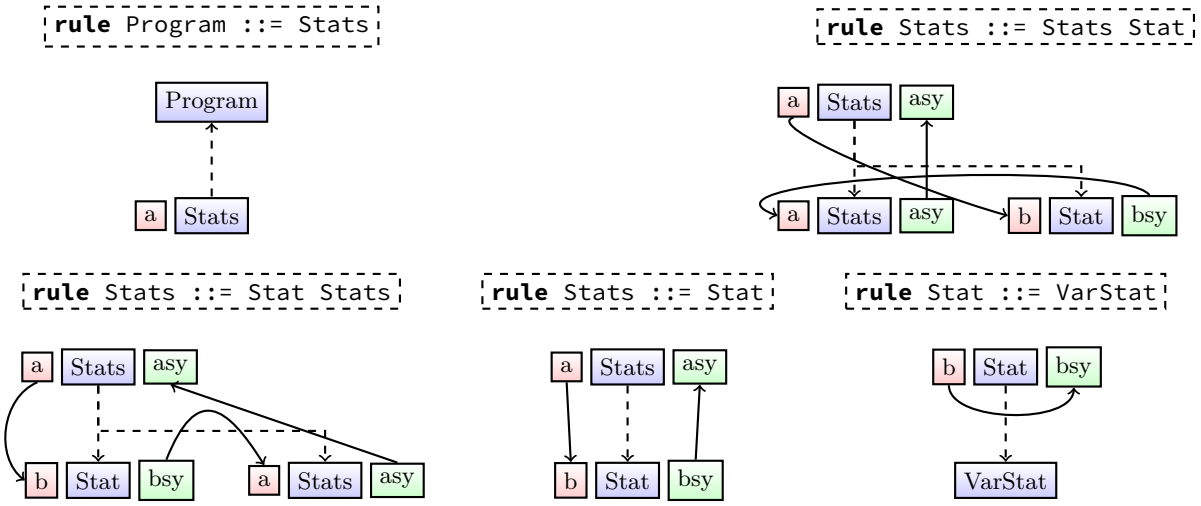


Abbildung 3.7. – Direkte Abhängigkeiten der Attributgrammatik aus Beispiel 3.1 mit abstrakter Syntax aus Grammatik 3.1. Nicht aufgeführt sind Attribute, die in einer Produktion weder verwendet noch definiert werden obwohl diese in einer den Symbolen zugeordneten Produktionen definiert oder gelesen werden.

In Beispiel 3.1 (Seite 33) ist eine Attributgrammatik gegeben, die nicht mit einer fest vorgegebenen Besuchsstrategie berechnet werden kann (siehe u. a. [69, 117]), jedoch geordnet ist. Die abstrakte Syntax zu dieser Attributgrammatik ist in Grammatik 3.1 angegeben.

Die (direkten) Abhängigkeiten der Attributgrammatik aus Beispiel 3.1 sind grafisch in Abbildung 3.7 dargestellt. Aus Abbildung 3.7 kann erkannt werden, dass die lokalen Abhängigkeitsgraphen azyklisch sind, d. h. die Attributgrammatik lokal azyklisch ist. Auf die Darstellung induzierter Abhängigkeiten wurde verzichtet, da diese sehr schnell die Graphen unübersichtlich machen, stattdessen werden die den Symbolen zugeordneten induzierten Abhängigkeiten in Tabelle 3.2 angegeben.

Neben dieser Partitionierung muss zur Prüfung, ob die Attributgrammatik geordnet ist, gezeigt werden, dass die Mengen EDP_p für alle Produktionen p der Attributgrammatik azyklisch sind. Im Beispiel kommen also zu den induzierten Abhängigkeiten der Produktionen genau die Kanten den induzierten Abhängigkeiten der Symbole für die jeweilige Produktion hinzu. Dies muss so nicht immer der Fall sein.

Der Vorteil geordneter Attributgrammatiken ist, dass alle Entscheidungen bzgl. der Attributauswertung bereits zur Erstellungszeit eines Übersetzers aus der Spezifikation der Attributgrammatik getroffen sind. Dies erlaubt verschiedene Optimierungen, wie die Verwendung globaler Variablen für Attribute, die in mehreren Teilbäumen genutzt werden [75, 76, 117]. Diese Optimierung ermöglicht auch die Verwendung großer Datentypen und Tabellen als Attribute.

Es lässt sich auch zeigen, dass jede zerlegbare Attributgrammatik in eine geordnete Attributgrammatik überführt werden kann, indem zusätzliche Abhängigkeiten innerhalb der induzierten Abhängigkeiten für ein Symbol hinzugefügt werden [117]. Kastens hat darüber hinaus in [75] bereits gezeigt, dass für alle Attributgrammatiken mit fest vorgegebener Berechnungsstrategie (z. B. eine links-rechts Tiefensuche oder beliebig viele links-rechts- und rechts-links Tiefensuchen) diese Attributgrammatik auch geordnet ist, oder sich in eine solche durch Hinzufügen zusätzlicher Abhängigkeiten überführen lässt.

```
1  rule Program ::= Stats
2  attr Stats.a ← 10
3
4  rule Stats ::= Stats Stat
5  attr Stat.b ← Stats₁.a
6    Stats₂.a ←Stat.bsy
7    Stats₁.asy ←Stats₂.asy
8
9  rule Stats ::= Stat Stats
10 attr Stat.b ← Stats₁.a
11   Stats₂.a ←Stat.bsy
12   Stats₁.asy ←Stats₂.asy
13
14 rule Stats ::= Stat
15 attr Stats.asy ← Stat.bsy
16   Stat.b ← Stats.a
17
18 rule Stat ::= VarStat
19 attr Stat.bsy ← Stat.b
```

Beispiel 3.1 – Beispiel einer geordneten Attributgrammatik für Grammatik 3.1, die für keine fest vorgegebene Besuchsstrategie berechenbar ist.

Das Hinzufügen zusätzlicher Attributabhängigkeiten um bspw. Zyklen im erweiterten Abhängigkeitsgraphen zu verhindern, erfolgt durch Hinzufügen dieser Attribute zu einer Berechnung hinter einem zusätzlichen Pfeil ↞.

Definitionstabellen werden üblicherweise als große Objekte durch den Baum transportiert oder mittels globaler Objekte definiert, die mit Seiteneffekten beschrieben werden können.

Das Beschreiben mittels Seiteneffekten führt dann entweder dazu, dass Änderungen zu Attributberechnungen propagiert werden müssen, wie in [58] oder [56] beschrieben, oder dass die Reihenfolge der

Symbol	**IDS**	**T(1)**	**T(2)**	**A(1)**	**A(2)**
`Program`	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
`Stats`	$\{\mathtt{a} \rightarrow \mathtt{asy}\}$	$\{asy\}$	$\{a\}$	$\{a\}$	$\{asy\}$
`Stat`	$\{\mathtt{b} \rightarrow \mathtt{bsy}\}$	$\{bsy\}$	$\{b\}$	$\{b\}$	$\{bsy\}$
alle anderen	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

Tabelle 3.2. – Induzierte Abhängigkeiten für Symbole sowie die resultierende Zerlegung für die Symbole zur Attributgrammatik aus Beispiel 3.1.

Seiteneffekte relevant wird. In geordneten Attributgrammatiken kann dies mittels der bereits erwähnten virtuellen Abhängigkeiten dargestellt werden.

Attribute, die rein für die „Ordnung“ der Seiteneffekte notwendig sind, werden *void*-Attribute genannt. Diese Attribute und die dazugehörigen Attributierungsregeln repräsentieren selbst keine semantisch relevanten Werte, dienen jedoch dem Erzwingen einer Partitionierung. Für diese Arbeit ausreichend zu wissen ist, dass *void*-Attribute keinen lesbaren Wert haben, aber in Form von Abhängigkeiten in die Bestimmung der Zerlegung eingehen. Somit können *void*-Attribute zwar geschrieben, jedoch nicht direkt gelesen werden. Der „lesende“ Zugriff erfolgt über das Anhängen nach dem Abhängigkeitspfeil ↞.

Auf Einträge in der Definitionstabelle wird über ein ausgezeichnetes Attribut zugegriffen. Für ein Attribut `b` des Symbols `X` `X.b` einer Attributgrammatik wird auf die Spalten der Definitionstabelle lesend wie schreibend mit `X.b : s` für die Spalte `s` zugegriffen.

Kapitel 4.

Typische Muster auf Attributgrammatiken

Weiterleitung, Referenzattribute und verschiedene Methoden der Komposition von Attributgrammatiken, sowie auch die Programmierung von Attributierungsregeln mit Termersetzungssystemen, sind einige der möglichen Alternativen zu „klassischen“ Attributgrammatiken mit geringerem Spezifikationsumfang. Nachteil dieser ist unter anderem, dass die resultierenden Übersetzer und Werkzeuge zur Programmanalyse während der Laufzeit abbrechen. Ursache für solche Abbrüche ist unter anderem, dass es für einige dieser Klassen unentscheidbar ist ob die zugrunde liegende Attributgrammatik berechenbar ist[26].

Die in dieser Arbeit vorgeschlagene Lösung zur Überbrückung dieser Diskrepanz wird als Muster bezeichnet. Im allgemeinen sind Muster wiederkehrende Komponenten oder Strukturelemente mit derer andere Muster aufgebaut werden können[6, 70]. Muster auf Attributgrammatiken müssen somit folgende Eigenschaften erfüllen:

- Kompaktheit bzw. höherer Abstraktionsgrad gegenüber klassischen Attributgrammatiken;
- Möglichkeit der Komposition zur Herstellung neuer Muster und, dass
- resultierende Übersetzer nicht weniger performant sind als ohne Verwendung von Mustern.

Aus bisherigen Arbeiten ist bekannt, dass geordnete Attributgrammatiken geeignet sind, um performante Übersetzer zu erstellen[16, 17, 76, 78]. Existiert für eine Attributgrammatik eine berechenbare Reihenfolge, die statisch bestimmbar ist, so kann diese Attributgrammatik geordnet werden [75]. Im Gegensatz zu Referenzattributgrammatiken oder ähnlichen Ansätzen sollen Muster (und deren Komposition) so gestaltet sein, dass eine berechenbare Reihenfolge für alle möglichen abstrakten Syntaxbäume statisch bestimmbar ist. Damit kann erreicht werden, dass die Eigenschaft der Performanz gewährleistet werden kann.

Im Rahmen dieser sind Muster als Abbildungen auf Attributgrammatiken mit fixierter abstrakter Syntax definiert; Ein Muster ist eine Abbildung, die eine gegebene Attributgrammatik in eine andere Attributgrammatik überführt, sodass die resultierende Attributgrammatik zerlegbar ist. Definition 4.1 formalisiert diese Aussage:

Definition 4.1. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax G. Ein **Muster** ist eine Abbildung $\mathcal{M}_G \colon \mathcal{AG} \to \mathcal{AG}$.

Ein Beispiel eines Musters ist die Identitätsfunktion, bei der keine Änderung an der Attributgrammatik geschieht, sodass eine zerlegbare Attributgrammatik weiterhin eine zerlegbare Attributgrammatik bleibt. Grundsätzlich können Transformationen einer Attributgrammatik aus einer beliebigen, sogar inkonsistenten Attributgrammatik, eine zerlegbare Attributgrammatik erzeugen. Dies wäre bspw. der Fall, wenn Attributierungsregeln hinzu kommen, die eine unvollständige Attributgrammatik so erweitern, dass eine Bestimmung einer Zerlegung überhaupt erst möglich ist.

Definition 4.2. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax G und $\mathcal{M}_G \colon \mathcal{AG} \to \mathcal{AG}$ eine Abbildung. Seien $AG, AG' \in \mathcal{AG}_G$ Attributgrammatiken und $AG' \triangleq \mathcal{M}_G(AG)$ die resultierende

Attributgrammatik zu AG nach Anwendung von $\mathcal{M}_G$. Ist AG nicht zerlegbar heißt $\mathcal{M}_G$ **zerlegungsherstellendes Muster**, falls die resultierende Attributgrammatik AG' zerlegbar ist; und **zerlegungserhaltendes Muster**, wenn AG und AG' zerlegbar sind.

Ein Problem von Abbildungen, die die zugrunde liegende abstrakte Syntax ändern würden, ist, dass bei dieser Änderung eine bestehende Zerlegung unmittelbar zerstört wird und ggf. die Attributgrammatik nicht mehr zerlegbar sein kann. Die (zerlegungserhaltenden) Muster dieser Arbeit sollen allerdings gerade sicherstellen, dass die resultierende Attributgrammatik wieder zerlegbar ist. Ein Muster, welches die zugrunde liegende abstrakte Syntax um beliebige neue Produktionen oder gar Symbole erweitert, kann schnell in einer Attributgrammatik resultieren, die nicht mehr die bisherigen Eigenschaften (z.B. zerlegbar oder geordnet) einhalten kann. Wird Beispiel 3.1 um eine Produktion `Stats` ::= `Stats Stats` mit, den in Beispiel 4.1 angegebenen, dazugehörigen Attributierungsregeln erweitert, so ist diese Attributgrammatik nicht mehr geordnet. Selbst unter Angabe zusätzlicher Abhängigkeiten existieren Zyklen in den induzierten Abhängigkeitsgraphen.

```
rule Stats ::= Stat
attr Stats.asy ← 10
     Stat.b ← Stats.a

rule Stats ::= Stats Stats
attr Stats₃.a ←Stats₂.asy ←Stats₁.a
     Stats₂.a ←Stats₃.asy ← Stats₁.a
     Stats₁.asy ←Stats₃.asy ← Stats₁.a
```

Beispiel 4.1 – Erweiterung der abstrakten Syntax mit dazugehörigen Attributierungsregeln zu Beispiel 3.1 führt zu nicht geordneter Attributgrammatik.

Eine einfache Variante bei der durch Änderung der abstrakten Syntax eine nicht berechenbare Attributgrammatik erzeugt wird, ist das Hinzufügen eines bestehenden Nichtterminals mit anderen Attributen zu einer bestehenden Produktion. So könnte die Bestimmung eines Attributs dann nicht mehr in den Attributierungsregeln enthalten sein – die Attributgrammatik wäre unvollständig.

Da Muster keine Änderung der abstrakten Syntax vornehmen, kommen für Muster nur Änderungen an den Attributen und Attributierungsregeln sowie den Bedingungen als Möglichkeit in Betracht. In dieser Arbeit werden Änderungen der Bedingungen nicht betrachtet, da diese in der Praxis durch Attributierungsregeln emuliert werden (können). Diese Vorgehensweise ist üblich [27, 76]. Eine Bedingung kann als eigenes Attribut definiert werden, wobei in einem bedingten Ausdruck die Definition der Bedingung ausgewertet wird. Als Ergebnis steht dann entweder wahr oder das Programm wird abgebrochen.

Ein Muster wird nun dadurch definiert, dass Attribute oder Attributierungsregeln einer Attributgrammatik entfernt oder hinzugefügt werden können. Folgende Definition stellt dies dar:

Definition 4.3. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax G und $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik. Ein Tupel $\mathsf{M}_{AG} \triangleq (\mathsf{A}_-, \mathsf{A}_+, \mathsf{R}_-, \mathsf{R}_+)$ definiert die resultierende Attributgrammatik $AG' \triangleq (G', A', R', B')$ mit

- $G' = G$,
- $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$,
- $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ und
- $B' = B$.

M_{AG} heißt dann **Attributgrammatik-abhängige Musterdefinition** zu AG; die Mengen des Tupels $(\mathsf{A}_-, \mathsf{A}_+, \mathsf{R}_-, \mathsf{R}_+)$ heißen **Änderungsmengen (einer Attributgrammatik)**.

Eine Attributgrammatik-abhängige Musterdefinition M_{AG} kann somit als auf eine konkrete Attributgrammatik angepasste Instanz der allgemeinen Formulierung eines Musters als Abbildung $\mathcal{M}_G$ verstanden werden.

Nach Definition 4.3 existieren grundsätzlich zwei Operationen zur Änderung einer Attributgrammatik:

1. Hinzufügen von Attributen und Attributierungsregeln und

2. Entfernen von Attributen und Attributierungsregeln.

Mit Definition 4.3 ist damit sichergestellt, dass Attribute bzw. Attributierungsregeln, die in einem Muster „bearbeitet“ werden, erst entfernt und dann hinzugefügt werden. Damit kann sichergestellt werden, dass die Änderung eines Attributs, bzw. einer Attributierungsregel nicht durch das dafür notwendige Entfernen zu einer inkonsistenten Attributgrammatik führt.

Hintergrund des folgenden Lemmas ist der Schrittweise Aufbau von Attributgrammatik-unabhängigen Musterdefinitionen.

Lemma 4.1. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax G und $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik. Sei M_{AG} eine Attributgrammatik-abhängige Musterdefinition und resultierender Attributgrammatik AG' nach Definition 4.3 zu AG. M_{AG} definiert ein zerlegungserhaltendes Muster genau dann, wenn AG' zerlegbar ist.

Beweis. Durch Angabe einer zerlegungserhaltenden Funktion

$$\mathcal{M}_G(ag) = \begin{cases} AG' & \text{für } ag = AG \\ ag & \text{sonst} \end{cases}$$

□

Ein Muster nach Lemma 4.1 wird in Beispiel 4.2 vorgestellt. Die Definition einer zusätzlichen Funktion zur Unterscheidung auf welche Attributgrammatik die Mengenvereinigung angewandt wird, ist nur notwendig um die Eigenschaften aus Definition 4.1 zu erhalten. Im Folgenden wird auf diese (explizite) Definition verzichtet.

Die Motivation dieser Herangehensweise ergibt sich daraus, bestehende Attribute einer Attributgrammatik verwenden zu können und dennoch (garantiert) eine konsistente Attributgrammatik mit berechenbarer Reihenfolge der Attribute zu erhalten. Letztendlich ist die Aussage von Definition 4.3 und Lemma 4.1, dass das Resultat einer Musteranwendung eine zerlegbare Attributgrammatik sein muss. Abweichungen können sich nur ergeben, wenn die Änderung kein Muster ist oder die Eingabe bereits nicht zerlegbar war.

Im Folgenden werden zuerst Eigenschaften der Änderungsmengen untersucht. Ausgehend von einer zerlegbaren Attributgrammatik soll gezeigt werden, welche Eigenschaften (zerlegungserhaltende) Muster und die damit verbundenen Änderungsmengen einhalten müssen, damit eine zerlegbare Attributgrammatik als resultierende Attributgrammatik erzeugt wird. Gewonnen wird dadurch die Aussagen, dass die resultierende Attributgrammatik nicht nochmals auf Zerlegbarkeit untersucht werden muss. Die Information, dass das Muster zerlegungserhaltend ist und auf eine zerlegbare Attributgrammatik angewendet wurde, soll zur Entscheidung, dass die resultierende Attributgrammatik zerlegbar ist, ausreichen. Darauf aufbauend sollen diese Änderungsmengen aus einer, von der Attributgrammatik unabhängigen, Definition hergeleitet werden. Aufbauend auf diesem System ist dann zu untersuchen, welche Eigenschaften zerlegungserhaltende Muster einhalten müssen, damit geordnete Attributgrammatiken verwendet werden können. Im weiteren Verlauf dieser Arbeit wird davon ausgegangen, dass Muster zerlegungserhaltend sind (oder sein sollen).

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr
7  rule VarStat ::= id number attr
```

a) Initiale Attributgrammatik ohne Attributierungsregeln

$$
\begin{aligned}
\mathsf{A}_- &= \emptyset \\
\mathsf{R}_- &= \emptyset \\
\mathsf{A}_+ &= \{\texttt{VarStat.number}\} \\
\mathsf{R}_+ &= \{\textbf{rule}\ \texttt{VarStat} ::= \texttt{id id} \\
&\qquad \textbf{attr}\ \texttt{VarStat.number} \leftarrow 0, \\
&\qquad \textbf{rule}\ \texttt{VarStat} ::= \texttt{id number} \\
&\qquad \textbf{attr}\ \texttt{VarStat.number} \leftarrow 1 \qquad \}
\end{aligned}
$$

b) Explizite Mengen

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr VarStat.number = 0
7  rule VarStat ::= id number attr VarStat.number = 1
```

c) Resultierende Attributgrammatik

Beispiel 4.2 – Explizite Angabe der Mengen zur Musteranwendung und resultierende Attributgrammatik für eine initial leere Attributgrammatik zur Abstrakten Syntax aus Grammatik 3.1.

4.1. Eigenschaften von Änderungsmengen

Die Änderungsmengen aus Definition 4.3 haben eine Reihe von Eigenschaften zu erfüllen, damit das Hinzufügen und Entfernen von Attributen und Attributierungsregeln einer zerlegbaren Attributgrammatik in einer zerlegbaren Attributgrammatik mündet.

Eine Frage bei dem Hinzufügen von Attributierungsregeln ist, welche Eigenschaften diese neuen Regeln erfüllen müssen, damit die resultierende Attributgrammatik nicht bspw. unmittelbar zyklisch wird. Zur Analyse dieser Problematik wird der folgende Begriff der vereinbar hinzufügbaren Attributierungsregel eingeführt:

Definition 4.4. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, für ein Attribut $a \notin AD_p$ einer Produktion $p \in P$ und einem Symbol $X \in \Sigma$ und $\lceil X \rceil_p \geq 1$, $b_1, \cdots, b_n \in A$ und für alle b_i, $1 \leq i \leq n$ und $n \in \mathbb{N}$ gilt, dass $b_i \neq a$ und es existieren $q_i \in P$ sodass $b_i \in AD_{q_i}$, $b_i \in A(Y_i)$, $\lceil Y_i \rceil_{q_i} - |Y_i|_{q_i} = 1$ für $Y_i \in \Sigma$ und $\lceil Y_i \rceil_p > 0$. Eine zu AG **vereinbar hinzufügbare** Attributierungsregel $r^{(a,p)}$ bezüglich dem Attribut a und der Produktion p, sodass $AG' = (G, A', R', B)$ mit $A' = A \cup \{a\}$, $a \in A'(X)$, $R' = R \cup \{r^{(a,p)}\}$ ist induktiv wie folgt definiert:

1. $r^{(a,p)} = a \leftarrow c$ für eine Konstante c ist vereinbar hinzufügbar,
2. $r^{(a,p)} = a \leftarrow f(b_1, \cdots, b_n)$ ist vereinbar hinzufügbar, falls $r^{(a,p)} \in R'_p$ und DG_p bzgl. R' azyklisch ist.

Bezüglich Definition 4.4 ist zu beachten, dass falls a bereits in der ursprünglichen Attributgrammatik existiert, $A' = A$ ist. Die Idee hinter dieser Definition ist, dass beliebige Attributierungsregeln einer Attributgrammatik hinzugefügt werden können, solange dieses Attribut bisher in dieser Produktion nicht definiert wurde, und durch Hinzufügen dieser Attributierungsregel kein lokaler Zyklus eingeführt wird.

Das Hinzufügen mehrerer, jeweils vereinbar hinzufügbarer Attributierungsregeln nach Definition 4.4 bedeutet nur, dass durch diese Attributierungsregeln selbst keine unmittelbaren Zyklen hinzugefügt werden können. Grundsätzlich kann durch Hinzufügen vereinbar hinzufügbarer Attributierungsregeln die resultierende Attributgrammatik Zyklen (in den induzierten Abhängigkeiten) enthalten, inkonsistent sein oder

```
rule Stats ::= Stats Stat
attr Stat.a ← Stats₂.b + Stats₂.asy
     Stats₂.b ←Stats₁.b + Stat.asy

rule Stats ::= Stat Stats
attr Stat.a ← Stats₂.b + Stats₂.asy
     Stats₂.b ←Stats₁.b

rule Stats ::= Stat attr Stat.a ← Stats.b + Stats.asy
rule Program ::= Stats attr Stats.b ←Stats.asy
rule Stat ::= VarStat attr Stat.asy ← Stat.a + Stat.b + Stat.bsy
```

a) Vereinbar hinzufügbare Attributierungsregeln, die Zyklen in induzierten Abhängigkeiten erzeugen.

```
rule Stats ::= Stats Stat attr Stat.a ← 5
```

b) Vereinbar hinzufügbare Attributierungsregeln, die Unvollständigkeit in resultierender Attributgrammatik erzeugen (würden).

```
rule Program :: Stats attr Stats.a ← 1000
```

c) Vereinbar hinzufügbare Attributierungsregeln, die Inkonsistenz in resultierender Attributgrammatik erzeugen (würden).

Beispiel 4.3 – Beispiel dreier Varianten in denen mehrere vereinbar hinzufügbare Attributierungsregeln zu Beispiel 3.1 hinzugefügt werden, wobei Zyklen in induzierten Abhängigkeiten, Unvollständigkeit oder Inkonsistenz Resultat sind

auch unvollständig. Beispiel 4.3 erweitert Beispiel 3.1 um vereinbar hinzufügbare Attributierungsregeln. In 4.3a verursachen die vereinbar hinzufügbaren Attributierungsregeln einen Zyklus in den induzierten Abhängigkeiten ähnlich wie Beispiel 4.1. In Beispiel 4.3b ist die resultierende Attributgrammatik unvollständig, in Beispiel 4.3c ist die resultierende Attributgrammatik inkonsistent.

Lemma 4.2. Sei $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und $r_1^{(a_1,p_1)}, \cdots, r_n^{(a_n,p_n)}$ jeweils zu AG vereinbar hinzufügbare Attributierungsregeln. Seien weiterhin die Mengen

$$
\begin{aligned}
R_1 &= \{r_1^{(a_1,p_1)}\} \\
&\cdots \\
R_n &= \{r_n^{(a_n,p_n)}\} \\
A_1 &= \{a_1\} \\
&\cdots \\
A_n &= \{a_n\} \\
R_\cup &= R_1 \cup \cdots \cup R_n \\
A_\cup &= A_1 \cup \cdots \cup A_n
\end{aligned}
$$

und $AG' = (G, A', R', B)$ wobei $A' = A \cup A_\cup$ und $R' = R \cup R_\cup$ dann ist für jede Produktion $p \in P$ der lokale Abhängigkeitsgraph DG_p azyklisch oder AG' ist inkonsistent.

Beweis. Angenommen es existiert eine Produktion $p \in P$ bzgl. AG' für die DG_p einen Zyklus enthält. Dann ist zu zeigen, dass AG' inkonsistent ist oder AG nicht zerlegbar. AG' ist inkonsistent bedeutet, es existiert ein Attribut $a \in A$ und (mindestens) zwei Regeln $r_i, r_j \in R'$ mit $r_i = a \leftarrow \cdots$ und $r_j = .a \leftarrow \cdots$ für $i, j \in [1, n]$ und $i \neq j$.

Dann existieren folgende Fälle:

1. $r_i, r_j \in R$, beide Regeln existieren bereits in AG.
2. Sei $r_i \in R_\cup$ ist vereinbar hinzufügbare Regel und $r_j \in R$ bereits in AG.
3. r_i und r_j sind vereinbar hinzufügbare Regeln, d. h. $r_i, r_j \in R_\cup$.

Fall 1 steht im Widerspruch zur Voraussetzung, dass AG zerlegbar (und damit konsistent) ist.

Für Fall 2 existiert eine Produktion $p \in P$ für die DG_p zyklisch für AG' nach Voraussetzung ist. Damit dies der Fall ist, muss ein gerichteter Kantenzug $c_1, \cdots, c_n, c_1$ in DG_p bzgl. AG' existieren. Da nach Voraussetzung dieser Kantenzug in DG_p bzgl. AG nicht existiert muss eine neue Attributierungsregel eine Kante in DG_p hinzugefügt haben, damit dieser gerichtete Kantenzug existieren kann. Sei (c_i, c_j) diese neue Kante, damit existiert eine Regel $r^{(c_j,p)} = c_j \leftarrow f(\cdots, c_i, \cdots) \in R_\cup$ und da $r^{(c_j,p)}$ vereinbar hinzufügbar ist $c_j \notin AD_p$. Allerdings ist $c_j \in A$, da sonst keine Kante (c_k, c_j) in DG_p hätte existieren können ($(c_j, c_k) \in DDP_p$ bzgl. AG). Da AG zerlegbar und damit vollständig war, existiert eine Regel $r = c_j \leftarrow f(\cdots)$, $r \in R$. Dann ist $r^{(c_j,p)} = r_i$ und $r = r_j$. Damit ist AG' inkonsistent.

Für Fall 3 existiert ebenso eine Produktion $p \in P$ für die DG_p zyklisch bzgl. AG' ist. Analog der Argumentation in Fall 2 bedeutet dies, dass ein Kantenzug $c_1, \cdots, c_n, c_1$ in DG_p bzgl. AG' existiert. Schließt eine Kante diesen zyklischen Kantenzug mit einer Kante (c_i, c_j), dann gilt die Argumentation von Fall 2 und r_i und r_j definieren dasselbe (bereits in AG vorhandene) Attribut. Weren (c_i, c_j) und (c_j, c_k) zum Schluss des Zyklus benötigt, dann gilt die Argumentation aus Fall 2 ebenfalls, jedoch für bestehende Regeln $r_{c_j}, r_{c_k} \in R$. Analoges für weitere Regeln $r' \in R_\cup$. □

Definition 4.4 und Lemma 4.2 lassen sich wie folgt zusammenfassen: durch Hinzufügen vereinbar hinzufügbarer Attributierungsregeln können nur Zyklen erzeugt werden, wenn dadurch auch die Attributgrammatik inkonsistent wird. Der Ausschluss dieser Inkonsistenz (sowie der möglichen Unvollständigkeit) wird im Folgenden gezeigt.

Bemerkung (Vereinbar hinzufügbare Attributierungsregeln mit konstanter Attributierung)**.** Im Beweis zu Lemma 4.2 werden nur vereinbar hinzufügbare Attributierungsregeln der zweiten Form (Def. 4.4) betrachtet. Attributierungsregeln der ersten Form, die zwar zu einer inkonsistenten Attributgrammatik führen (Attribut existiert in A der ursprünglichen Attributgrammatik wird aber in Produktion p nicht definiert, aber in q in anderer Richtung – ererbt statt synthetisiert bspw.) kann aber keinen Zyklus hinzufügen.

Folgender Satz beschreibt die Haupteigenschaften, die eine Änderung in Form eines Musters einhalten muss, damit die Anwendung des Musters in einer zerlegbaren Attributgrammatik resultiert. Die Formulierung der ersten beiden Eigenschaften kann ebenfalls durch Definition über den transitiven Abschluss bezüglich der Abhängigkeitsgraphen definiert werden. Jedoch ist die hier gewählte Formulierung geeignet mittels Fixpunktiteration über die ersten beiden Eigenschaften die Berechnungsvorschrift eines zu ersetzenden Attributs herzuleiten.

Satz 4.1. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G. Die Änderungsmengen

- A_-,
- A_+,
- R_- und
- R_+

erzeugen eine zerlegbare resultierende Attributgrammatik $AG' = (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$, wenn folgende Bedingungen bzgl. der Änderungsmengen gelten:

1.

$$\begin{aligned}
&\{b\colon \exists \langle a, \cdots, b\rangle \in DG_p,\ a \in AD_p\ \wedge\ a \in \mathsf{A}_-\ \wedge\ \neg(\exists r \in \mathsf{R}_+ : r = a \leftarrow f(\cdots, b, \cdots))\}\\
&\cup \{a'\colon \exists r \in \mathsf{R}_-,\ r = a' \leftarrow f(\cdots)\ \wedge\ \neg(\exists r \in \mathsf{R}_+, r = a' \leftarrow f(\cdots))\}\\
&\subseteq \mathsf{A}_- \subseteq A\\
&\text{und}\\
&\{r\colon r \in R\ \wedge\ \exists a \in \mathsf{A}_-,\ r = a \leftarrow f(\cdots)\}\\
&\cup \{r\colon r \in R, r = a \leftarrow f(\cdots) \wedge (\exists r' = a \leftarrow f(\cdots) \in \mathsf{R}_- \wedge \forall r'' \in \mathsf{R}_+ : r'' \neq a \leftarrow f(\cdots))\}\\
&\cup \{r\colon r \in R, r = a \leftarrow f(\cdots, b, \cdots) : b \in \mathsf{A}_- \wedge \neg(\exists r' \in \mathsf{R}_+, r' = a \leftarrow f(\cdots, b, \cdots))\}\\
&\subseteq \mathsf{R}_- \subseteq R
\end{aligned}$$

2.

$$\begin{aligned}
\mathsf{R}_{p,+,a,s,X} = \{&r\colon r \in \mathsf{R}_{p,+}, r = a \leftarrow f(\cdots), a \in \mathsf{A}_+, a \in A'(X), a \text{ synthetisiert},\\
&p = X ::= u \in P, u \in \Sigma^*\}\\
\mathsf{R}_{+,X,a,s} = &\biguplus_{p \in P, p = X\ ::=\ u, u \in \Sigma^*} \mathsf{R}_{p,+,a,s,X}
\end{aligned}$$

und $\left|\mathsf{R}_{p,+,a,s,X}\right| = 1$ und $\left|\mathsf{R}_{+,X,a,s}\right| = \left|\{p\colon p \in P, p = X ::= u, u \in \Sigma^*\}\right|$. Sowie

$$\{a\colon r = a \leftarrow f(\cdots), r \in \mathsf{R}_{p,+,a,s,X}\} \subseteq \mathsf{A}_+$$

ist.

3.

$$\begin{aligned}
\mathsf{R}_{p,+,a,i,X} = \{&r\colon r \in \mathsf{R}_{p,+}, r = X_i.a \leftarrow f(\cdots), a \in \mathsf{A}_+, a \in A'(X), a \text{ ererbt},\\
&p = Y ::= u\ X\ v, X \in \Sigma, Y \in N, u, v \in \Sigma^*, 1 \leq i \leq |X|_p, r \neq Y.a \leftarrow f(\cdots)\}\\
\mathsf{R}_{+,X,a,i} = &\biguplus_{p \in P, p = Y\ ::=\ u\ X\ v, u, v \in \Sigma^*, Y \in N} \mathsf{R}_{p,+,a,i,X}
\end{aligned}$$

wobei $\left|\mathsf{R}_{p,+,a,i,X}\right| = |X|_p$, $r_i \neq r_j, r_i, r_j \in \mathsf{R}_{p,+,a,i,X}$, $i \neq j, 1 \leq i \leq \left|\mathsf{R}_{p,+,a,i,X}\right|$, $1 \leq j \leq \left|\mathsf{R}_{p,+,a,i,X}\right|$ und $\left|\mathsf{R}_{+,X,a,i}\right| = \sum_{p \in P, p = \mathsf{Y}\ ::=\ u\ \mathsf{X}\ v} |\mathsf{X}|_p$ für $\mathsf{Y} \in N, u, v \in \Sigma^*$ Sowie

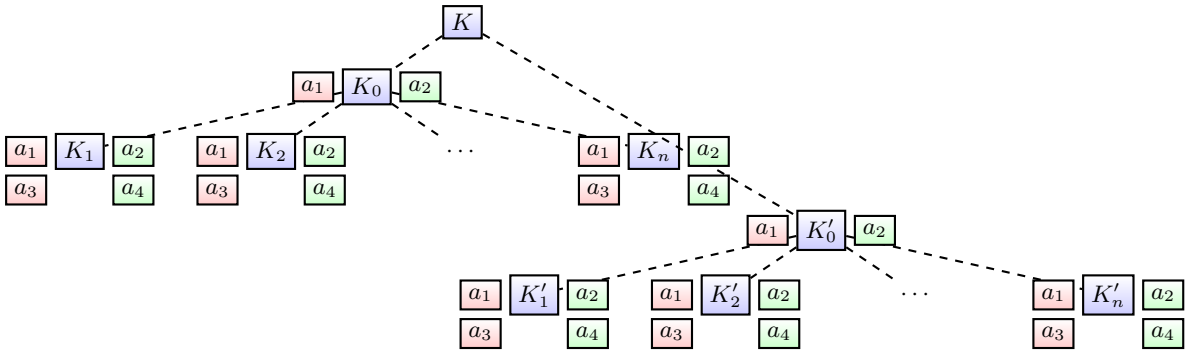
$$\{a\colon r = a \leftarrow f(\cdots), r \in \mathsf{R}_{p,+,a,i,X}\} \subseteq \mathsf{A}_+$$

ist.

4. Für alle $r \in \mathsf{R}_+$ gilt r ist vereinbar hinzufügbar zu R'' mit $R'' = R \setminus \mathsf{R}_-$
5. Für alle Produktionen $p \in P$, alle Symbole $X \in \Sigma$, alle Attribute $a \in \mathsf{A}_+$ gilt: ist $r = a \leftarrow f(\cdots) \in \mathsf{R}_{p,+,a,s,X}$, dann ist $r \notin \mathsf{R}_{q,+,a,i,X}$ für $q \in P$; analog ist $r = a \leftarrow f(\cdots) \in \mathsf{R}_{p,+,a,i,X}$ dann $r \notin \mathsf{R}_{q,+,a,s,X}$ für ein $q \in P$.
6. $\mathsf{A}_+ \subseteq \{a\colon r = a \leftarrow f(\cdots) \in \mathsf{R}_+\}$ und

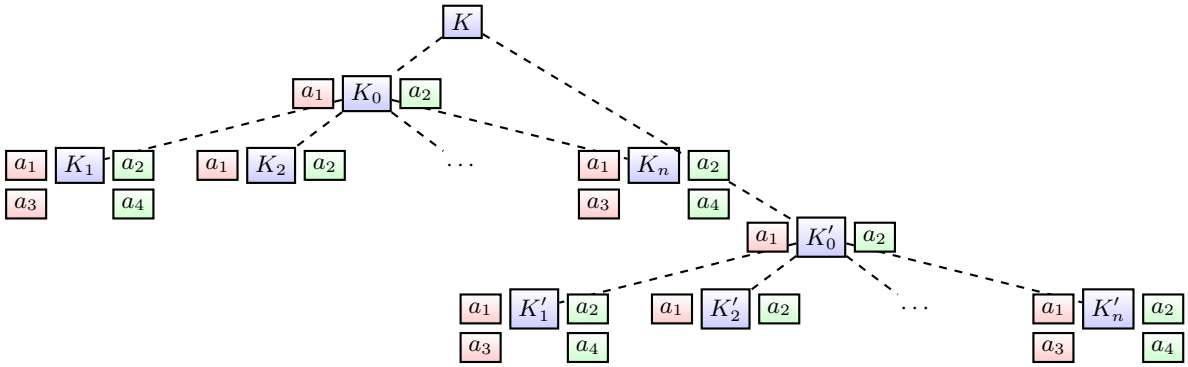
$$\mathsf{R}_+ = \biguplus_{p \in P, a \in \mathsf{A}_+, X \in \Sigma} (\mathsf{R}_{p,+,a,i,X} \uplus \mathsf{R}_{p,+,a,s,X})$$

Änderungsmengen, die die Eigenschaften von Satz 4.1 einhalten, können aus initialen Änderungsmengen mittels Fixpunktiteration oder Arbeitslisten-Algorithmus bestimmt werden. Eigenschaft 1 von Satz 4.1 stellt sicher, dass, wenn ein Attribut entfernt werden soll und keine neue definierende Attributierungsregel dafür hinzugefügt wird, die definierende Regel auch entfernt wird. Darüber hinaus ist auch sichergestellt, dass *alle* Attribute, die von diesem Attribut abhängig sind, ebenfalls entfernt werden. Für diese Attribute gelten dann entsprechende Aussagen ebenso. Für synthetisierte Attribute, die neu definiert werden, stellt Eigenschaft 2 sicher, dass für synthetisierte Attribute eines Symbols genau eine



$[K_0.a_1, K_1.a_1, K_1.a_2, K_2.a_1.K_2.a_2, \cdots, K_n.a_1, K_n.a_2, K_0.a_2, K'_0.a_1, K'_1.a_1, K'_1.a_2, \cdots,$
$K'_n.a_1, K'_n.a_2, K_1.a_3, K_1.a_4, K_2.a_3, K_2.a_4, \cdots, K_n.a_3, K_n.a_4, K'_1.a_3, K'_1.a_4, K'_2.a_3, K'_2.a_4, \cdots, K'_n.a_3, K'_n.a_4]$

a) Abstrakter Syntaxbaum vor Entfernung der Attribute (Ausschnitt)



$[K_0.a_1, K_1.a_1, K_1.a_2, K_2.a_1.K_2.a_2, \cdots, K_n.a_1, K_n.a_2, K_0.a_2, K'_0.a_1, K'_1.a_1, K'_1.a_2, \cdots,$
$K'_n.a_1, K'_n.a_2, K_1.a_3, K_1.a_4, \cdots, K_n.a_3, K_n.a_4, K'_1.a_3, K'_1.a_4, \cdots, K'_n.a_3, K'_n.a_4]$

b) Abstrakter Syntaxbaum nach Entfernung der Attribute $\mathsf{X}.a_3, \mathsf{X}.a_4$ in Knoten K_2 und K'_2

In a) ist der ursprüngliche abstrakte Syntaxbaum dargestellt, in b) der gleiche abstrakte Syntaxbaum jedoch nach Entfernen der Attribute $\mathsf{X}.a_3$ *und* $\mathsf{X}.a_4$*; dabei ist* $Typ(K_i) = Typ(K'_i)$ *für* $0 \leq i \leq n, n \in \mathbb{N}$ *und* $Typ(K_2) = \mathsf{X}$ *sowie* $\mathsf{X}.a_3 \in \mathsf{A}_{-}$ $\mathsf{X}.a_4$ *hängt von* $\mathsf{X}.a_3$ *ab und muss daher nach Eigenschaft 1 von Satz 4.1 ebenfalls entfernt werden. Die dargestellte berechenbare Reihenfolge bezieht sich auf den dargestellten Ausschnitt des abstrakten Syntaxbaums. Auf eine Darstellung von Abhängigkeiten wurde verzichtet, bzgl. der Attribute werden zuerst die Attribute* a_1 *und* a_2 *ausgewertet, dann im nächsten Besuch eines Knotens die Attribute* a_3 *und* a_4*. Für* $Typ(K_0) = \mathsf{Y}$ *sind* $a_3, a_4 \notin A_{\mathsf{Y}}$*. Aus Gründen der Übersichtlichkeit wurde auf Einfügen von Abhängigkeiten der Attributgrammatik verzichtet. Jedoch sind* a_1 *und* a_2 *intrinsische Attribute und Attribut* a_3 *hängt von* a_2 *ab,* a_4 *von* a_3*.*

Beispiel 4.4 – Beispiel zweier abstrakter berechenbarer Reihenfolgen für eine abstrakten Syntaxbaum vor und nach Entfernen zweier Attribute folgend den Eigenschaften von Satz 4.1

definierende Attributierungsregel je Produktion mit diesem Symbol auf linker Seite existiert. Analoges für ererbte Attribute und jedes Vorkommen diesen Symbols in der Produktion. Die unmittelbar mögliche Inkonsistenz bei ererbten – das Symbol kommt auf beiden Seiten der Produktion vor – wird ebenfalls hier bereits ausgeschlossen. Entsprechende Inkonsistenz wird ebenfalls durch Eigenschaft 5 ausgeschlossen – ist ein Attribut für ein Symbol ererbt attributiert in einer Produktion, dann gibt es keine Attributierungsregel für eine Produktion, bei der dieses Attribut für dieses Symbol synthetisiert ist. Letztendlich müssen alle hinzugefügten Attributierungsregeln vereinbar hinzufügbar sein. Implizit bedeuten Eigenschaft 1 und Eigenschaft 4 sowie die Eigenschaften 2, 3, dass bei der Ersetzung (Änderung) von Attributierungsregeln alle ursprünglich vorhandenen Regeln entfernt werden müssen: die zu ändernde Regel ist in R_{-}, damit die Anzahl der hinzugefügten Regeln bzgl. dem Attribut und dem Symbol mit Eigenschaft 3 bzw. Eigenschaft 2 gültig ist, müssen andere definierende Regeln ebenfalls entfernt sein. Aufgrund der Definition

vereinbar hinzufügbarer Attributierungsregeln (siehe Def. 4.4) ist sichergestellt, dass selbst bei Änderung einer Attributierungsregel nur zu existierenden Attributen Abhängigkeiten erlaubt sind. Durch den Ausschluss von Inkonsistenz und Unvollständigkeit sowie der Eigenschaft, dass alle Attributierungsregeln nur vereinbar hinzufügbar sein dürfen, ist sichergestellt, dass keine Zyklen im lokalen Abhängigkeitsgraphen eingeführt werden können. Im Folgenden wird gezeigt, dass, wenn alle Eigenschaften dieses Satzes eingehalten werden, die resultierende Attributgrammatik zerlegbar ist.

In Satz 4.1 ist in Eigenschaft 3 der Index am Symbol X zur Unterscheidung der einzelnen X in der Produktion zu verstehen. Ist $|X|_p = 1$, so ist dieser Index nicht notwendig. Für jedes Symbolvorkommen und jedes Attribut ist für synthetisierte und ererbte Attribute exakt eine Attributierungsregel notwendig. Im Folgenden wird ggf. darauf verzichtet die Attributierungsregeln für ererbte Attribute für jedes Symbolvorkommen zu erwähnen. Die Anzahl der ererbten Attributierungsregeln eines Attributs a in einer Produktion p zu einem Symbol X ist genau der Anzahl der Symbolvorkommen auf der rechten Seite der Produktion: $|X|_p$. Im Allgemeinen wird bei der Darstellung der Attributierungsregeln darauf verzichtet das Symbol mit aufzuführen.

Für den Beweis von Satz 4.1 werden in den folgenden Lemmata die geforderten Eigenschaften und die der Zerlegbarkeit zugrunde liegenden Eigenschaften genauer untersucht.

Lemma 4.3. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und Änderungsmengen A_-, A_+, R_-, R_+ die die Bedingungen aus Satz 4.1 einhalten. Die resultierende Attributgrammatik $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist *vollständig*.

Beweis. Sei AG' nicht vollständig, dann ist zu zeigen, dass AG nicht zerlegbar war oder eine der Bedingungen von Satz 4.1 nicht eingehalten wurde. Ist AG' nicht vollständig, so heißt dies, es existiert ein abstrakter Syntaxbaum AST und ein Attribut a, sodass für einen Knoten k mit $Typ(k) = \mathsf{X}$, $X \in \Sigma$ die Regel $r = \mathsf{X}.a \leftarrow e$ nicht in R' ist ($r \notin R'$).

Folgende Fälle werden betrachtet:

1. es existiert $r_d = \mathsf{X}.a \leftarrow e$, $r_d \in \mathsf{R}_-$ und für $r_a = \mathsf{X}.a \leftarrow e$ ist $r_a \notin \mathsf{R}_+$;
2. es gibt kein $r_d = \mathsf{X}.a \leftarrow e$, also $r_d \notin \mathsf{R}_-$, jedoch ein $r_a = \mathsf{X}.a \leftarrow e$, $r_a \in \mathsf{R}_+$;
3. es existieren r_a und r_d mit $r_a = \mathsf{X}.a \leftarrow e$, $r_d = \mathsf{X}.a \leftarrow e$ und $r_a \in \mathsf{R}_+$ und $r_d \in \mathsf{R}_-$ sowie
4. es existieren keine r_a, r_d mit $r_a = \mathsf{X}.a \leftarrow e$, $r_d = \mathsf{X}.a \leftarrow e$ somit also $r_a \notin \mathsf{R}_+$ und $r_d \notin \mathsf{R}_-$.

Fall 1: Ist $\mathsf{X}.a \in \mathsf{A}_-$, dann verletzt R_- Eigenschaft 1, da dann für jede solche Regel, insbesondere auch r, diese in R_- wäre. Ist $\mathsf{X}.a \in \mathsf{A}_+$, dann verletzt entweder A_+ Eigenschaft 6, da $r_a \notin \mathsf{R}_+$ oder R_+ verletzt eine der Eigenschaften 2 oder 3, da die Anzahl solcher Regeln bzgl. dem Symbol X entsprechend $\mathsf{R}_{+,X,a,i}$ oder $\mathsf{R}_{+,X,a,s}$ ist. Da AG' unvollständig existiert ein $p \in P$ für dass $\mathsf{R}_{p,+,X,a,s} = \emptyset$ oder $\left|\mathsf{R}_{p,+,X,a,i}\right| < |X|_p$.

Fall 2: Analog Fall 1 ist $\mathsf{X}.a \in \mathsf{A}_-$, dann hält R_- nicht Eigenschaft 1 ein. Ist $\mathsf{X}.a \in \mathsf{A}_+$ dann verletzt R_+ eine der Eigenschaften 2 oder 3 analog Fall 1.

Fall 3: Ist $\mathsf{X}.a \in \mathsf{A}_-$ hält R_- analog Fall 1 und Fall 2 nicht Eigenschaft 1 ein. Analog ist $\mathsf{X}.a \in \mathsf{A}_+$, dann verletzt R_+ eine der Eigenschaften 2 oder 3.

Fall 4: Ist $\mathsf{X}.a \in \mathsf{A}_+$ oder $\mathsf{X}.a \in \mathsf{A}_-$ analog bisheriger Fälle: A_- oder A_+ oder R_- oder R_+ verletzen die Eigenschaften wie gefordert. Existiert ein $r' = b \leftarrow f(\cdots, \mathsf{X}.a, \cdots) \in \mathsf{R}_+$ so verletzt dies die Eigenschaft 4 (für alle b_i – hier $\mathsf{X}.a$ – existiert eine definierende Attributierungsregel in R). Trifft nichts davon zu, dann ist bereits AG nicht zerlegbar.

Somit halten die Änderungsmengen mindestens eine der Eigenschaften von Satz 4.1 nicht ein oder AG ist nicht zerlegbar.

□

Lemma 4.3 zeigt, dass, wenn die Eigenschaften aus Satz 4.1 eingehalten werden, dass keine unvollständigen Attributierungsregeln über bleiben oder komplett attributiert wurde.

Lemma 4.4. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und einer Menge von Änderungsmengen A_-, A_+, R_-, R_+ die die Bedingungen aus Satz 4.1 einhalten. Die resultierende Attributgrammatik $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist *konsistent*.

Beweis. Angenommen die resultierende Attributgrammatik AG' ist nicht konsistent, d. h. es existiert ein abstrakter Syntaxbaum AST in dem es für ein Attribut $a \in A'$ in einem Knoten k mehr als eine definierende Regel für $\mathsf{X}.a$ gibt, wobei $Typ(k) = \mathsf{X}$. Somit existiert eine Produktion $p \in P$ sodass Regeln $r_i = \mathsf{X}.a \leftarrow e_i$, mit $1 \leq i \leq n$, $n \in \mathbb{N}, n > 1$, für beliebige Ausdrücke e_i existieren, sodass $r_i \in R'_p$. Zu zeigen ist dann, dass AG nicht zerlegbar war oder die Änderungsmengen die Eigenschaften von Satz 4.1 nicht einhalten. Sei $r = a \leftarrow f(\cdots)$ die definierende Regel mit $r \in R_p$ für AG, so existieren folgende Fälle:

1. es existiert kein $r \in \mathsf{R}_+$ mit $r = \mathsf{X}.a \leftarrow e$ sowie
2. es existiert $r \in \mathsf{R}_+$ mit $r = \mathsf{X}.a \leftarrow e$.

In Fall 1 sei ein $r_i \in \mathsf{R}_-$ und $\mathsf{X}.a \in \mathsf{A}_-$, dann verletzt R_- Eigenschaft 1 von Satz 4.1; ist $r_i \notin \mathsf{R}_-$ für beliebige i, dann war AG bereits nicht zerlegbar.

Fall 2 verletzt Eigenschaft 2 bzw. Eigenschaft 3 von Satz 4.1. Ist a synthetisiert bedeutet die Existenz einer Produktion p mit $r_i = \mathsf{X}.a \leftarrow e_i \in R'_p$ mit $1 \leq i \leq n$, $n \in \mathbb{N}$, $n >$ einen Widerspruch zu $\left|\mathsf{R}_{p,+,a,s,X}\right| = 1$. Ist a ererbt, steht dies im Widerspruch dazu, dass $\left|\mathsf{R}_{p,+,a,i,X}\right| = |X|_p$ *und* $r_i \neq r_j$ für $i \neq j$, $i, j = 1, \cdots, \mathsf{R}_{p,+,a,i,X}$ ist.

□

Folgende zwei Lemmata vereinfachen den darauf folgenden Beweis zu Satz 4.1. Ausgehend von einer zerlegbaren Attributgrammatik wird gezeigt, dass das Entfernen und Hinzufügen von Attributen erlaubt eine zulässige Zerlegung zu bestimmen.

Lemma 4.5. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und einer Menge von Änderungsmengen A_-, A_+, R_-, R_+ die die Bedingungen aus Satz 4.1 einhalten. Darüber hinaus sei $\mathsf{A}_+ = \emptyset$ und $\mathsf{R}_+ = \emptyset$. Für die resultierende Attributgrammatik $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-)$ und $R' = (R \setminus \mathsf{R}_-)$ existiert eine zulässige Zerlegung, sodass für jeden abstrakten Syntaxbaum AST, die Attribute des Knotens K mit $Typ(K) = \mathsf{X}$ in der Reihenfolge $A_\mathsf{X}(1) \cdots, A_\mathsf{X}(m_\mathsf{X})$ berechnet werden können.

Beweis. Da AG zerlegbar, existiert eine zulässige Zerlegung für AG, sodass für jedes Symbol $\mathsf{X} \in \Sigma$ gilt $A_\mathsf{X} = A_\mathsf{X}(1) \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$ wobei $A_\mathsf{X}(i) \subseteq AS_\mathsf{X}$ für $i = m_\mathsf{X}, m_\mathsf{X} - 2, \cdots$ und $A_\mathsf{X}(i) \subseteq AI_\mathsf{X}$ für $i = m_\mathsf{X} - 1, m_\mathsf{X} - 3, \cdots$. Zu zeigen ist, dass für AG' ebenfalls eine zulässige Zerlegung existiert.

Für $A'_\mathsf{X}(i) = A_\mathsf{X}(i) \setminus \mathsf{A}_-$, da $A'_\mathsf{X}(i) \subseteq A_\mathsf{X}(i)$ für $1 \leq i \leq m_\mathsf{X}$ ist die Eigenschaft erfüllt.

Für jeden AST ist dann die Reihenfolge $A'_\mathsf{X}(1), \cdots, A'_\mathsf{X}(m_\mathsf{X})$ für Knoten K mit $Typ(K) = \mathsf{X}$. □

Lemma 4.5 zeigt, dass das Entfernen von Attributen weiterhin eine zulässige Zerlegung ergibt. Darüber hinaus zeigt folgendes Lemma, dass das Hinzufügen von Attributen ebenfalls eine zulässige Zerlegung ergibt. Im folgenden Lemma wird zuerst nur gezeigt, dass solch eine zulässige Zerlegung existiert und erst

im darauf folgenden Lemma gezeigt, dass diese zulässige Zerlegung auch für jeden abstrakten Syntaxbaum berechenbar ist.

Lemma 4.6. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und einer Menge von Änderungsmengen A_-, A_+, R_-, R_+ die die Bedingungen aus Satz 4.1 einhalten. Darüber hinaus sei $\mathsf{A}_- = \emptyset$ und $\mathsf{R}_- = \emptyset$. Für die resultierende Attributgrammatik $AG' \triangleq (G, A', R', B)$ mit $A' = A \cup \mathsf{A}_+$ und $R' = R \cup \mathsf{R}_+$ existiert eine zulässige Zerlegung, d. h. für jedes Symbol $\mathsf{X} \in \Sigma$ ist $A_\mathsf{X}(1) \uplus \cdots A_\mathsf{X}(m_\mathsf{X})$ die zulässige Zerlegung.

Beweis. Mittels Induktion über die Anzahl der hinzugefügten Attribute:

Sei $\mathsf{A}_+ = \mathsf{A}_+^{(1)} \uplus \cdots \uplus \mathsf{A}_+^{(n)}$ mit $\left|\mathsf{A}_+^{(j)}\right| = 1$, für alle $1 \leq j \leq n$ und $n \in \mathbb{N}$ und somit $\mathsf{A}_+^{(j)} = \{a_j\}$.

Induktionsanfang: Sei $n = 1$, d. h. $\mathsf{A}_+ = \mathsf{A}_+^{(1)} = \{a_1\}$. Sei $a_1 \in \mathsf{A}_{\mathsf{X},+}$ des Symbols $\mathsf{X} \in \Sigma$. Für alle Symbole $\mathsf{Y} \in \Sigma$, $\mathsf{X} \neq \mathsf{Y}$ ist $A'_\mathsf{Y}(k) = A_\mathsf{Y}(k)$, mit $1 \leq k \leq m_\mathsf{Y}$; für X sei $m'_\mathsf{X} = m_\mathsf{X} + 2$, dann ist die Zerlegung $A'_\mathsf{X}(i)$ wie folgt definiert:

$$A'_\mathsf{X}(i) = \begin{cases} A_\mathsf{X}(i) & \text{falls } i \leq m_\mathsf{X} \\ \{a_1\} & \text{falls } i = m_\mathsf{X} + 1 \text{ und } a_1 \in AI(\mathsf{X}) \\ \{a_1\} & \text{falls } i = m_\mathsf{X} + 2 \text{ und } a_1 \in AS(\mathsf{X}) \\ \emptyset & \text{sonst} \end{cases}$$

Diese Zerlegung ist zulässig, da die bisherige Zerlegung beibehalten wird und das neue Attribut in die entsprechende, neue, Zerlegungsmenge hinzugefügt wird.

Induktionshypothese: Für beliebiges $n > 1$ und $\mathsf{A}_+ = \mathsf{A}_+^{(1)} \uplus \cdots \uplus \mathsf{A}_+^{(n)}$ mit $\left|\mathsf{A}_+^{(j)}\right| = \{a_j\}$, für alle $1 \leq j \leq n$ existiert eine zulässige Zerlegung für AG'.

Induktionsschritt: Zu zeigen ist, für $\mathsf{A}_+ = \mathsf{A}_+^{(1)} \uplus \cdots \uplus \mathsf{A}_+^{(n+1)}$ existiert eine zulässige Zerlegung. A_+ ist dann $\mathsf{A}_+^{(1)} \uplus \cdots \uplus \mathsf{A}_+^{(n)} \uplus \mathsf{A}_+^{(n+1)}$ wobei $\mathsf{A}_+^{(i)} = \{a_i\}$ für $1 \leq i \leq n+1$.

Sei AG'' die resultierende Attributgrammatik nach der Induktionshypothese, d. h. $AG'' = (G, A'', R'', B)$ wobei $A'' = (A \setminus \mathsf{A}_-) \cup (\mathsf{A}_+^{(1)} \uplus \cdots \mathsf{A}_+^{(n)})$. Für jedes Symbol $\mathsf{X} \in \Sigma$ existiert dann eine zulässige Zerlegung. Sei dies für AG'' dann $A''_\mathsf{X}(1) \uplus \cdots \uplus A''_\mathsf{X}(m''_\mathsf{X})$ nach Induktionshypothese.

Für die resultierende Attributgrammatik $AG' = (G, A', R', B)$ mit $A' = A'' \cup \mathsf{A}_+^{(n+1)}$ ist dann die Zerlegung für das Symbol $\mathsf{X} \in \Sigma$ wie folgt definiert:

$$A'_\mathsf{X}(i) = \begin{cases} A''_\mathsf{X}(i) & \text{falls } i \leq m''_\mathsf{X} \\ \{a_{n+1}\} & \text{falls } i = m''_\mathsf{X} + 1 \text{ und } a_{n+1} \in AI(\mathsf{X}) \\ \{a_{n+1}\} & \text{falls } i = m''_\mathsf{X} + 2 \text{ und } a_{n+1} \in AS(\mathsf{X}) \\ \emptyset & \text{sonst} \end{cases}$$

für $a_{n+1} \in \mathsf{A}_{\mathsf{X},+}$. Für Symbole $\mathsf{Y} \in \Sigma$, $Y \neq X$ ist $A'_\mathsf{Y}(i) = A''_\mathsf{Y}(i)$ für $1 \leq i \leq m''_\mathsf{Y}$.

Diese Zerlegung ist zulässig, da die nach Induktionshypothese zulässige Zerlegung beibehalten wird und das neue Attribut entsprechend der Klasse des neuen Attributs a_{n+1} in $A'_\mathsf{X}(m''_\mathsf{X} + 1)$ oder $A'_\mathsf{X}(m''_\mathsf{X} + 2)$ eingeordnet wird. Da $A'_\mathsf{X}(m''_\mathsf{X} + 1) \subseteq AI(\mathsf{X})$ und $A'_\mathsf{X}(m''_\mathsf{X} + 2) \subseteq AS(\mathsf{X})$ ist die Zerlegung zulässig nach Definition 3.12.

□

Die Lemmata 4.5 und 4.6 zeigen, wie für diese iterative Konstruktion einer resultierenden Attributgrammatik zulässige Zerlegungen bestimmt werden können. Obwohl es „bessere“ Zerlegungen gibt, ist für diese Arbeit nur entscheidend, dass solche Zerlegungen für eine resultierende Attributgrammatik existieren.

Ein weiteres, wesentliches Lemma um Satz 4.1 beweisen zu können, ist folgendes Lemma, Lemma 4.7:

Lemma 4.7. Sei $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und Änderungsmengen A_-, A_+, R_- und R_+, die die Eigenschaften nach Satz 4.1 einhalten sowie $\mathsf{A}_- = \emptyset$ und $\mathsf{R}_- = \emptyset$. Sei $AG' = (G, A', R', B)$ mit $A' = A \cup \mathsf{A}_+$ und $R' = R \cup \mathsf{R}_+$, für jeden abstrakten Syntaxbaum AST und jeden Knoten k dieses Baumes mit $Typ(k) = \mathsf{X}$, $\mathsf{X} \in \Sigma$ ist ist die zulässige Zerlegung nach Lemma 4.6 in der Reihenfolge $A'_\mathsf{X}(1), \cdots, A'_\mathsf{X}(m''_\mathsf{X}), A'_\mathsf{X}(m''_\mathsf{X}+1), A'_\mathsf{X}(m''_\mathsf{X}+2)$ berechenbar; dabei ist $|\mathsf{A}_+| = n$ die Anzahl der hinzugefügten Attribute und daher nach Lemma 4.6 $m''_\mathsf{X} = m_\mathsf{X} + 2n$

Beweis. Angenommen für einen abstrakten Syntaxbaum AST' sei die Reihenfolge $A'_\mathsf{X}(1), \cdots, A'_\mathsf{X}(m''_\mathsf{X}), A'_\mathsf{X}(m''_\mathsf{X}+1), A'_\mathsf{X}(m''_\mathsf{X}+2)$ nicht berechenbar, d. h. es existiert eine Menge $A'_\mathsf{X}(i)$, $1 \leq i \leq m_\mathsf{X} + 2$ mit $a_k \in A'_\mathsf{X}(i)$ und Regeln $r = a_k \leftarrow f(\cdots, a_l, \cdots) \in R'$ wobei $a_l \in A'_\mathsf{X}(j)$ mit $j > i$.

Es existieren drei Fälle:

1. $i \leq m_\mathsf{X}, j < m_\mathsf{X}$
2. $i \leq m_\mathsf{X}, j > m_\mathsf{X}$
3. $i > m_\mathsf{X}$, $j > i$

Fall 1 steht im direktem Widerspruch dazu, dass AG zerlegbar ist – die betroffenen Attribute und Zerlegungsmengen existieren bereits in AG.

Fall 2 ist ebenfalls im Widerspruch dazu, dass AG zerlegbar ist, da das existierende Attribut a_k von einem noch hinzuzufügenden Attribut a_l abhängt.

Fall 3 beschreibt den Fall, dass zwei Attribute hinzugefügt werden und nicht bzgl. der Abhängigkeiten in die Zerlegungsmengen eingefügt wurden. Dies steht im Widerspruch zu Eigenschaft 4 von Satz 4.1, da nach Definition vereinbar hinzufügbarer Attributierungsregeln die definierten Attribute nur von Attributen abhängig sein dürfen, die bereits in AG existieren. □

Mit den bisher bewiesenen Lemmata kann nun Satz 4.1 gezeigt werden.

Beweis zu Satz 4.1. Zu zeigen ist: Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G mit den Änderungsmengen A_-, A_+, R_-, R_+ die die Eigenschaften nach Satz 4.1 einhalten, dann ist die resultierende Attributgrammatik $AG' = (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ *zerlegbar*.

Folgende Eigenschaften für AG' sind nachzuweisen:

1. AG' ist lokal azyklisch und
2. für AG' existiert eine zulässige Zerlegung, sodass für jeden abstrakten Syntaxbaum eine berechenbare Reihenfolge existiert, für die die Attribute in der Reihenfolge der Zerlegung berechnet werden können.

Zu Eigenschaft 1: Nach Lemma 4.2 ist AG' lokal azyklisch oder inkonsistent. Wäre AG' inkonsistent, so wäre dies im Widerspruch dazu, dass AG zerlegbar und die Änderungsmengen Eigenschaften 1 bis 3 und Eigenschaft 5 einhalten. AG' ist außerdem konsistent nach Lemma 4.4.

Zu Eigenschaft 2: Nach Lemma 4.5 und 4.6 existiert eine zulässige Zerlegung. Da Nach Lemmata 4.3 und 4.4 AG' ebenfalls konsistent und vollständig ist und AG' lokal azyklisch (aus Lemma 4.2 und Lemma 4.4) existiert für jeden abstrakten Syntaxbaum eine berechenbare Reihenfolge. Für jeden Knoten k mit

```
1   rule Decls ::= Decls Decl
2   attr Decls2.declsIn ←Decls1.declsIn
3        Decl.declsIn ← Decls2.declsOut
4        Decls1.declsOut ←Decl.declsOut
5        Decls2.env ←Decls1.env
6        Decl.env ← Decls1.env
7
8   rule Decl ::= RqDecl
9   attr RqDecl.declsIn ← Decl.declsIn
10       Decl.declsOut ← RqDecl.declsOut
11       RqDecl.env ← Decl.env
12
13  rule RqDecl ::= RqDefId RqReferences
14  attr RqDefId.declsIn ← RqDecl.declsIn
15       RqDecl.declsOut ← RqDefId.declsOut
16       RqReferences.env ← RqDecl.env
17       RqDefId.env ← RqDecl.env
18
19  rule Decls ::= ε
20  attr Decls.declsOut ← Decls.declsIn
21
22  rule RqReferences ::= RqReferences RqReference
23  attr RqReferences2.env ←RqReferences1.env
24       RqReference.env ← RqReferences1.env
25
26  -- ....
```

a) Ausschnitt des ursprünglichen Beispiels aus Abschnitt 1.1

$$\mathsf{A}_{\_} = \{\texttt{RqDefId.env}\}$$

b) Initial zu entfernendes Attribut.

$$\begin{aligned}\mathsf{A}_{\_} = \{&\texttt{RqDefId.env}, \texttt{RqDecl.env}\\ &\texttt{RqReferences.env}, \texttt{Decl.env}\\ &\texttt{Decls.env}, \texttt{RqReference.env}\\ &\texttt{Decls.declsOut}, \cdots, \texttt{RqDefId.declsIn}\\ &\texttt{RqDefId.declsOut}\}\end{aligned}$$

c) Ergebnis der zu entfernenden Attribute nach Anwendung von Satz 4.1.

$$[] \textit{(leere Liste bzw. Berechnungsreihenfolge)}$$

d) Berechenbare Reihenfolge der resultierenden Attributgrammatik nach Anwendung der durch Satz 4.1 erzeugten Menge zu entfernender Attribute und Attributierungsregeln für jeden abstrakten Syntaxbaum (leere Liste).

Beispiel 4.5 – Berechenbare Reihenfolge nach Entfernung des Attributs `RqDefId.env` und Bestimmung der zu entfernenden Attribute nach Satz 4.1 zum Erhalt der Zerlegbarkeit. Nicht aufgeführt ist $\mathsf{R}_{\_}$.

$Typ(k) = \mathsf{X}$ für alle $X \in \Sigma$ ist die berechenbare Reihenfolge über die zulässige Zerlegung nach Lemma 4.5 und Lemma 4.6 gegeben.

Dass die zulässige(n) Zerlegung(en) nach Lemma 4.6 berechenbar sind, folgt aus Lemma 4.7 □

Auf den expliziten Beweis, dass das Ersetzen einer Attributierungsregel in einer zulässigen Zerlegung resultiert, wird verzichtet. Die Ersetzung wäre identisch mit dem schrittweisen Löschen und wieder Hinzufügen mittels zweier Muster. Gelöschte Attributierungsregeln sind dann geändert in den hinzugefügten Attributierungsregeln vorhanden. In der Konstruktion wird das Attribut aus der ursprünglichen Partitionierungsmenge entfernt und in eine entsprechende neue Partitionierungsmenge für $m'_{\mathsf{X}} > m_{\mathsf{X}}$ gepackt. Eben so, als hätte dieses Attribut mit Attributierungsregel nicht vor dem Ändern existiert.

Der Beweis zu Satz 4.1 lässt sich wie folgt zusammenfassen. Entfernen von Attributen und Attributierungsregeln nach Eigenschaft 1 von Satz 4.1 sorgt nur dafür, dass diese Attribute (und Attribute, die von diesen abhängig sind) aus der Zerlegung entfernt wird. Die berechenbare Reihenfolge bzgl. der übrigen Attribute bleibt gleich. Es werden somit nur die Attribute nicht mehr berechnet, die entfernt wurden. Dies folgt indirekt aus Lemma 4.3 und direkt aus Lemma 4.5. Beim Hinzufügen von Attributen werden diese Schrittweise in neue Zerlegungsmengen hinzugefügt (folgt aus dem Beweis zu Lemma 4.6), diese sind dann zulässig. Weiterhin folgt die Azyklizität aus den Anforderungen von Satz 4.1, und den Lemmata 4.2 und 4.4 da sonst bereits kein vereinbares Hinzufügen der Attribute möglich ist. Weiterhin wird gezeigt, dass nicht nur die Zerlegung zulässig ist, sondern auch die Attributgrammatik zerlegbar ist.

In Eigenschaft 1 von Satz 4.1 wird sichergestellt, dass nur dann Attribute und alle damit einhergehenden Regeln entfernt werden, wenn es für dieses Attribut nicht eine neue definierende Regel gibt. Zusammen mit Eigenschaft 4 ist sichergestellt, dass die Ersetzung von Attributierungsregeln keine Zyklen einführt,

Eigenschaft 5 stellt sicher, dass keine Inkonsistenz eingeführt wird. Darüber hinaus impliziert diese Konstellation, dass Attribute, für die eine Ersetzung der Berechnungsvorschrift statt findet in A_- *und* A_+ enthalten sind. In solch einem Fall werden dann, im Gegensatz zu Beispiel 4.4, die übrigen Attributierungsregeln, die vom gelöschten Attribut abhängig sind, nicht entfernt.

Ausgehend von Beispiel 4.4 und dem einleitenden Beispiel 1.2 gibt folgendes Beispiel das ursprünglich entfernte Attribut und die berechenbare Reihenfolge der resultierenden Attributgrammatik an.

Beispiel 4.5 beschreibt somit das Erstellen einer zerlegbaren Attributgrammatik durch Entfernen aller Attribute.

Symbol `X`	$A_X(1)$
`Program`	$\emptyset$
`Stats`	$\emptyset$
`Stat`	$\emptyset$
`VarStat`	{`VarStat.number`}

Tabelle 4.1. – Zerlegung der Attributgrammatik nach Bestimmung einer ursprünglichen Zerlegung und Anwendung von Satz 4.1 auf die Attributgrammatik und die dazugehörigen Änderungsmengen aus Beispiel 4.2.

Bezug nehmend auf Beispiel 4.2 führt die Anwendung von Satz 4.1 zu der in Tabelle 4.1 angegebenen Zerlegung.
Die Eigenschaften 1, 2 und 3 von Satz 4.1 stellen sicher, dass die resultierende Attributgrammatik konsistent und vollständig ist. Die mögliche Inkonsistenz durch mehrfache Attributierung desselben Attributs innerhalb einer Produktion wird durch Einschränkung der Anzahl solcher Regeln r bzgl. einer Produktion p mit den darüber definierten Eigenschaften eingeschränkt.
Nachdem nun gezeigt ist, dass die Änderung von Attributgrammatiken durch Änderungsmengen erfolgen kann ist der nächste Schritt die Beschreibung und Herleitung solcher Mengen aus einer Beschreibung, die unabhängig von einer Attributgrammatik ist. Dabei ist zu beachten, dass die daraus jeweils inferierbaren Mengen *für alle* zerlegbaren Attributgrammatiken die obig genannten Eigenschaften einhalten müssen. Im Folgenden wird gezeigt wie solch eine Beschreibung aufgebaut werden kann.

Zusammenfassend können Attributierungsregeln einer zerlegbaren Attributgrammatik durch zwei grundlegende Mechanismen geändert werden:

- Löschen bestehender Attribute und deren Berechnungsvorschrift(en) sowie
- Hinzufügen neuer, bisher unbekannter, Attribute mit dazugehöriger Berechnungsvorschrift.

Dabei müssen die, in Abschnitt 4.1 vorgestellten, Eigenschaften eingehalten werden, um ausgehend von einer zerlegbaren Attributgrammatik, Mengen der zu entfernenden sowie hinzuzufügenden Attribute und Attributierungsregeln, wiederum eine zerlegbare Attributgrammatik zu erhalten.

Wenn Attribute entfernt werden so muss ebenfalls der transitive Abschluss des zu diesem Attribut inversen Abhängigkeitsgraphen entfernt werden, außer ein identisches Attribut bzw. eine Attributierungsregel für dieses Attribut wird wieder hinzugefügt. In Satz 4.1 werden darüber hinaus weitere wichtige Eigenschaften, die Konsistenz und Vollständigkeit der resultierenden Attributgrammatik sicherstellen, vorgestellt. Ausgeschlossen werden direkt Attributierungsregeln, die zu einem lokalen Zyklus führen.

Der bisher vorgestellte Mechanismus funktioniert auf konkreten, zerlegbaren Attributgrammatiken. Bisher ungelöst ist die Frage, wie diese Mengen aus einer abstrakten Beschreibung hergeleitet werden können und wie diese Beschreibung aussehen kann. Müssen diese Änderungsmengen für jede zerlegbare Attributgrammatik neu angegeben werden, so ist keine Abstraktion geschaffen.

4.2. Herleitung von Mustern aus Beschreibungen

Bisher sind die Änderungsmengen zur Überführung zerlegbarer Attributgrammatiken in eine zerlegbare Attributgrammatik explizit angegeben. In diesem Abschnitt wird gezeigt wie solche Änderungsmengen

aus einer Beschreibung hergeleitet werden können, die (weitestgehend) unabhängig von der konkreten, zu ändernden Attributgrammatik ist, und wie diese Beschreibung zur Herleitung aussehen kann. Ebenso müssen die Berechnungsvorschriften in Abhängigkeit zu den existierenden Attributen einer zerlegbaren Attributgrammatik herleitbar sein. Können nur trivialste Attributierungsregeln – bspw. nur Attributierungsregeln mit einer Konstanten auf der rechten Seite der Regel – hinzugefügt werden, so kann nicht von abstrakteren Mechanismus als Attributgrammatik gesprochen werden.

Ausgehend von der Grundidee Änderungsmengen als Funktionsanwendung auf ursprünglichen Attribut- und Regelmengen herzuleiten werden Terme, Prädikate auf zerlegbaren Attributgrammatiken, Regeln zur Anwendung einer Substitution auf diesen Termen bzgl. einer zerlegbaren Attributgrammatik eingeführt. Mittels einer Semantikdefinition dieser Substitution lässt sich eine abstrakterer Mechanismus als klassische Attributgrammatiken aufbauen, der genau dem Verhalten der Änderungsmengen aus Abschnitt 4.1 entspricht.

Die Bestimmung der Mengen aus Definition 4.3 lässt sich durch die Anwendung von Funktionen auf den jeweiligen Elementen einer Attributgrammatik erreichen. Eine wichtige Eigenschaft solcher Funktionen ist die Definition solcher Funktionen unabhängig von einer konkreten Attributgrammatik und somit auch unabhängig von einer abstrakten Syntax. Die folgende Definition (Def. 4.5) entwickelt die dafür verwendbaren Begriffe.

Definition 4.5. Sei U_A die (Grund-)Menge aller möglichen Attribute, U_R die (Grund-)Menge aller möglichen Attributierungsregeln eines Attributsdann ist $\mathsf{A} \triangleq 2^{U_A}$ die **Familie möglicher Attribute** und $\mathsf{R} \triangleq 2^{U_R}$ die **Familie möglicher Attributierungsregeln**. Sei dann $\mathcal{AG}_G = (G, \mathsf{A}, \mathsf{R}, \mathsf{B})$ dann ist $AG \in \mathcal{AG}_G$ mit $AG = (G, A, R, B)$ so ist $A \in \mathsf{A}$, $R \in \mathsf{R}$ und $B \in \mathsf{B}$.

Sei $\mathcal{M}_G^{(F)} \triangleq (\mathcal{M}_G^{(F_{A,+})}, \mathcal{M}_G^{(F_{A,-})}, \mathcal{M}_G^{(F_{R,+})}, \mathcal{M}_G^{(F_{R,-})})$ wobei

1. $\mathcal{M}_G^{(F_{A,+})} \colon \mathcal{AG}_G \to \mathsf{A}$,
2. $\mathcal{M}_G^{(F_{A,-})} \colon \mathsf{A} \to \mathsf{A}$,
3. $\mathcal{M}_G^{(F_{R,+})} \colon \mathcal{AG}_G \to \mathsf{R}$ und
4. $\mathcal{M}_G^{(F_{R,-})} \colon \mathsf{R} \to \mathsf{R}$

Funktionen zur Bestimmung von Attributen bzw. Attributierungsregeln sind. Dann heißt $\mathcal{M}_G^{(F)}$ **funktionsbasierte Musterdefinition**.

Die funktionsbasierte Musterdefinition formalisiert die Idee Funktionen auf zerlegbaren Attributgrammatiken zu verwenden um die Änderungsmengen einer Attributgrammatik herzuleiten. Formalisiert müssen diese Funktionen auf Potenzmengen der jeweiligen Universen, bspw. alle möglichen Attribute in Attributgrammatiken, definiert werden. Ein Element aus der Grundmenge aller möglichen Attribute ist immer nur *ein* Attribut; Ein Element aus der Grundmenge der Attributierungsregeln ist *eine* mögliche Regel um ein Attribut zu attributieren. Wobei auch für ein Attribut beliebig viele Attributierungsregeln in dieser Grundmenge enthalten sind. Analog zur Grundmenge der Attributierungsregeln ist die Grundmenge der Bedingungen aufgebaut.

Lemma 4.8. Sei $\mathcal{AG}_G \triangleq (G, \mathsf{A}, \mathsf{R}, \mathsf{B})$ die Menge aller zerlegbaren Attributgrammatiken mit abstrakter Syntax G und mit Familien möglicher Attribute A, möglicher Attributierungsregeln R und möglicher Bedingungen B. Sei weiterhin $\mathcal{M}_G^{(F)} \triangleq (\mathcal{M}_G^{(F_{A,+})}, \mathcal{M}_G^{(F_{A,-})}, \mathcal{M}_G^{(F_{R,+})}, \mathcal{M}_G^{(F_{R,-})})$ eine funktionsbasierte Musterdefinition.

$\mathcal{M}_G^{(F)}$ definiert ein Muster genau dann, wenn für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit $AG = (G, A, R, B)$ die resultierende Attributgrammatik $AG' = (G, A', R', B)$ zerlegbar ist, wobei

- $A' = (A \setminus \mathcal{M}_G^{(F_{A,-})}(A)) \cup \mathcal{M}_G^{(F_{A,+})}(AG)$ und

$$\mathcal{M}_G^{(F_{A,+})}(ag) = \begin{cases} \emptyset & \textit{falls } \texttt{VarStat.number} \in A(ag) \\ \{\texttt{VarStat.number}\} & \textit{sonst} \end{cases}$$

$$\mathcal{M}_G^{(F_{A,-})}(attrs) = \emptyset$$

$$\mathcal{M}_G^{(F_{R,-})}(rules) = \begin{cases} \emptyset & \textit{falls } \texttt{Varstat.number} \notin AD_p \\ & \forall p \in P(G(ag)) \\ \{r \in R(ag) \colon r = \texttt{VarStat.number} \leftarrow c, c \in \mathbb{N}, \} & \textit{sonst} \end{cases}$$

$$\begin{aligned} \mathcal{M}_G^{(F_{R,+})}(ag) = \{&\textbf{rule } \texttt{VarStat} ::= \texttt{id id} \\ &\textbf{attr } \texttt{VarStat.number} \leftarrow 10, \\ &\textbf{rule } \texttt{VarStat} ::= \texttt{id number} \\ &\textbf{attr } \texttt{VarStat.number} \leftarrow 100\} \end{aligned}$$

a) Funktionsbasierte Musterdefinition resultierend in Attributgrammatik-abhängiger Musterdefinition

$$\begin{aligned} A_- &= \emptyset \\ A_+ &= \emptyset \\ R_- &= \{\textbf{rule } \texttt{VarStat} ::= \texttt{id id} \\ &\quad \textbf{attr } \texttt{VarStat.number} \leftarrow 0, \\ &\quad \textbf{rule } \texttt{VarStat} ::= \texttt{id number} \\ &\quad \textbf{attr } \texttt{VarStat.number} \leftarrow 1 \quad \} \\ R_+ &= \{\textbf{rule } \texttt{VarStat} ::= \texttt{id id} \\ &\quad \textbf{attr } \texttt{VarStat.number} \leftarrow 10, \\ &\quad \textbf{rule } \texttt{VarStat} ::= \texttt{id number} \\ &\quad \textbf{attr } \texttt{VarStat.number} \leftarrow 100 \quad \} \end{aligned}$$

b) Zwischenergebnisse nach Anwendung der Funktionen auf die resultierende Attributgrammatik aus Beispiel 4.2c

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr VarStat.number = 10
7  rule VarStat ::= id number attr VarStat.number = 100
```

c) Resultat nach Anwendung von Lemma 4.9.

In diesem Beispiel wird in der Funktion $\mathcal{M}_G^{(F_{A,+})}$ geprüft, ob das Attribut `VarStat.number` *bereits vorhanden ist, und wenn nicht, soll dieses der Attributgrammatik hinzugefügt werden. Da die Attributierung für dieses Attribut ggf. geändert werden soll, sollte dieses Attribut bereits in Attributierungsregeln bestimmt werden, so werden durch die Funktion $\mathcal{M}_G^{(F_{R,-})}$ alle definierenden Regeln für dieses Attribut entfernt und mittels $\mathcal{M}_G^{(F_{R,+})}$ diese für die Attributgrammatik wieder hinzugefügt.*

Beispiel 4.6 – Funktionen zur Attributgrammatik-unabhängigen Musterdefinition mit dazugehörigen resultierenden Zwischenmengen zur Darstellung einer Attributgrammatik-abhängigen Musterdefinition für die resultierende Attributgrammatik aus Beispiel 4.2c

- $R' = (R \setminus \mathcal{M}_G^{(F_{R,-})}(R)) \cup \mathcal{M}_G^{(F_{R,+})}(AG)$

ist.

Beweis. Mit $\mathsf{A}_- = \mathcal{M}_G^{(F_{A,-})}(A)$, $\mathsf{A}_+ = \mathcal{M}_G^{(F_{A,+})}(AG)$, $\mathsf{R}_- = \mathcal{M}_G^{(F_{R,-})}(R)$ und $\mathsf{R}_+ = \mathcal{M}_G^{(F_{R,+})}(AG)$ folgt der Beweis aus dem Beweis zu Lemma 4.1. □

Lemma 4.9 verallgemeinert Lemma 4.1 indem die Konstruktion der resultierenden Attributgrammatik indirekt über eine Anwendung von Funktionen auf der ursprünglichen Attributgrammatik erreicht wird. Definition 4.5 erlaubt die Generierung Attributgrammatik-abhängiger Musterdefinitionen für viele Attributgrammatiken. Ausgehend von Funktionen können beliebige (zerlegbare) Attributgrammatiken verändert werden.

Lemma 4.9. Sei $\mathcal{AG}_G \triangleq (G, \mathsf{A}, \mathsf{R}, \mathsf{B})$ die Menge aller Attributgrammatiken mit abstrakter Syntax G. Sei $\mathcal{M}_G^{(F)} \triangleq (\mathcal{M}_G^{(F_{A,+})}, \mathcal{M}_G^{(F_{A,-})} \mathcal{M}_G^{(F_{R,+})}, \mathcal{M}_G^{(F_{R,-})})$ ein Tupel, wobei

- $\mathcal{M}_G^{(F_{A,+})} \colon \mathcal{AG}_G \to \mathsf{A}$,
- $\mathcal{M}_G^{(F_{A,-})} \colon \mathcal{AG}_G \to \mathsf{A}$,
- $\mathcal{M}_G^{(F_{R,+})} \colon \mathcal{AG}_G \to \mathsf{R}$ und
- $\mathcal{M}_G^{(F_{R,-})} \colon \mathcal{AG}_G \to \mathsf{R}$

Funktionen auf den Attributen bzw. Attributierungsregeln solcher Attributgrammatiken sind. $\mathcal{M}_G^{(F)}$ definiert ein Muster genau dann, wenn für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$, $AG \triangleq (G, A, R, B)$ die resultierende Attributgrammatik $AG' \in \mathcal{AG}_G, AG' \triangleq (G', A', R', B')$ zerlegbar ist, wobei

- $G' = G$,
- $A' = (A \setminus \mathcal{M}_G^{(F_{A,-})}(A)) \cup \mathcal{M}_G^{(F_{A,+})}(AG)$
- $R' = (R \setminus \mathcal{M}_G^{(F_{R,-})}(R)) \cup \mathcal{M}_G^{(F_{R,+})}(AG)$
- $B' = B$

Beweis. Für eine zerlegbare Attributgrammatik AG folgt der Beweis unmittelbar aus dem Beweis zu Lemma 4.1. □

Die Anwendung von Lemma 4.9 auf eine konkrete Attributgrammatik führt zu einer Attributgrammatik-abhängigen Musterdefinition nach Definition 4.3.

Definition 4.5 liefert somit eine mögliche Darstellung einer Funktion, die eine zerlegbare Attributgrammatik in eine andere zerlegbare Attributgrammatik überführt bzw. überführen kann. Somit ist Definition 4.5 eine Möglichkeit Muster nach Definition 4.1 zu definieren.

Eine grundsätzliche Eigenschaft, die bereits in Lemma 4.1 vorgestellt wurde, ist, dass die Zwischenmengen nicht immer zu Änderungen führen müssen. In Beispiel 4.6 wird dies anhand der Änderung der Attribute vorgestellt.

Folgende Definition führt Attributwertterme ein, die notwendig sind um eine Berechnungsvorschriften bzw. Attributierungsregel durch Substitution in einer zerlegbaren Attributgrammatik einfügen zu können. Terme und Sorten werden allgemein u. a. in [51] vorgestellt. Anhang C gibt die in dieser Arbeit dafür

benötigten Definitionen wieder. An dieser Stelle wird nur die notwendige Definition der Signatur und Belegung wiedergegeben.

Definition 4.6. (u. a. [51]) Eine **Signatur** $\Sigma = (S, F)$ wobei

- S eine Menge von **Sorten** und
- $F = (F_{w,s})_{w \in S^*, s \in S}$ eine Menge von **Operationssymbolen** ist.

$f \in F$ mit $f\colon s_1 \times \cdots \times s_n \to s$ ist ein Operationssymbol der Stelligkeit n. Für $n = 0$ heißt f **Konstantensymbol** oder Konstante.

Signaturen und Terme werden allgemein in Anhang B.1 (siehe Def. 4.6 und Def. B.7) eingeführt und an dieser Stelle nicht wiedergegeben.

Definition 4.7. Seien a_i, $0 \leq i \leq n, n \in \mathbb{N}$ Variablen der Sorte $\mathfrak{A}$ – der Musterattribute, $f \in \mathfrak{F}$ eine Variable der Sorte der Musterfunktionen und $c \in \mathfrak{C}$ eine Variable der Sorte der Musterkonstanten. Ein **Attributwertterm** $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ist induktiv wie folgt definiert:

a) c ist ein Attributwertterm;

b) a_i ist ein Attributwertterm und

c) sind $t^{(0)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}, \cdots, t^{(n)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ Attributwertterme, dann ist auch $f(t^{(0)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}, \cdots, t^{(n)}_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}})$ ein Attributwertterm.

Im Folgenden wird in dieser Arbeit davon ausgegangen, dass bei der Substitution nur „sortentreue“ Ersetzungen stattfinden, d.h. Variablen der Sorte $\mathfrak{A}$ durch Attribute ersetzt werden, Variablen der Sorte $\mathfrak{F}$ nur durch Funktionen und Variablen der Sorte $\mathfrak{C}$ nur durch nullstellige Funktionen substituiert werden. Details zu dieser Form der Substitution wird in Anhang C vorgestellt. Das Erlauben nicht-sortentreuer Substitutionen würde die Verwendung von Referenzattributgrammatiken erlauben. Wie bereits ausgeführt sind diese nicht im Rahmen dieser Arbeit vorgesehen, stattdessen werden geordnete Attributgrammatiken verwendet.

Nach dieser „sortentreuen“ Substitution werden die dadurch erzeugten Terme in einer Attributierungsregel angewandt und einer zerlegbaren Attributgrammatik hinzugefügt.

Vor der Einführung der Substitution auf Attributwerttermen ist es notwendig Prädikate zu definieren, die eine Auswahl bezüglich der Attributierungsregeln und Produktionen sowie Attribute einer zerlegbaren Attributgrammatik erlauben. Diese Prädikaten werden ebenfalls aus Prädikattermen bestimmt.

4.2.1. Prädikate und Prädikatterme

Attributwertterme beschreiben den Wert, den die rechte Seite einer Attributierungsregel einnehmen kann. Für die Verwendung über beliebige Attributgrammatiken mit dem Ziel Änderungsmengen herzuleiten, werden weitere Terme benötigt:

Definition 4.8. Seien a_i für $0 \leq i \leq n$, $n \in \mathbb{N}$, a_i der Sorte der Musterattribute $\mathfrak{A}$, $c, true, false$ der Sorte der Konstanten $\mathfrak{C}$. Für Variablen erster Ordnung $a_i, c, true, false, f$, einen Term erster Ordnung $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ und Variablen $S_0, S_1, \cdots, S_m$ der Sorte der Symbolvariablen $\mathfrak{V}$ zweiter Ordnung ist ein **Prädikatterm** $t_{\{\mathcal{P}\}}$ induktiv wie folgt definiert:

1. $true$ und $false$ sind (konstante) Prädikatterme;
2. ein Attributwertterm $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ist ein Prädikatterm (zweiter Ordnung);

Prädikatterm	**informelle Semantik**
$\forall S : \mathfrak{V} \bullet \exists S' : \mathfrak{V} \bullet (S ::= S \ / \wedge S ::= S')$	Es existiert ein Nichtterminal mit linksrekursiver Produktion und einer Produktion, die in einem anderen Symbol endet.
$\forall a : \mathfrak{A} \bullet a \in AI(S) \wedge S : \mathfrak{V}$	Alle ererbten Attribute eines Symbols.
$a_1, a_2, a_3 : \mathfrak{A}, S_0, S_1, S_2 : \mathfrak{V} \wedge S_0 \neq S_2 \wedge S_1 \neq S_2 \wedge \exists S_0 ::= S_1 \ S_2 \bullet a_1 \in AS(S_0) \wedge a_2 \in AS(S_1) \wedge a_3 \in AS(S_2) \wedge a_1 \leftarrow a_2 + a_3$	Gesucht sind drei synthetisierte Attribute, sodass das in der aktuellen Regel definierte Attribut sich genau aus der Addition dieser synthetisierten Attribute bestimmt; dabei ist die definierende Produktion mit zwei Symbolen auf der rechten Seite der Produktion versehen wobei linke Seite und zweites Symbol auf der rechten Seite nicht identisch sein dürfen, ebenso dürfen beide Symbole der rechten Seite nicht übereinstimmen.

Tabelle 4.2. – Beispiele nutzbarer Prädikate bei Bestimmung prädikatabhängiger Teil-Attributgrammatiken

3. für eine Variable a_j der Sorte $\mathfrak{A}$ ist $a_j \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ein Prädikatterm;
4. Seien $t^{(0)}_{\{\mathcal{P}\}}$ und $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme, so auch $t^{(0)}_{\{\mathcal{P}\}} \vee t^{(1)}_{\{\mathcal{P}\}}$, $t^{(0)}_{\{\mathcal{P}\}} \wedge t^{(1)}_{\{\mathcal{P}\}}$ und $t^{(0)}_{\{\mathcal{P}\}} = t^{(1)}_{\{\mathcal{P}\}}$;
5. Ist $t_{\{\mathcal{P}\}}$ ein Prädikatterm, so auch $\neg t_{\{\mathcal{P}\}}$
6. Seien $t^{(0)}_{\{\mathcal{P}\}}$ und $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme, so auch $\forall t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}}$ und $\exists t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}}$;
7. $S_0 ::= S_1 \ \cdots \ S_n$ ist ein Prädikatterm, ebenso $p : S_0 ::= S_1 \ \cdots \ S_n$ für einen neuen Bezeichner p; statt einer Variablen kann auch der Platzhalter _ stehen; statt S_n kann der Platzhalter / stehen;
8. für eine Variable a der Sorte $\mathfrak{A}$ (Musterattributvariable) sind $a \in AS$ und $a \in AI$ sowie $a \in AS(S_1)$ und $a \in AI(S_1)$ für eine Variable S_1 der Sorte $\mathfrak{V}$;

Prädikatterme der Form 1, 2, 3 heißen Prädikatterme erster Ordnung, Prädikatterme der Form 6, 7, 8 heißen Prädikatterme zweiter Ordnung und Prädikatterme der Form 4 und 5 heißen Prädikatterme erster oder zweiter Ordnung wenn $t_{\{\mathcal{P}\}}$, $t^{(0)}_{\{\mathcal{P}\}}$ oder $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme erster oder zweiter Ordnung sind.

In der Definition des Prädikatterms wurde keine Rücksicht auf Klammerung und Präzedenzen getroffen. Im Folgenden werden die üblichen Klammerungsregeln und Präzedenzen der Operatoren (wie $\vee$, $\wedge$, $\forall$, usw.) verwendet. Auf die übliche Betrachtung freier und gebundener Variablen wird nicht weiter eingegangen. Freie und Gebundene Variablen werden auch im Lambda-Kalkül eingesetzt – ebenso wie Substitution, Terme, Termersetzung und die Auswertung von Termen. Anhang B.1 gibt hierzu einen Überblick, verwandte Themenbereiche werden in [63] bearbeitet. Notationen dieser Arbeit werden in Anhang A präsentiert.

Tabelle 4.2 zeigt Beispiele einiger Prädikate und eine informelle Semantik zu diesen.

Muster werden im Folgenden durch Substitution aus solchen Termen hergeleitet. Ein Prädikatterm dient dann im Folgenden dazu eine Teilmenge einer zerlegbaren Attributgrammatik auszuwählen bzgl. derer die Attributterme substituiert werden können. Die jeweiligen Substitutionen stellen somit die Beschreibung der Anwendung eines Muster(terms) auf eine zerlegbare Attributgrammatik dar.

Definition 4.9. Seien $t_{\{\mathcal{P}\}}$, $t^{(0)}_{\{\mathcal{P}\}}$ und $t^{(1)}_{\{\mathcal{P}\}}$ Prädikatterme, $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ein Attributwertterm, sei ferner v_i eine Variable und t_i ein (variablenfreier) Prädikatterm, die **Substitution σ auf Prädikattermen** mit

$$a \leftarrow c$$

a) Prädikatterm für die Anwendung einer Substitution, Variable a der Sorte $\mathfrak{A}$ und v der Sorte $\mathfrak{C}$

$$\sigma = \left[\texttt{VarStat.number}/a, 100/c\right]$$

b) Substitution σ

$$\texttt{VarStat.number} \leftarrow 100$$

c) Resultat nach Ausführung der Substitution aus 4.7b auf den Term aus 4.7a

Beispiel 4.7 – Ausführung einer Substitution auf einem einfachen Prädikatterm

$\sigma = \left[t_i/v_i\right]$ als

$$t_{\{\mathcal{P}\}}[t_i/v_i] = \begin{cases} true & \text{falls } t_{\{\mathcal{P}\}} \doteq true \\ false & \text{falls } t_{\{\mathcal{P}\}} \doteq false \\ t_i & \text{falls } t_{\{\mathcal{P}\}} \doteq v_i \\ t_i \in AS & \text{falls } t_{\{\mathcal{P}\}} \doteq v_i \in AS \\ t_i \in AI & \text{falls } t_{\{\mathcal{P}\}} \doteq v_i \in AI \\ t_i \in AS(S_1\left[t_i/v_i\right]) & \text{falls } t_{\{\mathcal{P}\}} \doteq v_i \in AS(S_1) \\ t_i \in AI(S_1\left[t_i/v_i\right]) & \text{falls } t_{\{\mathcal{P}\}} \doteq v_i \in AI(S_1) \\ S_0\left[t_i/v_i\right] ::= S_1\left[t_i/v_i\right] \cdots S_n\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq S_0 ::= S_1 \cdots S_n \\ t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} = t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \\ a\left[t_i/v_i\right] \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \\ t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] \vee t^{(1)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq t^{(0)}_{\{\mathcal{P}\}} \vee t^{(1)}_{\{\mathcal{P}\}} \\ t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] \wedge t^{(1)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq t^{(0)}_{\{\mathcal{P}\}} \wedge t^{(1)}_{\{\mathcal{P}\}} \\ t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] = t^{(1)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq t^{(0)}_{\{\mathcal{P}\}} = t^{(1)}_{\{\mathcal{P}\}} \\ \forall t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] \bullet t^{(1)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq \forall t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}} \\ \exists t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] \bullet t^{(1)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq \exists t^{(0)}_{\{\mathcal{P}\}} \bullet t^{(1)}_{\{\mathcal{P}\}} \\ \neg t^{(0)}_{\{\mathcal{P}\}}\left[t_i/v_i\right] & \text{falls } t_{\{\mathcal{P}\}} \doteq \neg t^{(0)}_{\{\mathcal{P}\}} \\ t_{\{\mathcal{P}\}} & \text{sonst} \end{cases}$$

Die Anwendung einer Substitution mit mehreren Elementen $1 \leq i \leq n$ für $n \in \mathbb{N}$ entspricht der „Hintereinanderausführung“ der einzelnen Substitutionen. Beispiel 4.7 zeigt die Ausführung einer Substitution auf einem einfachen Prädikatterm.

In Definition 4.9 wird zum Vergleich der Prädikatterme in der Fallunterscheidung $\doteq$ (siehe rechten Seiten der Fallunterscheidung) für die syntaktische Gleichheit verwendet, da diese Art von Vergleich ebenfalls in den Prädikattermen vorkommen kann. Ist die Sorte der Variablen nicht unmittelbar aus dem Aufbau des Prädikatterms klar, wird in den folgenden Beispielen die Sorte nach einem Doppelpunkt der Variablen nachgestellt. Folgende Zeile zeigt einen gültigen Prädikatterm nach Definition 4.8:

$$\exists S : \mathfrak{V} \bullet \forall a : \mathfrak{A} \bullet a \in AS(S) \wedge \exists a_0 : \mathfrak{A} \bullet \exists a_1 : \mathfrak{A} \bullet a \leftarrow plus(a_0, a_1)$$

Analog lässt sich die Substitution auf Attributwerttermen definieren. Auf die explizite Definition dieser Substitution wird an dieser Stelle verzichtet. In Anhang C wird darauf eingegangen. Wie bereits im vorherigen Abschnitt beschrieben, wird in dieser Arbeit davon ausgangen, dass die Substitution „sortentreu“ geschieht. Signatur, Typ, Art der Variablen und des ersetzenden Terms müssen „passen“. In Anhang C wird auf diese Details genauer eingegangen.

Prädikatterme werden herangezogen um, zusammen mit einer gegebenen Attributgrammatik und einer Substitution, all jene Regeln und Attribute zu finden, die für die Änderungsmengen aus Abschnitt 4.1 relevant sind. Ziel ist eine Formulierung von Mustern als Terme. Zusammen mit einer Substitution und einer Attributgrammatik sollen sich dann die Änderungsmengen ergeben.

Ausgehend von der Substitution des Prädikatterms ist das Prädikat gebildet. Durch Prüfung auf einer zerlegbaren Attributgrammatik kann dieses ausgewertet werden. Neben der Notwendigkeit, dass das Prädikat zu „wahr“ bzw. *true* ausgewertet wird, sind weitere, implizite, Bedingungen bei der Auswahl der zu ändernden Teil-Attributgrammatik zu beachten. Für die allgemeine Definition der Substitution siehe Anhang C, die Definition von Termen in Anhang B.1.

Definition 4.10. Sei $t_{\{\mathcal{P}\}}$ eine Prädikatterm und σ eine Substitution von Prädikattermen. Sei $AG \triangleq (G, A, R, B)$, $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Sei $t = t_{\{\mathcal{P}\}}\sigma_{AG}$ das durch Substitution erzeugte Prädikat und t *variablenfrei*. Die **Semantik des Prädikats** t ist definiert durch

$$
\begin{aligned}
[\![true]\!]_{AG} &= true \\
[\![false]\!]_{AG} &= false \\
[\![t_1 \vee t_2]\!]_{AG} &= [\![t_1]\!]_{AG} \vee [\![t_2]\!]_{AG} \\
[\![t_1 \wedge t_2]\!]_{AG} &= [\![t_1]\!]_{AG} \wedge [\![t_2]\!]_{AG} \\
[\![t_1 = t_2]\!]_{AG} &= [\![t_1]\!]_{AG} \doteq [\![t_2]\!]_{AG} \\
[\![t_1 \vee t_2]\!]_{AG} &= [\![t_1]\!]_{AG} \vee [\![t_2]\!]_{AG}
\end{aligned}
$$

$$
[\![X_0 ::= X_1 \cdots X_n]\!]_{AG} = \begin{cases} true & \text{falls } X_0 ::= X_1 \cdots X_n \in P \\ false & \text{sonst} \end{cases}
$$

$$
[\![X_0 ::= X_1 \cdots X_{i-i}\_X_{i+1} \cdots X_n]\!]_{AG} = \begin{cases} true & \text{falls } p \in P, p = X_0 ::= X_1 \cdots X_{i-1} X_i X_{i+1} \cdots X_n \\ & \text{existiert mit } X_i \in \Sigma \\ false & \text{sonst} \end{cases}
$$

$$
[\![X_0 ::= X_1 \cdots X_{i-i}/]\!]_{AG} = \begin{cases} true & \text{falls } p \in P, p = X_0 ::= X_1 \cdots X_{i-1} X_i X_{i+1} \cdots X_n \in P \\ & \text{existiert} \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' : \mathfrak{A}]\!]_{AG} = \begin{cases} true & \text{falls } a' \in A \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' \in AS]\!]_{AG} = \begin{cases} true & \text{falls } a' \in AS \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' \in AI]\!]_{AG} = \begin{cases} true & \text{falls } a' \in AI \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' \in AS(X_1)]\!]_{AG} = \begin{cases} true & \text{falls } a' \in AS(X_1) \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' \in AI(X_1)]\!]_{AG} = \begin{cases} true & \text{falls } a' \in AI(X_1) \\ false & \text{sonst} \end{cases}
$$

$$
[\![a' \leftarrow t_a]\!]_{AG} = \begin{cases} true & \text{falls } a' \leftarrow t_a \in R \\ false & \text{sonst} \end{cases}
$$

$$
[\![\neg t]\!]_{AG} = \begin{cases} true & \text{falls } [\![t]\!]_{AG} \doteq false \\ false & \text{sonst} \end{cases}
$$

wobei t_1 und t_2 ebenfalls durch Substitution auf Prädikattermen erzeugte Prädikate sind und t_a ein durch Substitution auf einem Attributwertterm $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ erzeugte Attributterm ist und a' das durch Substitution aus der Variablen a erezugte Attribut. X_0 das aus S_0 durch Anwendung der Substitution erzeugte Nichtterminal und $X_1, \cdots X_n$ die durch Substitution erzeugten Symbole sind.

Für eine Variable x mit $t_{\{\mathcal{P}\}} = \exists x \bullet t^{(1)}_{\{\mathcal{P}\}}$ ist die Semantik des Prädikats

$$[\![(\exists x \bullet t^{(1)}_{\{\mathcal{P}\}})\sigma_{AG}]\!]_{AG} = \begin{cases} true & \text{falls} [\![t^{(2)}_{\{\mathcal{P}\}}\sigma_{AG}]\!]_{AG} \doteq true \text{ für ein } t' = t^{(1)}_{\{\mathcal{P}\}}\sigma_{AG} \\ false & \text{sonst} \end{cases}$$

und für $t_{\{\mathcal{P}\}} = \forall x \bullet t^{(1)}_{\{\mathcal{P}\}}$

$$[\![(\forall x \bullet t^{(1)}_{\{\mathcal{P}\}})\sigma_{AG}]\!]_{AG} = \begin{cases} true & \text{falls} [\![t^{(1)}_{\{\mathcal{P}\}}\sigma_{AG}]\!]_{AG} \doteq true \text{ für jedes } t' = t^{(1)}_{\{\mathcal{P}\}}\sigma_{AG} \\ false & \text{sonst} \end{cases}$$

Weitere Details zur Anwendung der Substitution auf Prädikattermen und der Semantik von Prädikaten bzgl. einer Attributgrammatik werden in Anhang C vorgestellt. Die Auswertung eines Prädikats folgt den intuitiven Regeln, sodass die üblichen Operationen die übliche Semantik enthalten. So ist die Semantik der Konjunktion und Disjunktion intuitiv klar auf Basis der üblichen mathematischen Regeln.

Hinweis: In Definition 4.10 wird eine Produktion mit Platzhaltern dargestellt – diese Platzhalter sind _ und /.

Die Prüfung ob ein Prädikat zu wahr auswertet wird, folgt den intuitiven Ansätzen, wobei Existenz- und Allquantor nur zur Variablenbindung dienen. Die Variablen werden in der Substitution bereits ersetzt, sodass die Auswertung dieses Teilterms nicht genauer beachtet wird.

Die Bestimmung der Teilmenge der zerlegbaren Attributgrammatik für die die Änderungsmengen herangezogen werden müssen, lässt sich mit diesen Prädikate bestimmen. Intuitiv werden alle „Teile“ einer zerlegbaren Attributgrammatik – abstrakte Syntax, Attribute, Attributierungsregeln – herangezogen, für die ein Prädikat zu wahr auswertet.

Definition 4.11. Sei $AG \in \mathcal{AG}_G, AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik. Seien weiterhin $Pr_+ = \{t^{(0,+)}_{\{\mathcal{P}\}}, \cdots, t^{(n,+)}_{\{\mathcal{P}\}}\}$ und $Pr_- = \{t^{(0,-)}_{\{\mathcal{P}\}}, \cdots, t^{(m,-)}_{\{\mathcal{P}\}}\}$ Mengen von Prädikattermen und σ_{AG} eine Substitution von Prädikattermen bzgl. AG. Das Paar (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$ heißt **prädikatabhängige Teil-Attributgrammatik** (bzgl. AG und Pr_+ und Pr_-) genau dann, wenn die Anzahl der Elemente in $G_+, A_+, R_+, G_-, A_-, R_-$ minimal ist und folgende Bedingungen bzgl. dieser Mengen eingehalten werden, d. h. es gibt keine Menge, die dieselben Bedingungen einhält und weniger Elemente enthält: Für alle Prädikatterme $pred \in Pr_+$ und Substitution σ_{AG} mit $[\![pred]\!]_{AG}$ ist $AG_+ = (G_+, A_+, R_+, B)$ mit $G_+ = (N_+, T_+, P_+, Z_+)$ dann halten

$$\begin{aligned} &\{\mathsf{X}_0 \colon \mathsf{X}_0 \in N \wedge \exists p \in P_+ : p = \mathsf{X}_0 ::= u, u \in \Sigma^* \wedge N_+ \setminus \{\mathsf{X}_0\} \implies [\![pred]\!]_{AG} \doteq false\} \\ \cup &\{\mathsf{Y} \colon \mathsf{Y} \in N, \exists p \in P_+, p = \mathsf{X} :: \ u\ \mathsf{Y}\ v, \mathsf{Y} \in N, u, v \in \Sigma^* \wedge N_+ \setminus \{\mathsf{Y}\} \implies [\![pred]\!]_{AG} \doteq false\} \\ &\subseteq N_+ \subseteq N \end{aligned}$$

$$\begin{aligned} &\{\mathsf{X} \colon \mathsf{X} \in T \wedge \exists p \in P_+ : p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*, \mathsf{Y} \in N_+ \wedge T_+ \setminus \{\mathsf{X}\} \implies [\![pred]\!]_{AG} \doteq false\} \\ &\subseteq T_+ \subseteq T \end{aligned}$$

$$\begin{aligned} &\{q \colon q \in P, q = \mathsf{X} ::= u, u \in \Sigma^*, \mathsf{X} \in N_+ \wedge \{P_+ \setminus \{q\} \implies [\![pred]\!]_{AG} \doteq false\} \\ \cup &\{p \colon p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*, \wedge(\exists q \in P_+ : q = \mathsf{X} ::= v, v \in \Sigma^* \vee a \in A_+ \wedge a \in AS(\mathsf{X})\} \\ \cup &\{p \colon p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*, \mathsf{X} \in \Sigma, \mathsf{Y} \in N \wedge \exists a \in A_+, a \in AI(\mathsf{X})\} \\ \cup &\{p \colon p \in P \wedge \exists r \in R_+, r \in R_p \wedge R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false\} \\ &\subseteq P_+ \subseteq P \end{aligned}$$

$$Z_+ \in N_+ \wedge \{\mathsf{X} \colon \mathsf{X} \in N_+ \wedge \mathsf{X} \overset{+}{\rightsquigarrow} Z_+\} = \emptyset \wedge \mathsf{X} \in N_+, \mathsf{X} \neq Z_+ \implies Z_+ \overset{+}{\rightsquigarrow} \mathsf{X}$$

$$\{b_i : \exists r = a \leftarrow f(b_1, \cdots, b_n) \in R_+ \vee (a \in A_+ \wedge r \in R) \wedge (R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false \vee A_+ \setminus \{a\} \implies [\![pred]\!]_{AG} \doteq false)\} \subseteq A_+ \subseteq A$$

$$\{r : r = a \leftarrow f(\cdots), a \in A_+ \wedge R_+ \setminus \{r\} \implies [\![pred]\!]_{AG} \doteq false\} \subseteq R_+ \subseteq R$$

ein.

Analoges gilt für die jeweiligen G_-, A_-, R_-, sowie N_-, T_-, P_-, Z_-.

Die Auswertung der Bedingungen für die prädikatabhängigen Teil-Attributgrammatiken ist über Fixpunktiterationen lösbar. In einer prädikatabhängigen Teil-Attributgrammatik soll nach Definition 4.11 ein konsistenter, vollständiger Teilausschnitt der Attributgrammatik enthalten sein. Solche prädikatabhängigen Teil-Attributgrammatiken dienen bspw. der Erzeugung von Attributgrammatiken ausgehend von einer neuen Wurzel und der Attributierung dieser bspw. unabhängig von den sonstigen Attributen der Attributgrammatik. Die letzten vier Bedingungen in Definition 4.11 stellen sicher, dass eine eindeutige Wurzel für die Teilgrammatik existiert und diese nicht aufgrund von ererbten Attributierungen inkonsistent ist.

In den Bedingungen von Definition 4.11 sind bereits einige Einschränkungen mit aufgeführt, die üblicherweise zwingend notwendig sind, damit die Teil-Attributgrammatik und die darauf angewendeten Änderungsmengen verträglich mit den Sätzen und Lemmata des vorherigen Abschnitts sind.

4.2.2. Erweiterte Attributwertterme und Änderungsmengen

Die aufgrund von Substitution auf Prädikattermen und Anwendung auf einer Attributgrammatik gewonnenen Teilmengen dienen im nächsten Schritt der Herstellung der Änderungsmengen für die die Eigenschaften aus Abschnitt 4.1 gelten müssen. Dafür werden die Attributwertterme erweitert und auf diese erweiterten Attributwertterme dieselben Substitutionen angwendet, wie auf die Prädikatterme (neben weiterer Substitutionen). Somit erweitert folgende Definition Attributwertterme, so dass die Herleitung der Änderungsmengen auf Basis der Substitution(en) ermöglicht wird.

Definition 4.12. Sei a eine Variable der Sorte der Musterattribute $\mathfrak{A}$, $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ ein Attributwertterm, $S_0, S_1, \cdots, S_n$ Variablen der Sorte der Symbolvariablen $\mathfrak{V}$. Ein Term $t_{\{\mathfrak{P}\}}$ heißt **erweiterter Attributwertterm** genau dann, wenn $t_{\{\mathfrak{P}\}}$ eine der folgenden Formen hat

1. **symbol** S_0 **attr** $\uparrow.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$
2. **symbol** S_0 **attr** $\downarrow.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$
3. **rule** $S_0 ::= S_1 \ \cdots \ S_n$ **attr** $S_j.a \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$

wobei $0 \leq j \leq n$ ist.

Ein erweiterter Attributwertterm erlaubt somit die Darstellung von Berechnungen, die eine Attributierungsregel bezüglich einer oder mehrerer Produktionen mit den Variablen S_0 bis S_n darstellen. Gleichermaßen erlaubt Definition 4.12 die Darstellung von Berechnungen für Attribute eines Symbols unabhängig von der Produktion. Die Terme aus Definition 4.12 stellen somit die grundlegenden Attributierungsmöglichkeiten von Attributgrammatiken in dieser Arbeit dar. In folgenden Abschnitten werden diese erweiterten Attributwertterme auch als Basismuster verstanden und so vorgestellt.

Im Folgenden wird die *Attributgrammatik-unabhängige Musterdefinition* verwendet um die Änderungsmengen zu bilden.

Definition 4.13. Das Tupel $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ heißt **Attributgrammatik-unabhängige Musterdefinition** wobei

- $V = \mathfrak{V} \uplus \mathfrak{A} \uplus \mathfrak{C} \uplus \mathfrak{F}$ die Menge der Variablen;
- $Pr = Pr_+ \cup Pr_-$ die Menge der Prädikate unter Verwendung der Variablen aus V sind und
- $At = At_+ \cup At_-$ die Menge erweiterter Attributwertterme über Variablen der Sorten aus V sind.

Die Herleitung der Änderungsmengen aus einer Attributgrammatik-unabhängigen Musterdefinition erfolgt unter Verwendung der Substitution. Solch eine Substitution benötigt einige Eigenschaften, sodass nach Anwendung der Substitution Änderungsmengen entstehen, die in einem Muster münden. Der grundlegende Aufbau der Terme, die für Variablen substitutiert werden, ist in folgender Definition gegeben. Erst danach wird über den konkreten Aufbau der Substitutionsergebnisse und deren Eigenschaften argumentiert.

Definition 4.14. Sei $S \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge von Substitutionen, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Eine Substitution $\sigma_i = \left[t_{i_1}/v_{i_1} \cdots t_{i_m}/v_{i_m}\right]$, $0 \leq i \leq n$ heißt **passende Substitution** zu AG und $\mathcal{M}_u$ genau dann, wenn für alle Ersetzungen t_{i_j}/v_{i_j} mit $1 \leq j \leq m, m \in \mathbb{N}$ folgende Eigenschaften erfüllt sind:

1. $v_{i_j} \in V$, v_{i_j} der Sorte $Sort$
2. t_{i_j} ist ein Wort über dem Alphabet $\Sigma \triangleq N \cup T \cup \{::=, \_, /, \leftarrow, \varepsilon\} \cup A \cup F \cup A'$, wobei F eine Menge an Operationssymbolen ist und $A' \cap A = \emptyset$ und t_{i_j} der Sorte $Sort$;
3. $v_{i_k} = v_{i_l} \implies t_{i_k} = t_{i_l}, 0 \leq k \leq m, 0 \leq l \leq m$.

σ_i heißt **anwendbare Substition** auf $t_{\{\mathcal{P}\}}$ oder anwendbare Substitution auf $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$, wenn sie passend ist und mindestens eine der Variablen v_{i_j} in $t_{\{\mathcal{P}\}}$ oder $t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}}$ vorkommt.

Eine Menge $Subs$ heißt **Menge passender Substitutionen** genau dann, wenn jedes $\sigma_i \in Subs$ eine passende Substitution ist und falls $v_{i_j} = v_{k_l}$ dann auch $t_{i_j} = t_{k_l}$ mit $0 \leq i \leq n, 0 \leq j \leq m$ und $0 \leq k \leq n, 0 \leq l \leq o, o \in \mathbb{N}$.

Eine weitere Definition erweitert Definition 4.14 um die Anwendung einer Menge passender Substitutionen auf mehreren Termen.

Definition 4.15. Sei $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Die **Semantik der Anwendung der Menge passender Substitutionen** $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ auf einem Term t ist definiert als $[\![t\ Subs]\!] \triangleq \{t\sigma_{\pi(1)} \cdots \sigma_{\pi(n)} : \pi\colon \{1, \cdots, n\} \to \{1, \cdots, n\}$ ist Permutation$\}$ wobei $\sigma_{\pi_i} \in Subs$ für alle Permutationen $\pi_1, \cdots, \pi_n$ von 1 bis n, wenn $\pi_i = \pi_j$ dann auch $i = j, 1 \leq i \leq n$, $1 \leq j \leq n$.

Mit den Definitionen 4.13, 4.12 und Definition 4.14 ist die notwendige Basis geschaffen, um die Herleitung der Änderungsmengen aus einer Attributgrammatik-unabhängigen Musterdefinition vorzustellen. Dabei ist in Definition 4.15 zu beachten, dass das Resultat der Anwendung einer Menge passender Substitutionen eine Menge (über die Permutation der Substitutionsanwendung) ist.

Definition 4.16. Sei $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition mit $Pr = Pr_+ \cup Pr_-$ und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, sowie der prädikatabhängigen Teil-Attributgrammatik (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$, wobei $G_+ \triangleq (N_+, T_+, P_+, Z_+)$ und $G_- \triangleq (N_-, T_-, P_-, Z_-)$. Sei $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Das Tupel $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ heißt **Musteranwendung** bzgl. AG und $\mathcal{M}_u$ genau dann, wenn sich $\mathcal{M}_{+,A}, \mathcal{M}_{+,R}, \mathcal{M}_{-,A}$ und $\mathcal{M}_{-,R}$ mit folgenden Regeln bilden:

1. Die Menge $\mathcal{M}_{+,A}$ sind alle Attribute für alle variablenfreien, aus erweiterten Attributwerttermen aus At_+, durch passende Substitutionen, erzeugten Attribute für jede der möglichen Formen des erweiterten Attributwertterms:

$$\begin{aligned}\mathcal{M}_{+,A} = \{&a\colon a \in [\![a'\ Subs]\!]_{AG}, a \text{ variablenfrei } \wedge a \notin A_+ \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \text{ mit}\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \vee\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\\ &f(b_1, \cdots, b_d) \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \text{ mit } b_g \in \mathcal{M}_{+,A} \cup A_+, 1 \leq g \leq d, d \in \mathbb{N}\\ &f \text{ variablenfrei}, \exists X \in [\![S_0\ Subs]\!]_{AG} \wedge X \in \Sigma\}\\ \cup \{&a\colon a \in [\![S_j.a'\ Subs]\!]_{AG} \wedge a \text{ variablenfrei } \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \text{ mit}\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1\ \cdots\ S_n \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\\ &f(b_1, \cdots, b_d) \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \text{ mit } b_g \in \mathcal{M}_{+,A} \cup A_+, 1 \leq g \leq d, d \in \mathbb{N}\}\end{aligned}$$

2. Die Menge $\mathcal{M}_{-,A}$ sind alle Attribute für alle variablenfreien, aus erweiterten Attributwerttermen aus At_-, durch passende Substitutionen, erzeugten Attribute für jede der möglichen Formen des erweiterten Attributwertterms:

$$\begin{aligned}\mathcal{M}_{-,A} = \{&a\colon a \in [\![a'\ Subs]\!]_{AG}, a \text{ variablenfrei } \wedge a \in A_- \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_- \text{ mit}\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \vee\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\\ &\exists X \in [\![S_0\ Subs]\!]_{AG} \wedge X \in \Sigma\}\\ \cup \{&a\colon a \in [\![S_j.a'\ Subs]\!]_{AG} \wedge a \text{ variablenfrei } \wedge t^{(i)}_{\{\mathfrak{P}\}} \in At_- \text{ mit}\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1\ \cdots\ S_n \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\}\\ \subseteq\ &A\end{aligned}$$

3. Die Menge $\mathcal{M}_{+,R,s}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_+, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat synthetisierte Attributierungsregeln sind:

$$\begin{aligned}\mathcal{M}_{+,R,s} = \{&r\colon r = \ \textbf{rule } \mathtt{X}_0 ::= \mathtt{X}_1\ \cdots\ \mathtt{X}_n \textbf{ attr } \mathtt{X}_0.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \wedge\\ &\mathtt{X}_0 \in [\![S_0\ Subs]\!]_{AG} \wedge \exists p \in P_+, p = \mathtt{X}_0 ::= \mathtt{X}_1\ \cdots\ \mathtt{X}_n \wedge\\ &a \in [\![a'\ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{+,A}, \mathtt{X}_0 \in N,\\ &\mathtt{X}_k \in \Sigma_+, 1 \leq k \leq n,\ a, t' \text{ variablenfrei}\}\\ \cup\{&r\colon r = \ \textbf{rule } \mathtt{X}_0 ::= \mathtt{X}_1\ \cdots\ \mathtt{X}_n \textbf{ attr } \mathtt{X}_0.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+\\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1\ \cdots\ S_m \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}} \text{ mit}\\ &\mathtt{X}_0 \in [\![S_0\ Subs]\!]_{AG}, \mathtt{X}_l \in [\![S_k\ Subs]\!]_{AG}, 1 \leq l \leq n, 1 \leq k \leq m, m \geq n,\\ &\exists p \in P_+, p = \mathtt{X} ::= \mathtt{X}_1\ \cdots\ \mathtt{X}_n \wedge \mathtt{X}_0.a \in [\![S_j.a']\!]_{AG}, t' \in [\![t_{\{\mathfrak{E},\mathfrak{A},\mathfrak{F}\}}\ Subs]\!]_{AG},\\ &\mathtt{X}_0.a \in \mathcal{M}_{+,A}, \mathtt{X}_0 \in N_+, \mathtt{X}_l \in \Sigma_+, \mathtt{X}_0.a, t' \text{ variablenfrei}\}\end{aligned}$$

4. Die Menge $\mathcal{M}_{+,R,i}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_+, durch passende Substitutionen, erzeugten Attributierungsregeln deren

Resultat ererbte Attributierungsregeln sind:

$$\begin{aligned}\mathcal{M}_{+,R,i} = \{&r\colon r^{(k)} = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_j \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_j^{(k)}.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \wedge \\ &\mathtt{X}_j \in [\![S_0 \ Subs]\!]_{AG} \wedge \exists p \in P_+, p = \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_j \ \cdots \ \mathtt{X}_n \wedge \\ &a \in [\![a' \ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{+,A}, \mathtt{X}_0 \in N, \\ &\mathtt{X}_v \in \Sigma_+, 1 \leq v \leq n, \ a,t \text{ variablenfrei}, 1 \leq k \leq \left|\mathtt{X}_j\right|_p\} \\ \cup\{&r\colon r = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_h \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_j.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_+ \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1 \ \cdots \ S_m \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \text{ mit} \\ &\mathtt{X}_0 \in [\![S_0 \ Subs]\!]_{AG}, \mathtt{X}_l \in [\![S_k \ Subs]\!]_{AG}, 1 \leq l \leq n, 1 \leq k \leq m, m \geq n, \\ &\mathtt{X}_h \in [\![S_j \ Subs]\!]_{AG} \exists p \in P_+, p = \mathtt{X} ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \wedge \mathtt{X}_h.a \in [\![S_j.a']\!]_{AG}, \\ &t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG}, \mathtt{X}_h.a \in \mathcal{M}_{+,a}, \mathtt{X}_0 \in N_+, \mathtt{X}_l \in \Sigma_+, \mathtt{X}_0.a, t' \text{ variablenfrei}\}\end{aligned}$$

5. Die Menge $\mathcal{M}_{-,R,s}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_-, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat synthetisierte Attributierungsregeln sind:

$$\begin{aligned}\mathcal{M}_{-,R,s} = \{&r\colon r = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_0.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \uparrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \wedge \\ &\mathtt{X}_0 \in [\![S_0 \ Subs]\!]_{AG} \wedge \exists p \in P_-, p = \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \wedge \\ &a \in [\![a' \ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{-,A}, \mathtt{X}_0 \in N, \\ &\mathtt{X}_k \in \Sigma_-, 1 \leq k \leq n, \ a,t \text{ variablenfrei}\} \\ \cup\{&r\colon r = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_0.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1 \ \cdots \ S_m \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \text{ mit} \\ &\mathtt{X}_0 \in [\![S_0 \ Subs]\!]_{AG}, \mathtt{X}_l \in [\![S_k \ Subs]\!]_{AG}, 1 \leq l \leq n, 1 \leq k \leq m, m \geq n, \\ &\exists p \in P_-, p = \mathtt{X} ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \wedge \mathtt{X}_0.a \in [\![S_j.a']\!]_{AG}, t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG}, \\ &\mathtt{X}_0.a \in \mathcal{M}_{-,a}, \mathtt{X}_0 \in N_-, \mathtt{X}_l \in \Sigma_-, \mathtt{X}_0.a, t' \text{ variablenfrei}\}\end{aligned}$$

6. Die Menge $\mathcal{M}_{-,R,i}$ sind alle Attributierungsregeln für alle variablenfreien, aus erweiterten Attributwerttermen aus At_-, durch passende Substitutionen, erzeugten Attributierungsregeln deren Resultat ererbte Attributierungsregeln sind:

$$\begin{aligned}\mathcal{M}_{R,i,-} = \{&r\colon r^{(k)} = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_j \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_j^{(k)}.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{symbol } S_0 \textbf{ attr } \downarrow .a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \wedge \\ &\mathtt{X}_j \in [\![S_0 \ Subs]\!]_{AG} \wedge \exists p \in P_-, p = \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_j \ \cdots \ \mathtt{X}_n \wedge \\ &a \in [\![a' \ Subs]\!]_{AG} \wedge t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG} \wedge a \in \mathcal{M}_{-,A}, \mathtt{X}_0 \in N, \\ &\mathtt{X}_v \in \Sigma_-, 1 \leq v \leq n, \ a,t \text{ variablenfrei}, 1 \leq k \leq \left|\mathtt{X}_j\right|_p\} \\ \cup\{&r\colon r = \textbf{ rule } \mathtt{X}_0 ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_h \ \cdots \ \mathtt{X}_n \textbf{ attr } \mathtt{X}_j.a \leftarrow t' \text{ und } \exists t^{(i)}_{\{\mathfrak{P}\}} \in At_- \\ &t^{(i)}_{\{\mathfrak{P}\}} = \textbf{rule } S_0 ::= S_1 \ \cdots \ S_m \textbf{ attr } S_j.a' \leftarrow t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \text{ mit} \\ &\mathtt{X}_0 \in [\![S_0 \ Subs]\!]_{AG}, \mathtt{X}_l \in [\![S_k \ Subs]\!]_{AG}, 1 \leq l \leq n, 1 \leq k \leq m, m \geq n, \\ &\mathtt{X}_h \in [\![S_j \ Subs]\!]_{AG} \exists p \in P_-, p = \mathtt{X} ::= \mathtt{X}_1 \ \cdots \ \mathtt{X}_n \wedge \mathtt{X}_h.a \in [\![S_j.a']\!]_{AG}, \\ &t' \in [\![t_{\{\mathfrak{C},\mathfrak{A},\mathfrak{F}\}} \ Subs]\!]_{AG}, \mathtt{X}_h.a \in \mathcal{M}_{-,a}, \mathtt{X}_0 \in N_-, \mathtt{X}_l \in \Sigma_-, \mathtt{X}_0.a, t' \text{ variablenfrei}\}\end{aligned}$$

7. für alle $r \in \mathcal{M}_{+,R} = \mathcal{M}_{R,s,+} \cup \mathcal{M}_{+,R,i}$, ist $R_+ \cup \{r\}$ lokal azyklisch

mit $\Sigma_+ = T_+ \uplus N_+$ und $\Sigma_- = T_- \uplus N_-$ und $\mathcal{M}_{-,R} = \mathcal{M}_{-,R,s} \cup \mathcal{M}_{-,R,i}$,

Durch Definition 4.16 und die Definition 4.15 ist es möglich aus wenigen kurzen Prädikaten und wenigen erweiterten Attributwerttermen eine große Menge von Attributierungsregeln herzuleiten. Beispiel 4.8 beschreibt, wie die, in diesem und vorherigen Abschnitt dargestellten, Terme und Substitutionen in einer vollständigen Attributgrammatik münden.

Beispiel 4.8 zeigt, wie aus wenigen Prädikaten und Attributierungstermen und wenigen Substitutionen mindestens fünfmal so viele Attributierungsregeln hergeleitet werden können. Zwischenschritte, die in Beispiel 4.8 aufgrund der in Definition 4.16 vorkommenden Zwischenmengen zur Bestimmung des Resultats notwendig sind, wurden nicht aufgeführt.

Die Herstellung der für Satz 4.1 notwendigen Eigenschaften lässt sich in der Praxis leicht erreichen. Für diese Arbeit wichtig ist dieser Prozess nicht, lediglich die damit einhergende Definition:

$$
\begin{aligned}
V &= \{S_0, S_1, S_2, S_3\} \uplus \{a, b\} \\
Pr_+ &= \{true\} \\
Pr_- &= \{false\} \\
At_+ &= \{at_1 = \textbf{rule}\ S_0 ::= S_1\ \textbf{attr}\ \uparrow.a \leftarrow 10, \\
&\qquad at_2 = \textbf{rule}\ S_0 ::= S_2\ \textbf{attr}\ \uparrow.a \leftarrow 100, \\
&\qquad at_3 = \textbf{symbol}\ S_0\ \textbf{attr}\ \uparrow.b \leftarrow S_1.b + S_2.b\} \\
At_- &= \emptyset
\end{aligned}
$$

a) Prädikate und erweiterte Attributwertterme

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr
7  rule VarStat ::= id number attr
```

b) Attributgrammatik zur Anwendung

$$
\begin{aligned}
S = \{&\sigma_1 = \left[VarStat/S_0, id\ number/S_1, cnt/a\right], \\
&\sigma_2 = \left[VarStat/S_0, id\ id/S_2, cnt/a\right], \\
&\sigma_3 = \left[Stat/S_0\right], \sigma_4 = \left[Stats/S_0\right], \\
&\sigma_5 = \left[VarStat/S_1\right], \sigma_6 = \left[\varepsilon/S_2\right], \\
&\sigma_7 = \left[0/S_2.b\right], \sigma_8 = \left[Stats/S_1\right], \\
&\sigma_9 = \left[Stats/S_2\right] \sigma_{10} = \left[Stat/S_1\right], \\
&\sigma_{11} = \left[Stat/S_2\right], \sigma_{12} = \left[Program/S_0\right], \\
&\sigma_{13} = \left[stats/b\right], \sigma_{14} = \left[VarStat.cnt/S_1.b\right]\}
\end{aligned}
$$

c) Substitutionen

$$
\begin{aligned}
\mathcal{M}_{+,A} &= \{VarStat.cnt, Stat.stats, Stats.stats, Program.stats\} \\
\mathcal{M}_{-,A} &= \{\} \\
\mathcal{M}_{+,R} = \{&\textbf{rule}\ Program ::= Stats\ \textbf{attr}\ Program.stats \leftarrow Stats.stats + 0, && \textit{mit } at_3, \sigma_12, \sigma_13, \sigma_6, \sigma_7, \sigma_8 \\
&\textbf{rule}\ Stats ::= Stats\ Stat\ \textbf{attr}\ Stats_1.stats \leftarrow Stats_2.stats + Stat.stats, && \textit{mit } at_3, \sigma_4, \sigma_8, \sigma_11, \sigma_13 \\
&\textbf{rule}\ Stats ::= Stat\ Stats\ \textbf{attr}\ Stats_1.stats \leftarrow Stat.stats + Stats_2.stats, && \textit{mit } at_3, \sigma_4, \sigma_10, \sigma_13, \sigma_9 \\
&\textbf{rule}\ Stats ::= Stat\ \textbf{attr}\ Stats.stats \leftarrow Stat.stats + 0, && \textit{mit } at_3, \sigma_4, \sigma_10, \sigma_13, \sigma_7, \sigma_6 \\
&\textbf{rule}\ Stat ::= VarStat\ \textbf{attr}\ Stat.stats \leftarrow VarStat.cnt + 0, && \textit{mit } at_3, \sigma_7, \sigma_14, \sigma_3, \sigma_13 \\
&\textbf{rule}\ VarStat ::= id\ id\ \textbf{attr}\ VarStat.cnt \leftarrow 100, && \textit{mit } at_2, \sigma_2 \\
&\textbf{rule}\ VarStat ::= id\ number\ \textbf{attr}\ VarStat.cnt \leftarrow 10\} && \textit{mit } at_1, \sigma_1
\end{aligned}
$$

d) Mengen nach Anwendung von Definition 4.16

Beispiel 4.8 – Umformung der Terme und Substitutionen mit Zwischenmengen zur Herleitung der vollständigen Attributierung einer Attributgrammatik

Definition 4.17. Sei $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition mit $Pr = Pr_+ \cup Pr_-$ und $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, sowie der prädikatabhängigen Teil-Attributgrammatik (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$, wobei $G_+ \triangleq (N_+, T_+, P_+, Z_+)$ und $G_- \triangleq (N_-, T_-, P_-, Z_-)$. Sei $Subs \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge passender Substitutionen zu AG und $\mathcal{M}_u$. Das Tupel $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ ist. Halten die Mengen nach $\mathcal{M}_u$ die Eigenschaften von Satz 4.1 ein, wobei $\mathsf{A}_- = \mathcal{M}_{-,A}$, $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_- = \mathcal{M}_{-,R}$ und $\mathsf{R}_+ = \mathcal{M}_{+,R}$ so heißt $\mathcal{M}_u$ mit den Substitutionen $Subs$ **schlichte Musteranwendung**.

Eine Musteranwendung, die Definition 4.17 nicht genügt, kann bspw. zustande kommen, wenn komplexere Zyklen eingefügt werden oder die Musterdefinition keine Rücksicht auf bestehende Attributierungsregeln legt. Bei einer Definition von Mustern ist die Berücksichtigung dieser Eigenschaften notwendig. Im Folgenden wird die Anwendung von Mustern im Zusammenhang mit geordneten Attributgrammatiken entwickelt.

4.3. Muster und geordnete Attributgrammatiken

In den bisher betrachteten Abschnitten und Definitionen dieser Arbeit wurden zerlegbare Attributgrammatiken behandelt. Für geordnete Attributgrammatiken ist bekannt, dass die dazugehörigen Generatoren in performanten Übersetzern münden (siehe u. a. [16, 17]). Wie Kastens bereits in [75] gezeigt hat, lässt sich zu jeder Attributgrammatik, in der Zyklen ausschließlich im erweiterten Abhängigkeitsgraphen vorkommen, durch zusätzliche Abhängigkeiten ordnen.

In diesem Abschnitt folgt nun die Betrachtung der Eigenschaften, die Muster erfüllen müssen um nicht nur zerlegbare Attributgrammatiken in zerlegbare Attributgrammatiken zu überführen, sondern geordnete Attributgrammatiken in geordnete. Folgende Definition formalisiert diese Betrachtung:

Definition 4.18. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax G und $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik, $\mathcal{M}_G \colon \mathcal{AG} \to \mathcal{AG}$ ein Muster und $AG' \triangleq \mathcal{M}_G(AG)$, AG' zerlegbar. Jedes Muster $\mathcal{M}_G$ heißt **zerlegungserhaltend**, sind AG und AG' geordnet, heißt $\mathcal{M}_G$ **ordnungserhaltend**. Ist AG' geordnet heißt $\mathcal{M}_G$ auch **ordnendes Muster**.

Jedes ordnungserhaltende Muster ist nach Definition 4.18 auch ein ordnendes Muster. Insbesondere in Fällen, in denen über die Ordnungseigenschaft der ursprünglichen Attributgrammatik nichts bekannt ist, findet diese Definition Anwendung. In dieser Arbeit ist es Ziel nachzuweisen, dass für viele Muster diese ordnungserhaltend sind. Muster, die weder zerlegungserhaltend noch ordnungserhaltend sind, sind für diese Arbeit nicht relevant.

Dass ein Muster ordnungserhaltend ist, bedeutet, dass das Hinzufügen neuer Attribute und Attributierungsregeln einer Attributgrammatik für kein Symbol X und keine Produktion p Zyklen in den Mengen IDS_{X} und IDP_p erzeugt werden. Darüber hinaus darf im erweiterten Abhängigkeitsgraphen bzgl. der Partitionierung der Attributgrammatik kein Zyklus erzeugt werden.

Initial lässt sich feststellen, dass jede geordnete Attributgrammatik auch zerlegbar ist bzw. jedes ordnungserhaltende Muster zerlegungserhaltend ist.

Lemma 4.10. Sei $\mathcal{AG}_G$ die Menge aller Attributgrammatiken mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik, $\mathcal{M}_G \colon \mathcal{AG} \to \mathcal{AG}$ ein zerlegungserhaltendes Muster und $AG' \triangleq \mathcal{M}_G(AG)$ die resultierende Attributgrammatik. Ist $\mathcal{M}_G$ ordnungserhaltend, so ist $\mathcal{M}_G$ auch zerlegungserhaltend.

Beweis. Angenommen AG' sei nicht zerlegbar, dann existieren zwei Fälle:

1. AG' ist nicht lokal azyklisch oder

2. für ein Symbol $\mathsf{X} \in \Sigma$ ist die Zerlegung $A_\mathsf{X} = A_\mathsf{X}(1) \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$ nicht zulässig oder es existiert ein abstrakter Syntaxbaum AST für den in einem Knoten K mit $Typ(K) = \mathsf{X}$ die Attribute nicht in der Reihenfolge $A_\mathsf{X}(1), \cdots, A_\mathsf{X}(m_\mathsf{X})$ berechnet werden können.

Fall 1 steht im Widerspruch, dass AG' geordnet ist: für jedes Symbol $\mathsf{Y} \in \Sigma$ ist IDS_Y azyklisch, genauso auch für jede Produktion $p \in P$ ist IDP_p azyklisch, damit ist AG' auch lokal azyklisch für jede Produktion p. Wäre dies nicht der Fall wäre IDP_p ebenfalls zyklisch, da IDP_p alle Kanten aus DDP_p enthält oder äquivalente transitive Kanten[1].

Fall 2 steht im Widerspruch, dass AG' geordnet ist. Sei $\mathsf{X} \in \Sigma$ genau dieses Symbol mit der Zerlegung $A_\mathsf{X} = A_\mathsf{X}(1) \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$, sodass A_X nicht zulässig zerlegt ist. Sei $A_\mathsf{X}(i) \subseteq AI_\mathsf{X}$ für $i = m_\mathsf{X}, m_\mathsf{X}-2, \cdots$ und $A_\mathsf{X}(i) \subseteq AS_\mathsf{X}$ für $i = m_\mathsf{X}-1, m_\mathsf{X}-3, \cdots$, dann sei $m'_\mathsf{X} = m_\mathsf{X}+1$ und $A_\mathsf{X}(m_\mathsf{X}) = \emptyset$ wieder zulässig zerlegt. Existiert ein abstrakter Syntaxbaum so, dass die Attribute des Knoten K mit $Typ(K) = \mathsf{X}$ *nicht* in der Reihenfolge $A_\mathsf{X}(1), \cdots, A_\mathsf{X}(m_\mathsf{X})$ berechnet werden können. Dies steht im Widerspruch zur grundsätzlichen Eigenschaft geordneter Attributgrammatiken, dass für jedes Symbol und jeden abstrakten Syntaxbaum die Attribute in der Reihenfolge der Zerlegung berechenbar sind. □

Im Folgenden wird gezeigt, welche zusätzlichen Anforderungen an die Änderungsmengen gestellt werden, damit ein zerlegungserhaltendes Muster auch ordnungserhaltend ist.

Lemma 4.11. Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen

- A_-,
- A_+,
- R_- und
- R_+

gegeben mit $\mathsf{A}_+ = \mathsf{R}_+ = \emptyset$, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-)$ und $R' = (R \setminus \mathsf{R}_-)$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 4.1 einhalten.

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_X definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_X die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N} \text{ mit } \mathsf{X}.a \in A'_\mathsf{X}(h) \wedge \mathsf{X}.b \in A'_\mathsf{X}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Beweis. Seien $r_i \in \mathsf{R}_-$ mit $r_i = a_i \leftarrow f(\cdots)$ für $a_i \in A \cap \mathsf{A}_-$ die entfernten Attributierungsregeln und die entfernten Attribute, dann ist $IDS'_\mathsf{X} \subseteq IDS_\mathsf{X}$ für alle Symbole X, $IDP'_p \subseteq IDP_p$ für alle Produktionen p und $A'_\mathsf{X} \subseteq A_\mathsf{X}$ mit $A_\mathsf{X} = A_\mathsf{X} \uplus \cdots \uplus A_\mathsf{X}(m_\mathsf{X})$ dann ist $A'_\mathsf{X}(j) = A_\mathsf{X}(j) \setminus \{a_i : a_i \in A \cap \mathsf{A}_- \text{ und } a_i \in A_\mathsf{X}(j)\}$. Angenommen IDS'_X sei zyklisch, dann kann dies nur dann der Fall gewesen sein, dass IDS_X zyklisch gewesen ist. Dies ist im Widerspruch dazu, dass AG geordnet war. Angenommen IDP'_p wäre zyklisch, dann kann dies, ebenfalls nur der Fall gewesen sein, dass IDP_p zyklisch war, wieder im Widerspruch dazu, dass AG geordnet war. Analog ist $EDP'_p \subset EDP_p$, es werden dieselben Kanten wie in IDP_p entfernt, die Zerlegung bleibt erhalten, h, k bleiben unverändert. Wäre EDP'_p zyklisch, dann nur, wenn EDP_p zyklisch gewesen wäre. Dies steht im Widerspruch dazu, dass AG geordnet war. □

Lemma 4.11 sagt in dem Sinne aus, dass an der „Ordnung“ der Attributgrammatik durch Entfernen von Attributen und dazugehörigen Regeln, nicht zu zyklischen Bedingungen für eine geordnete Attri-

[1] Dies bezieht sich auf die Konstruktion der normalisierten direkte Abhängigkeiten $NDDP_p$ über den transitiven Abschluss von DDP_p und dem gleichzeitigen entfernen von durch diesen transitiven Abschluss eingeführten Kanten zu in dieser Produktion definierten Attributen.

butgrammatik führen kann. In den bestehenen Lemmata (wie Lemma 4.3, 4.4 oder Lemma 4.5) wurde gezeigt, dass notwendige Vorbedingungen einer geordneten Attributgrammatik – Konsistenz, Vollständigkeit, Existenz einer zulässigen Zerlegung – bei Mustern auf Basis solcher Änderungsmengen eingehalten werden. Es ist dabei zu beachten, dass aufgrund der Eigenschaften aus Satz 4.1 (insb. Eigenschaft 1) Attribute und Attributierungregeln vollständig entfernt sind und daher diese auch komplett aus der Zerlegung und den Graphen entfernt werden können.

Ein weiterer Schritt ist der Nachweis, dass das einfache, vereinbare Hinzufügen zu einer geordneten Attributgrammatik eine geordnete Attributgrammatik erzeugt. Auf die Ersetzung einer Attributierungsregel – dann werden nicht alle Attributierungsregeln für ein Attribut eines Symbols an allen Vorkommen dieses Attributs entfernt – wird an dieser Stelle keine Rücksicht genommen. Im Gegensatz zu den bisherigen Mustereigenschaften haben Muster für geordnete Attributgrammatiken zusätzliche Anforderungen. Diese zusätzliche Anforderung wird in folgendem Lemma vorgestellt und erlaubt den Nachweis, dass die induzierten Graphen zyklenfrei sind.

Lemma 4.12. Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen

- A_-,
- A_+,
- R_- und
- R_+

gegeben, mit $\mathsf{R}_- = \mathsf{A}_- = \emptyset$, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = A \cup \mathsf{A}_+$ und $R' = R \cup \mathsf{R}_+$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 4.1 einhalten und für alle $r_i, r_j \in \mathsf{R}_{+,a}$ mit $\mathsf{R}_{+,a} = \{r \colon r \in \mathsf{R}_+ \text{ und } r = a \leftarrow f(b_1, \cdots, b_n)\}$ dann ist $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_j = a \leftarrow f(b_{j_1}, \cdots, b_{j_m})$ und $\{b_{i_1}, \cdots, b_{i_n}\} = \{b_{j_1}, \cdots, b_{j_m}\}$ mit $\left|\{b_{i_1}, \cdots, b_{i_n}\}\right| = \left|\{b_{j_1}, \cdots, b_{j_m}\}\right| = \left|\{b_1, \cdots, b_n\}\right|$[2]

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_X definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_X die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N} \text{ mit } \mathsf{X}.a \in A'_\mathsf{X}(h) \wedge \mathsf{X}.b \in A'_\mathsf{X}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Beweis. Folgende Fälle sind zu betrachten:

1. IDS'_X ist zyklisch für ein $\mathsf{X} \in \Sigma$;
2. IDP'_p ist zyklisch für ein $p \in P$ oder
3. EDP'_p ist zyklisch für ein $p \in P$.

Fall 1 bedeutet, es gibt eine Kante $(b, a) \in (IDS'_\mathsf{X} \setminus IDS_\mathsf{X})$, die in diesem Zyklus enhalten ist. Somit existiert eine Kante $(\mathsf{X}.a, \mathsf{X}.b) \in IDP_p^+$ einer Produktion p mit $\lceil X \rceil_p \geq 1$. Dies steht jedoch im Widerspruch dazu, dass $a \notin AD_p$ nach Definition 4.4 und Eigenschaft 4 von Satz 4.1.

Fall 2 bedeutet analog, dass falls $(\mathsf{X}.b, \mathsf{X}.a) \in IDP'_p$, dann auch $(\mathsf{X}.a, \mathsf{X}.b) \in IDP_p^+$ (sonst wäre kein Zyklus in IDP'_p möglich), im Widerspruch zur Voraussetzung, dass die Eigenschaften nach Satz 4.1 gelten (insb. dass Hinzufügen lokal azyklisch ist).

Für Fall 3 wird die Konstruktion nach Lemma 4.6 (und Lemma 4.7) herangezogen. Sei $k = \left|\mathsf{A}_{+,\mathsf{X}}\right|$ die Anzahl der zu X hinzugefügten Attribute. Für alle $a_j \in \mathsf{A}_{+,\mathsf{X}}$ mit $1 \leq j \leq k$ ist dann

[2] In den neu hinzugefügten Regeln können die Attriute beliebig permutiert und beliebig oft vorkommen.

$$A'_{\mathsf{X}}(i) = \begin{cases} A_{\mathsf{X}}(i) & \text{falls } i \leq m_{\mathsf{X}} \\ \{a_j\} & \text{falls } i = m_{\mathsf{X}} + 2 \cdot j + 1 \text{ und } a_j \in AI(\mathsf{X}) \\ \{a_j\} & \text{falls } i = m_{\mathsf{X}} + 2 \cdot j + 2 \text{ und } a_j \in AS(\mathsf{X}) \\ \emptyset & \text{sonst} \end{cases}$$

Sei in EDP'_p ein Zyklus durch die Kante (a_u, a_v) eingeführt, wobei $1 \leq u, v \leq k$, d. h. es existieren $r_u, r_v \in \mathsf{R}_{+,p}$ mit $r_u = a_u \leftarrow f(b_{u_1}, \cdots, b_{u_n})$ und $r_v = a_v \leftarrow f(b_{v_1}, \cdots, b_{v_m})$ mit $n, m \in \mathbb{N}$. Da Fall 2 zeigt, dass IDP'_p azyklisch ist, muss (a_u, a_v) durch die Konstruktion von EDP'_p als Kante hinzugefügt worden sein. O. B. d. A. sei $a_u \in A_{\mathsf{X}}(u)$ und $a_v \in A_{\mathsf{X}}(v)$ mit $u < v$. Damit (a_u, a_v) einen Zyklus verursacht muss eines der b_{u_i} von a_v abhängig sein, d. h. $(b_{u_i}, a_v) \in IDP'_p$. Damit b_{u_i} abhängig von a_v ist, muss $a_v \in A$ der ursprünglichen Attributgrammatik sein, dies steht im Widerspruch zu den Eigenschaften von Satz 4.1. □

Bemerkung (Transitiver Abschluss)**.** Im Beweis zu Lemma 4.12 werden, wie in Abschnitt 3.3 die Abschlüsse induzierter Abhängigkeiten der Produktionen mit IDP_p^+ bezeichnet.

Die ersten beiden Fälle im Beweis zu Lemma 4.12 sagen nur aus, dass, da bereits die Eigenschaften aus Satz 4.1 gelten und für alle Produktionen und alle Symbole die Abhängigkeiten identisch sein müssen ($\{b_{i_1}, \cdots, b_{i_n}\} = \{b_{j_1}, \cdots, b_{j_n}\}$), kann so ein Zyklus in den induzierten Abhängigkeiten der Symbole oder Produktionen nur aufgrund dessen eingeführt werden, wenn ein Attribut bereits in der ursprünglichen Attributgrammatik existierte und dessen Attributierungsregel ersetzt wird. Dies ist bereits aufgrund von Definition 4.4 und der Eigenschaften von Satz 4.1 verboten.

Weiterhin kann ein Zyklus im erweiterten Abhängigkeitsgraphen (Fall 3 von Lemma 4.12) nur durch den Erweiterungsschritt eingeführt werden. Diese Kante stellt nur die Partitionierung (analog derer aus Lemma 4.7) grafisch dar. Damit diese zusätzliche Kante einen Zyklus einführt, muss eines der Attribute von denen Quelle oder Ziel der Kante abhängig sind, eben zu Quelle oder Ziel dieser Kante abhängig sein. Dies kann nur dann der Fall sein, wenn Quelle oder Ziel der Kante und die zur Berechnung notwendigen Attribute, bereits in der ursprünglichen Attributgrammatik existieren. Dies steht jedoch im Widerspruch zur Annahme von Lemma 4.12, dass $\mathsf{A}_- = \emptyset$ ist und somit Eigenschaft 1 von 4.1 nicht greift.

Folgendes Lemma beschreibt nun die in Lemma 4.12 ausgelassene Eigenschaft für ersetzende Attributierungsregeln.

Bemerkung (Unterschied folgenden Lemmas zu bisherigen Aussagen)**.** Folgendes Lemma ist notwendig, da zwar in Lemma 4.12 ausgesagt wurde, dass die hinzugefügten Attributierungsregeln nur von den gleichen Attributen abhängen darf, aber nicht, was passiert wenn solch eine Attributierungsregel bisher existiert hat und die Attributgrammatik geordnet war. Wesentlicher Unterschied sind somit die Eigenschaften der hinzugefügten Attributierungsregeln. Wesentliche Eigenschaft folgenden Lemmas ist, dass Abhängigkeiten höchstens entfernt werden können. Hinzufügen bestehender Attribute als Abhängigkeit zu einer bestehenden Attributierungsregel kann zu Zyklen in den induzierten Abhängigkeiten führen.

Lemma 4.13. Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und ein Muster durch die Änderungsmengen

- A_-,
- A_+,
- R_- und
- R_+

gegeben, sodass $AG' \triangleq (G, A', R', B)$ mit $A' = (A \setminus \mathsf{A}_-) \cup \mathsf{A}_+$ und $R' = (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist, wobei die Änderungsmengen die Eigenschaften von Satz 4.1 einhalten und für alle $r_i, r_j \in \mathsf{R}_{+,a}$ mit $\mathsf{R}_{+,a} = \{r \colon r \in \mathsf{R}_+ \text{ und } r = a \leftarrow f(b_1, \cdots, b_n)\}$ dann ist $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_j = a \leftarrow f(b_{j_1}, \cdots, b_{j_n})$ und $\{b_{i_1}, \cdots, b_{i_n}\} = \{b_{j_1}, \cdots, b_{j_n}\}$. Für $r_i \in \mathsf{R}_-$ mit $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i'_n})$ ist dann $\{b_{i_1}, \cdots, b_{i'_n}\} \subseteq$

$\{b_{i_1}, \cdots, b_{i_n}\}$ wobei $n' \leq n$, $n', n \in \mathbb{N}$. Dabei gilt $b_{i_j} \in A$ für alle $i \in \mathbb{N}$ und $i_1, \cdots, i_n, i'_n \in \mathbb{N}$ und für alle $r \in (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ ist $r \neq b \leftarrow f(\cdots, a, \cdots)$.

Für jede Produktion $p \in P$ und jedes Symbol $\mathsf{X} \in \Sigma$ sind dann die durch IDP'_p und IDS'_X definierten Graphen azyklisch, wobei IDP'_p die induzierten Abhängigkeiten zwischen Attributvorkommen der resultierenden Attributgrammatik AG' sind, analog IDS'_X die induzierten Abhängigkeiten zwischen Symbolattributen der resultierenden Attributgrammatik sind. Der erweiterte Abhängigkeitsgraph $EDP'_p \triangleq IDP'_p \cup \{\mathsf{X}.a \to \mathsf{X}.b : \exists h, k \in \mathbb{N}$ mit $\mathsf{X}.a \in A'_\mathsf{X}(h) \wedge \mathsf{X}.b \in A'_\mathsf{X}(k) \wedge h < k\}$ ist ebenfalls azyklisch.

Beweis. Folgende Fälle sind zu betrachten:

1. IDS'_X ist zyklisch für ein $\mathsf{X} \in \Sigma$;
2. IDP'_p ist zyklisch für ein $p \in P$ oder
3. EDP'_p ist zyklisch für ein $p \in P$.

Für Fall 1 gelten die Aussagen aus Lemmas 4.11 und 4.12. Sei die Kante $(b, a) \in (IDS'_\mathsf{X} \setminus IDS_\mathsf{X})$, die den Zyklus einführende Kante. Dann kann diese Kante nur existieren, weil ein $r \in (R \setminus \mathsf{R}_-) \cup \mathsf{R}_+$ mit $r = b \leftarrow f(\cdots, a, \cdots)$ ist. Dies steht im Widerspruch zur Voraussetzung, dass für alle r diese ungleich genau dieser Form sind.

Für Fall 2 folgt die Beweisführung Fall 1: Angenommen $(\mathsf{X}.b, \mathsf{Y}.a) \in IDP'_p$ für $\mathsf{X}, \mathsf{Y} \in \Sigma$ und $\lceil \mathsf{X} \rceil_p > 0$ und $\lceil \mathsf{Y} \rceil_p > 0$, und $(\mathsf{X}.b, \mathsf{Y}.a)$ erzeugt einen Zyklus in IDP'_p, so existiert ein $r \in R'_p$ mit $r = \mathsf{X}.b \leftarrow f(\cdots, \mathsf{Y}.a, \cdots)$ und mit $b = \mathsf{X}.b$ und $a = \mathsf{Y}.a$ steht dies ebenfalls im Widerspruch zur Voraussetzung.

Für Fall 3 sei $r_i \in R'_p$ und EDP_p azyklisch. Sei $a \in A_\mathsf{X}(j)$ in der Partitionierung j des Symbols X und $r_i \in \mathsf{R}_{+,a}$ und es existiert ein $r'_i \in \mathsf{R}_{-,a}$ und $r'_i \in R_p$ wobei $r'_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i'_n})$ und $\{b_{i_1}, \cdots, b_{i_n}\} \subseteq \{b_{i_1}, \cdots, b_{i'_n}\}$ und $n \leq n'$. Ohne Beschränkung der Allgmeinheit sei $n' = n + 1$ und $b_{i'_n}$ in Partitionierung $A_\mathsf{X}(k)$. Damit in der Konstruktion von EDP'_p ein Zyklus durch die Änderung von r'_i in r_i, d. h. Hinzunahme der Abhängigkeit $(b_{i'_n}, a)$ zu EDP_p ein Zyklus erzeugt werden kann muss $k > j$ sein. Wie in der Anmerkung zum Beweis von Satz 4.1 vorgestellt ist dann jedoch $a \in A_\mathsf{X}(m'_\mathsf{X})$ mit $m'_\mathsf{X} > m_\mathsf{X}$, wie ein neues Attribut. Damit folgt die Aussage aus dem Beweis zu Lemma 4.12.

Alle anderen Fälle folgen aus den Beweisen zu Lemma 4.13 und Lemma 4.12. □

Die Bedingung in Lemma 4.13 beschreibt, dass bei der Ersetzung bestehender Attributierungsregeln höchstens existierende Attribute in die neue Abhängigkeit hinzugefügt werden können. Wenn eine Attributierungsregel ersetzt und mit Abhängigkeiten erweitert wird, dann darf keine Attributierungregel existieren, für die eine Abhängigkeit zu diesem ersetzten Attribut existiert. Für die erweiterten Abhängigkeitsgraphen gilt dann, dass die ersetzten Attributierungsregeln so eingeordnet werden, als wären sie neu hinzu gekommen. Siehe für letztere Aussage auch den Absatz nach dem Beweis zu Satz 4.1.

Weiterhin ist zu zeigen, dass die Komposition von Mustern die wesentlichen Eigenschaften von Mustern erhält. Die in Abschnitt 4.2 präsentierte Variante zum Aufbau zerlegungserhaltender Muster aus unabhängigen Beschreibungen erlaubt auch die Definition ordnungserhaltender Muster unter den in diesem Abschnitt vorgegebenen Bedingungen. Die Lemmata 4.11 und 4.12 geben die dafür notwendigen Voraussetzungen an. Für eine schlichte Musteranwendung, die darüber hinaus die Eigenschaften der Lemma 4.11, 4.12 und Lemma 4.13 einhält, heißt *schlichte geordnete Musteranwendung*:

Definition 4.19. Sei $\mathcal{M}_u \triangleq (V, Pr, At)$ eine Attributgrammatik-unabhängige Musterdefinition mit $Pr = Pr_+ \cup Pr_-$ und $AG \triangleq (G, A, R, B)$ eine geordnet Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$, sowie der prädikatabhängigen Teil-Attributgrammatik (AG_+, AG_-) mit $AG_+ \triangleq (G_+, A_+, R_+, B)$ und $AG_- \triangleq (G_-, A_-, R_-, B)$, wobei $G_+ \triangleq (N, T, P_+, Z)$ und $G_- \triangleq (N, T, P_-, Z)$. Sei $S \triangleq \{\sigma_0, \cdots, \sigma_n\}, n \in \mathbb{N}$ eine Menge passender Substitutionen zu AG und $\mathcal{M}_u$. Das Tupel $\Delta_\mathcal{M} \triangleq$

$(\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ ist. Halten die Mengen nach $\mathcal{M}_u$ die Eigenschaften von Satz 4.1, die Vorraussetzungen von Lemma 4.12, Lemma 4.11 und Lemma 4.13 ein, wobei $\mathsf{A}_- = \mathcal{M}_{-,A}$, $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_- = \mathcal{M}_{-,R}$ und $\mathsf{R}_+ = \mathcal{M}_{+,R}$ so heißt $\mathcal{M}_u$ mit den Substitutionen $Subs$ **schlichte *geordnete* Musteranwendung**.

Im folgenden Abschnitt wird der Zusammenhang Attributgrammatik-unabhängiger Musterdefinitionen und Attributgrammatik-abhängiger Muster in Zusammenhang gebracht. Damit wird der Schritt von Abschnitt 4.2 und diesem Abschnitt vervollständigt. Der folgende Abschnitt stellt somit den Zusammenhang zwischen Mustern aus Änderungsmengen und schlichter (und geordneten) Musteranwendungen her.

4.4. Zusammenhang Attributgrammatik-unabhängiger und Attributgrammatik-abhängiger Muster

In Abschnitt 4.1 wurde gezeigt, was Muster sind und unter welchen Bedingungen das Hinzufügen und Entfernen von Attributierungsregeln einer konkreten Attributgrammatik die Zerlegbarkeit dieser Attributgrammatik erhält. In Abschnitt 4.2 wurde gezeigt, wie Attributgrammatik-unabhängige Muster definiert bzw. aufgebaut werden können. Darauf aufbauend wurde gezeigt, wie diese Musteranwendung zu den Mengen aus Abschnitt 4.1 führt. In Abschnitt 4.3 wurde für eben diese Mengen noch gezeigt, wie diese, angewandt auf eine geordnete Attributgrammatik, zu einer geordneten Attributgrammatik führen.

Der folgende letzte wichtige Satz verknüpft nun die in Abschnitt 4.1 vorgestellten Eigenschaften und Satz 4.1 mit der Konstruktion aus Abschnitt 4.2.

Satz 4.2. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G und $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition sowie einer Menge passender Substitutionen $Subs$. Für jede Musteranwendung $\Delta_\mathcal{M} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$ bzgl. AG und $\mathcal{M}_u$ und $Subs$ mit $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ existiert eine schlichte Musteranwendung, wobei $\mathsf{A}_+ = \mathcal{M}_{+,A}$, $\mathsf{R}_+ = \mathcal{M}_{+,R}$, $\mathsf{A}_- = \mathcal{M}_{-,A}$ und $\mathsf{R}_- = \mathcal{M}_{-,R}$ sind.

Der Beweis zu Satz 4.2 benötigt eine Reihe weiterer Hilfslemmata. Gleichwohl ist dieser Satz, neben Satz 4.1 zentral für diese Arbeit. Eine wesentliche Eigenschaft, die immer wieder in den folgenden Lemmata benutzt wird ist, dass, falls ein Attribut oder eine Attributierungsregel in A_+ oder R_+ ist, dass dann in der Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u$ in At_+ ein erweiterter Attributwertterm existiert haben muss, der, zusammen mit der Substitution und der Anwendung der Definition der Musteranwendung zum Hinzufügen in $\mathcal{M}_{+,A}$ und $\mathcal{M}_{+,R}$ geführt haben muss. Analoges für At_-. Wie in Definition 4.19 und Definition 4.17 werden die $\mathcal{M}_{+,A}$, $\mathcal{M}_{+,R}$, $\mathcal{M}_{-,A}$ und $\mathcal{M}_{-,R}$ als die Attributgrammatik-abhängigen Mengen nach Definition 4.3 verwendet.

Folgende grundsätzliche Aussage stellt sicher, dass Mengen $\mathcal{M}_{+,R}$ und $\mathcal{M}_{+,A}$ existieren, die, sind diese nicht leer, entsprechende erweiterte Attributwertterme in At_+ existiert haben müssen.

Lemma 4.14. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik und $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition mit passenden Substitutionen $Subs$. Ist $\mathcal{M}_{+,R} \neq \emptyset$ oder $\mathcal{M}_{+,A} \neq \emptyset$, dann ist $At_+ \neq \emptyset$.

Beweis. Angenommen At_+ wäre leer, dann ist zu zeigen, dass $\mathcal{M}_{+,A} = \mathcal{M}_{+,R} = \emptyset$ sind.

Für $\mathcal{M}_{+,A}$ bedeutet dies, da kein $t_{\{\mathfrak{P}\}} \in At_+$, dass auch kein $a \in [\![a'\ Subs]\!]_{AG}$ existiert, da a' ein Teil von $t_{\{\mathfrak{P}\}}$ sein muss. Nach Eigenschaft 3 muss ein $t^{(i)}_{\{\mathfrak{P}\}}$ zur Konstruktion von $\mathcal{M}_{+,R,s}$ existieren, dies ist nicht der Fall, somit ist $\mathcal{M}_{+,R,s} = \emptyset$, analog nach Eigenschaft für 4 für $\mathcal{M}_{+,R,i}$. Da nach Definition 4.16 $\mathcal{M}_{+,R} = \mathcal{M}_{+,R,s} \cup \mathcal{M}_{+,R,i}$ ist $\mathcal{M}_{+,R} = \emptyset$ wie gefordert. □

Die Umkehrung von Lemma 4.14 gilt nicht unmittelbar. Dafür sind zusätzliche Annahmen über die passenden Substitutionen notwendig. An dieser Stelle wird nicht darauf eingegangen. In den folgenden Lemmata wird gezeigt, welche Eigenschaften diese Substitutionen haben müssen, sodass die aus erweiterten Attributtermen erzeugten Mengen nicht leer sind.

In Definition 4.16 werden die erweiterten Attributwertterme anhand der Konstruktion des Attributwertterms unterschieden. Dies ist auch im Beweis der folgenden Lemmata notwendig. Gäbe es solche Terme nicht, die nach passender Substitution in den entsprechenden Mengen eingeordnet werden, dann wäre die Substitution nicht passend oder die daraus resultierende Menge leer. Auch dann würde jede Aussage dazu gelten, da für leere Mengen eine Aussage über alle Elemente der Menge trivialerweise erfüllt ist.

Ein notwendiges Lemma zur Konstruktion in den folgenden Beweisen ist nun noch zu zeigen. Grundsätzliche Aussage folgenden Lemmas ist, dass aus Attributierungsregeln einer Attributgrammatik ein erweiterter Attributwertterm erzeugt werden – mit der passenden Substitution dazu.

Lemma 4.15. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G. Sei $r \in R$ eine Attributierungsregel, dann existiert ein $t_{\{\mathfrak{P}\}}$ mit passender Substition $Subs$, sodass $r \in \mathcal{M}_{+,R}$.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u = (V, Pr_+, Pr_-, At_+, At_-)$ und Angabe einer passenden Substitution $\sigma \in Subs$.

O. B. d. A. sei $r = \textbf{rule } \mathsf{X}_0 ::= \mathsf{X}_1 \ \cdots \ \mathsf{X}_n \textbf{ attr } \mathsf{X}_k.a \leftarrow f(b_1, \cdots, b_m)$ mit $\mathsf{X}_0 \in N, \mathsf{X}_j \in \Sigma$ für $1 \leq j \leq n$, $n \in \mathbb{N}\ 0 \leq k \leq n$, $\mathsf{X}_k.a \in A$, $b_1, \cdots, b_m \in A$.

$$
\begin{aligned}
V &= \{S_0, S_1, \cdots, S_t, a_1, c_1, \cdots, c_l, g\} \\
Pr_+ &= \{true\} \\
Pr_- &= \{false\} \\
At_+ &= \{\textbf{rule } S_0 ::= S_1 \ \cdots \ S_t \textbf{ attr } S_j.a_1 \leftarrow g(c_1, \cdots, c_l)\} \\
At_- &= \emptyset \\
\sigma &= [\mathsf{X}_0/S_0, \mathsf{X}_1/S_1, \cdots, \mathsf{X}_n/S_n, \varepsilon/S_{n+1}, \cdots, \varepsilon/S_t, \\
&\qquad a/a_1, b_1/c_1, \cdots, b_m/c_m, \varepsilon/c_{m+1}, \cdots, \varepsilon/c_l, f/g, \mathsf{X}_k/S_j]
\end{aligned}
$$

für $t, l \in \mathbb{N}$, $t \geq n$, $l \geq m$.

Die Aussage folgt dann aus der Definition der Musteranwendung und der Konstruktion von $\mathcal{M}_{+,R}$. Ist $\mathsf{X}_k.a$ synthetisiert nach Eigenschaft 3, analog ist $\mathsf{X}_k.a$ ererbt nach Eigenschaft 4 von Definition 4.16. r entspricht dann jeweils genau einem Element aus den zweiten Mengen mit $t' = f(b_1, \cdots, b_m)$. □

Eine analoge Aussage zu Lemma 4.15 mit der Formulierung für zu entfernende Attributierungsregeln wird nicht geführt. Ebenfalls wird auf die Präsentation entsprechender Lemmata für zu entfernende oder hinzuzufügende Attribute verzichtet. Die Beweise sind ebenso konstruktiv führbar und folgen aus der Definition der Musteranwendung.

In den folgenden Lemmata werden die einzelnen Eigenschaften von Satz 4.1 untersucht. Somit zeigt 4.2 dann, dass *jede* Musteranwendung, d. h. jede Konstruktion der Änderungsmengen unter Einhaltung der Eigenschaften aus Definition 4.16, eine schlichte Musteranwendung ist.

Lemma 4.16. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit $Subs$ einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen *Subs* existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'- = (At_- \setminus At_-^-) \cup At_-^+$ welche Eigenschaft 1 von Satz 4.1 einhält, unter der Annahme, dass $\mathsf{A}_- = \mathcal{M}_{-,A}$ und $\mathsf{R}_- = \mathcal{M}_{-,R}$

Beweis. Angenommen $\mathcal{M}_{-,A}$ oder $\mathcal{M}_{-,R}$ halten Eigenschaft 1 von Satz 4.1 nicht ein, dann ist $\mathcal{M}_{-,A} \neq \emptyset$ oder $\mathcal{M}_{-,R} \neq \emptyset$. Zu zeigen ist, dann, es existiert ein $\mathcal{M}'_{-,A}$ oder $\mathcal{M}'_{-,R}$, das Eigenschaft 1 einhält.

Für folgende Fälle werden die Mengen $\mathcal{M}'_{-,A}$ und $\mathcal{M}'_{-,R}$ aus $\mathcal{M}_{-,A}$ und $\mathcal{M}_{-,R}$ (schrittweise) konstruiert:

1. $a \in \mathcal{M}_{-,A}$, und Regeln $r \in R$, $r = a \leftarrow f(\cdots, b, \cdots)$ ist $r \notin \mathcal{M}_{-,R}$ und es existiert kein $r' \in \mathcal{M}_{+,R}$ mit $r' = a \leftarrow f(\cdots, b, \cdots)$, und $b \notin \mathcal{M}_{-,A}$.
2. $r \in \mathcal{M}_{-,R}$ mit $r = a \leftarrow f(\cdots)$ und es existiert kein $r' \in \mathcal{M}_{+,R}$ mit $r = a \leftarrow f(\cdots)$, und $a \notin \mathcal{M}_{-,A}$.
3. $a \in \mathcal{M}_{-,A}$ ist $a \notin A$.
4. Wie Fall 1 für ein $r \notin \mathcal{M}_{-,R}$.
5. Es existiert ein $r \in \mathcal{M}_{-,R}$, $r = a \leftarrow f(\cdots)$ und es ist kein $r' \in \mathcal{M}_{+,R}$ mit $r' = a \leftarrow f(\cdots)$ dann sind alle $r'' \in R$ mit $r'' = a \leftarrow f(\cdots)$, mit $r'' \neq r$, $r'' \notin \mathcal{M}_{-,R}$.
6. Ist $b \in \mathcal{M}_{-,A}$ und es existiert kein $r \in \mathcal{M}_{+,R}$ mit $r = a \leftarrow f(\cdots, b, \cdots)$, dann sei $r' \notin \mathcal{M}_{-,R}$ mit $r' \in R$ und $r' = a \leftarrow f(\cdots, b, \cdots)$
7. Es existiert ein $r \in \mathcal{M}_{-,R}$ und $r \notin R$.

Für Fall 1 ergibt sich $\mathcal{M}'_{-,A}$ aus $\mathcal{M}_{-,A}$ vereinigt mit all jenen b aus Fall 1. In Fall 2 füge a zu $\mathcal{M}'_{-,A}$ aus Fall 2 hinzu. Für die Elemente a aus Fall 3 werden diese aus $\mathcal{M}'_{-,A}$ entfernt. Für Fall 4 ergibt sich $\mathcal{M}'_{-,R}$ aus $\mathcal{M}_{-,R}$ vereinigt mit allen r aus Fall 4. Alle r'' werden für Fall 5 in $\mathcal{M}'_{-,R}$ hinzugefügt. Im Fall 6 füge jedes r' in $\mathcal{M}'_{-,R}$ hinzu. Für r nach Fall 7 entferne diese r aus $\mathcal{M}'_{-,R}$.

Damit sind $\mathcal{M}'_{-,R}$ und $\mathcal{M}'_{-,A}$ so konstruiert, dass alle Unterpunkte von Eigenschaft 1 von Satz 4.1 einhalten. Nach Lemma 4.15 existieren entsprechende Mengen $Subs'$ und At'_+. Die in At_+ zu entfernenden und hinzuzufügenden Elemente zur Konstruktion von At'_+ können wie in Lemma 4.15 konstruiert werden. □

Mit dem Beweis von Lemma 4.16 ist bereits gezeigt, dass es zu einer Musteranwendung eine Musteranwendung gibt, die die erste Eigenschaft von Satz 4.1 einhält. Nachfolgendes Lemma zeigt, dass für eine Musteranwendung ebenso eine Musteranwendung existiert, die die Eigenschaften 2 und 3 von Satz 4.1 einhalten.

Lemma 4.17. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit *Subs* einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen *Subs* existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'+ = (At_+ \setminus At_+^-) \cup At_+^+$ $At'- = (At_- \setminus At_-^-) \cup At_-^+$ welche Eigenschaft 2 von Satz 4.1 einhält, unter der Annahme, dass $\mathsf{A}_- = \mathcal{M}_{-,A}$ und $\mathsf{R}_- = \mathcal{M}_{-,R}$, $\mathsf{R}_+ = \mathcal{M}_{+,R}$ und $\mathsf{A}_+ = \mathcal{M}_{+,A}$

Beweis. Angenommen Eigenschaft 2 gilt nicht, dann ist zu zeigen, es existiert ein $\mathcal{M}'_{+,A}$ und $\mathcal{M}'_{+,R}$.

Folgende Fälle sind möglich:

1. Es existiert ein $a \in \mathcal{M}_{+,A}$ und a ist synthetisiert für ein Symbol X und es existiert keine Regel $r = \mathsf{X}.a \leftarrow \cdots \in \mathcal{M}_{+,R}$.

2. Für ein $r \in \mathcal{M}_{+,R}$ mit $r = a \leftarrow f(\cdots)$ ist $a \notin \mathcal{M}_{+,A}$.

3. Für ein $a \in \mathcal{M}_{+,A}$, a synthetisiert für ein Symbol X ist

$$\left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right| > 1$$

für ein $p \in P$ mit $p = \mathsf{X} ::= u$, bzw. $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right| > \left|\{p\colon p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right|$.

4.
$$\left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right| \leq 1$$

und $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right| < \left|\{p\colon p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right|$.

In Fall 1 steht $r \notin \mathcal{M}_{+,R}$ im Widerspruch zur Konstruktion von $\mathcal{M}_{+,A}$ und $\mathcal{M}_{+,R}$: damit $a \in \mathcal{M}_{+,A}$ muss ein Symbol existieren, sodass eine Regel nach Punkt 3 konstruiert werden kann. Dann greift Fall 3 oder Fall 4.

In Fall 2 ergibt sich $\mathcal{M}'_{+,A}$ aus $\mathcal{M}_{+,A}$ und allen Attributen $a \notin \mathcal{M}_{+,A}$ für die die Eigenschaft aus Fall 2 zutrifft.

Ist die Anzahl der Regeln in $\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}$ größer als 1 für Fall 3, dann entferne r_i aus $\mathcal{M}'_{+,R}$ mit $2 \leq i \leq \left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right|$. Damit ist nach Definition der Menge $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right|$ (da a synthetisiert für ein Symbol X sein muss, entspricht dies genau der Vereinigung über $p \in P$ mit $p = \mathsf{X} ::= u$ für eben $\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots)\}$) oder Eigenschaft 5 von Satz 4.1 wird verletzt. Dies ist aber nicht die betrachtete Aussage.

Ist in Fall 4 für ein Symbol $\mathsf{X} \in N$ die Anzahl der Regeln geringer als die Anzahl von Produktionen mit linker Seite X und greift Fall 1 nicht, dann existiert ein $r \in \mathcal{M}_{+,R}$ mit $r \in \mathsf{R}_{+,p}$ einer Produktion p mit linker Seite X wobei $r = a \leftarrow f(\cdots)$ ist. Dann bildet sich $\mathcal{M}'_{+,R}$ aus $\mathcal{M}_{+,R}$ ohne alle solche Regeln und $\mathcal{M}'_{+,A}$ aus $\mathcal{M}_{+,A}$ ohne a.

Die Aussage folgt aus Lemma 4.15 (bzw. analoger Aussagen). □

In Lemma 4.22 werden im letzten Fall die Attributierungsregeln und das hinzugefügte Attribut entfernt. Dies ist notwendig, da nicht garantiert werden kann, dass für andere Produktionen ebenso solch eine Regel konstruiert werden kann. Dies würde nur zutreffen, wenn die rechte Seite der Attributierungsregel eine Konstante ist. Dies gilt im Allgemeinen nicht.

Analog ist das Lemma zum Nachweis, dass Eigenschaft 3 von Satz 4.1 eingehalten werden kann. Der Beweis zu Lemma 4.23 verläuft analog zum Beweis von Lemma 4.22:

Lemma 4.18. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit $Subs$ einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen $Subs$ existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'+ = (At_+ \setminus At^-_+) \cup At^+_+$ $At'- = (At_- \setminus At^-_-) \cup At^+_-$ welche Eigenschaft 3 von Satz 4.1 einhält, unter der Annahme, dass $\mathsf{A}_- = \mathcal{M}_{-,A}$ und $\mathsf{R}_- = \mathcal{M}_{-,R}$, $\mathsf{R}_+ = \mathcal{M}_{+,R}$ und $\mathsf{A}_+ = \mathcal{M}_{+,A}$

Beweis. Angenommen Eigenschaft 3 gilt nicht, dann ist zu zeigen, es existiert ein $\mathcal{M}'_{+,A}$ und $\mathcal{M}'_{+,R}$.

Folgende Fälle sind möglich:

1. Es existiert ein $a \in \mathcal{M}_{+,A}$ und a ist ererbt für ein Symbol X und es existiert keine Regel $r = \mathsf{X}.a \leftarrow \cdots \in \mathcal{M}_{+,R}$.

2. Für ein $r \in \mathcal{M}_{+,R}$ mit $r = a \leftarrow f(\cdots)$ ist $a \notin \mathcal{M}_{+,A}$.

3. Für ein $a \in \mathcal{M}_{+,A}$, a ererbt für ein Symbol X ist

$$\left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*, Y \in \mathsf{N}\}\right| > |\mathsf{X}|_p$$

für ein $p \in P$ mit $p = \mathsf{Y} ::= u\ \mathsf{X}\ v$, bzw. $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right| > \sum_{p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v} |\mathsf{X}|_p$.

4.

$$\left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*, Y \in \mathsf{N}\}\right| < |\mathsf{X}|_p$$

und $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right| < \left|\{p\colon p \in P, p = \mathsf{X} ::= u, u \in \Sigma^*\}\right|$.

In Fall 1 steht $r \notin \mathcal{M}_{+,R}$ im Widerspruch zur Konstruktion von $\mathcal{M}_{+,A}$ und $\mathcal{M}_{+,R}$: damit $a \in \mathcal{M}_{+,A}$ muss ein Symbol existieren, sodass eine Regel nach Punkt 3 konstruiert werden kann. Dann greift Fall 3 oder Fall 4.

In Fall 2 ergibt sich $\mathcal{M}'_{+,A}$ aus $\mathcal{M}_{+,A}$ und allen Attributen $a \notin \mathcal{M}_{+,A}$ für die die Eigenschaft aus Fall 2 zutrifft.

Ist die Anzahl der Regeln in $\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*\}$ größer als die Anzahl der Vorkommen des Symbols innerhalb der Produktion für Fall 3, dann entferne r_i aus $\mathcal{M}'_{+,R}$ mit $|\mathsf{X}|_p < i \leq \left|\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots), p \in P, p = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*\}\right|$. Damit ist nach Definition der Menge $\left|\{r\colon r \in \mathsf{R}_+, r = a \leftarrow f(\cdots)\}\right|$ (da a ererbt für ein Symbol X sein muss, entspricht dies genau der Vereinigung über $p \in P$ mit $p = \mathsf{Y} ::= u\ \mathsf{X}\ v$ für eben $\{r\colon r \in \mathsf{R}_{+,p}, r = a \leftarrow f(\cdots)\}$) oder Eigenschaft 5 von Satz 4.1 wird verletzt.

Ist in Fall 4 für ein Symbol $\mathsf{X} \in N$ die Anzahl der Regeln geringer als die Anzahl der rechten Seiten X über alle Produktionen mit eben rechter Seite X und greift Fall 1 nicht, dann existiert ein $r \in \mathcal{M}_{+,R}$ mit $r \in \mathsf{R}_{+,p}$ einer Produktion p mit rechter Seite X wobei $r = a \leftarrow f(\cdots)$ ist. Dann bildet sich $\mathcal{M}'_{+,R}$ aus $\mathcal{M}_{+,R}$ ohne alle solche Regeln und $\mathcal{M}'_{+,A}$ aus $\mathcal{M}_{+,A}$ ohne a.

Die Aussage folgt aus Lemma 4.15 (bzw. analoger Aussagen). □

Lemma 4.22 und Lemma 4.23 zeigen somit, dass und wie aus einer Musteranwendung eine Musteranwendung hergeleitet werden kann, die die Eigenschaften 2 und 3 von Satz 4.1 einhält.

Lemma 4.19. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit *Subs* einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen *Subs* existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'+ = (At_+ \setminus At_+^-) \cup At_+^+$ $At'- = (At_- \setminus At_-^-) \cup At_-^+$ welche Eigenschaft 4 von Satz 4.1 einhält.

Beweis. Angenommen es gibt eine Regel $r \in \mathcal{M}_{+,R}$, die nicht vereinbar hinzufügbar zu $R' = R \setminus \mathcal{M}_{-,R}$ ist. Dann ist zu zeigen, dass es $\mathcal{M}'_{+,R}$ gibt, für die jede Regel $r \in \mathcal{M}'_{+,R}$ vereinbar hinzufügbar ist.

Es existieren folgende Fälle:

1. Für $R' \cup \{r\}$ existiert eine Produktion $p \in P$ für die dann DG_p bzgl. R' zyklisch ist.

2. Ist $r = a \leftarrow f(b_1, \cdots, b_n)$ oder $r = a \leftarrow c$ für eine Konstante c dann ist $a \in AD_p$ mit $r \in \mathsf{R}_{+,p}$ der Produktion p.

3. Ist $r = a \leftarrow f(b_1, \cdots, b_n)$ und für ein b_i existiert kein q_i, sodass $b_i \in AD_{q_i}$ mit $1 \leq i \leq n$.

4. Ist $r = a \leftarrow f(b_1, \cdots, b_n)$ dann ist ein $b_i \notin (A \setminus \mathcal{M}_{-,A}) \cup \mathcal{M}_{+,A}$ mit $1 \leq i \leq n$.

Für Fall 1 ist $\mathcal{M}_u$ keine Musteranwendung nach Definition 4.16: Eigenschaft 7 ist verletzt. Dies steht im Widerspruch dazu, dass $\mathcal{M}_u$ Musteranwendung ist.

Für Fall 2 und Fall 3 ist $\mathcal{M}'_{+,R} = \mathcal{M}_{+,R} \setminus \{r\}$.

Ist in Fall 4 für alle $b_i \in A$ mit $1 \leq i \leq n$ und exitieren $q_i \in P$ sodass $b_i \in AD_{q_i}$, dann ist $b_i \in \mathcal{M}_{-,A}$. Dann ist $\mathcal{M}'_{-,A} = \mathcal{M}_{-,A} \setminus \{b_i \colon b_i \in \mathcal{M}_{-,a}, r = a \leftarrow f(b_1, \cdots, b_n), 1 \leq i \leq n\}$. Sonst ist ist $\mathcal{M}'_{+,R} = \mathcal{M}_{+,R} \setminus \{r\}$.

Die Aussage folgt aus Lemma 4.15. □

Es gibt Alternativen für die Lösung im dritten Fall, bspw. durch vorheriges Hinzufügen eben dieser Attributierungsregeln für b_i. Diese Lösung steht jedoch an dieser Stelle nicht zur Verfügung.

Bevor der Beweis zu Satz 4.2 nun geführt werden kann ist noch folgendes Lemma zur letzten Eigenschaft von Satz 4.1 nachzuweisen.

Lemma 4.20. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit $Subs$ einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen $Subs$ existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'+ = (At_+ \setminus At^-_+) \cup At^+_+$ $At'- = (At_- \setminus At^-_-) \cup At^+_-$ welche Eigenschaft 5 von Satz 4.1 einhält.

Beweis. Angenommen $\mathcal{M}_u$ hält Eigenschaft 5 von Satz 4.1 nicht ein, d. h. es existiert ein Attribut $a \in \mathcal{M}_{+,A}$ mit Regeln $r_i, r_j \in \mathcal{M}_{+,R}$ mit $r_i \neq r_j$ und $r_i = a \leftarrow f(\cdots)$ und $r_j = a \leftarrow f(\cdots)$ wobei $r_i \in \mathsf{R}_{+,p}$ und $r_j \in \mathsf{R}_{+,q}$, $p, q \in P$. Weiterhin ist dann $p = \mathsf{X} ::= u$, $u \in \Sigma^*$ und $q = \mathsf{Y} ::= u\ \mathsf{X}\ v$ mit $u, v \in \Sigma^*$ und $\mathsf{Y} \in N$. Es existiert also ein Attribut a eines Symbols $\mathsf{X} \in \Sigma$ für das a synthetisiert und ererbt vorkommt, dann wird $\mathcal{M}'_{+,R}$ gebildet als $\mathcal{M}'_{+,R} = \mathcal{M}_{+,R} \setminus \{r_j \colon r_j \in \mathsf{R}_{+,q}, q \in P, q = \mathsf{Y} ::= u\ \mathsf{X}\ v, u, v \in \Sigma^*, Y \in N, \mathsf{X} \in \Sigma\}$

Die Aussage folgt aus Lemma 4.15. □

In Lemma 4.20 werden zur Sicherstellung von Eigenschaft 5 Attributierungsregeln entfernt in denen a ererbt vorkommt. Alternativ können auch Attributierungsregeln entfernt werden, in denen a synthetisiert vorkommt. Dies ist eine Entwurfsentscheidung, die keine unmittelbaren Einfluss auf die Aussage aus Lemma 4.20 hat.

Die letzte Eigenschaft, Eigenschaft 6 folgt unmittelbar aus den bisherigen Aussagen:

Lemma 4.21. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax G, $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ eine Attributgrammatik-unabhängige Musterdefinition und $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$, wobei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ eine Musteranwendung bzgl. AG und $\mathcal{M}_u$ mit $Subs$ einer Menge passender Substitutionen zu AG und $\mathcal{M}_u$.

Für jede Musteranwendung $\mathcal{M}_u$ und Substitutionen $Subs$ existieren $\mathcal{M}'_u \triangleq (V', Pr_+, Pr_-, At_+, At'_-)$ mit $At'+ = (At_+ \setminus At^-_+) \cup At^+_+$ $At'- = (At_- \setminus At^-_-) \cup At^+_-$ welche Eigenschaft 6 von Satz 4.1 einhält.

Beweis. Zu zeigen ist $\mathcal{M}'_{+,A} \subseteq \{a \colon r = a \leftarrow f(\cdots) \in \mathcal{M}'_{+,R}$ und für alle Produktionen, alle $a \in \mathcal{M}'_{+,A}$ und alle Produktionen $p \in P$ existiert ein $r \in \mathcal{M}'_{+,R}$ mit $r \in \mathsf{R}_{+,p}$ und $r = a \leftarrow f(\cdots)$. Beide Teilaussagen folgen aus Lemma 4.22 und Lemma 4.23. In Eigenschaft 6 ist (implizit) gefordert, dass die ererbt hinzugefügten Attribute nicht synthetisiert für dasselbe Attribut vorkommen, dies folgt aus der Konstruktion in Lemma 4.20.

Damit folgt die Aussage. □

Letztendlich erlauben die hier vorgestellten Beweise der Lemmata den Beweis zu Satz 4.2.

Beweis zu Satz 4.2. Zu zeigen ist also, für jede Musteranwendung existiert eine schlichte Musteranwendung, d. h. jede Eigenschaft von Satz 4.1 wird eingehalten.

Eigenschaft 1 ist durch Lemma 4.16 gezeigt; Eigenschaften 2 und 3 gezeigt durch Lemma 4.22 und Lemma 4.23; Eigenschaft 4 durch Lemma 4.19 sowie Eigenschaft 5 gezeigt durch Lemma 4.20. Eigenschaft 6 ist gezeigt durch Lemma 4.21.

Damit folgt Satz 4.2. □

Nachdem nun der Zusammenhang zwischen Attributgrammatik-unabhängigen Musterdefinitionen und Mustern als Änderungsmengen gezeigt ist, können darauf aufbauend Basismuster gefunden und diese konstruiert werden.

4.5. Basisoperationen, Basismuster sowie deren Darstellungsform(en)

Basismuster stellen die grundlegenden Formen von Attributierungsregeln in Attributgrammatiken dar. Für die Definition von Mustern sind darüber hinaus noch, die bisher über Mengenoperationen definierten, Operationen auf Attributgrammatiken notwendig. Folgende drei Basisoperationen existieren auf Attributgrammatiken:

1. Löschen bestehender Attributierungsregeln
2. Hinzufügen neuer Attributierungsregeln
3. Ändern bestehender Attributierungsregeln

Letztere *Operation* lässt sich durch die hintereinander Ausführung des Löschens und des Hinzufügens ausdrücken, sodass nur Löschen und Hinzufügen im Rahmen dieser Arbeit von Relevanz sind und eigene Muster bekommen.

Für das Hinzufügen von Attributierungsregeln existieren bereits eine ganze Reihe von Arbeiten, die solche Attributierungen vorstellen. Einen großen Überblick liefert [101]. In der Regel wird diese Quelle, aufgrund der Betrachtung als „Paradigmen“, in der Angabe zu den Mustern vorgezogen. Gleichwohl existieren frühere Arbeiten, die bestimmte Mechanismen oder Attributierungen vorstellen, wie [73], die auch in [101] vorgestellt werden. So sind zwar Kettenattributierungen erstmals in dieser Form in [76] vorgestellt, sind aber ursprünglich in [93] vorgestellt. Im Allgemeinen wird bei den Basismustern dann nur die Quelle angegeben, an deren Darstellung sich das Muster orientiert. Alternative, frühere Quellen werden nicht angegeben.

Somit ist das erste Basismuster das Löschen bestehender Attributierungsregeln.

4.5.1. Löschoperation(en) als Basismuster

Grundsätzliches Ziel einer Löschoperation ist das unmittelbare Löschen einer Attributierungsregel. Aufgrund der geforderten Eigenschaften aus Satz 4.1 wäre es jedoch notwendig *alle* Attribute mit zu löschen,

```
1  rule Program ::= Stats
2  attr Program.stats ← Stats.stats + 0
3  rule Stats ::= Stats Stat
4  attr Stats1.stats ←Stats2.stats + Stat.stats
5  rule Stats ::= Stat Stats attr
6  attr Stats1.stats ←Stat.stats + Stats2.stats
7  rule Stats ::= Stat
8  attr Stats.stats ← Stat.stats + 0
9  rule Stat ::= VarStat
10 attr Stats.stats ← VarStat.cnt + 0
11 rule VarStat ::= id id
12 attr VarStat.cnt ← 100
13 rule VarStat ::= id number
14 attr VarStat.cnt ← 10
15
16 delete VarStat.cnt ← 10 in VarStat ::= id number
```

Beispiel 4.9 – Resultierende Attributgrammatik aus Beispiel 4.8 (Zeilen 1 - 14) und Löschanweisung (Zeile 16) für eine Implementierung einer Sprache zur Nutzung von Mustern auf Attributgrammatiken

die von diesem Attribut abhängig sind. Die Realisierung als Attributgrammatik-unabhängige Musterdefinition stellt folgende Definition des Basismusters dar.

Basismuster 1. ***Löschen einer Attributierungsregel***

Die Attributgrammatik-unabhängige Musterdefinition

$$
\begin{aligned}
V &= \{S_0, S_1, S_2, \cdots, S_n, a, b_1, \cdots, b_m\} \\
Pr_+ &= \emptyset \\
Pr_- &= \{S_0 ::= S_1 \ \cdots \ S_n\} \\
At_+ &= \emptyset \\
At_- &= \{\textbf{rule } S_0 ::= S_1 \ \cdots S_n \textbf{ attr } a \leftarrow f(b_1, \cdots, b_m)\}
\end{aligned}
$$

beschreibt das Muster ***Löschen einer Attributierungsregel*** *für* $n, m \in \mathbb{N}$.

Die Anwendung von Basismuster 1 löscht somit, in Abhängigkeit von der dazu verwendeten Substitution, eine oder mehrere Attributierungsregeln sowie ggf. die dazugehörigen Attribute. Die Implementierung von Mustern ist nicht Gegenstand dieser Arbeit, jedoch bietet Anhang D dazu einen kurzen Überblick.

Ein Beweis, dass Löschen ein (zerlegungserhaltendes) Muster ist, ist nicht notwendig. Die Darstellung in Basismuster 1 ist nach Definition 4.17 eine Attributgrammatik-unabhängige Musterdefinition.

Beispiel 4.9 gemeinsam mit Basismuster 1 zeigt bereits in Ansätzen, die in dieser Arbeit verwendete Darstellungsform von Mustern. Diese Darstellungsform entspricht im wesentlichen der bereits in [18, 19] verwendeten Form. Eine Zusammenfassung dieser Darstellung folgt im folgenden Abschnitt.

4.5.2. Darstellung von Mustern und Basismustern

Zur vereinfachten und intuitiveren Darstellung der in dieser Arbeit präsentierten Muster wird in diesem Abschnitt eine Darstellung eingeführt, die an eine Implementierung angelehnt ist. Eine mögliche Grundlage solch einer Implementierung ist in Anhang F vorgestellt.

Statt Variablen und Substitution zu verwenden, um Muster darzustellen, werden „Meta-Produktionen“ und „Meta-Attribute“ verwendet. Dabei werden die „Muster-Anweisungen“ – bspw. **delete** – auf sol-

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr
7  rule VarStat ::= id number attr
8
9  symbol VarStat attr ↑cnt ← 10
```

a) Attributgrammatik und Nutzung von Basismuster 2

$$
\begin{aligned}
V &= \{S_0, S_1, \cdots, S_n, a, b_1, \cdots, b_m\} \\
Pr_+ &= \{S_0 ::= S_1 \ \cdots \ S_n\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{symbol}\ S_0\ \textbf{attr}\ a \leftarrow f(b_1, \cdots, b_m)\} \\
At_- &= \emptyset
\end{aligned}
$$

b) Darstellung als Attributgrammatik-unabhängiges Muster

$$
\begin{aligned}
Subs = \{ &\left[\texttt{VarStat}/S_0, \texttt{id}/S_1, \texttt{id}/S_2, \varepsilon/S_3, \cdots, \varepsilon/S_n\right], \\
&\left[\texttt{VarStat}/S_0, \texttt{id}/S_1, \texttt{number}/S_2, \varepsilon/S_3, \cdots, \varepsilon/S_n\right], \\
&\left[\texttt{VarStat.cnt}/a, 10/f(b_1, \cdots, b_m)\right]\}
\end{aligned}
$$

c) Explizite Substitutionen zur Herstellung der Anwendung von Basismuster 2 in der Darstellung aus 4.13b bzw. Definition 4.13. In Basismuster 2 bzw. der Anwendung ist die Substitution implizit enthalten.

```
1  rule VarStat ::= id id attr
2  rule VarStat ::= id number attr
```

d) AG_+, ermittelt nach Anwendung der Substitutionen aus 4.10c.

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr VarStat.cnt ← 10
7  rule VarStat ::= id number attr VarStat.cnt ← 10
```

e) Ergebnis nach Anwendung von Basismuster 2 unter Verwendung von AG_+ aus 4.10d und den Substitutionen aus 4.10c

Beispiel 4.10 – Symbolberechnungen aus Basismuster 2 angewandt auf die Attributgrammatik aus Beispiel 4.8 zur Gegenüberstellung der Quelltext-artigen Darstellung und der Attributgrammatik-unabhängigen Darstellung typischer Muster.

chen Meta-Produktionen und Meta-Attributen in andere Meta-Produktionen und Meta-Attribute überführt.

Anhand folgenden Basismusters, Basismuster 2 wird die Darstellungsform dieser Arbeit demonstriert.

Basismuster 2. ***Synthetisierte Symbolattributierung*** *(vorgestellt in [73])*

Sei $AG \triangleq (G, A, R, B)$ *eine zerlegbare Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für alle Symbole* $\mathsf{X} \in \Sigma$ *mit beliebigem Attribut* $\mathsf{X}.b \in AS_{\mathsf{X}}$ *und allen Produktionen* $p : \mathsf{X} ::= u \in P$, $\mathsf{X} \in N$, $u \in \Sigma^*$ *steht:*

```
symbol X
attr ↑b ← e
```

für

```
rule p: X ::= u
attr X.b ← e
```

Dabei ist `e` *ein beliebiger Ausdruck in denen das Attribut* `X.b` *nicht verwendet wird. Für Terminalsymbole* $\mathsf{X} \in T$ *sei* $u = \varepsilon$ *angenommen.*

Den Zusammenhang von Basismuster 2 und (zerlegungserhaltenden) Mustern, stellt folgendes Lemma her:

Lemma 4.22. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Basismuster 2 ein Muster.

Beweis. Durch Angabe der Attributgrammatik-unabhängigen Musterdefinition zu Basismuster 2 $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$\begin{aligned} V &= \{S, S_0, \cdots, S_n, a_1, a_2, b_{d_1}, \cdots b_{d_m} \\ &\quad\; b_1, \cdots, b_k, c_1, \cdots, c_j, f, g\} \\ Pr_+ &= \{\forall p : p = S ::= S_1 \ \cdots \ S_n \wedge \ \neg \ a_1 \leftarrow f(b_{d_1}, \cdots, b_{d_m})\} \\ Pr_- &= \emptyset \\ At_+ &= \{\textbf{symbol } S \textbf{ attr } \uparrow .a_1 \leftarrow f(b_1, \cdots, b_k)\} \\ At_- &= \emptyset \end{aligned}$$

wobei $n, m, k \in \mathbb{N}$ sind. □

Basismuster 3. ***Ererbte Symbolattributierung*** *(vorgestellt in [73])*

Sei $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für alle Symbole $\mathsf{X} \in \Sigma$ und $\mathsf{Y} \in N$ mit $p \in P$, $p = \mathsf{Y} ::= u\ \mathsf{X}_i\ v$, $u, v \in \Sigma^*$ und $|\mathsf{X}|_p = n$ steht

```
symbol X
attr ↓a ← e
```

für

```
rule p: Y ::= u X u
attr X1.a ←e
     ···
     Xn.a ←e
```

Dabei ist e *ein beliebiger Ausdruck in denen das Attribut* X.a *nicht verwendet wird.*

Für die Attributgrammatik-unabhängige Darstellung eines Muster sind die Variablen anzugeben und herzuleiten. Jedoch kommen in Basismuster 2 und Basismuster 3 keine Variablen vor, ebenfalls sind keine zu ersetzenden Terme aufgeführt. Generell entsprichen Basismuster 2 und Basismuster 3 den ersten beiden Formen erweiterter Attributwertterme aus Definition 4.12. Ursprünglich vorgestellt wurden Symbolattributierungen von Kastens und Waite in [73] als syntaktische Erweiterung. Damit folgt die Darstellung in Basismuster 2 und Basismuster 3 eher der Darstellung von [73] als den bisherigen Definitionen. Folgendes Lemma schließt diese Lücke für Basismuster 3, nachdem bereits Lemma 4.22 dies für Basismuster 2 gezeigt hat:

Lemma 4.23. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Basismuster 3 ein Muster.

Beweis. Durch Angabe der Attributgrammatik-unabhängigen Musterdefinition zu Basismuster 3 $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$\begin{aligned} V &= \{S, S_0, \cdots, S_n, a_1, c_{d_1}, \cdots, c_{d_l}, \\ &\quad\; b_1, \cdots, b_k, c_1, \cdots, c_j, f\} \\ Pr_+ &= \{\forall q : q = S_0 ::= S_1 \cdots \ S_{i-1} \ S \ S_{i+1} \ \cdots \ S_n \wedge \ \neg \ a_1 \leftarrow f(c_{d_1}, \cdots, c_{d_l})\} \\ &\quad\; f \\ At_+ &= \{\textbf{symbol } S \textbf{ attr } \downarrow .a_1 \leftarrow f(c_1, \cdots, c_j)\} \\ At_- &= \emptyset \end{aligned}$$

Wobei $n, l, j \in \mathbb{N}$ sind. □

Durch Auswahl einer passenden Substitution und der Definition der Musteranwendung (Def. 4.16) definiert Basismuster 2 ein ordnungserhaltendes bzw. zerlegungserhaltendes Muster; analog Basismuster

3. Die Lemma 4.22 und 4.23 zeigen die Definition als Attributgrammatik-unabhängige Musterdefinition. In der Literatur wird üblicherweise gefordert, dass Regelattributierungen Symbolattributierungen überschreiben [73]. Für die Muster dieser Arbeit ist die hier präsentierte Variante ausreichend.

In der Darstellung der Basismuster 2 und 3 sind implizit bereits die Stellen für die Substitution vorgegeben, d. h. in einer Implementierung können die substituierten Variablen auf Basis der dargestellten abstrakten Syntax hergeleitet werden. Beispiel 4.10 stellt präsentiert Attributgrammatik und synthetisierte Symbolattributierung mit den passenden Substitutionen vor.

Die Darstellung der Basismuster 2 und 3 erlaubt die Definition von Attributierungsregeln auch mit Rückgriff auf bestehende Attribute. Diese Basismuster entsprechen somit nicht nur der den ersten beiden Formen eines erweiterten Attributwertterms, sondern formalisieren die Beschreibung von Symbolberechnungen, wie diese in [73, 101] vorkommen.

Der Beitrag dieser Arbeit ist an dieser Stelle bereits die formale Darstellung dieses, bereits bekannten, Basismusters. An dieser Stelle muss ebenfalls erwähnt werden, dass eine abstrakte Syntax bei der Implementierung der Muster bereits die zu substituierenden Variablen vorgibt und bei Anwendung auf eine konkrete Attributgrammatik der Entwickler eines Übersetzers die Substitution angibt. Bezug nehmend auf Basismuster 2 und Basismuster 3 zeigt Abbildung 4.1 eine mögliche abstrakte Syntax. Die Herleitung der Änderungsmengen für die schlichte geordnete Musteranwendung ist dann über eine Auswertung der Prädikate für alle Produktionen bzw. alle Attributierungsregeln möglich.

$\langle SymbolComp \rangle$ ::= ‘`symbol`’ $\langle SubstedSymbolVar \rangle$ ‘`attr`’ $\langle SymbolAttributions \rangle$

$\langle SymbolAttributions \rangle$ ::= $\langle SymbolInherited \rangle$
| $\langle SymbolSynthesized \rangle$

$\langle SymbolInherited \rangle$ ::= $\downarrow$ ‘`.`’ $\langle SubstedDefAttribute \rangle \leftarrow \langle Computation \rangle$

$\langle SymbolSynthesized \rangle$ ::= $\uparrow$ ‘`.`’ $\langle SubstedDefAttribute \rangle \leftarrow \langle Computation \rangle$

$\langle Computation \rangle$ ::= $\langle SubstedFunctionVariable \rangle$ ‘`(`’ $\langle SubstedAttributeReferences \rangle$ ‘`)`’

$\langle SubstedAttributeReferences \rangle$ ::= ε

$\langle SubstedAttributeReferences \rangle$ ::= $\langle SubstedAttributeReferences \rangle$ ‘`,`’ $\langle SubstedAttributeReference \rangle$

$\langle SubstedAttributeReferences \rangle$ ::= $\langle SubstedAttributeReference \rangle$

Abbildung 4.1. – Mögliche abstrakte Syntax für die Beschreibung der Symbolattributierung als Muster zur Angabe der Substitution durch Übersetzer-Entwickler.

Die Herleitung der Substitution aus der, im Beweis zu Lemma 4.22 oder Lemma 4.23 angegebenen, Attributgrammatik-unabhängigen Musterdefinition ist nicht Gegenstand dieser Arbeit. In Anhang D.2 wird auf diese Herleitung auf Basis der abstrakten Syntax aus Abbildung 4.1 eingegangen.

Basismuster dieser Arbeit sind somit Instanzen von Attributgrammatik-unabhängigen Musterdefinitionen mit einem dazugehörigen (impliziten) Algorithmus zur Herleitung der Substitutionen unter Beachtung der Eigenschaften von Mustern und Musteranwendungen.

Darüber hinaus lässt sich für Basismuster 2 noch folgende wesentliche Aussage zeigen:

Satz 4.3. Für jede geordnete Attributgrammatik AG existiert eine geordnete Musteranwendung von Basismuster 2 und Basismuster 3.

Beweis. Sei Basismuster 2 als Attributgrammatik-unabhängige Musterdefinition $\mathcal{M}_u$ wie im Beweis zu Lemma 4.22 gegeben und sei $Subs$ eine passende Substitution für AG bzgl. $\mathcal{M}_u$, AG geordnet.

Für $\mathcal{M}_u$ existiert dann eine schlichte Musteranwendung $\mathcal{M}'_u$ nach Satz 4.2. Da laut Basismuster 2 keine Attribute und Attributierungsregeln zu entfernen sind, werden nur Attributierungsregeln hinzugefügt.

Angenommen dieses $\mathcal{M}'_u$ ist keine geordnete Musteranwendung, dann existieren (nach Definition 4.19) folgende Fälle:

1. die Voraussetzungen nach Lemma 4.11 sind verletzt;
2. die Voraussetzungen nach Lemma 4.12 sind verletzt;
3. die Voraussetzungen nach Lemma 4.13 sind verletzt oder
4. die Eigenschaften von Satz 4.1 werden verletzt.

Fall 1 und Fall 3 sind ausgeschlossen, da $\mathcal{M}'_{-,A} = \mathcal{M}'_{-,R} = \emptyset$ ist – Attribute und Attributierungsregeln werden nach Basismuster 2 weder gelöscht noch ersetzt. Dies ist ebenfalls nicht nach Lemma 4.22 vorgesehen.

Angenommen Fall 2 trifft zu, dann ist zu zeigen, dass eine Musteranwendung $\mathcal{M}''_u$ existiert, die die Voraussetzungen nach Lemma 4.12 einhält. Fall 2 heißt, es existieren Regeln $r_i, r_j \in \mathsf{R}_+$ wobei $r_i = a \leftarrow f(b_{i_1}, \cdots, b_{i_n})$ und $r_j = a \leftarrow f(b_{j_1}, \cdots, b_{j_n})$ ist und $\{b_{i_1}, \cdots, b_{i_n}\} \neq \{b_{j_1}, \cdots, b_{j_n}\}$. Für $\mathsf{R}_{+,a} = \{r \colon r = a \leftarrow f(b_1, \cdots, b_n)$ und $r \in \mathsf{R}_+\}$ und $Deps_r = \{b_i \colon r = a \leftarrow f(b_1, \cdots, b_n)$ für $1 \leq i \leq n\}$ ist dann $D' = \bigcup_{r \in \mathsf{R}_{+,a}} Deps_r$ die Obermenge der Abhängigkeiten zur Attributierung von a. Dann lässt sich $\mathsf{R}_{+,a}$ auch konstruieren als $\mathsf{R}'_{+,a} = \{r_i = a \leftarrow f'_i(d_1, \cdots, d_l), 1 \leq i \leq \left|\mathsf{R}_{+,a}\right|\}$ mit

$$f'_i = f(b_{i_1}, \cdots, b_{i_n}) \twoheadleftarrow d_1, \cdots, d_l, d_i \in D', 1 \leq i \leq l$$

Damit sind die Abhängigkeiten eines Attributs a identisch in jedem Kontext. Nach Satz 4.2 existiert zu dieser Musteranwendung eine Musteranwendung, die die Eigenschaften von Satz 4.1 erfüllt. Mit dieser Konstruktion und dieser Aussage sind die Vorbedingungen von Lemma 4.12 erfüllt. Damit folgt die Aussage.

Für Fall 4 steht im Widerspruch zu Satz 4.2, da $\mathcal{M}'_u$ bereits die Eigenschaften von Satz 4.1 einhält.

Somit existiert eine geordnete Musteranwendung für Basismuster 2 wie gewünscht.

Der Beweis für $\mathcal{M}_u$ nach Lemma 4.23 verläuft identisch. □

Im Beweis zu Satz 4.3 werden die zusätzliche Abhängigkeiten einer Attributierungsregel mit dem, bereits eingeführten, Symbol $\twoheadleftarrow$ hinzugefügt. Analog lässt sich eine Funktion f' statt f verwenden, die diese zusätzlichen Abhängigkeiten – gelesenen Attribute – als Argumente verwendet. Die Funktion f' würde dann f mit den ursprünglichen Attributen aufrufen und die zusätzlichen Argumente ignorieren.

4.5.3. Komposition von Mustern und deren Darstellungen

Muster, bzw. Musteranwendungen sind kombinierbar. Dies folgt unmittelbar aus der Definition eines Musters – siehe Definition 4.1. Da jedoch jede Musteranwendung die zugrundeliegende Attributgrammatik ändert, ist die Reihenfolge der Musteranwendung relevant. Die Komposition von Mustern kann somit als „Meta-Muster“ verstanden werden und gleichwohl auch als Basismuster.

Definition 4.20. Seien $\mathcal{M}_{u,1}$ und $\mathcal{M}_{u,2}$ Attributgrammatik-unabhängige Musterdefinitionen und $AG \triangleq (G, A, R, B)$ eine Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$. Für die Mengen passender Substitutionen $Subs_1$ und $Subs_2$ ist

$$\mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}(AG, Subs_1, Subs_2) = \mathcal{M}_{u,2}(\mathcal{M}_{u,1}(AG, Subs_1), Subs_2)$$

Folgende Musterdefinitionen werden verwendet: $\mathcal{M}_{u,s1}$ entspricht der initialen Symbolattributierung wie in Beispiel 4.10, analog ist $Subs_1$ entsprechend der in Beispiel 4.10c aufgeführten Substitution. Weiterhin entspricht $\mathcal{M}_{u,d}$ der Löschung

```
1    delete VarStat.cnt ← 10 in VarStat ::= id number
```

allerdings mit der Einschränkung, dass $Pr_{-,d}$ nur zu wahr in genau dieser Produktion wird. Dann ist $\mathcal{M}_{u,s2}$ mit $Pr_{+,s2} = Pr_{-,d}$ wieder die Symbolattributierung

```
1    symbol Varstat attr ↑cnt ← 100
```

Somit ist S_2 für $\mathcal{M}_{u,s2}$ und $\mathcal{M}_{u,d}$ identisch. Die resultierende Attributgrammatik

```
1  rule VarStat ::= id id attr VarStat.cnt ← 10
2  rule VarStat ::= id number attr VarStat.cnt ← 100
```

gebildet aus $\mathcal{M}_{u,s2} \circ \mathcal{M}_{u,d}(\mathcal{M}_{u,s1}(AG, Subs_1), S_2)$

Beispiel 4.11 – Komposition von Symbolberechnungen und Löschung zur Erzeugung der Berechnungen für `VarStat.cnt` wie in Beispiel 4.8.

die **Komposition Attributgrammatik-unabhängiger Musterdefinitionen bzgl.** AG **und** $Subs_1$ **sowie** $Subs_2$.

Ist $Subs_1 = Subs_2$ entsteht ein Spezialfall von Definition 4.20 auf den hier nicht tiefer eingegangen wird. Sind Attributgrammatik und Substitution je Musteranwendung bei der Komposition aus dem Kontext erkennbar wird auf die Angabe dieser verzichtet. Die Musterkomposition ist im allgemeinen weder kommutativ noch assoziativ. Zur einfacheren Darstellung steht $M_{u,n} \circ \cdots \circ M_{u,1}(AG, Subs_n, \cdots, Subs_1)$ für $M_{u,n} \circ (M_{u,n-1} \circ (\cdots \circ (M_{u,2} \circ M_{u,1}) \cdots))(AG, Subs_n, \cdots, Subs_1)$ mit Anwendung der jeweiligen geklammerten Paare wie in Definition 4.20.

Die Musterkomposition hat einen breiten Anwedungsbereich. Definition 4.20 kann genutzt werden um initial mit Symbolattributierung (Basismuster 2 oder Basismuster 3) Attributierungsregeln für ein Symbol festzulegen, wie in Beispiel 4.10 geschehen. Durch eine Komposition mit einer weiteren Symbolattributierung und einer vorherigen Löschung auf denselben Symbolen und einer Produktion, kann die in Beispiel 4.8 gefundenen Attributierungsregeln für die Produktionen mit linker Seite `VarStat` erzeugt werden. Beispiel 4.11 stellt dies genauer vor.

Weiterhin ist zur Definition weiterer Muster folgender Abschluss der Komposition notwendig:

Definition 4.21. Sei $AG \triangleq (G, A, R, B)$ eine zerlegbare Attributgrammatik mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$. Für beliebige anwendbare Substitutionen $Subs_1, \cdots, Subs_n$ ist der Abschluss der Musteranwendung definiert als

$$\begin{aligned}
\mathcal{M}_u^{(0)} &= AG \\
\mathcal{M}_u^{(1)} &= \mathcal{M}_u(AG, Subs_1) \\
\mathcal{M}_u^{(i+1)} &= \mathcal{M}_u \circ \mathcal{M}_u^{(i)}(AG, Subs_i, Subs_{i+1}) \\
\mathcal{M}_u^{+} &= \mathcal{M}_u^{(n)}
\end{aligned}$$

für $n \in \mathbb{N}, n > 0$.

Die letzte Zeile der Formeln in Definition 4.21 stellt den Zusammenhang zwischen den Substitutionen und den Musteranwendungen dar. Letztendlich sind folgende zwei Lemmata für die folgenden Nachweise dieser Arbeit notwendig.

Lemma 4.24. Sei $AG \in \mathcal{AG}_G$ eine zerlegbare Attributgrammatik mit abstrakter Syntax G und $\mathcal{M}_{u,1}$ und $\mathcal{M}_{u,2}$ (zerlegungserhaltende) Attributgrammatik-unabhängige Musterdefinitionen und $Subs_1$ und $Subs_2$ Mengen passender Substitutionen. Die Musterkomposition $\mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}(AG, Subs_1, Subs_2)$ ist zerlegungserhaltend.

Beweis. Angenommen $AG'' = \mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}(AG, Subs_1, Subs_2)$ wäre nicht zerlegbar, dann existieren drei Möglichkeiten:

1. $AG' = \mathcal{M}_{u,1}(AG, Subs_1)$ ist nicht zerlegbar;
2. AG' ist zerlegbar aber AG'' nicht, wobei nach Definition 4.20 $AG'' = \mathcal{M}_{u,2}(AG', Subs_2)$; sowie
3. AG ist nicht zerlegbar.

Fall 3 steht im Widerspruch zur Voraussetzung, dass AG zerlegbar ist; Fall 2 steht im Widerspruch dazu, dass $\mathcal{M}_{u,2}$ eine Attributgrammatik-unabhängige Musterdefinition ist und $Subs_2$ passend ist und Fall 1 steht ebenfalls im Widerspruch zur Voraussetzung, dass AG zerlegbar und $\mathcal{M}_{u,1}$ eine Attributgrammatik-unabhängige Musterdefinition mit passenden Substitutionen $Subs_1$ ist. □

Analog erfolgt der Beweis für folgendes Lemma:

Lemma 4.25. Sei $AG \in \mathcal{AG}_G$ eine geordnete Attributgrammatik mit abstrakter Syntax G und $\mathcal{M}_{u,1}$ und $\mathcal{M}_{u,2}$ (ordnungserhaltende) Attributgrammatik-unabhängige Musterdefinitionen und $Subs_1$ und $Subs_2$ Mengen passender Substitutionen. Die Musterkomposition $\mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}(AG, Subs_1, Subs_2)$ ist ordnungserhaltend.

Beweis. Angenommen $AG'' = \mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}(AG, Subs_1, Subs_2)$ wäre nicht geordnet, dann existieren folgende Fälle:

1. $AG' = \mathcal{M}_{u,1}(AG, Subs_1)$ ist nicht geordnet; sowie
2. AG' ist geordnet, aber AG'' nicht;

Für Fall 1 ist entweder AG bereits nicht geordnet, was im Widerspruch zur Voraussetzung steht, oder $\mathcal{M}_{u,1}$ ist nicht ordnungserhaltend, ebenfalls im Widerspruch zur Voraussetzung.

Fall 2 steht im Widerspruch zur Voraussetzung, dass $\mathcal{M}_{u,2}$ ordnungserhaltend ist. □

4.5.4. Weitere grundlegende Muster

Dieser Abschnitt beschreibt weitere grundlegende Muster, die zum Teil Komposition benötigen. Die in diesem Abschnitt vorgestellten Muster sind zwingend notwendig für die einfachere Darstellung im weiteren Teil der Arbeit, lassen sich jedoch durch Komposition mit Basismustern ersetzen. Üblicherweise sind die in diesem Abschnitt vorgestellten Muster bereits in anderen Arbeiten in anderer Form vorgestellt.

Lemma 4.26. Sei $\mathcal{M}_{u,1}$ eine Attributgrammatik-unabhängige Musterdefinition nach Lemma 4.22 und $\mathcal{M}_{u,2}$ eine Attributgrammatik-unabhängige Musterdefinition nach Lemma 4.23. $\mathcal{M}_s = \mathcal{M}_{u,2} \circ \mathcal{M}_{u,1}$ ist ein ordnungserhaltendes Muster.

Beweis. $\mathcal{M}_s$ ist zerlegungserhaltend nach Lemma 4.22, Lemma 4.22 und Lemma 4.24. $\mathcal{M}_s$ ist ordnungserhaltend nach Satz 4.3 und Lemma 4.25. Damit folgt die Aussage. □

Neben Symbolberechnungen existiert ein weiterer, bereit in [73] bzw. [101] vorgestellter Ansatz um Attributierungen auszudrücken – Kettenberechnungen. Grundlage für dieses Basismuster ist dann in dieser Arbeit die dritte Form erweiterter Attributwertterme (siehe Def. 4.12) in mehreren Instanzen.

Kettenberechnungen finden vielfältige Verwendung in Attributgrammatiken – in Beispiel 1.2 werden diese verwendet um die Namensanalyse durchzuführen. In [27] hingegen wird unter Verwendung solcher

Kettenattributierungen gezeigt, dass es zerlegbare Attributgrammatiken gibt, die nicht geordnet sind. Wenngleich bereits Kastens in [75] gezeigt hat, dass sich diese durch Hinzufügen zusätzlicher Abhängigkeiten der Attributierung in geordnete Attributgrammatiken überführen lassen.

Typisches Muster 1. ***Kettenberechnungen*** *(u. a. [101])*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und* $\mathsf{X}_0 \in N$, $\mathsf{X}_i \in \Sigma$, $i \in [1, n]$, $n \in \mathbb{N}$ *mit* $p \in P, p : \mathsf{X}_0 ::= \mathsf{X}_1 \cdots \mathsf{X}_n$, *dann ist*

```
chain a head X0 ←e
```

äquivalent mit

```
rule X0 ::= X1 ··· Xn
attr X1.aIn ←e
     X2.aIn ←X1.aOut
     ···
     Xn.aIn ←Xn−1.aOut
     X0.aOut ←Xn.aOut

chain a head X1 ←X1a.In
···
chain a head Xn ←Xna.In
```

Für alle $q \in P, q : \mathsf{X} ::= \varepsilon$ *ist*

```
chain a head X ← e
```

äquivalent mit

```
rule X ::= ε
attr X.aOut ← X.aIn
```

Das Muster der Kettenberechnungen (Typisches Muster 1) stellt bereits implizit eine Kombination unter Verwendung von Basismuster 2, Basismuster 3 mit der dritten Form von Definition 4.12 dar. Das Muster der Kettenberechnungen beschreibt somit die Initialisierung in einem Symbol und eine Art Links-Tief-Rechts Attributierungsform. Statt einer vollständigen Rückführung auf einzelne Regelattributierungen wird in dieser Arbeit eine Attributgrammatik-unabhängige Musterdefinition unter Verwendung der Komposition von Basismustern verwendet. Die Implementierung der zugrundeliegenden Spezifikationssprache wird in Anhang F präsentiert. Ähnliche Konzepte können in einer, darauf aufbauenden, Transformationssprache herangezogen werden.

Folgendes Lemma zeigt, dass das Typische Muster 1 sich als Attributgrammatik-unabhängiges Muster darstellen lässt.

Lemma 4.27. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 1 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$:

$$
\begin{aligned}
V &= \{S_0, \cdots, S_n, aI_0, \cdots, aI_n, aO_0, \cdots aO_n, f, b_1, \cdots, b_m\} \\
Pr_+ &= \{\forall p : p = S_0 ::= S_1 \ \cdots \ S_n\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{rule } S_0 ::= S_1 \ \cdots \ S_n \textbf{ attr } aI_1 \leftarrow aI_0, \\
&\qquad \textbf{rule } S_0 ::= S_1 \ \cdots \ S_n \textbf{ attr } aI_2 \leftarrow aO_1, \\
&\qquad\quad \cdots \\
&\qquad \textbf{rule } S_0 ::= S_1 \ \cdots \ S_n \textbf{ attr } aO_0 \leftarrow aO_n, \\
&\qquad \textbf{rule } S_0 ::= S_1 \ \cdots \ S_n \textbf{ attr } aI_1 \leftarrow f(b_1, \cdots, b_m), \\
&\qquad \textbf{rule } S_0 ::= \varepsilon \textbf{ attr } aO_0 \leftarrow aI_0\} \\
At_- &= \emptyset
\end{aligned}
$$

Für $n, m \in \mathbb{N}$, wobei die Anwendung vom typischen Muster 1 dann $\mathcal{M}_u^+$ entspricht. □

Die Anwendung des Attributgrammatik-unabhängigen Musters erfolgt durch iterative Selektion einer neuen Substitution unter Ausnutzung von Definition 4.21. Dies bedeutet, dass die im typischen Muster 1 angegebene Rekursion auf den weiteren Symbolen der Produktion durch den Abschluss der Musteranwendung erfolgt. Die Substitutionen können, analog wie in Basismuster 2 oder Basismuster 3, durch Anwendung auf eine Attributgrammatik hergeleitet werden.

Sollen Kettenattributierungen und Symbolattributierungen kombiniert werden, wird für die Spezifikation in Attributgrammatiken eine Möglichkeit benötigt auf die Attribute, unabhängig von der konkreten Produktion oder gar der konkreten Attributgrammatik, zuzugreifen.

Typisches Muster 2. ***Symbol-lokaler Zugriff auf Kettenattribute*** *(u. a. [101])*

Für eine Attributgrammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und einem Nichtterminal* $\mathsf{Y} \in N$ *und Symbolen* $\mathsf{A}, \mathsf{B} \in \Sigma$ *und* $u, v \in \Sigma^*$ *existieren folgende mögliche Produktionen*

1. $p_1 \in P : \mathsf{Y} ::= \varepsilon$
2. $p_2 \in P : \mathsf{Y} ::= \mathsf{A}u$
3. $p_3 \in P : \mathsf{Y} ::= v\mathsf{B}$

Dann ist

```
symbol Y
attr head.a ← e1
     ↑b ← head.a
     ↑d ← tail.c default e2
```

äquivalent mit

```
rule p1: Y ::= ε
attr Y.b ← e1
     Y.d ← e2

rule p2: Y ::= A u
attr A.a ← e1
     Y.b ← A.a

rule p3 Y ::= v B
attr Y.d ← B.c
```

e_1 *und* e_2 *beliebige Ausdrücke sind, die die Attribute* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}$ *nicht verwenden und* c *ein bestehendes Attribut eines Symbols ist.*

Das Muster 2 lässt sich ebenso als Kombination der Basismuster Kettenattributierung und Symbolattributierung definieren. Bei der Anwendung des Musters ist sicher zu stellen, dass für die Abbildung auf solch ein Nichtterminal eine Kettenattributierung vorliegt. Neben der Initialisierung von Kettenattributierungen unabhängig von konkreten Produktionen erlauben Kettenattributierungen die Verwendung des Ergebnisses. Dieses Ergebnis kann dann durch die in [112] vorgestellte Methode propagiert werden. Dieses „Kopieren nach unten“ lässt sich wie folgt darstellen:

Typisches Muster 3. ***Kopieren nach Unten*** *(u. a. [112])*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und beliebigen Symbolen* $\mathsf{X} \in N, \mathsf{Y} \in \Sigma$*, wobei es Produktionen gibt, so dass* $\mathsf{X} \overset{+}{\rightsquigarrow} \mathsf{Y}$*. Sei* $\mathsf{b} \in A_{\mathsf{X}}$*, sowie allen Produktionen* $p : \mathsf{V} ::= u\mathsf{X}v$*,* $q : \mathsf{X} ::= w\mathsf{W}y$ *und* $r : \mathsf{W} ::= x\mathsf{Y}z$*, sodass* $\mathsf{V} \overset{*}{\rightsquigarrow} \mathsf{X}$*,* $\mathsf{X} \overset{*}{\rightsquigarrow} \mathsf{W}$ *und* $\mathsf{W} \overset{*}{\rightsquigarrow} \mathsf{Y}$*, wobei* $u, v, w, x, y, z \in \Sigma^*$*,* $X, W, V \in N$*,* $Y \in \Sigma$ *und* $r \in P$ *mit* $|Y|_r = k$ *und* $1 \leq i \leq k$*,* $k \in \mathbb{N}$*, dann steht*

```
autocopy a
```

für

```
rule p: V ::= u X v
attr X.a ← V.a

rule q: X ::= w W y
attr W.a ← X.a

rule r: W ::= x Y z
attr Yi.a ←W.a
```

für alle $1 \leq i \leq k$.

Existiert eine Produktion in der a *definiert wird, so bleibt diese Produktion und die Berechnung von* a *in dieser Produktion von diesem Muster unberührt.*

Die letzte Aussage der Definition des Musters 3 lässt sich, analog der Beschreibung von Symbolattributierungen, durch Einschränkung der Prädikate erreichen. Die Bedingung Y für alle Vorkommen in r entsprechend zu attributieren, ist auf Eigenschaft 3 von Satz 4.1, auf den Aufbau ererbter Attributierungsregeln sowie Definition 4.16 zurückzuführen. Ist

Im Folgenden wird gezeigt, wie Muster 2 und 3 als Attributgrammatik-unabhängige Musterdefinitionen realisiert werden können.

Lemma 4.28. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 2 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$
\begin{aligned}
V &= \{S_i, A_1, \cdots, A_n, f, g, b_1, \cdots, b_m, c_1, \cdots, c_l, aI_1, bO_0, bO_1, aO_n\} \\
Pr_+ &= \{S_i ::= \varepsilon \\
&\qquad S_i ::= A_1 \ \cdots \ A_n, \\
&\qquad S_i ::= A_1 \ \cdots \ A_n\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{rule } S_i ::= \varepsilon \textbf{ attr } bO_0 \leftarrow f(b_1, \cdots, b_m), \\
&\qquad \textbf{rule } S_i ::= A_1 \ \cdots \ A_n \textbf{ attr } aI_1 \leftarrow f(b_1, \cdots, b_m), \\
&\qquad \textbf{rule } S_i ::= \varepsilon \textbf{ attr } bO_1 \leftarrow g(c_1, \cdots, c_l), \\
&\qquad \textbf{rule } S_i ::= A_1 \ \cdots \ A_n \textbf{ attr } bO_1 \leftarrow aO_n\} \\
At_- &= \emptyset
\end{aligned}
$$

für $n, m \in \mathbb{N}$. □

Im Beweis von Lemma 4.28 wird, wie in den anderen Attributgrammatik-unabhängigen Musterdefinitionen, die beliebigen Ausdrücke auf Funktionsanwendung mit bestehenden Attributen abgebildet.

Lemma 4.29. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 3 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$
\begin{aligned}
V &= \{S_0, S_1, \cdots, S_n, a_0, b_1, \cdots, b_m, f, base, copy, copytarget, A_i, B_0, \cdots, B_l\} \\
Pr_+ &= \{S_0 ::= S_1 \ \cdots \ S_n, \\
&\quad\ A_i ::= B_0 \ \cdots \ B_l, \\
&\quad\ a_0 \leftarrow f(b_1, \cdots, b_m)\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{rule}\ S_0 ::= S_1 \ \cdots \ S_n\ \textbf{attr}\ copytarget \leftarrow copy, \\
&\quad\ \textbf{rule}\ A_i ::= B_0 \ \cdots \ B_l\ \textbf{attr}\ copy \leftarrow base, \\
&\quad\ \textbf{rule}\ A_i ::= B_0 \ \cdots \ B_l\ \textbf{attr}\ copytarget \leftarrow copy, \\
&\quad\ \textbf{symbol}\ A_i\ \textbf{attr}\ \downarrow.base \leftarrow a_0\} \\
At_- &= \emptyset
\end{aligned}
$$

für $n, m \in \mathbb{N}$, wobei die Anwendung von Basismuster 3 dann $\mathcal{M}_u^+$ entspricht. □

In der Konstruktion beim Beweis von Lemma 4.29 wird davon ausgegangen, dass beim Abschluss in der Substitution für die ererbten, kopierten Attribute eben die Substitution für die zu kopierenden Attribute auf allen Symbolen angewandt wird. Der Algorithmus zum Finden solch einer Substitution ist nicht Gegenstand dieser Arbeit. In Anhang D.3 wird die Grundidee jedoch kurz vorgestellt.

Eine andere Variante, die in Beispiel 2.7 aus [80] vorgestellt wird, realisiert dieses Kopieren durch ein *programmiertes* Termersetzungssystem. In dieser Arbeit kommen jedoch nur die Basismuster und geeignete Abbildungen auf eine bestehende Attributgrammatik zum Einsatz. Die Verwendung eines programmierbaren (bzw. programmierten) Termersetzungssystems kann eine Grundlage zur Implementierung des in dieser Arbeit vorgestellten Verfahrens sein. Das typische Muster 4 realisiert die in Beispiel 2.7 als Termersetzungssystem vorgestellte Lösung, wie sie in [76, 101] präsentiert wird. Ein programmierbares bzw. programmiertes Termersetzungssystem vereint die Spezifikation von Termersetzungsregeln mit, durch einen Programmierer definierten, Besuchsstrategien[3]. Anwendungsfall solcher sind vor allem Termersetzungssysteme, die nicht konfluent sind[90].

Typisches Muster 4. ***Kopieren nach Unten (von oben)*** *(u. a. [73, 78])*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und beliebigen Symbolen dass* $\mathtt{X} \overset{+}{\rightsquigarrow} \mathtt{Y}$. *Seien* $\mathtt{b} \in A_{\mathtt{Y}}$ *und* $\mathtt{a} \in A_{\mathtt{X}}$, *sowie allen Produktionen* $p : \mathtt{V} ::= u\mathtt{X}v$, $q : \mathtt{X} ::= w\mathtt{W}y$ *und* $r : \mathtt{W} ::= x\mathtt{Y}z$, *sodass* $\mathtt{V} \overset{*}{\rightsquigarrow} \mathtt{X}$, $\mathtt{X} \overset{*}{\rightsquigarrow} \mathtt{W}$ *und* $\mathtt{W} \overset{*}{\rightsquigarrow} \mathtt{Y}$, *wobei* $u, v, w, x, y, z \in \Sigma^*$, $X, W, V \in N$, $Y \in \Sigma$, *weiterhin sei* $|X|_p = l$, $|W|_q = m$ *und* $|Y|_r = n$ *dann ist für all* $1 \leq i \leq l$, $1 \leq j \leq m$ *und* $1 \leq k \leq n$, $l, m, n \in \mathbb{N}$ *dann*

```
symbol Y
attr this.b ← including X.a
```

entspricht

```
rule p: V ::= u X v
attr X_i.incl_1 ←X.a

rule q: X ::= w W y
attr W_j.incl_1 ←X.incl_1

rule r: W ::= x Y z
attr Y_k.b ←W.incl_1
```

Wobei $\mathtt{incl}_1$ *ein neues Attribut ist, welches bisher nicht in der Attributgrammatik verwendet wurde, d.h.* $\mathtt{incl}_1 \notin A$.

Bemerkung (Verallgemeinerung auf mehrer $\mathtt{X}_i$)**.** Das typische Muster 4 lässt sich auf mehrere $\mathtt{X}_i$ verallgemeinern und ebenso auch auf Nichtterminale A, die auf dem Pfad von X zu Y liegen. Intuitiv wird dann an der Stelle, an der A abgeleitet wird, statt dem weiterkopieren von `incl_1` das von A kopierte Attribut eingesetzt.

[3] Diese Strategien können ebenfalls bereits Bestandteil des programmierbaren Termersetzungssystems sein.

```
symbol Description
attr ↑env ← tail.declsOut

symbol UseId
attr this.env ← including Description.env
```

Beispiel 4.12 – Beispiel 1.2 ergänzt um Verwendung des typischen Musters 4 und 2.

Wiederum ist dieses typische Muster als Symbolattributierung realisierbar und darüber hinaus mit geeigneten Abbildungen als Hintereinanderausführung auf derselben Attributgrammatik. Für ein Muster der Symbolattributierung kann das Kopieren nach Unten (von Oben) dann mittels Definition 4.21 realisiert werden. Lemma 4.30 zeigt diese Konstruktion.

Lemma 4.30. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 4 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$
\begin{aligned}
V &= \{A_0, \cdots, A_n, B_1, \cdots, B_k, f, targ, from, tmp, a_1, \cdots, a_m, S, S_0, S_1, \cdots, S_l\} \\
Pr_+ &= \{A_0 ::= A_1 \ \cdots \ A_n \\
&\qquad from \leftarrow f(a_1, \cdots, a_m)\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{symbol } S_0 \textbf{ attr } tmp \ \leftarrow \ from \\
&\qquad \textbf{rule } S_0 ::= S_1 \ \cdots \ S_l \textbf{ attr } S.tmp \leftarrow S_0.tmp \\
&\qquad \textbf{rule } S_i ::= B_1 \ B_{i-1} \cdots \ A_i \ B_{i+1} \ \cdots \ B_k \textbf{ attr } targ \leftarrow tmp\} \\
At_- &= \emptyset
\end{aligned}
$$

für $n, m, l \in \mathbb{N}$, wobei die Anwendung vom typischen Muster 4 dann $\mathcal{M}_u^+$ entspricht. □

Für Beispiel 1.2 lässt sich das Kopieren des Attributs `env` durch das typische Muster 4 realisieren. Statt dem manuellen Kopieren wird dieses Kopieren durch Angabe des zu kopierenden Attributs realisiert. Beispiel 4.12 erweitert und ersetzt die in Beispiel 1.2 (siehe Seite 6) mit dem typischen Muster 4 um Kopieranweisungen zu streichen.

Darüber hinaus existieren verschiedene Varianten das Kopieren in obige Richtung, d. h. von synthetisierten Attributen zu vermeiden:

Typisches Muster 5. ***Kopieren von Unten (einfach)*** *(u. a. [101, 73])*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Sei* $\mathsf{Y} \in \mathsf{N}$ *ein Nichtterminal und Produktionen* $p_i : \mathsf{Y} ::= \mathsf{X}_i \in P$ *für Symbole* $\mathsf{X}_i \in \Sigma$, $i \in [1, n]$, *dann ist*

```
symbol Y
attr this.b ←
       constituent (X1.a1, ···, Xn.an)
```

äquivalent zu

```
rule Y ::= X1
attr Y.b ← X1.a1

···

rule Y ::= Xn
attr Y.b ← X1.an
```

genau dann, wenn der Typ der Attribute $\mathsf{a}_1, \cdots, \mathsf{a}_n$ *identisch ist und für jedes der* X_i *gilt:* $\mathsf{X}_i \neg \overset{+}{\rightsquigarrow} \mathsf{Y} \wedge \mathsf{X}_i \neg \overset{+}{\rightsquigarrow} \mathsf{X}_j$ *für alle* $j \in [1, n]$.

Mit diesem Muster lassen sich viele Alternativen in der Ableitung eines Nichtterminals in einer Berechnung zusammen fassen. Dies ist insbesondere dann nützlich, wenn über Alternativen und Standardfälle eine komplexe Berechnung mit Terminalsymbolen verknüpft werden soll. Folgendes Lemma zeigt die Attributgrammatik-unabhängige Musterdefinition zu Muster 5:

Lemma 4.31. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 4 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$\begin{aligned} V &= \{S, T, U, a, b\} \\ Pr_+ &= \{S ::= T \wedge \neg\exists T ::= U\} \\ Pr_- &= \emptyset \\ At_+ &= \{\textbf{rule } S ::= T \textbf{ attr } a \leftarrow b\} \\ At_- &= \emptyset \end{aligned}$$

□

In der Anwendung des Attributgrammatik-unabhängigen Musters aus Lemma 4.31 ist zu beachten, dass die Substitution alle Regeln von dem mit S substitutierten Symbol einnehmen muss. Dieses Muster kann dann nicht angewendet werden, wenn es bspw. für ein Symbol Y auch Produktionen mit mehreren herleitbaren Symbolen in derselben Produktion, bspw. $\mathsf{Y} ::= \mathsf{X}_1\ \mathsf{X}_2$, gibt.

Darauf aufbauend kann das Muster des Kopierens von Unten (komplex) als Abschluss von Muster 5, synthetisierter Symbolattributierung (Basismuster 2) und deren Komposition betrachtet werden.

Typisches Muster 6. *Kopieren von Unten (komplex)* *(u. a. [73, 101])*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für* $\mathsf{X}_0 \in N$ *und alle Symbole* $\mathsf{X}_i \in \Sigma$ *mit* $\Sigma' = \{X_i : \mathsf{X}_0 \overset{+}{\rightsquigarrow} \mathsf{X}_i\}$, *sei* $P' = \{p \in P, \lceil \mathsf{X}_i \rceil_p > 0, i \geq 0\}$ *so existieren in* P' *folgende Arten von Produktionen:*

$$\begin{aligned} p_0 &= \mathsf{X}_i ::= u\ \mathsf{X}_j\ v \\ p_1 &= \mathsf{X}_i ::= u\ \mathsf{X}_j\ v\ \mathsf{X}_j\ w \\ p_2 &= \mathsf{X}_i ::= u \end{aligned}$$

Wobei $\left|\mathsf{X}_j\right|_{p_0} = 1$ *und* $\left|\mathsf{X}_j\right|_{p_1} = n$, $n > 1$ *ist,* $\left|\mathsf{X}_j\right|_{p_2} = 0$ *für alle* $X_i, X_j \in \Sigma'$ *und* $i \geq 0, j > 0$. *Darüber hinaus ist für jede Produktion*

$$p_3 = \mathsf{X}_j ::= u\ \mathsf{X}_0\ v$$

für $u, v, w \in \Sigma^*$ *so steht*

```
symbol X0
attr this.b ←
       constituents (Y1.a1, ···, Ym.am)
       with ⊕, f, e
```

für

```
rule Xi ::= u Xj v
attr Xi.c ← Xj.c

rule Xi ::= u Xj v Xj w
attr Xi.c ← Xj,1 ⊕ ··· ⊕ Xj,n

rule Xi ::= u
attr Xi ← e

rule Xj ::= u Yk v
attr Xj.c ← f(Yk.ak)

...

rule Xj ::= u X0 v
attr Xj.c

symbol X0
attr ↑b ← ↑c
```

Dabei sind die $\mathsf{Y}_k \in \Sigma', k \in \mathbb{N}$*, Sei* τ *ein Typ und* $\oplus\colon \tau \times \tau \to \tau$*, dann bildet* $(\tau, \oplus, e)$ *einen Monoid.*

Sei μ der Typ der Attribute $\mathsf{Y}_k.a_k, 0 < k$ dann ist $f\colon \mu \to \tau$ eine Operation zur Typkonversion in Muster 6.

Letztendlich bleibt auch noch für dieses Muster zu zeigen, dass eine Attributgrammatik-unabhängige Musterdefinition existiert, sodass dieses Muster Anwendung finden kann.

Lemma 4.32. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 4 ein Attributgrammatik-unabhängiges Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$ wobei

$$
\begin{aligned}
V &= \{S_0, \cdots, S_n, A_i, B_0, \cdots, B_m, tmp, target, f, g, e, a\} \\
Pr_+ &= \{S_0 ::= S_1 \ \cdots \ S_n, \\
&\quad S ::= B_0 \ \cdots \ B_{i-1} \ A_i \ B_{i+1} \ \cdots \ B_m, \\
&\quad A_i ::= A_j, \\
&\quad A_i ::= \varepsilon\} \\
Pr_- &= \emptyset \\
At_+ &= \{\textbf{symbol } S_0 \textbf{ attr } target \leftarrow tmp, \\
&\quad \textbf{rule } S_0 ::= S_1 \ \cdots \ S_n \textbf{ attr } S_0.tmp \leftarrow f(S_1.tmp, \cdots, S_n.tmp), \\
&\quad \textbf{rule } S ::= B_0 \ \cdots \ B_{i-1} \ A_i \ B_{i+1} \ \cdots \ B_m \textbf{ attr } S.tmp \leftarrow A_i.tmp, \\
&\quad \textbf{rule } A_i ::= A_j \textbf{ attr } A_i.tmp \leftarrow g(A_j.a), \\
&\quad \textbf{rule } A_i ::= \varepsilon \textbf{ attr } A_i.tmp \leftarrow e\} \\
At_- &= \emptyset
\end{aligned}
$$

□

wobei $n, m \in \mathbb{N}$.

Beispiel 2.6 verwendet Vererbung für Symbolattributierung. Dies ist im Rahmen dieser Arbeit nichts anderes als die Festlegung einer Substitution für mehrere Muster der Symbolattributierung. Weitere Beispiele verwenden diese Art der Attributierung nicht, weswegen diese Thematik nur am Rande erwähnt wird.

4.6. Bemerkungen zur Semantik der Musteranwendung bzgl. der Musterdarstellung der Arbeit

Die Darstellung von Mustern in dieser Arbeit abstrahiert damit, ebenso wie Muster selbst, von den formal zugrunde liegenden Methoden und erlaubt eine knappe Formulierung. Bei der Anwendung von Mustern, wie sie hier dargestellt sind, ist implizit anzunehmen, dass bei fehlender Substitution bzw. falls die Prädikate unerfüllt bleiben, keine Änderungsmengen existieren und somit keine Änderung erfolgt. Ist also bei einer Musteranwendung des Muster $\mathcal{M}_u$ bzgl. einer Attributgrammatik AG und passender Substitutionen *Subs* das Tupel $\Delta_{\mathcal{M}} \triangleq (\mathcal{M}_+, \mathcal{M}_-)$ und dabei $\mathcal{M}_+ \triangleq (\mathcal{M}_{+,A}, \mathcal{M}_{+,R})$ und $\mathcal{M}_- \triangleq (\mathcal{M}_{-,A}, \mathcal{M}_{-,R})$ (siehe Def. 4.16) die Mengen zur Bestimmung der Änderungen jeweils leer, weil die Prädikate bspw. bereits zu falsch ausgewertet werden, dann sind auch die jeweiligen Mengen R_+, R_-, A_+ und A_- eben leer. Praktisch bedeutet dies, dass ein Muster nicht anwendbar war. Für die in dieser Arbeit entwickelte

```
1  rule Program ::= Stats attr
2  rule Stats ::= Stats Stat attr
3  rule Stats ::= Stat Stats attr
4  rule Stats ::= Stat attr
5  rule Stat ::= VarStat attr
6  rule VarStat ::= id id attr VarStat.number = 0
7  rule VarStat ::= id number attr VarStat.number = 1
```

a) Initiale (und finale) Attributgrammatik für dieses Beispiel nach Anwendung der Änderungsmengen aus Beispiel 4.2.

```
1  symbol VarStat attr ↑number = 10
```

b) Anwendung von Basismuster 2 auf die Attributgrammatik aus 4.13a, wobei keine Änderungsmengen generiert werden können.

Beispiel 4.13 – Beispiel für Anwendung von Basismuster 2 ohne praktische Änderungsmengen durch Auswertung der Prädikate zu falsch ausgehend von der Attributgrammatik nach Anwendung der Änderungsmengen aus Beispiel 4.2

Theorie ist dies dennoch nicht relevant, da diese Mengen die Eigenschaften nach Satz 4.1 einhalten und bei einer zerlegbaren Attributgrammatik die Anwendung solch eines Musters ebenfalls wieder zerlegbar ist. Im Fall von geordneten Attributgrammatiken bleiben diese ebenfalls geordnet.

Ein Beispiel für die Anwendung eines Musters für welches die Prädikate zu falsch ausgewertet werden unter Verwendung von Basismuster 2 ist Beispiel 4.13. In diesem Beispiel ist die Substitution nicht explizit angegeben. An dieser Stelle sei dazu nur folgendes angemerkt: alle Variablen bzgl. a_2 und c_i werden durch ε ersetzt und a_1 durch `number` bzw. `VarStat.number`, die Vorschrift $f(b_{d_1}, \cdots, b_{d_m})$ eben durch 0 bzw. 1, sodass dieser Term für alle Produktionen in denen S durch `VarStat` ersetzt wurde nicht zutrifft. Wäre das Muster anwendbar, so würde $a_1 \leftarrow f(b_1, \cdots, b_k)$ noch durch `number` $\leftarrow 10$ substituiert werden.

Damit schließt die Vorstellung der bereits bekannten Muster und Basismuster. Jedoch ist das Löschen von Attributierungsregeln sowie die Komposition von Mustern hervorzuheben, da dies bisher in der Literatur noch nicht betrachtet wurde. Darüber hinaus sind in der Darstellung der Muster dieser Arbeit neben der Attributgrammatik-unabhängigen Musterdefinition mit den erweiterten Attributwerttermen, ebenso die dafür notwendigen Prädikate, und – implizit – die zu substituierten Terme bei der Anwendung auf eine Attributgrammatik angegeben. Auf die Substitution und der Bestimmung der Substitution wird in dieser Arbeit nicht genauer eingegangen, eine Reihe von Ausführungen dazu finden sich in Anhang D.

Kapitel 5.

Komplexe Muster auf Attributgrammatiken

Grundsätzlich sind die in Kapitel 4 vorgestellten Muster und deren Eigenschaften bereits ausreichend zur kompakteren Beschreibung von Attributgrammatiken. Einerseits sind diese jedoch zum Teil bereits aus der Literatur bekannt andererseits existieren weitere Muster, die genutzt werden können um Attributgrammatiken kompakt zu spezifizieren.

In Abschnitt 4.1 wurden die Muster dieser Arbeit in Bezug auf zerlegbare Attributgrammatiken definiert. In der Praxis der Sprachentwicklung im Übersetzerbau werden jedoch (u. a.) geordnete Attributgrammatiken genutzt[1] Bei geordneten Attributgrammatiken sind ggf. zusätzliche Attributabhängigkeiten notwendig, um sicherzustellen, dass die zerlegbare Attributgrammatik geordnet ist. In Abschnitt 4.3 wurde auf die dafür notwendigen Eigenschaften eingegangen.

Aufbauend auf den Mustern aus Abschnitt 4.5 werden somit in diesem Kapitel weitere Muster präsentiert. Die Muster dieses Kapitels bestehen vorwiegend aus den bereits bekannten Mustern. In diesem Kapitel erfolgt nicht immer eine ausführliche Rückführung auf die Attributgrammatik-unabhängige Darstellung, sondern nur die, die aufbauend auf den in Lemma 4.24, 4.28, 4.27, Lemma 4.22 und Lemma 4.23 vorgestellten Darstellungen notwendig ist um diese mittels Komposition zu den Mustern dieses Kapitels zu erweitern.

In der weiteren Darstellung wird ggf. unterschieden in bestehende und neue Attribute bzw. auf bestehende Attributierungsregeln eingegangen. Im Rahmen dieser Arbeit ist dies bspw. als Bedingung gemeint, dass für diese Attribute ein Prädikat zu wahr auswertet wird. Dabei soll dieses Attribut (bzw. diese Attributierungsregel) dann dennoch nicht in der Menge der zu entfernenden Attribute (oder Attributierungsregeln) aufgenommen werden.

Dieses Kapitel stellt wenige Beispiele vor, zeigt allerdings in einer Reihe von Beweisen die Rückführung auf Basismuster. Die Anwendung der, in diesem Kapitel präsentierten, Muster erfolgt im folgenden Kapitel.

5.1. Einführung komplexerer Muster

In Beispiel 2.3 werden in den ersten Zeilen Variablen initialisiert. Nach dem Einlesen eines Programms liegen die Werte des dazugehörigen Terminalsymbols häufig nur als Zeichenfolge oder als Eintrag in der Symboltabelle vor. Zur Realisierung von Konstantenfaltung, wie dies in Beispiel 1.1 geschieht, müssen solche Konstanten Werte erkannt werde können. Dafür ist der Wert, der in der Zeichenfolge beschrieben ist notwendig. Bei Verwendung Typ-sicherer Sprachen kann dafür ein neues Attribut verwendet werden, welches diesen konvertierten Wert speichert.

In folgendem Muster wird noch ein Schritt darüber hinaus durchgeführt, indem für alle Nichtterminale in denen ein Attribut vorliegt durch Aufruf einer geeigneten Konvertierungsfunktion berechnet werden:

[1] Ebenfalls in der Praxis werden Attributgrammatiken mit Attributen höherer Ordnung, Referenzattributgrammatiken und andere Formen von (dynamischen) Attributgrammatiken genutzt. Weiterhin existieren manuell programmierte Lösungen.

Typisches Muster 7. ***Attributabbildung***

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *mit Symbolen* $\mathsf{X}_i \in \Sigma$, $1 \leq i \leq n$ *für* $n \in \mathbb{N}$ *und einem Attribut* $b \in A_{\mathsf{X}_i}$, *dann ist*

```
a is f(𝔛.b_1, ···, 𝔛.b_m)
```

äquivalent mit

```
symbol X_1
attr ↑a ← f(this.b_1,\cdots,this.b_m)
···
symbol X_n
attr ↑a ← f(this.b_1,\cdots,this.b_m)
```

für alle $\mathsf{X}_i \in \mathfrak{X}$, $\mathfrak{X} \subseteq \Sigma$ *und* $b_j \in A(\mathsf{X}_i)$ *für* $1 \leq i \leq n$, $n \in \mathbb{N}$, $1 \leq j \leq m$, $m \in \mathbb{N}$ *und beliebige Funktionen* f.

Das Muster 7 dient also der Verringerung des Schreibaufwands bei geeignet entwickelten Attributgrammatiken. Dieses Muster wurde bisher in der Literatur noch nicht vorgestellt. Auch dieses Muster lässt sich auf das Basismuster 2 zurückführen.

Lemma 5.1. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 7 ein Muster.

Beweis. Nach Lemma 4.22 ist die Symbolattributierung $\mathcal{M}_u \triangleq (V, Pr_+, Pr_-, At_+, At_-)$, ebenso dann $\mathcal{M}_u^+$ nach Lemma 4.24. □

Nach Lemma 5.1 und $|\mathfrak{X}| = 1$ fällt Muster 7 mit Basismuster 2 zusammen. Dabei ist dann die Angabe der passenden Substitutionen nach Angabe durch den Entwickler herleitbar. In Anhang D.4 wird die Anwendung dieses Musters in der Implementierung der dazugehörigen Sprache mit der Herleitung der Substitution genauer besprochen.

Zum Aufbau komplexerer Muster werden darüber hinaus folgende zwei typischen Muster verwendet. Diese beiden Muster sind angelehnt an „contributions“ (engl. contribution: Beitrag) wie diese von Boyland in [25] sowie Hedin in [60] vorgestellt wurden. Die in dieser Arbeit vorgestellten „Beitrags“-Muster gehen jedoch über die von Boyland oder Hedin vorgestellten hinaus: neben der Beschreibung Mengen-artiger Attributierungen können beliebige Strukturen attributiert werden. Darüber hinaus erlauben „Beitrags“-Muster zusätzliche Attributierungsregeln, sodass in einer geordneten Attributgrammatik deren Ordnung erzwungen wird.

Bei geordneten Attributgrammatiken können globale Datenstrukturen, auch als Attribute abgelegt, bei der Definition (anderer) Attribute manipuliert werden. Diese Manipulation in einer Attributierungsregel eines anderen Attributs wird als „Seiteneffekt“ bezeichnet. Diese Seiteneffekte und auch Seiteneffekt-basierte Attributierungen[2] sind ein Alleinstellungsmerkmal der „Beiträge“ in der vorliegenden Arbeit.

Typisches Muster 8. ***Einfache Beiträge*** *(u.a. [60, 25])*

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *gegeben. Für ein Symbol* X *mit Attribut* $a \in A_{\mathsf{X}}$ *sowie einem Nichtterminal* Y, $\mathsf{Y} \overset{+}{\rightsquigarrow} \mathsf{X}$ *steht*

```
contribute X.a to Y.c ←e using ⊕
```

für

```
chain c head Y ←e

symbol X
attr ↑c ← ↓c ⊕ this.a
```

Wobei e *ein beliebiger Ausdruck ist und* c *ein neues Attribut;* $\oplus\colon \alpha \to \beta \to \alpha$ *ist eine binäre Operation. Stimmen* β *und* α *überein und ist dann* $\oplus$ *assoziativ, so wird, analog Muster 6, ein Monoid gebildet.*

[2] Beim Aufbau der Definitionstabelle mit den Methoden aus [74] werden bei einer Attributierungsregel Funktionen auf, letztendlich, globalen Objekten aufgerufen, die nicht direkt im Attribut mitgeführt werden müssen. Dieser Funktionsaufruf bewirkt Zustandsänderungen an diesem Objekt, sodass dies als Seiteneffekt bezeichnet werden kann. Dieses Prinzip kann auch mit anderen Datentypen verwendet werden.

Das typische Muster 8 entspricht den *Collection*-Attributen, wie diese von Boyland oder auch Hedin verwendet werden. Äquivalente Berechnungen können auch mittels traditioneller Faltungen, wie sie Muster 6 präsentiert, erreicht werden. Der Unterschied zwischen einfachen Beiträgen und dem Kopieren von Unten besteht in der Möglichkeit, dass letztere bei allen Attributgrammatiksystemen zu einer Ausführungszeit proportional zur Baumgröße führen, selbst wenn das Symbol `X` im abstrakten Syntax hergeleitet wurde (bspw. durch mehrfache Verknüpfung des Initialausdrucks e für alle Blätter und alle Zwischenknoten ausgehend von einem `Y`). Bei einfachen Beiträgen können in geordneten Attributgrammatiken die Kopieropertationen für die „Kette“ (a) wegoptimiert werden, sodass nur eine implizite Reihenfolge durch diese Kopieroperationen hinzugenommen wird. Details zur Optimierung in geordneten Attributgrammatiken beschreibt Kastens in [77].

Dementsprechend sind einfache Beiträge für das Aufsammeln von Attributen dem „Kopieren von Unten (Monoid)“ vorzuziehen, da hierbei die Performanz, insbesondere bei teuren Operationen ($\oplus$), potentiell besser ist.

Üblicherweise wird gefordert, dass e das neutrale Element bzgl. einer Operation $\oplus\colon \alpha \to \alpha \to \alpha$ bildet. Dies wird an dieser Stelle nicht explizit gefordert und in den folgenden Mustern auch nicht vorausgesetzt. Dennoch wird in dieser Arbeit davon ausgegangen, dass diese Eigenschaften eingehalten werden. Mittels zusätzlicher Attributabhängigkeiten kann bspw. eine Besuchsreihenfolge sichergestellt werden, sodass, wenn e nicht neutrales Element bzgl. $\oplus$ ist, notwendige Eigenschaften eingehalten werden. Im weiteren Verlauf dieser Arbeit wird nur noch begrenzt auf diese Eigenschaften eingegangen.

Ein Beispiel, bei der die typischen Muster „einfacher Beitrag“ und „Attributabbildung“ verwendet werden um die maximal benötigte Anzahl an Abhängigkeiten einer Anforderung zu bestimmen ist in Beispiel 5.1 aufgefürt.

```
1  size is length(uses)
2  contribute RqDefId.size to Description.max_used using max
```

Beispiel 5.1 – Ausschnitt aus einem Beispiel zur Anwendung der typischen Muster „Attributabbildung“ und „einfacher Beitrag“ zur Berechnung der maximal benötigten Spaltenanzahl beim Export der Abhängigkeiten als Ergänzung zu Beispiel 1.2 auf Seite 6.

In Beispiel 5.1 werden die Funktionen `length` und `max` benutzt, welche wie in Haskell wie folgt definiert sind:

```
length [] = 0
length (x:xs) = 1 + length(xs)

max a b = if a > b then a else b
```

Folgendes Muster abstrahiert von einfachen Beiträgen hin zu komplexen Beiträgen:

Typisches Muster 9. *Komplexe Beiträge*

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *gegeben. Für Symbole* $\mathtt{X}_i$ *mit Attributen* $a_i \in A_{\mathtt{X}_i}$ *sowie einem Nichtterminal* $\mathtt{Y}$, $\mathtt{Y} \overset{+}{\rightsquigarrow} \mathtt{X}_i$ *für alle* $i \in [0..n], n \in \mathbb{N}$, $\mathtt{X}_j \neq \mathtt{X}_k$ *für* $k \neq j$, $k, j \in [0..n]$

```
contribute X1.a1,···,Xn.an to Y.c ←e
          using ⊕
via b1 ←e1,
    ···,
    bm ←em
```

für

```
chain c head Y ←e

symbol X1
attr ↑b1 ←e1 ↞↓c
     ···
     ↑bm ←em ↞this.bm−1
     ↑c ← ↓c ⊕ this.a1 ↞this.bm
···

symbol Xn
attr ↑b1 ←e1 ↞↓c
     ···
     ↑bm ←em ↞this.bm−1
     ↑c ← ↓c ⊕ this.an ↞this.bm
```

Alternativ besteht folgende Äquivalenz:

```
contribute X_1.a_1,···,X_n.a_n to Y.c ←e
via b_1 ←e_1,
    ···
    b_m ←e_m
    chain ← f(···,chain,···, tribute,···)
```

für

```
chain c head Y ←e

symbol X_1
attr ↑b_1 ←e_1 ↞↓c
     ···
     ↑b_m ←e_m ↞this.b_{m-1}
     ↑c ← f(···, ↓c, ···, this.a_1, ···)
          ↞ this.b_m

···

symbol X_n
attr ↑b_1 ←e_1 ↞↓c
     ···
     ↑b_m ←e_m ↞this.b_{m-1}
     ↑c ← f(···, ↓c, ···, this.a_n, ···)
          ↞ this.b_m
```

$b_1, \cdots, b_m$ sind neue Attribute und $e_1, \cdots, e_m$ beliebige Ausdrücke zur Berechnung dieser Attribute, für ein $m \in \mathbb{N}$. ***tribute*** *und* ***chain*** *sind Abkürzungen zum Zugriff auf den Beitrag bzw. an die beizutragende Kette (siehe dazu auch Muster 1). f sei eine beliebige Funktion mit beliebigen Argumenten. Dabei kann die Berechnungsvorschrift für* ***chain*** *an beliebiger Stelle, auch zwischen* ***via****-Attributen stehen.*

Die Alternativen in Muster 9 unterscheiden sich in der „Vorauswahl“ der Substitution. Während in der ersten Alternative (implizit) gefordert ist, dass in der Substitution eine binäre Operation für die zu substituierende Funktion angegeben wird, ist dies in der zweiten Alternative frei. Wird $\oplus$ als Funktion dargestellt ist die letzte Zeile des Resultats auch darstellbar als:

```
↑c ← f_⊕(↓c, this.a_i)
     ↞ this.b_m
```

Für jedes „beigetragene“ $\mathsf{X}_i.a_i$. Die Verwendung der zweiten Alternative erlaubt die flexiblere Konstruktion der Seiteneffekte. So können für die neuen Attribute b_i ebenfalls Funktionen über das Kettenattribut und das hinzugefügte Attribut konstruiert werden. Die erste Alternative ist damit zwar eingeschränkter, kann aber auch kürzer formuliert werden.

Muster 9 ist eine Verallgemeinerung von Muster 8. Somit wird für Muster 8 kein eigener Beweis geführt. Wie bereits in Lemma 5.1 wird dieses Muster auf Kettenattribute, Symbolattributierung sowie Musterkomposition zurückgeführt.

Lemma 5.2. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 9 ein Muster.

Beweis. Sei $\mathcal{M}_c$ eine Kettenattributierung (siehe Lemma 4.27) mit passender Substitution $Subs_c$ mit $\left[\mathsf{chainIn}/aI_h\right] \in Subs_c$, `chain` dem Kettenattribut (und somit `chainIn` als ererbtes und `chainOut` als synthetisiertes) für alle $1 \leq h \leq n$, $h \in \mathbb{N}$. $\mathcal{M}_s$ eine Symbolattributierung (siehe Lemma 4.26) mit passenden Substitution $Subs_i$ mit $\left[\mathsf{b}_i/a_1\right]$ für alle $1 \leq i \leq m$ und $m \in \mathbb{N}$. Dann ist $\left[\mathsf{chainIn}/b_r\right] \in Subs_1$ und $\left[\mathsf{b}_{i-1}/b_r\right] \in Subs_i$. Mit $\mathcal{M}_{acc}$ einer Symbolattributierung und passender Substition $Subs_{acc}$ wobei $\left[\mathsf{chainOut}/a_1\right]$, $\left[\mathsf{b}_m/b_r\right]$ und $\left[\mathsf{chainIn}/b_{r+1}\right] \in Subs_{acc}$. Dabei ist $r \in \mathbb{N}$, $1 \leq r \leq k$.

Dann wird Muster 9 gebildet als $\mathcal{M}_{cont} = \mathcal{M}_{acc} \circ \mathcal{M}_s^m \circ \mathcal{M}_c$ und den angegebenen, passenden, Substitutionen. □

Im Beweis von Lemma 5.2 wurde nicht jede mögliche Substitution aufgeführt, wie bspw. die Substitionen um aus der Attributierungsregel $\uparrow .a_1 \leftarrow f(b_1, \cdots, b_k)$ einen Ausdruck e_i zu erzeugen. Für den Beweis sind diese Teilaussagen nicht von Relevanz, da sie ausschließlich die passenden Substitutionen betreffen und nicht die Attributgrammatik-unabhängige Beschreibung.

Wie in Definition 4.20 vorgestellt, wird auf das Anhängen von Substitutionen und Attributgrammatik verzichtet.

Ausgehend von den bisher präsentierten Mustern lassen sich viele komplexere Muster konstruieren:

Typisches Muster 10. ***Aufsammeln***

Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *ist*

```
collect X_1.a_1, ···, X_n.a_n in Y.c
```

äquivalent mit

```
chain c head Y ← []

symbol X_1
attr ↑c ← ↓c ++ [this.a_1]
···
symbol X_n
attr ↑c ← ↓c ++ [this.a_n]
```

Dabei sind $\mathsf{X}_1, \cdots, \mathsf{X}_n \in \Sigma$ *und* $\mathsf{Y} \in N$*, sowie* $a_i \in A_{X_i}$ *für* $0 \leq i \leq n$ *und* c *ein neues Attribut.* [] *bezeichnet die leere Liste,* ++ *den Konkatenationsoperator auf Listen.*

Alternativ lässt sich das typische Muster 10 auch durch Ausnutzung des Musters 6 beschreiben, diese Variante ist jedoch, aufgrund der Optimierung des Attributspeichers nach [77], nicht zu verwenden. Bei Muster 6 müssen alle erreichbaren Symbole attributiert werden, sodass sehr häufig, ggf. Laufzeitintensive, Operationen mit neutralen Elementen oder leeren Mengen oder Initialwerten ausgeführt werden müssen. Mit der Optimierung der Besuchsreihenfolge nach [77] werden unter Verwendung von Kettenattributen die Kopieranweisungen nur zur Bestimmung der Reihenfolge jedoch nicht zur Ausführung der Kopieroperationen verwendet. Somit ist die Variante, wie sie im typischen Muster 10 vorgestellt wurde, vorzuziehen. Auf die Verwendung von Mengen kann verzichtet werden, da im Bereich der Entwicklung Domänen-spezifischer Sprachen die Reihenfolge bei der Eingabe häufiger von Relevanz ist als die Sicherstellung, dass ein Element nur einmal betrachtet wird.

Auf die Angabe einer Attributgrammatik-unabhängigen Darstellung für Muster 10 könnte an dieser Stelle verzichtet werden, wenn in der Beschreibung des Musters 10 eine Rückführung auf komplexe Beiträge (Muster 9) verwendet worden wäre. Dies wäre jedoch verglichen mit der vorgestellten Variante aus Muster 10 bei der Expansion zum Vergleich eines Abstraktionsgrades wesentlich umfangreicher als die hier gewählte Variante. Gleichwohl kann eine Attributgrammatik-unabhängige Musterdefinition gewählt werden, die der aus dem Beweis von Lemma 5.2 ähnelt:

Lemma 5.3. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 10 ein Muster.

Beweis. Sei $\mathcal{M}_c$ eine Kettenattributierung (siehe Lemma 4.27) mit passender Substitution $Subs_c$ mit $[\mathtt{chainIn}/aI_h, \mathtt{chainOut}/aO_h] \in Subs_c$, `chain` dem Kettenattribut für alle $1 \leq h \leq n$, $h \in \mathbb{N}$. $\mathcal{M}_s$ eine Symbolattributierung (siehe Lemma 4.26) mit passenden Substitution $Subs_i$ mit $[\mathtt{chainOut}/a_1]$, $[(++)/f]$, $[\mathtt{chainIn}/b_1]$, $[\mathtt{a}_i/b_2]$ für alle $1 \leq i \leq n'$ und $n' \in \mathbb{N}$. Dann ist $[\mathtt{chainIn}/b_r] \in Subs_1$ und $[\mathtt{b}_{i-1}/b_r] \in Subs_i$. Mit $\mathcal{M}_{acc}$ einer Symbolattributierung und passender Substition $Subs_{acc}$ wobei $[\mathtt{chainOut}/a_1, \mathtt{b}_m/b_r, \mathtt{chainIn}/b_{r+1}] \in Subs_{acc}$. Dabei ist $r \in \mathbb{N}$, $1 \leq r \leq k$.

Dann wird Muster 10 gebildet als $\mathcal{M}_s^{n'} \circ \mathcal{M}_c$ und den angegebenen, passenden, Substitutionen, wobei n' genau dem n aus Muster 10 entspricht. □

Die Möglichkeit den Quelltext von Attributgrammatiken durch die Spezifikation aufzusammelnder Attribute zu verringern, wurde bereits von Boyland und auch Hedin beschrieben. Muster und komplexe Beiträge als Muster wurden jedoch bisher nur in dieser Arbeit präsentiert. Alle nun folgenden Muster basieren nicht auf aus der Literatur bekannten Möglichkeiten der Quellreduktion in Attributgrammatiken sondern sind nicht nur in der Präsentation, sondern auch inhaltlich bezüglich Attributgrammatiken neu.

Häufig ist die Akkumulation von Werten notwendig. Durch Verwendung von Präfixoperationen können Reihenfolge und Zwischenwerte für weitere Berechnungen zur Verfügung gestellt werden. Dieses Muster ließe sich nutzen, um das Beispiel aus [27] zur Bestimmung eindeutiger Marken in Blättern eines Baums zu realisieren.

Typisches Muster 11. ***Präfixsummen***

Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *mit Symbolen* $\mathsf{X}_1, \cdots, \mathsf{X}_n \in \Sigma$ *und Attributen* $a_i \in A_{\mathsf{X}_i}$ *für* $1 \leq i \leq n$, $n \in \mathbb{N}$, *dann*

```
scan X1.a1, ···, Xn.an
  to Y.c ← e with b using ⊕
```

entspricht

```
contribute X1.a1, ···, Xn.an to Y.tmp ←e
  via b ← chain ⊕ tribute
      chain ← chain ⊕ tribute
collect X1.b, ···, Xn.b in Y.c
```

für ein Nichtterminal $\mathsf{Y} \in N$ *mit* $\mathsf{Y} \overset{*}{\rightsquigarrow} \mathsf{X}_i$ *für* $1 \leq i \leq n$. *Dabei sei* τ *ein Typ und* $\oplus \colon \tau \to \tau \to \tau$. *Der Ausdruck* e *sei das neutrale Element in einem Monoid* $(\tau, \oplus, e)$. *Die Attribute* b *und* $\mathtt{tmp}$ *seien neue Attribute.*

Die Angabe einer Attributgrammatik-unabhängigen Musterdefinition folgt direkt aus den Beweisen zu Lemma 5.2 und Lemma 5.3 durch Komposition. Einzig relevant ist die Fixierung der Substitution angegeben durch Auswahl in der Musterbeschreibung und Referenzierung dieser bzgl. der angegebenen Attributgrammatik-unabhängigen Darstellungen in den Lemmata.

Lemma 5.4. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 11 ein Muster.

Beweis. Durch Angabe der Komposition Attributgrammatik-unabhängiger Musterdefinition: Sei $\mathcal{M}_{cont}$ die Attributgrammatik-unabhängige Musterdefinition nach Lemma 5.2 sowie, nach Lemma 5.3, $\mathcal{M}_{coll}$ eine Attributgrammatik-unabhängigen Musterdefinition für Muster 10. Für passende Substitutionen $Subs_{cont}$ und $Subs_{coll}$ sei dann $[\mathsf{b}/a_1] \in Subs_{cont}$ und $[\mathsf{b}/a_i] \in Subs_{coll}$ für $1 \leq i \leq n$, dann bildet $\mathcal{M}_{coll} \circ \mathcal{M}_{cont}$ das Attributgrammatik-unabhängige Muster 11. □

Die Relevanz von Muster 11 wird von der Liste in [23] verdeutlicht. In [23] führt Blelloch 13 Anwendungsfälle des Prinzips der Bestimmung von Präfixsummen auf. Neben der Implementierung von Quicksort taucht dabei auch ganz allgemein an Position 12 dieser Liste die Implementierung einiger Baumoperationen auf. Ein für den Übersetzerbau bzw. die Implementierung von DSLs wichtige Baumoperation ist die Bestimmung eines Index oder einer Marke. Die Umsetzung auf Basis von Präfixsummen ist in Muster 12 präsentiert.

Typisches Muster 12. *Indexbestimmung*

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und Symbolen* $\mathtt{X}_i \in \Sigma$, $1 \leq i \leq n$ *für ein* $n \in \mathbb{N}$ *und* $\mathtt{X}_0 \in N$ *mit* $\mathtt{X}_0 \overset{*}{\rightsquigarrow} \mathtt{X}_i$

```
count X_1, ···, X'_n from X_0
  in s start e
```

entspricht

```
symbol X_1
attr ↑t ← 1
···

symbol X'_n
attr ↑t ← 1
scan X_1.t, ··· X'_n.t to X_0.cnt ←e
  to X_0.irrev with s using +
```

wobei `cnt`, `irrev`, `t` *und* `s` *neue Attribute sind und* e *ein Initialisierungsausdruck eines Zahlentyps ist.*

Wie bereits in der Konstruktion zu Lemma 5.4 gezeigt, müssen für die Beweise von Mustern die durch Rückführung auf andere Muster angegeben sind, Substitutionen fixiert werden:

Lemma 5.5. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 12 ein Muster.

Beweis. Durch Angabe der Komposition Attributgrammatik-unabhängiger Musterdefinition: Sei $\mathcal{M}_{scan}$ die Attributgrammatik-unabhängige Musterdefinition nach Lemma 5.2 sowie $\mathcal{M}_s$ der Attributgrammatik-unabhängigen Musterdefinition für Symbolattributierungen nach Lemma 4.26. Sei $Subs_i$ die Menge passender Substitutionen für die Symbolattributierungen mit $1 \leq i \leq n'$ mit $n' \in \mathbb{N}$, und seien $\left[\mathtt{X}_i/S\right]$, $\left[\mathtt{t}/a_1\right] \in Subs_i$. Ist für $Subs_{scan}$ der Menge passender Substitutionen ebenfalls $\left[\mathtt{t}/a_i\right] \in Subs_{scan}$, dann bildet $\mathcal{M}_{scan} \circ \mathcal{M}_s^+$ das Attributgrammatik-unabhängige Muster 12. □

Typisches Muster 13. *Abhängigkeitsaufbau*

Für beliebige Symbolpaare $(\mathtt{X}_i, \mathtt{Y}_i)$, $1 \leq i \leq n$ *für* $n \in \mathbb{N}$ *einer attributierten Grammatik* $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *gelte für jedes* i *jeweils eine der folgenden Bedingungen:*

1. $\mathtt{Y}_i \overset{*}{\rightsquigarrow} \mathtt{X}_i$, $\mathtt{Y}_i \in N$, $\mathtt{X}_i \in \Sigma$ *oder*
2. $\mathtt{X}_i \overset{*}{\rightsquigarrow} \mathtt{Y}_i$, $\mathtt{X}_i \in N$, $\mathtt{Y}_i \in \Sigma$.

Darüber hinaus sei $\mathtt{Y} \in N$ *und* $\mathtt{Y} \overset{*}{\rightsquigarrow} \mathtt{X}_i$ *und* $\mathtt{Y} \overset{*}{\rightsquigarrow} \mathtt{Y}_i$ *für alle* $i \in [0, n]$. *Es seien die Attribute* $a_i \in A_{\mathtt{Y}_i}$ *und* $b_i \in A_{\mathtt{X}_i}$, *dann ist für den ersten Fall*

```
deptype (Y_1.a_1, X_1.b_1),
        ··· (Y_n.a_n, X'_n.b'_n) in Y.d
```

äquivalent mit

```
symbol X_1
attr ↑t_1 ←including Y_1.a_1
     ↑t_2 ←(this.t_1, this.b_1)

···

symbol X'_n
attr ↑t_1 ←including Y_n.a_n
     ↑t_2 ←(this.t_1, this.b'_n)
collect X_1.t_2, ···, X_n.t_2 in Y.d
```

Der zweite Fall ist analog durch Austausch der $\mathtt{X}_i$ *und* $\mathtt{Y}_i$ *in der erweiterten Variante herzustellen. Die Attribute* $\mathtt{t}_1, \mathtt{t}_2$ *sowie* `d` *sind neue Attribute.*

Es ist zu beachten, dass für $\mathtt{Y}_i \overset{*}{\rightsquigarrow} \mathtt{Y}_i$ die Abhängigkeiten in $\mathtt{Y}$ mehrfach vorkommen.

Der Nachweis, dass Muster 13 ein Muster ist, erfolgt durch folgendes Lemma:

Lemma 5.6. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 13 ein Muster.

Beweis. Durch Angabe der Komposition bekannter Muster und deren Komposition. Sei $\mathcal{M}_{incl}$ die Attributgrammatik-unabhängige Musterdefinition nach Lemma 4.30. Sei $Subs_i$ die Menge passender Substitutionen für $1 \leq i \leq n'$ wobei $[\mathtt{X}_i/S_0]$, $[\mathtt{t}_1/target]$ und $[a_i/from] \in Subs_i$. Sei $\mathcal{M}_{coll}$ die Attributgrammatik-unabhängige Musterdefinition für Muster 10 nach Lemma 5.3 mit Menge passender Substitionen $Subs_{coll}$ wobei $[\mathtt{t}_2/b_2] \in Subs_{coll}$ dann ist $\mathcal{M}_{coll} \circ \mathcal{M}_{incl}^{n'}$ ein Muster. □

Es existieren zwei weitere typische Muster, die für alle bisher vorgestellten typischen Muster verallgemeinert werden können. Die dafür notwendige Definition als Attributgrammatik-unabhängige Darstellung benötigt dafür im ersten Fall nur die, in den **`without`**-Klauseln angegebenen, Nichtterminale als Prädikat in Pr_+ anzugeben. Im Zweiten Fall ist eine bestimmte Attributierungsregel bezüglich eines Attributs vorgegeben, sodass auch hier die Definition als Attributgrammatik-unabhängige Darstellung nur begrenzt von der ursprünglichen Variante abweicht.

Typisches Muster 14. ***Filterung über Knoten für komplexe Beiträge***

Für eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und einem Nichtterminal $\mathtt{Y} \in N$ sowie Symbolen $\mathtt{X}_i \in \Sigma$ für $1 \leq i \leq n$, $n \in \mathbb{N}$ wobei für X_i, $1 \leq i \leq n-k$ gilt: $Y \overset{*}{\rightsquigarrow} X_i$ und es existieren j und l, $1 \leq j \leq n-k$, und $n-k < l \leq n$ sodass $\mathtt{X}_j \overset{*}{\rightsquigarrow} \mathtt{X}_l$ und für Werte von j und l $\mathtt{X}_l\neg \overset{*}{\rightsquigarrow} \mathtt{X}_j$ gilt.

Existiert ein l und j, sodass $\mathtt{X}_l \overset{+}{\rightsquigarrow} \mathtt{X}_j$, dann ist ohne Beschränkung der Allgmeinheit

```
contribute X_1.a_1,···,
           X_{n-k}.a_{n-k} to Y.c ←e
           using ⊕
via b_1 ←e_1,
    ···
    b_m ←e_m
without X_{n-k+1}, ···, X_n
```

äquivalent mit

```
contribute X_1.a_1,···,
           X_{n-k}.a_{n-k} to Y.c ←e
           using ⊕
via b_1 ←e_1
    ···
    b_m ←e_m

symbol X_{n-k+1}
attr ↑c ← ↓c

···

symbol X_l
attr ↑c ← ↓c
     head.c ← e

···

symbol X_n
attr ↑c ← ↓c
```

Existiert kein solches $\mathtt{X}_l$, dann wird die Attributierung von $\mathtt{X}_l$ durch die Attributierung von $\mathtt{X}_{n-k+1}$ ersetzt.

Das typische Muster 14 stellt somit einen Fall der Filterung über Knotentypen dar. Die Filterung über Attributwerte hingegen wird im typischen Muster 15 dargestellt. Für Muster 14 wird die Attributgrammatik-unabhängige Darstellung nicht weiter angegeben, diese stimmt, bis auf Pr_+, mit der im Beweis zu Lemma 5.2 angegebenen überein.

Typisches Muster 15. ***Filterung über Attribute für komplexe Beiträge***

Sei $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$. *Für* $\mathsf{Y} \in N$ *und* $\mathsf{X}_i \in \Sigma$ *für* $0 \leq i \leq n$, $n \in \mathbb{N}$ *und Attribute* $\mathsf{X}_i.\mathsf{a}_i, \mathsf{X}_i.\mathsf{b}_i \in A$, *für alle* $i, 0 \leq i \leq n$, *wobei* $\mathsf{Y} \overset{+}{\rightsquigarrow} \mathsf{X}_i$, *sowie* $\mathsf{X}_i \neg \overset{+}{\rightsquigarrow} \mathsf{Y}$, *dann ist*

```
contribute X1.a1 on X1.b1,
           ...,
           Xn.an on Xn.bn
           to Y.c ← e
           using ⊕
via c1 ←e1,
    ...
    cm ←em
```

äquivalent mit

```
chain c head Y ←e

symbol X1
attr ↑c1 ←e1 ↞↓c
     ...
     ↑cm ←em ↞this.bm−1
     ↑c ← if this.b1
            then ↓c ⊕ this.a1
            else ↓c
               ↞ this.bm

...

symbol Xn
attr ↑c1 ←e1 ↞↓c
     ...
     ↑cm ←em ↞this.bm−1
     ↑c ← if this.bm
            then ↓c ⊕ this.an
            else ↓c
               ↞ this.bm
```

Mit den bisher noch nicht verwendeten Attributen c_j, $0 < j \leq m$ *für* $m \in \mathbb{N}$ *und eine binäre Operation* $\oplus$.

Die beiden Muster 15 und 14 lassen sich darüber hinaus kombinieren. Auf eine Präsentation dieser Kombination sowie der Kombination mit anderen Mustern, wie „Aufsammeln“ oder dem typischen Muster 12 wird verzichtet. Intuitiv werden diese anderen Muster durch Anhängen der Bedingungen (**on** und **without**) an das eigentliche Muster und die vorgestellte Ersetzung erreicht. Als Muster lassen sich die Filterungen (Muster 14 und 15) durch Kombination der eigentlichen Muster unter Verwendung einschränkender Prädikate realisieren. Wie bereits beschrieben wird für Muster 14 mittels einer Einschränkung in Pr_+ diese Filterung realisiert, hingegen in Muster 15 mittels Vorgabe einer Substitution bei der Attributierungsfunktion.

5.2. Muster der Definitionstabelle

Eine Definitionstabelle ist eine Datenstruktur zum Speichern von Informationen, wie bspw. Typ oder erstes Vorkommen, zu einem Bezeichner. Grundsätzlich werden in der Definitionstabelle nur definierende Vorkommen gespeichert und bei der Referenz eines Bezeichners diese Tabelle abgefragt. In Abhängigkeit von der Sprachsemantik erfolgt dieses Nachschlagen anhand einer Blockstrucktur oder erst nach dem kompletten Befüllen dieser Tabelle. Ebenfalls kann die Definitionstabelle genutzt werden um Teile der Codegenerierung vorzugeben.

Die Definitionstabelle wird häufig in Domänen-spezifischen, aber auch allgemeinen Programmiersprachen, verwendet. Der erste Schritt bei der Arbeit mit der Definitionstabelle ist das Befüllen dieser. Häufig gibt es in Programmiersprachen eine Beziehung zwischen Definition und Benutzung, welche durch die Namensanalyse erstellt bzw. geprüft werden muss. In verschiedenen Sprachen ist es bspw. erlaubt, dass Elemente vor der Definition benutzt werden können, in anderen nicht. Darüber hinaus ist es gehäuft notwendig Informationen gleich in der Definitionstabelle abzulegen um diese an anderer Stelle wieder zur weiteren Verarbeitung aus dieser herauszuholen.

Basierend auf der Arbeit [74], in der ein abstrakter Datentyp zur Namensanalyse vorgestellt wird, existieren Muster, die geeignet sind die Definitionstabelle mit den gewünschten Kriterien zu füllen. Im Gegensatz zum reinen Datentyp aus [74] besteht bei den Mustern direkt die Möglichkeit Informationen in der

`Environment` Eine Definitionstabelle.

`Binding` Ein Zeiger auf einen Eintrag in die Definitionstabelle.

`StringTableKey` Ein Index in die Symboltabelle.

`String` Eine Zeichenfolge.

Sev Eine Meldungsart, $Sev = \{FATAL, ERROR, WARNING, NOTE\}$.

`bindKey`: `Environment` × `StringTableKey` → `Binding` Erstellung eines neuen Eintrags in der Definitionstabelle.

`bindingInEnv`: `Environment` × `StringTableKey` → `Binding` Suche eines Eintrags in der Definitionstabelle; falls nicht vorhanden liefert `NoBinding` als Rückgabewert.

`newscope`: `Environment` → `Environment` Erstellung eines neuen Gültigkeitsbereich „unterhalb“ des übergebenen Gültigkeitsbereichs

Tabelle 5.1. – Signaturen der verwendeten Funktionen aus [74] und Informationen zu den verwendeten Typen zur Umsetzung der Namensanalysen als Muster.

Definitionstabelle im Schritt des Definitionstabellenaufbaus abzulegen. In [74] ist der Anwendungsfall, dass dies erst in späteren Schritten geschieht.

Die Notwendigkeit einer Definitionstabelle zur Analyse der Definition-Benutzt-Struktur ergibt sich aus den vielen Beispielen in denen dies für weitere Analysen und die Codegenerierung von Relevanz ist. Selbst in Referenzattributgrammatiken kommt eine Definitionstabelle zum Einsatz. Die Referenz auf eine Definition wird in solch einer Definitionstabelle gespeichert und am Ort der Definition als Referenzattribut gesucht und geladen. Der Unterschied zur Definitionstabelle dieser Arbeit ist, dass Attribute, die am Ort der Definition bekannt sind und in der Definitionstabelle gespeichert werden, in Referenzattributgrammatiken am Ort der Referenz durch Aufruf an den Ort der Definition nur bei Bedarf berechnet werden. Dies bildet auch einen Nachteil, da, wie Boyland in [26] zeigt, es nicht entscheidbar ist, ob solch eine Attributgrammatik berechnet werden kann. Erst zur Laufzeit eines Evaluators dieser Attributgrammatik ist dann entscheidbar, ob für das gegebene Programm, die Attributgrammatik berechenbar ist.

In Abschnitt 2.1 wurden die typischen Arten der Namensanalyse für verschiedene DSLs gefunden: „Benutzung vor Definition erlaubt“ sowie „Definition vor Benutzung erzwungen“. Zur Umsetzung dieser Arten der Namensanalyse werden die Methoden aus [74] verwendet. Tabelle 5.1 gibt einen kurzen Überblick über die verwendeten Funktionen, deren Signatur und die dazugehörigen Typen.

In Anhang E.3 werden weitere Typen und Funktionen vorgestellt, die im Rahmen dieser Arbeit verwendet werden können. Im weiteren Verlauf dieser Arbeit ist ausreichend zu wissen, dass der Typ `Binding` einen Zeiger in die Definitionstabelle darstellt und `NoBinding` ein überprüfbarer, aber ungültiger Zeiger ist. `NoBinding` repräsentiert also einen nicht vorhandenen Eintrag der Definitionstabelle. Folgende Muster repräsentieren dann, die bereits erwähnten Möglichkeiten der Namensanalyse.

Typisches Muster 16. *Definition vor Benutzung erzwungen (Namensanalyse 1)*

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und Symbolen* $\mathsf{D}, \mathsf{R} \in \Sigma$*, sodass für ein Nichtterminal* $\mathsf{S} \in N$ *mit* $\mathsf{S} \overset{*}{\rightsquigarrow} D$ *und* $\mathsf{S} \overset{*}{\rightsquigarrow} R$ *gilt.*

Für Attribute $a \in A_{\mathsf{D}}, b \in A_{\mathsf{R}}$ *vom Typ* `StringTableKey`*, neuen Attributen* $\mathsf{bind}, \mathsf{t} \notin A$ *und* $sev_1, sev_2 \in Sev$ *steht*

```
def_before_use D.bind of D.a,
               R.bind of R.b in S.env
with sev1 undef
with sev2 predef
via D.bind:k1 ←e1
    ...
    D.bind:kn ←en
```

für

```
contribute D.a to S.t ← ∅
  using bindKey
  via D.bind ← bindKey(chain, tribute)
      D.bind:k1 ←e1
      ...
      D.bind:kn ←en
      chain ← chain ↞ D.bind:kn
symbol S
attr ↑env ← this.t

symbol D
attr cond D.bind ≠ NoBinding
  ⇒ report(sev2, "Already defined: "
                 ++ this.a)

symbol R
attr ↑bind ← bindInEnv(↓t, this.b)
     ↑t ← ↓t ↞ ↑bind
cond ↑bind ≠ NoBinding
   ⇒ report(sev1, "Unknown reference: "
                  ++ this.b)
```

Dabei sind $e_1, \cdots, e_n$ *beliebige Ausdrücke und* $k_1, \cdots, k_n$ *beliebige Spalten der Definitionstabelle,* $n \in \mathbb{N}$*.*

Typisches Muster 17. *Benutzung vor Definition erlaubt (Namensanalyse 2)*

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und Symbolen* $\mathsf{D}, \mathsf{R} \in \Sigma$*, sodass für ein Nichtterminal* $\mathsf{S} \in N$ *mit* $\mathsf{S} \overset{*}{\rightsquigarrow} D$ *und* $\mathsf{S} \overset{*}{\rightsquigarrow} R$ *gilt.*

Für Attribute $a \in A_{\mathsf{D}}, b \in A_{\mathsf{R}}$ *vom Typ* `StringTableKey` *steht*

```
use_before_def D.bind of D.a,
               R.bind of R.b in S.env
with sev1 undef
with sev2 predef
via D.bind:k1 ←e1
    ...
    D.bind:kn ←en
```

für

```
contribute D.a to S.t ← ∅
  using bindKey
  via D.bind ← bindKey(chain, tribute)
      D.bind:k1 ←e1
      ...
      D.bind:kn ←en
symbol S
attr ↑env ← this.t

symbol D
attr cond D.bind ≠ NoBinding
  ⇒ report(sev2, "Already defined: "
                 ++ this.a)

symbol R
attr ↑bind ←
      bindInEnv(including S.env,
                this.b)
    cond ↑bind ≠ NoBinding
   ⇒ report(sev1, "Unknown reference: "
                  ++ this.b)
```

⟨*LEntity*⟩ ::= *'List'* ⟨*EDef*⟩ *'='* *'discover'* *'('* ⟨*EType*⟩ *')'*

⟨*ForLoop*⟩ ::= *'for'* *'('* ⟨*EType*⟩ ⟨*LVarDef*⟩ *':'* ⟨*ERef*⟩ *')'* ⟨*ForStats*⟩

⟨*EDef*⟩ ::= *id*

⟨*ERef*⟩ ::= *id*

⟨*EType*⟩ ::= *id*

a) Ausschnitt einer abstrakten Syntax zur Beschreibung der Sprache aus Beispiel 2.1.

```
1  symbol LEntity attr ↑sym ← constituent EDef.sym
2  def_before_use LEntity.bind of LEntity.sym
3                 ERef.bind of ERef.sym in Program.names
4  with error unknown
5  with error predef
6  via LEntity.bind:type ← constituent EType.sym
7  symbol ForLoop
8  attr ↑type ← constituent EType.sym
9       ↑ref ← constituent ERef.bind
10 cond ↑ref:type = ↑type  ⇒ error "type mismatch" ++ ↑ref:type ++
11                                              " vs. " ++ ↑type
```

b) Attributgrammatik zur Verwendung der Definitionstabelle mit Übertragung von Typinformationen.

Beispiel 5.2 – Beispiel einer Typanalyse mit Verwendung der Definitionstabelle zur Veranschaulichung der Notwendigkeit des Informationstransports mittels Definitionstabelle.

Dabei sind `bind`, `t` *neue Attribute,* sev_1 *und* sev_2 *Elemente vom Typ Sev,* $e_1, \cdots, e_n$ *beliebige Ausdrücke und* $k_1, \cdots, k_n$ *beliebige Spalten der Definitionstabelle.* `bindKey` *ist dabei der Operator* $\oplus$ *nach Muster 9.*

In den beiden Mustern 16 und 17 wird ein Attribut (`bind`) als Kopie des Resultats der Anwendung der Operation ($\oplus$) zum lokalen Zugriff auf den aktuellen Zustand der Definitionstabelle hinzugefügt. Mit diesem Attribut werden dann die Spalten der Definitionstabelle zusätzlich befüllt. Die Spalten der Definitionstabelle sind typisiert, genauso wie der Zugriff und das Befüllen der Spalten.

Die in diesen Mustern präsentierte Namensanalyse lässt sich nutzen um zu prüfen, dass in Beispiel 2.1 die abgebildeten Ereignisse auch korrekt zugegriffen werden. In Beispiel 2.1 auf Seite 12 wird beim Behandeln des Ereignisses der Temperaturänderung ein anderer Name benutzt, um auf die Werte zuzugreifen, als dieser noch direkt darüber definiert wurde (siehe Zeilen 17 und 18 von Beispiel 2.1).

Ein weiteres Beispiel zur Notwendigkeit um Informationen vom Ort der Definition zum Ort der Benutzung, bspw. durch Nutzung der Definitionstabelle, zu transportieren, wird in Beispiel 5.2 gezeigt.

In anderen Sprachen, wie der Anforderungsbeschreibungssprache (Beispiel 1.2) oder der Sprache aus Beispiel 2.3, ist die Verwendung von Bezeichnern vor der eigentlichen Benutzung sinnvoll.

Im Rahmen dieser Arbeit werden die in Muster 16 und 17 vorgestellten Varianten in verschiedene Attributgrammatik-unabhängige Darstellungen überführt. Folgendes Lemma beschreibt die erste Variante – Definition vor Benutzung erzwungen:

Lemma 5.7. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 16 ein Muster für die erste Variante.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{d1}$. Sei $\mathcal{M}_{cont}$ eine Attributgrammatik-unabhängige Musterdefinition für Muster 9 nach Lemma 5.2, $\mathcal{M}_{cont} = \mathcal{M}_{acc} \circ \mathcal{M}_s^{m'} \circ \mathcal{M}_c$. Dann ist für $\mathcal{M}_{d1}$ für alle m' $\left[\mathtt{D}/S\right] \in Subs_i$, $1 \leq i \leq m'$ und $m' = n + 1$ (der Darstellung

in Muster 16). Dabei ist dann $\left[\texttt{chainIn}/b_1, \texttt{bindKey}/f, \texttt{a}/b_2\right] \in Subs_1$ der passenden Substitutionen für die Symbolattributierungen $\mathcal{M}_s^1$ sowie $\left[\texttt{bind} : \texttt{k}_i/a_i, \texttt{bind} : \texttt{k}_{i-1}/b_r\right] \in Subs_i$ für alle $1 < i \leq m'$. Weiterhin sei $\mathcal{M}_{s'}$ eine Symbolattributierung mit passenden Substitutionen $Subs_{s'}^j$ für $j \in \mathbb{N}$ wobei $\left[\texttt{D}/S, \texttt{cond}/a_1, \textbf{if}/f, \cdots\right] \in Subs_{s'}^1$, $\left[\texttt{R}/S, \texttt{bind}/a_1, \cdots\right] \in Subs_{s'}^2$ und $\left[\texttt{R}/S, \texttt{cond}/a_1\textbf{if}/f, \cdots\right] \in Subs_{s'}^2$.

Dann ist $\mathcal{M}_{d1} = \mathcal{M}_{s'}^+ \circ \mathcal{M}_{cont}$. □

Die Variante, die Benutzung vor Definition erlaubt, wird durch folgendes Lemma beschrieben:

Lemma 5.8. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 17 ein Muster für die zweite Variante.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{d2}$. Sei $\mathcal{M}_{cont}$ eine Attributgrammatik-unabhängige Musterdefinition für Muster 9 nach Lemma 5.2, $\mathcal{M}_{cont} = \mathcal{M}_{acc} \circ \mathcal{M}_s^m \circ \mathcal{M}_c$. Dann ist für $\mathcal{M}_{d1}$ für alle m' $\left[\texttt{D}/S\right] \in Subs_i$, $1 \leq i \leq m'$ und $m' = n+1$ (der Darstellung in Muster 17). Dabei ist dann $\left[\texttt{chainIn}/b_1, \texttt{bindKey}/f, \texttt{a}/b_2\right] \in Subs_1$ der passenden Substitutionen für die Symbolattributierungen $\mathcal{M}_s^1$ sowie $\left[\texttt{bind} : \texttt{k}_i/a_i, \texttt{bind} : \texttt{k}_{i-1}/b_r\right] \in Subs_i$ für alle $1 < i \leq m'$. Sei $\mathcal{M}_{incl}$ eine Attributgrammatik- unabhängige Musterdefinition für Muster 4 nach Lemma 4.30 und passenden Substitutionen $Subs_{incl}^{(1)}$ wobei $\left[\texttt{env}/from, \texttt{S}/S_0\right] \in Subs_{incl}^{(1)}$ und $\left[\texttt{R}/A_i, \texttt{env}/targ\right] \in Subs_{incl}^{(v)}$ für $v \in \mathbb{N}$. Weiterhin sei $\mathcal{M}_{s'}$ eine Symbolattributierung mit passenden Substitutionen $Subs_{s'}^i$ für $i \in \mathbb{N}$ wobei $\left[\texttt{D}/S, \texttt{cond}/a_1, \textbf{if}/f, \cdots\right] \in Subs_{s'}^1$, $\left[\texttt{R}/S, \texttt{bind}/a_1, \cdots\right] \in Subs_{s'}^2$ und $\left[\texttt{R}/S, \texttt{cond}/a_1, \textbf{if}/f, \cdots\right] \in Subs_{s'}^2$.

Dann ist $\mathcal{M}_{d2} = \mathcal{M}_{s'}^+ \circ \mathcal{M}_{incl}^+ \circ \mathcal{M}_{cont}$ □

Anmerkung: In Muster 16 wird die Definitionstabelle im Attribut `env` des Symbols `S` zur Verfügung gestellt. Dies spiegelt sich nicht in den Lemmata 5.7 und 5.8 wider. Die Darstellung in Muster 16 orientiert sich in diesem Punkt an der Notwendigkeit dieses Attributs in Muster 17 ist jedoch nicht notwendig für den Beweis.

Ausgehend von Beispiel 5.2 und der Vorstellung der Muster der Definitionstabelle dieser Arbeit kann gezeigt werden, wie viel Umfang an Attributierungsregeln – insbesondere semantisch irrelevanter Attributierungsregeln – durch Verwendung von Mustern eingespart werden kann. Einen Ausschnitt – nicht in Beispiel 5.2a aufgeführte Produktionen werden nicht attributiert – aus der aus Beispiel 5.2 erzeugbaren Attributgrammatik, wird in Beispiel 5.3 gezeigt.

Bemerkung (Modulsysteme und Namensanalysen)**.** Andere Varianten der Namensanalyse oder auch die Analyse von Importsystemen oder Modulsystemen sind mit Mustern ebenfalls möglich. Diese würden jedoch den Rahmen dieser Dissertation überschreiten. Darüber hinaus benötigt ein Importsystem oder ein Modulsystem weitere Unterstützung, die über geordnete Attributgrammatiken hinaus gehen (können). So werden, je nach Art des Importsystems, Attributgrammatiken höherer Ordnung verwendet um die importierten Dateien einzulesen und als Teilbaum im abstrakten Syntaxbaum einzufügen. Alternativen über die Benutzung der Definitionstabelle sind mit hohem Aufwand über Hilfsfunktionen realisierbar. Allein die Vorstellung aller möglicher Alternativen und deren Umsetzung zur Namensanalyse über mehrere Dateien hinweg, würde den Rahmen dieser Arbeit überschreiten.

Ebenso häufig wie die Namensanalyse selbst kommt es vor, dass erst nach Durchführung der Namensanalyse, Werte bestimmt und abgespeichert werden können. Häufig werden diese Werte dann an anderer Stelle benötigt.

```
1  rule LEntities ::= LEntities LEntity
2  attr LEntities2.namesIn ← LEntities1.namesIn
3       LEntity.namesIn ← LEntities2.namesOut
4       LEntities1.namesOut ← LEntity.namesOut
5
6
7  rule LEntity ::= EDef EType
8  attr LEntity.sym ← EDef.sym
9       LEntity.type ← EType.sym
10      LEntity.bind ← bindKey(LEntity.namesIn, LEntity.sym)
11      LEntity.bind:type ← LEntity.type ↞ LEntity.bind
12      EDef.namesIn ← LEntity.namesIn ↞ LEntity.bind:type
13      EType.namesInt ← EDef.namesOut
14      LEntity.namesOut ← EType.namesOut
15 cond LEntity.sym ∉ LEntity.names ⇒ error "Already defined: " ++ LEntity.sym
16
17 rule ForLoop ::= EType LVarDef ERef ForStats
18 attr ForLoop.type ← EType.sym
19      ForLoop.ref ← ERef.bind
20      EType.namesIn ← ForLoop.namesIn
21      LVarDef.namesIn ← EType.namesOut
22      ERef.namesIn ← LVarDef.namesOut
23      ForStats.namesIn ← ERef.namesOut
24      ForLoop.namesOut ← ForStats.namesOut
25 cond ForLoop.ref:type = ForLoop.type ⇒ error "type mismatch" ++ ForLoop.ref:type ++
26                                       " vs. " ForLoop.type
27
28 rule EDef ::= id
29 attr EDef.namesOut ← EDef.namesIn
30
31 rule ERef ::= id
32 attr ERef.bind ← bindInEnv(ERef.namesIn, ERef.sym)
33      ERef.namesOut ← ERef.namesIn ↞ ERef.bind
34 cond ERef.sym ∈ ERef.namesIn ⇒ error "Unknown reference " ++ ERef.sym
35
36 rule EType ::= id
37 attr EType.namesOut ← EType.namesIn
```

Beispiel 5.3 – Aus Beispiel 5.2 hergeleitete Attributgrammatik für in Beispiel 5.2 angegebenene Grammatik unter Rückführung der Muster auf die Basisform von Attributgrammatiken ohne Darstellung aller Kopierregeln.

Typisches Muster 18. ***Speichern und Laden*** *(auch Transfer)*

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und Symbolen* $X, Y \in \Sigma$*, sowie Nichtterminal* $S \in N$ *mit* $S \overset{*}{\rightsquigarrow} X$ *und* $S \overset{*}{\rightsquigarrow} Y$ *und den Attributen* $\mathsf{a} \in A_X$ *und* $\mathsf{b} \in A_Y$ *vom Typ* `Binding` *sind,* e *ein beliebiger Ausdruck und* c *ein neues Attribut ist, dann*

```
store_load X.a:k_a ←e
  through Y.b in c from S
```

entspricht

```
contribute X.a:k to S.gotk_a ←()
  via X.a:k ← e
      chain ← chain

symbol S
attr ↓gotResk_a ←S.gotk_a

symbol Y
attr ↑c ← Y.b:k_a
       ↞ including S.gotResk_a
```

für neue void-Attribute $\mathsf{gotResk}_a$ *und* gotk_a *und einer Definitionstabellenspalte* k*.*
*Der Teil „***from** *S“ kann weggelassen werden, wenn* $\mathsf{S} = Z$*.*

Am Ende von Abschnitt 3.3 wurden *void*-Attribute und deren Bedeutung bereits eingeführt.

Es wird davon ausgegangen, dass bei dem Muster 18 die Attribute a und b der entsprechenden Symbole durch eine Namensanalyse entstanden sind. Ist der Ausdruck e unabhängig von durch ähnliche Muster nach der Namensanalyse durchgeführten Berechnungen von Attributen, dann hat dieses Muster in der Regel keinen Einfluss auf zusätzliche Laufzeit. Auch viele der zusätzlichen Attribute dieses Musters führen

nicht zu weiterem Speicherverbrauch. Einzig wirklich neues Attribut ist das Attribut `c` in welchem das Ergebnis nach dem Abspeichern in der Definitionstabelle nach dem „Transport“ gespeichert wird.

Die Namensanalyse kann häufig mittels Namensbereichen erweitert werden. Wenngleich dies im Rahmen der Entwicklung von Domänen-spezifischen Sprachen nicht so komplex ist, wie in richtigen Programmiersprachen, kann auch für diesen Anwendungsfall ein Muster angegeben werden. Auf eine Präsentation dieses Musters wird aufgrund der gesteigerten Komplexität und dem damit einhergehenden Quellumfang verzichtet. Für den noch komplexeren Fall der Datei-übergreifenden Namensanalyse unter Verwendung umfangreicher Bibliotheksfunktionalität mit gleichzeitigem Importsystem existerien weitere alternative Muster, die jedoch nicht der eigentlichen Definitionstabelle zuzuordnen sind.

Zum Nachweis der Darstellung als Attributgrammatik-unabhängige Musterdefinition für das Muster des Speicherns und Ladens können dieselben Prinzipien wie beim Muster der Definitionstabelle angewandt werden:

Lemma 5.9. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 18 ein Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{sl}$. Sei $\mathcal{M}_{cont}$ eine Attributgrammatik-unabhängige Musterdefinition für Muster 9 nach Lemma 5.2, $\mathcal{M}_{cont} = \mathcal{M}_{acc} \circ \mathcal{M}_s^m \circ \mathcal{M}_c$. Mit $Subs_1$ den passenden Substitutionen für $\mathcal{M}_s^1$ und $\left[\mathtt{a} : \mathtt{k}/a_1, \mathtt{X}/S\right] \in Subs_1$ und

$$\left[\mathtt{gotkOut}_a/a_1, \mathtt{gotkIn}_a/b_1, \mathtt{a} : \mathtt{k}/b_2, \mathtt{X}/S\right] \in Subs_2$$

passenden Substitutionen für $\mathcal{M}_s^2$. Sei weiterhin $\left[\mathtt{gotkIn}_a/aI_h\right] \in Subs_c$ den passenden Substitutionen für $\mathcal{M}_c$. Darüber hinaus Sei $\mathcal{M}_{s'}$ eine Attributgrammatik-unabhängige Musterdefinition der Symbolattributierung (Basismuster 2, Basismuster 3, Lemma 4.26) mit passenden Substitutionen $Subs_{s'}$ wobei $[\mathtt{gotResk}_a/a_2, \mathtt{gotReskOut}_a/b_1, \mathtt{S}/S] \in Subs_{s'}$ sowie $\mathcal{M}_{s,Y}$ und $\mathcal{M}_{incl}$ Attributgrammatik-unabhängigen Musterdefinitionen der Symbolattributierung und dem Kopieren nach unten mit passenden Substitutionen $Subs_{incl}$ und $Subs_{s,Y}$. Mit $[\mathtt{S}/S_0, \mathtt{Y}/A_i, \mathtt{gotRek}_a/from] \in Subs_{incl}$ und $[\mathtt{Y}/S, \mathtt{c}/a_1, \mathtt{b} : \mathtt{k}_a/b_1] \in Subs_{s,Y}$ ist

Damit ist $\mathcal{M}_{sl} = \mathcal{M}_{s,Y} \circ \mathcal{M}_{incl} \circ \mathcal{M}_{s'} \circ \mathcal{M}_{cont}$ ein Muster. □

Aufbauend auf dem Muster des Speichern und Ladens (typ. Muster 18) existiert das Muster der Fortsetzung. Bei dem Muster der Fortsetzung ist die Idee, dass ein Wert, der in der Definitionstabelle gespeichert ist, noch nicht vollständig ist. Damit ein Wert jedoch fortgesetzt werden kann, muss ein Schlüssel der Definitionstabelle vorhanden sein. Für eine Fortsetzung muss eine Unterstützung der abstrakten Syntax vorliegen. In den bisher verwendeten Grammatiken ist dies nicht der Fall. Ein Beispiel findet sich im Aufsammeln der Attributierungsregeln einer Attributgrammatik. Für die Bestimmung der Abhängigkeiten werden alle Attributierungsregeln aller Symbole und Regeln benötigt. In den üblichen Implementierungen sollen Attributierungen jedoch nach Semantik und nicht nach Symbol (bzw. Produktion) gruppiert werden. Selbst für separat angegebene Grammatiken und damit eindeutige Definitionen von Symbolen und Produktionen, wäre dann das Aufsammeln und die Fortsetzung unterhalb einer Verwendung anzuordnen. Da das Einfügen der Fortsetzung in die bestehenden Muster wegen dieser Argumentation schwer herzustellen ist, ist die Fortsetzung wie folgt definiert:

Typisches Muster 19. ***Fortsetzung***

Sei $AG \triangleq (G, A, R, B)$ *eine Attributgrammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *mit Nichtterminal* $\mathsf{Y} \in N$ *und Symbolen* $\mathsf{X}_i \in \Sigma$*, Attributen* $\mathsf{X}_i.\mathsf{k}_i$*, und Definitionstabelleneintrag* p *und Initialwert* e *mit* $0 \leq i \leq n$ *für ein* $n \in \mathbb{N}$*. Sei darüber hinaus* $\mathsf{Y} \overset{+}{\rightsquigarrow} \mathsf{X}_i$*, dann steht*

```
continue p in X1.k1 with X1.a1 ←e1,
                 ...,
              Xn.kn with Xn.an ←en
  below Y
```

für

```
chain c head Y ← true
symbol X1
attr ↓a1 ←e1 ↞↓c
     this.k1:p ←↑a1
     ↑c ← ↓c ↞ this.k1:p

...

symbol Xn
attr ↓an ←en ↞↓c
     this.kn:p ←↑an
     ↑c ← ↓c ↞ this.k1:p
```

falls $\mathsf{X}_i.\mathsf{a}_i$ *ebenfalls Kettenattribute sind, andernfalls wird jeweils die Richtung durch „***`this`***“ ersetzt. Die Ausdrücke* e_i *sind beliebige Ausdrücke.*

Im typischen Muster 19 wird keine Einschränkung bezüglich der Ausdrücke e_i gemacht. In der praktischen Anwendung dieses Musters wird in diesen Ausdrücken ein Wert aus der Definitionstabelle gelesen. Die Fortsetzung ist somit der Abschluss von Kettenattributierungen und Symbolattributierungen mit Definitionstabelle. Für Attribute, bei denen die Richtung (↑, ↓) durch **`this`** ersetzt wurde, wird die Richtung des Attributs hergeleitet.

Lemma 5.10. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 19 ein Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{continue}$. Sei $\mathcal{M}_{c,h}$ eine Attributgrammatik-unabhängige Musterdefinition nach Lemma 4.27 mit passenden Substitutionen $Subs_{c,h}$ wobei $\left[\mathsf{Y}/S_0, \mathtt{cIn}/aI_0, \mathtt{cOut}/aO_0, \cdots\right] \in Subs_{c,h}$. Sei weiterhin $\mathcal{M}_{s,i,j}$ eine Attributgrammatik-unabhängige Musterdefinition nach Lemma 4.26 mit passenden Substitutionen $Subs_i^j$ wobei $[\mathsf{X}_i/S_0, \mathsf{a}_i/a_2, \mathtt{cIn}/c_r]$ für alle $1 \leq i \leq n$ für $j = 1$ und weiterhin für $j = 2$ $\left[\mathsf{X}_i/S_0, \mathsf{k}_i : \mathsf{p}/a_1, \mathtt{aOut}_i/b_r\right]$ und für $j = 3$ $\left[\mathsf{X}_i/S_0, \mathtt{cOut}/a_1, \mathsf{k}_i : \mathsf{p}/b_r, \mathtt{cIn}/b_1\right]$ dann ist $\mathcal{M}_{continue} = (\mathcal{M}_{s,i}^{+})_j^{+} \circ \mathcal{M}_{c,h}$ □

Ein weiteres Muster, welches mit der Namensanalyse und der Verwendung der Definitionstabelle im Zusammenhang steht ist das Muster, welches die Verwendung von Gültigkeitsbereichen (engl. Scopes) erlaubt. Dieses Muster selbst ist wiederum abhängig von der Verwendung des Musters zur Namensanalyse (typ. Muster 16 oder typ. Muster 17). Da dieses Muster deshalb das typische Muster 16 bzw. typisches Muster 17 nur erweitert, wird nur eine Variante in dieser Arbeit präsentiert. Die anderen Fälle lassen sich analog definieren.

Typisches Muster 20. ***Gültigkeitsbereich (Benutzung vor Definition)***

Sei eine attributierte Grammatik $AG \triangleq (G, A, R, B)$ *mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$ *und Symbolen* $\mathtt{D}, \mathtt{R} \in \Sigma$*, sodass für ein Nichtterminal* $\mathtt{S} \in N$ *mit* $\mathtt{S} \overset{*}{\rightsquigarrow} D$ *und* $\mathtt{S} \overset{*}{\rightsquigarrow} R$ *gilt. Existiert darüber hinaus ein Nichtterminal* $\mathtt{M} \in N$*, sodass* $\mathtt{M} \overset{+}{\rightsquigarrow} D$*,* $\mathtt{M} \overset{+}{\rightsquigarrow} \mathtt{R}$*,* $\mathtt{S} \overset{+}{\rightsquigarrow} \mathtt{M}$ *und darüber hinaus* $\mathtt{M} \overset{+}{\rightsquigarrow} \mathtt{M}$ *und es existiert ein Attribut* `M.bind` *vom Typ* `Binding`*, dann ist*

```
use_before_def D.bind of D.a,
               R.bind of R.b in S.env
               scope on M.bind:s
with sev1 undef
with sev2 predef
via D.bind:k1 ←e1
    ...
    D.bind:kn ←en
```

entspricht

```
contribute D.a to S.t ← ∅
  via D.bind ← bindKey chain tribute
       D.bind:k1 ←e1
       ...
       D.bind:kn ←en
       chain ← chain

chain t head M ← new_scope ↓t
symbol M
attr ↑bind:s ← tail.t
     ↑t ← ↓t ← ↑bind:s
     ↑env ← tail.t

symbol S
attr ↑env ← this.t

symbol D
cond D.bind ≠ NoBinding
  ⇒ report(sev2, "Already defined: "
                 ++ this.a)

symbol R
attr ↑bind ←
    bindInEnv((including S.env, M.env,
                         MR.env), this.b)
    cond ↑bind ≠ NoBinding
     ⇒ report(sev1, "Unknown reference:"
                    ++ this.b)
```

Muster 20 kann als Erweiterung und Verallgemeinerung von Muster 19 aufgefasst werden. In Lemma 5.11 wurde nicht erzwungen, dass sie Attribute $\mathtt{a}_i$ implizit durch Kettenattributierung erzeugt wurden.

Lemma 5.11. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 20 ein Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{scope}$.

Sei $\mathcal{M}_{cont}$ eine Attributgrammatik-unabhängige Musterdefinition für Muster 9 nach Lemma 5.2, $\mathcal{M}_{cont} = \mathcal{M}_{acc} \circ \mathcal{M}_s^m \circ \mathcal{M}_c$. Dann ist für $\mathcal{M}_{d1}$ für alle m' $[\mathtt{D}/S] \in Subs_i$, $1 \leq i \leq m'$ und $m' = n{+}1$ (der Darstellung in Muster 16). Dabei ist dann $[\mathtt{chainIn}/b_1, \mathtt{bindKey}/f, \mathtt{a}/b_2] \in Subs_1$ der passenden Substitutionen für die Symbolattributierungen $\mathcal{M}_s^1$ sowie $[\mathtt{bind} : \mathtt{k}_i/a_i, \mathtt{bind} : \mathtt{k}_{i-1}/b_r] \in Subs_i$ für alle $1 < i \leq m'$. Sei $\mathcal{M}_{incl}$ eine Attributgrammatik- unabhängige Musterdefinition für Muster 4 nach Lemma 4.30 und passenden Substitutionen $Subs_{incl}^1$ wobei $[\mathtt{env}/from, \mathtt{S}/S_0] \in Subs_{incl}^1$ und $[\mathtt{R}/A_i, \mathtt{env}/targ] \in Subs_{incl}^v$ für $v \in \mathbb{N}$. Weiterhin sei $\mathcal{M}_{s'}$ eine Symbolattributierung mit passenden Substitutionen $Subs_{s'}^i$ für $i \in \mathbb{N}$ wobei $[\mathtt{D}/S, \mathtt{cond}/a_1, \mathbf{if}/f, \cdots] \in Subs_{s'}^1$, $[\mathtt{R}/S, \mathtt{bind}/a_1, \cdots] \in Subs_{s'}^2$ und $[\mathtt{R}/S, \mathtt{cond}/a_1, \mathbf{if}/f, \cdots] \in Subs_{s'}^2$. Mit $\mathcal{M}_{c'}$ einer Kettenattributierung mit $[\mathtt{M}/S_0, \mathtt{tIn}/aI_1, \mathtt{new\_scope}/f, \mathtt{tIn}/b_1] \in Subs_{c'}$ passenden Substitutionen, sowie $\mathcal{M}_{tails,M,1}$ mit $[\mathtt{M}/S_i, \mathtt{bind} : \mathtt{s}/bO_1, \mathtt{tOut}/a0_n] \in Subs_{tails,M,1}$ $\mathcal{M}_{tails,M,2}$ mit $[\mathtt{M}/S_i, \mathtt{env}/bO_1, \mathtt{tOut}/a0_n] \in Subs_{tails,M,2}$ und darüber hinaus $\mathcal{M}_{s,M}$ mit $[\mathtt{M}/S, \mathtt{tOut}/a_1, \mathtt{tIn}/b_1, \mathtt{bind} : \mathtt{s}/b_r] \in Subs_{s,M}$.

Dann ist $\mathcal{M}_{scope} = \mathcal{M}_{s,M} \circ \mathcal{M}_{tails,M}^+, \circ \mathcal{M}_{s'}^+ \circ \mathcal{M}_{incl}^+ \circ \mathcal{M}_{cont}$ □

Viele Eigenschaften von Elementen lassen sich mit Hilfe der bisher vorgestellten Muster bestimmen indem eine Namensanalyse durchgeführt wird und dann zusätzliche Eigenschaften mittels „Speichern und Laden“ (Muster 18) in der Definitionstabelle abgelegt werden. Eigenschaften, die durch Kombination dieser Definitionstabelleneinträge erzeugt werden, lassen sich durch folgende Muster ausdrücken:

Typisches Muster 21. ***Transformation***

Sei $AG \triangleq (G, A, R, B)$ *eine attributierte Grammatik mit abstrakter Syntax* $G \triangleq (N, T, P, Z)$*. Für ein Symbol* $\mathsf{S} \in N$ *mit Attribut* $\mathtt{env} \in A_{\mathsf{S}}$*, erzeugt durch die Namensanalyse (typisches Muster 16 oder typisches Muster 17) und identisch mit dem dortigen Attribut sowie Spalten der Definitionstabelle* $\mathsf{k}_1, \cdots, \mathsf{k}_n$ *und Symbolen* $\mathsf{D}, \mathsf{R} \in \Sigma$ *mit Attributen* $\mathtt{bind} \in A_{\mathsf{R}}$ *und* $\mathtt{bind} \in A_{\mathsf{D}}$*, ebenfalls entsprechend dem typischen Muster 16 (oder typ. Muster 17) erzeugt. Sei* A *entweder* R *oder* D*, dann ist*

```
each b of S.env : b:k ← f in A.c
```

äquivalent mit

```
symbol S
attr ↑gotk ← h(this.env)

symbol A
attr ↑c ← A.bind:k
       ← including S.gotk
```

wobei $\mathtt{c}$ *ein neues Attribut ist,* $\mathtt{gotk}$ *ein neues void-Attribut und* $\mathtt{h}$ *eine neue Hilfsfunktion ist, die für alle Elemente der Definitionstabelle* $\mathtt{b}$ *die Berechnung* f *durchführt und in der Spalte* $\mathtt{k}$ *ablegt.* f *ist eine beliebige Funktion über beliebige* k_i*,* $i \in [1, n]$ *für* $n \in \mathbb{N}$ *des Elements* $\mathtt{b}$*.*

Die Darstellung als Attributgrammatik-unabhängige Musterdefinition ist verglichen mit den übrigen Varianten dieses Kapitels einfach:

Lemma 5.12. Für alle zerlegbaren Attributgrammatiken $AG \in \mathcal{AG}_G$ mit abstrakter Syntax G, $AG \triangleq (G, A, R, B)$ ist Muster 21 ein Muster.

Beweis. Durch Angabe einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{trafo}$. Sei $\mathcal{M}_{s,S}$ eine Attributgrammatik-unabhängige Musterdefinition der Symbolattributierung nach Lemma 4.26, mit passenden Substitutionen $Subs_{s,S}$ mit $\left[\mathtt{gotk}/a_1, \mathsf{S}/S, \mathtt{env}/b_1, \mathsf{h}/f\right] \in Subs_{s,S}$ und einer Attributgrammatik-unabhängigen Musterdefinition $\mathcal{M}_{incl}$ nach Lemma 4.30 mit passenden Substitutionen $Subs_{incl}$, wobei

$$\left[\mathsf{S}/S_0, \mathtt{A}/A_i, \mathtt{gotK}/from, \mathtt{t}/targ\right] \in Subs_{incl}$$

Weiterhin sei $\mathcal{M}_{s,A}$ eine Attributgrammatik-unabhängige Musterdefinition der Symbolattributierung nach Lemma 4.26, mit passenden Substitutionen $Subs_{s,A}$ mit $\left[\mathtt{A}/S, \mathtt{c}/a_1, \mathtt{bind} : \mathsf{k}/b_1, \mathtt{t}/b_r\right] \in Subs_{s,A}$.

Dann ist $\mathcal{M}_{trafo} = \mathcal{M}_{s,A} \circ \mathcal{M}_{incl} \circ \mathcal{M}_{s,S}$. □

Beim Transformationsmuster wird somit ein neues virtuelles Attribut, d.h. Abhängigkeiten, sowie ein „Ergebnisattribut“ hinzugefügt. Zur Bestimmung der Laufzeit werden Informationen über die verwendete Funktion benötigt. Unabhängig von der verwendeten Funktion wird jedes Element der Definitionstabelle jedoch mindestens einmal besucht. Weitere Besuche könnten in Abhängigkeit von der Struktur der Definitionstabelle und der verwendeten Funktion verursacht sein. Solch eine „rekursive“ Struktur der Definitionstabelle kann durch das Muster 13 erzeugt sein.

5.3. Abschließende Bemerkungen zu Typischen Mustern

Im Rahmen dieser Arbeit wurde bisher beschrieben, wie typische Muster konstruiert werden können, sodass eine zerlegbare oder gar eine geordnete Attributgrammatik bei Anwendung auf ebensolche erzeugt wird. Gegenstand diesen Abschnitts ist der exemplarische Nachweis, dass die Anwendung eines Musters auf einer zerlegbaren Attributgrammatik ebenso eine zerlegbare Attributgrammatik erzeugt. Darüber hinaus wird in diesem Abschnitt auf ausgewählte Fragestellungen bzgl. Mustern, Antimustern und funktionaler Programmierung eingegangen.

```
1  rule Program ::= Decls
2  rule Decls ::= Decls Decl
3  rule Decls ::= ε
4  rule Decl ::= RqDecl
5  rule RqDecl ::= RqDefId RqReferences
6  rule RqDefId ::= id
7  rule RqReferences ::= RqReferences RqReference
8  rule RqReferences ::= RqReference
9  rule RqReference ::= id
```

a) Initiale, geordnete Attributgrammatik zu Beispiel 1.2.

```
1  contribute RqDefId.sym to Program.decls ← ∅
2             using bindKey
```

b) Anweisungen zur Anwendung von Muster 9 auf die initale Attributgrammatik.

Beispiel 5.4 – Ursprüngliche (geordnete) Attributgrammatik zu Beispiel 1.2 ohne Attributierungsregeln (5.4a) und Anweisungen zur Benutzung von Muster 9 (5.4b).

5.3.1. Zerlegbarkeit, Ordnung und komplexe Muster

Für den exemplarischen Nachweis, dass die resultierende Attributgrammatik zerlegbar (bzw. geordnet) ist, wenn ein Muster auf diese angewandt wird, wird in diesem Abschnitt das einleitende Beispiel, Beispiel 1.2, herangezogen und die Namensanalyse vorbereitend auf Muster 9 zurückgeführt.

Beispiel 5.4 zeigt ursprüngliche (zerlegbare und geordnete) Attributgrammatik, sowie die Anweisungen bzgl. Muster 9.

Nach Lemma 5.2 werden eine Reihe von Substitutionen durch die Anwendung des Musters festgelegt. Unmittelbar sind dies, nach Lemma 5.2 und Beispiel 5.4 für die einzelnen Mengen passender Substitutionen

$$\begin{aligned} \left[\texttt{Program}/S_0, \texttt{declsIn}/aI_1, \emptyset/f\right] &\in Subs_c && \text{für } 1 \leq i \leq i_{c,n} \\ \left[\texttt{RqDefId}/S, \texttt{declsOut}/a_1, \texttt{declsIn}/b_{r+1}\right] &\in Subs_{acc} && \text{für } r = 1 \\ \left[\texttt{bindKey}/f, \texttt{sym}/b_1\right] &\in Subs_{acc} \end{aligned}$$

Anhand dieser angegebenen Substitutionen, können für alle anderen möglichen Substitutionen und Attributwertterme der Attributgrammatik-unabhängigen Musterdefinition entsprechende Substitutionen hergeleitet werden. Initial gilt für $Subs_c$, dass S_0 durch `Program` ersetzt wird, damit dies gilt, werden für alle Produktionen mit linker Seite `Program` entsprechende Substitutionen in $Subs_c$ eingefügt, sodass Pr_+ nach Lemma 4.27 wahr wird. Somit, falls $\left[\texttt{Program}/S_0\right] \in Subs_c$, dann auch $\left[\texttt{Decls}/S_1\right] \in Subs_c$ und für alle S_j mit $1 < j \leq n$ – n Anzahl der zu substituierenden Symbole nach Darstellung der Attributgrammatik-unabhängigen Musterdefinition aus Lemma 4.27 – ist $\left[\varepsilon/S_j\right] \in Subs_c$. Damit ist dann auch nach der Definition von At_+ bzgl. Lemma 4.27 für diese Produktion aI_1 ersetzt durch `Decls.declsIn` sowie f durch $\emptyset$. Somit ist für alle b_k, $1 \leq k \leq m$ (siehe Beweis zu Lemma 4.27) $\left[\varepsilon/b_k\right] \in Subs_c$. Zur Anwendung von Muster 1 müssen weitere Substitutionen hergeleitet werden, da nach Beweis von Lemma 4.27 sich die Anwendung des Musters über den Abschluss nach Definition 4.21 bildet. Dafür ergeben sich dann für jede Produktion Substitionen dergestallt, dass die klassische Kettenattributierung (links-rechts Tiefensuche) erzeugt wird. Würde direkt dieses Muster so angewendet, wären eine ganze Reihe von Kopierattributierungen das Resultat. Insbesondere auch jene, sodass $\mathcal{M}_{acc}$ eine Löschoperation zur Anwendung vorher benötigt. Allerdings erlaubt Definition 4.21 auch die Verwendung anderer Substitionen, sodass die Substitution für die Produktion `RqDefId` ::= `id` eben leer bleiben könnte. Für diese Arbeit ist dies keine Einschränkung, die Definition von Mustern erlaubt es in $\mathcal{M}_{acc}$ auch die Löschung dieser einen Attributierungsregel zu definieren und die, aufgrund von Eigenschaft 1 von Satz 4.1 entfernten Attributierungsregeln wieder in $\mathcal{M}_{acc}$ aufzunehmen. Dies ist somit für die Anwendung von Mustern keine Einschränkung. Für die einfachere Präsentation wird auf diese Problematik nicht näher eingegangen.

Als weiterer Schritt ist die Substition für $\mathcal{M}_{acc}$ zu vervollständigen. Dies ist ebenfalls einfach, da bereits alle wesentlichen Informationen vorliegen. Es ist nur für alle b_l mit $l > r + 1$ und $l < m$ $\left[\varepsilon/b_l\right] \in Subs_{acc}$ abzuleiten. Zwischenstand sind somit eine Reihe von Mengen. Nach Konstruktion sollen diese die Eigenschaften von Satz 4.1 einhalten, sowie bei der Konstruktion Definition 4.16 einhalten.

Auf die Details der Zwischenmengen wird hier an dieser Stelle nicht eingangen. Eine mögliche Zerlegung ist eben jene, die der Zerlegung aus Tabelle 3.2 entspricht.

5.3.2. Zusammenhang Funktionaler Programmierung mit Mustern

Der Zusammenhang funktionaler Programmiersprachen und Attributgrammatiken sowie die Definition von Programmen auf (rekursiven) Datenstrukturen ist u. a. Gegenstand von [11, 42, 71]. In dieser Arbeit wird ein weiterer Zusammenhang zwischen funktionaler Programmierung und Attributgrammatiken dargestellt.

Die Dokumentation einer Reihe funktionaler Programme – `map`, `scan` und `fold` – entspricht nun genau der Dokumentation der in dieser Arbeit vorgestellten Muster. Somit entsprechen Muster funktionalen Programmen auf rekursiven Datenstrukturen. Damit gehen eine Reihe von Eigenschaften einher sowie eine Reihe wünschenswerter Eigenschaften in weiteren Mustern. Im Rahmen dieser Arbeit wird dieser Zusammenhang nicht weiter untersucht, sondern muss im Rahmen weiterer Forschung betrachtet werden.

5.3.3. Antimuster und deren Alternativen

Wie Bransen et. al. sehr gut in [27] beschreiben, lassen sich mit wenig Anstrengung Attributgrammatiken spezifizieren, für jene die erweiterten Abhängigkeitsgraphen (siehe Definition 3.16) zyklisch werden. Die Ursache dessen wurde in Kapitel 3 untersucht. Aufgrund dieser Untersuchung wird eine Umkehrung von Muster 1 (Kettenberechnungen) nicht angestrebt sondern als Antimuster – ein Muster, das nicht verwendet werden sollte – festgelegt.

Durch Kombination dieses Basismusters – Kettenberechnungen von links nach rechts – und einer umgekehrten Anwendung dieses Musters kann schnell ein Zyklus im erweiterten Abhängigkeitsgraphen erzeugt werden.

Es gibt eine Reihe von Anwendungsfällen in denen dieses Muster hätte Anwendung finden können. Für diese Anwendungsfälle sind jedoch die bestehenden Muster ausreichend. Das Beispiel aus [27] lässt sich leicht umsetzen indem der Listenkonstruktor „cons“ durch den Listenkonstruktor „snoc“ ersetzt wird. In realen Anwendungen wird mit diesen aufgesammelten Daten weiter gearbeitet, sodass die Frage im Raum steht, warum nicht an dieser Stelle die Umkehrung der Daten erfolgt, oder vor der Ergebnisverwertung bei normalem links-rechts Aufsammeln, bspw. mittels dem typischen Muster 10, das Ergebnis mittels `reverse` umgewandelt wird. Die Funktion `reverse` ist aus funktionalen Sprachen bekannt und kehrt eine Liste um.

Das Argument, dass bei fauler Auswertung in funktionalen Sprachen unendlich lange Listen möglich sind, ist nur von begrenzter Validität. Für diese Arbeit sind praktisch relevante Bäume aus endlichen Eingabedaten gewonnen.

Kapitel 6.

Fallstudien zur Anwendung typischer Muster

Anhand der motivierenden Beispiele aus Kapitel 1 und Kapitel 2 wird gezeigt, wie Muster genutzt werden können, um die dort bearbeiteten Probleme zu lösen. So wird in nahezu allen Beispielen eine Art der Namensanalyse verwendet und eine Ausgabe generiert.

Folgende Liste präsentiert die zu lösenden Probleme der Beispiele und welches Sprachkonzept damit in Verbindung steht. Die folgenden Abschnitte gehen noch einmal im Detail auf mögliche Lösungsansätze für die unterschiedlichen Beispiele ein und zeigen, wie eine Lösung mit Mustern gestaltet werden kann.

Benutzung vor Definition erlaubt Beispiele 1.2, 2.3

Definition vor Benutzung erzwungen Beispiele 1.1, 2.1 und Beispiel 2.2

Bestimmung von Abhängigkeiten Beispiel 2.5, 1.2 und Beispiel 2.7

Bestimmung von Marken Beispiel 2.1 und Beispiel 1.1

Codegenerierung Beispiel 1.1, 2.2, 2.1 und Beispiel 2.3

Für die Semantik der Beispielsprachen sei im wesentlichen auf Kapitel 2 sowie die Einleitung verwiesen. Das Ergebnis der Codegenerierung der Beispielsprachen ist nicht einheitliche – Format und Programmiersprache unterscheiden sich zum Teil erheblich. In der vorliegenden Arbeit wird, soweit übersetzbarer Code generiert werden muss, einheitlich `C++` generiert. Im Rahmen dieser Arbeit werden immer nur Ausschnitte präsentiert und nicht die komplette Codegenerierung der, teilweise sehr umfangreichen, Sprachen. Für Ziria allein enthält die aktuelle Implementierung bereits über 2000 Zeilen Grammatikspezifikationen für Happy[1] und insgesamt fast 20000 Zeilen an reiner Implementierung[2]. Somit beschränkt sich die vorliegende Arbeit damit die Grundideen potentieller Umsetzungen mit Mustern und Attributgrammatiken vorzustellen.

In dieser Arbeit werden für die Fallstudien eine Reihe einheitlicher Annahmen bei der Konstruktion der abstrakten Syntax getroffen: für Definition und Benutzung werden die Suffixe `Def` und `Ref` den Nichtterminalen hinzugefügt. Für einen wesentlichen Teil der Beispiele und Quellen wird keine abstrakte Syntax von den Autoren angegeben. Somit ist es möglich, dass die in dieser Arbeit verwendeten abstrakten Syntaxen nicht mit den von den Autoren verwendeten übereinstimmt. Änderungen der abstrakten Syntax haben unmittelbar Einfluss auf die Anzahl an erzeugten Attributierungsregeln bei Anwendung eines Musters in der resultierenden Attributgrammatik. In typischen Mustern spiegelt sich somit eine grundlegende Eigenschaft von Mustern wider: es ist eine Abwägung und Entwurfsentscheidung welches Muster bei welcher abstrakten Syntax einzusetzen ist. In diesem Kapitel vorgestellte Fallstudien können hierfür Hinweise geben und zeigen, wie Muster miteinander kombiniert werden können.

Weiterhin ist auch die Konstruktion der abstrakten Syntax immer eine Entwurfsentscheidung. Beispielsweise wird in dieser Arbeit die abstrakte Syntax der Ausdrucksgrammatik zur detailierteren Codege-

[1] Happy ist ähnlich yacc ein Parser-Generator für Haskell, siehe auch `http://www.haskell.org/happy`.
[2] Weitere Bibliotheksfunktionalität benötigt nochmals eine Größenordnung mehr an C-Quelltext mit ca. 150000 Zeilen.

nerierung, angelehnt an die konkrete Syntax, konstruiert. Alternativ könnten Informationen verworfen werden, sodass die abstrakte Syntax kompakter wird und somit Analysen einfacher sind.

Neben der Beschreibung verschiedener Sprachen mittels Mustern, zeigt dieses Kapitel ebenso, dass Sprachen, die nicht mit Attributgrammatiken entwickelt wurden, intuitiv mit Attributgrammatiken umsetzbar sind.

6.1. Anwendungen der Muster zur Namensanalyse und der Definitionstabelle

Für die Quellen der Beispiele sei auf Kapitel 1 und Kapitel 2 verwiesen. Bis auf die Sprache aus Beispiel 1.2 wurden die Beispiele nicht unter Verwendung von Attributgrammatiken entwickelt, teilweise wurde gar keine Implementierung vorgestellt. In diesem Kapitel wird somit gezeigt, dass die, teilweise intuitiv dargestellte, Semantik der Beispiele mit Attributgrammatiken und insbesondere mit Musterinstanzen auf geordneten Attributgrammatiken umgesetzt werden können.

In Abbildung 6.1 werden die in dieser Arbeit verwendeten Grammatiken zur Darstellung der abstrakten Syntax der Beispiele 1.1 und Beispiel 2.1 in Ausschnitten präsentiert, Abbildung 6.2 zeigt diese für Beispiel 2.2 und Beispiel 2.3.

Die abstrakte Syntax aus Abbildung 6.1b stellt einen Ausschnitt einer möglichen Variante zur Implementierung der Sprache aus [5] dar. Alternative Darstellungsformen sind ebenfalls möglich. Da in diesem Abschnitt nur die Namensanalyse von Relevanz ist, werden andere notwendige Eigenschaften zur Umsetzung dieser, und auch der anderen Sprachen, nicht aufgeführt. So ist für die praktsiche Implementierung neben der Namensanalyse auch eine Typanalyse notwendig. Ebenso muss in der Codegenerierung beachtet werden, dass für externe Referenzen (Nichtterminal `ExRef`) zusätzliche Prüfungen notwendig sind. In Beispiel 2.1 muss u. a. sichergestellt werden, dass diese externen Referenzen zur Laufzeit des Prozesses existieren. Somit beschreibt die Arbeit [5] eine Sprache mit statischer und dynamischer Semantik, wohingegen in diesem Abschnitt der vorliegenden Arbeit die statische Semantik der Beispiele Betrachtung finden soll.

Die Darstellung der abstrakten Syntax der Sprache Ziria in [106] ist an Haskell Datentypen angelehnt. In der vorliegenden Arbeit wird somit eine abgewandelte abstrakte Syntax wiedergegeben. Ebenso wird nicht die vollständige abstrakte Syntax angegeben. Es gibt weitere Steuerstrukturen – Schleifen, Wiederholungen – und vordefinierte Anweisungen zur Iteration über Listen oder einer Sequenz von Eingaben. Die Definition von Variablen (und – nicht aufgeführt – Funktionen) ist in Ziria, ähnlich wie beim einleitenden Beispiel 1.1, aufgebaut. Ein Unterschied ist, dass in den Beispielen in [106], entgegen der vereinfachten Syntax dieser Arbeit, keine **in**-Anweisungen (z.B. **let** *name* = ⟨*Expression*⟩ **in**) verwendet werden. Weiterhin wird in der abstrakten Syntax als Wurzel ⟨*Program*⟩ gewählt, wobei ein Programm dann bzgl. Ziria eine Funktion darstellt – in der vorliegenden Arbeit wird somit nur der Code für eine Funktion generiert. Ziria selbst erlaubt die Definition von Berechnungen und Funktionen und Programmen.

Die abstrakte Syntax der Sprache aus [29] wird in dieser Arbeit anhand eines Beispiels präsentiert, sodass die hier in 6.2b gezeigte Syntax nicht der in [29] entsprechen muss. Letztendlich besteht ein Programm in der Sprache von [29] aus einer Hauptmaschine mit unterschiedlich verschachtelten Zuständen und (Unter-)Maschinen. Ein Zustand dieser Maschine kann ebenfalls aus Maschinen bestehen. Wiederum in den Maschinen gibt es Variablen, Zustände und Ereignisse. Transitionen und Ereignisse führen zu Zustandsübergängen.

Für all diese Beispiele reichen die Musterinstanzen aus Abschnitt 5.2 zur Beschreibung aus. Die Umsetzung der Namensanalysen zeigt Beispiel 6.1.

In Beispiel 6.1a ist die Verwendung der Attributierung mit `namesIn` in Zeilen 9 – 11 der abstrakten Syntax geschuldet. Mittels Umordnung, Einführung anderer Nichtterminale und das Entfernen von Ket-

⟨*Program*⟩ ::= ⟨*Expr*⟩

⟨*Expr*⟩ ::= ⟨*Term*⟩
| ⟨*Expr*⟩ '+' ⟨*Term*⟩
| ⟨*Let*⟩

⟨*Let*⟩ ::= 'let' ⟨*VDef*⟩ '=' ⟨*Expr*⟩ 'in' ⟨*Expr*⟩

⟨*Term*⟩ ::= ⟨*Factor*⟩
| ⟨*Term*⟩ '*' ⟨*Factor*⟩

⟨*Factor*⟩ ::= ⟨*VRef*⟩
| ⟨*Int*⟩

⟨*VDef*⟩ ::= ID

⟨*VRef*⟩ ::= ID

a) abstrakte Syntax zu Beispiel 1.1

⟨*Program*⟩ ::= ⟨*Process*⟩

⟨*Process*⟩ ::= 'Process' ⟨*PDef*⟩ ⟨*PStats*⟩

⟨*PStat*⟩ ::= (⟨*SEntity*⟩ | ⟨*LEntity*⟩)* ⟨*PVar*⟩*
⟨*PEvent*⟩* ⟨*Prefs*⟩ ⟨*Main*⟩
⟨*EventHandle*⟩*

⟨*SEntity*⟩ ::= ⟨*EType*⟩ ⟨*EDef*⟩ '=' 'discover' '('
⟨*EType*⟩ ',' ⟨*ExRef*⟩ ')'

⟨*LEntity*⟩ ::= 'List' ⟨*EDef*⟩ '=' 'discover' '('
⟨*EType*⟩ ')'

⟨*PVar*⟩ ::= ⟨*CType*⟩ ⟨*VDef*⟩

⟨*PEvent*⟩ ::= 'subscribe' ⟨*SubscribeRef*⟩

⟨*EventHandle*⟩ ::= 'onEvent' ⟨*ERef*⟩, 'Map'
⟨*VDef*⟩ ⟨*CStat*⟩+

b) abstrakte Syntax zu Beispiel 2.1(Ausschnitt)

Abbildung 6.1. – Kontextfreie Grammatiken als Repräsentation der abstrakten Syntax (enthält Teile der konkreten Syntax zur besseren Veranschaulichung) der Ausdrucksgrammatik und der Gebäudebeschreibung

tenproduktionen könnte `VDef` auch als das Modul (`M`) für Muster 20 aufgefasst werden. Das Werkzeug aus [72] ermöglicht dafür notwendige Änderungen der abstrakten Syntax.

Die Verwendung der Definitionstabellenspalte `val` dient bereits nicht mehr direkt der Namensanalyse, sondern wird in der Codegenerierung sowie der Konstantenfaltung verwendet.

Beispiel 6.1 präsentiert die Namensanalyse für unterschiedliche Sprachen. Anhand diesen Beispiels kann gesehen werden, dass die Ausprägung der Sprache – die Anzahl semantisch relevanter Namensbereiche und weitere Eigenschaften dieser Namen – und die abstrakte Syntax wesentlichen Einfluss auf die Anwendung von Mustern der Namensanalyse hat. Grundsätzlich erlaubt der Mechanismus der Muster sowie die Ausprägungen der Muster 16 und Muster 17 auch, dass nur eine Musteranwendung benötigt wird in Beispiel 6.1b. Eine solche Variante kann wie folgt beschrieben werden:

```
1  def_before_use PDef.bind, EDef.bind, VDef.bind
2              of PDef.sym, Edef.sym, Vdef.sym
3                 PRef.bind, ERef.bind, VRef.bind
4              of PRef.sym, Eref.sym, VRef.sym
5    in Program.proc_names, Program.event_names, Program.var_names
6  with error unknown, error unknown, error unknown
7  with error predef, error predef, error predef
```

Die Erkenntnis unterschiedlicher Ausprägungen erlaubt in weiteren Arbeiten das Finden weiterer Muster.

Beispiel 6.2 zeigt die Attributierung der Namensanalysen für die noch fehlenden Sprachen aus Abbildung 6.2 – die Attributierung für die Sprache aus 2.2 und für 2.3

Für die Attributierung in Beispiel 6.2a wurden nur Variablen herangezogen, da der Umfang Zirias groß ist und den Rahmen dieser Arbeit übersteigen würde. Dennoch sind die verwendeten Mechanismen für

⟨*Program*⟩ ::= ⟨*Stat*⟩*

⟨*Stat*⟩ ::= ⟨*Sequence*⟩
| ⟨*Composition*⟩
| ⟨*Conditional*⟩
| ⟨*VarDef*⟩
| ...

⟨*Sequence*⟩ ::= 'seq' '⟨-' <*ComputeRef*⟩ ';' ⟨*Stat*⟩ (';' ⟨*Stat*⟩)* ''

⟨*Composition*⟩ ::= ⟨*Stat*⟩ '»>' ⟨*Stat*⟩

⟨*Conditional*⟩ ::= 'if' ⟨*Expression*⟩ 'then' ⟨*Stat*⟩ 'else' ⟨*Stat*⟩

⟨*VarDef*⟩ ::= 'var' ⟨*VarDefID*⟩ ':' ⟨*TypeReference*⟩ ':=' ⟨*Value*⟩ 'in' ⟨*Stat*⟩

⟨*Value*⟩ ::= VALUE

a) Ausschnitt der abstrakten Syntax zu Beispiel 2.2

⟨*Program*⟩ ::= ⟨*StateMachine*⟩

⟨*StateMachine*⟩ ::= '(' 'machine' ⟨*MachineDef*⟩ ⟨*MachineStat*⟩* ')'

⟨*MachineDef*⟩ ::= ID

⟨*MachineStat*⟩ ::= ⟨*StateDef*⟩
| ⟨*Transition*⟩
| ⟨*EventDef*⟩
| ⟨*Timeout*⟩
| ⟨*VarDef*⟩

⟨*StateDef*⟩ ::= '(' 'state' ⟨*StateDefId*⟩ ⟨*StateStat*⟩* ')'

⟨*StateDefId*⟩ ::= ID

⟨*StateStat*⟩ ::= ⟨*Entry*⟩
| ⟨*Running*⟩
| ⟨*StateMachine*⟩

⟨*Transition*⟩ ::= '(' 'on' ⟨*EventRef*⟩ ⟨*StateRef*⟩ '->' ⟨*StateRef*⟩ ⟨*TransitionDef*⟩ ')'

⟨*VarDef*⟩ ::= '(' 'var' ⟨*VarDefId*⟩ ':=' '[' ⟨*SmallCode*⟩ ']' ')'

b) abstrakte Syntax zu Beispiel 2.3

Abbildung 6.2. – Kontextfreie Grammatiken als Repräsentation der abstrakten Syntax (mit Teilen der konkreten Syntax zur besseren Veranschaulichung) der Sprachen für Ziria und der Roboterprogrammierung

die anderen Arten von Namen identisch. Weiterhin sei darauf hingewiesen, dass die Namensräume bei Ziria für Funktionen und Berechnungen getrennt sind und dies zu Problemen führt[3].

Eine Besonderheit ergibt sich in Beispiel 6.2b, da hier die Deklaration der Zustände und Maschinen verschachtelt sein können. Dies bedeutet, dass Beispiel 6.2b nicht die korrekte Semantik der Arbeitsweise von [29] widerspiegelt. Eine Lösung kann durch Verwendung von Muster 20 erreicht werden. Solch eine Lösung findet sich in Beispiel 6.3.

An der Lösung in Beispiel 6.3 hervorzuheben ist, dass aufgrund der vielen Möglichkeiten, die Muster 20 bietet, weitere Attributierungsregeln notwendig sind. Diese zusätzlichen Regeln sind notwendig, um die, aus dem Unterbaum gewonnenen, Namen an der Stelle der Definition des Zustands, bzw. der Zustandsmaschine, bekannt zu machen. In [29] ist keine Aussage zum Namensbereich der Transitionen oder deren Semantik bzgl. nicht Eindeutigkeit – doppelt vorkommende Bezeichner – definiert. Für diese Arbeit wird davon ausgegangen, dass diese entweder ebenso durch Geltungsbereich oder Namensanalyse ausgedrückt werden können. In der, in dieser Arbeit verwendeten, beispielhaften, Implementierung wurde ein globaler Namensraum für Transitionen, wie in Beispiel 6.2b, verwendet.

Eine Lösung für die Attributierung der Namensanalyse für Beispiel 1.2 stellt Beispiel 6.4 vor. Ebenfalls darin enthalten ist eine Erweiterung der ursprünglich verwendeten abstrakten Syntax. Grundsätzlich

[3] Ausführungen dazu sind dem Quelltext von Ziria (siehe `https://github.com/dimitriv/Ziria`) bei der Beschreibung der konkreten Syntex zu entnehmen.

```
1  use_before_def VDef.bind of VDef.sym,
2                 VRef.bind of VRef.sym
3  in Program.names
4  via VDef.bind:val ← VDef.value
5  with error unknown with error predef
6
7  rule Let ::= VDef Expr Expr
8  attr VDef.value ← Expr₁.value
9       Expr₁.namesIn ←Let.namesIn
10      Expr₂.namesIn ←VDef.namesOut
11        ← Expr₁.namesOut
12
13 symbol VRef
14 attr this.value ← this.bind:val
```

a) Attributierung der Namensanalyse unter Verwendung typischer Muster zum Beispiel 1.1

```
1  def_before_use PDef.bind of PDef.sym
2                 PRef.bind of PRef.sym
3  in Program.proc_names
4  with error unknown with error predef
5  def_before_use EDef.bind of EDef.sym
6                 ERef.bind of ERef.sym
7  via EDef.bind:type ← EDef.type
8  in Program.event_names
9  with error unknown with error predef
10 def_before_use VDef.bind of VDef.sym
11                VRef.bind of VRef.sym
12 in Program.var_names
13 with error unknown with error predef
14 rule LEntity ::= EDef EType
15 attr EDef.type ← EType.sym
16 rule SEntity ::= EType EDef EType ExRef
17 attr EDef.type ← EType₁.sym
18 cond EType₁.sym = EType₂.sym
19    ⇒ error "type mismatch " ++ EType₁.sym
20    ++ " vs. " ++ EType₂.sym
```

b) Attributierung der Namensanalyse für die Sprache aus [5] bzgl. der abstrakten Syntax aus Abbildung 6.1b

Beispiel 6.1 – Namensanalysen unter Verwendung von Mustern für Beispiel 1.1 und 2.1 bzgl. abstrakter Syntax aus Abbildung 6.1a und 6.1b

unterscheidet sich diese Lösung nur geringfügig von den anderen in dieser Arbeit vorgestellten Lösungen. Beispiel 6.4 ist jedoch für spätere Beispiele von Relevanz.

Für die Ausdrucksgrammatik aus Beispiel 1.1 werden dieselben Konzepte auf ähnliche Weise wieder verwendet. Ausschließlich für die Berechnung von Werten der Variablen, die mit **let**- Ausdrücken definiert wurden, wird die Namensanalyse benötigt. Somit wird in diesem Abschnitt auf eine Darstellung der dazugehörigen Namensanalyse verzichtet.

Weitere Muster auf Basis der Namensanalyse, für die bspw. die Abarbeitung einer Arbeitsliste notwendig ist, lassen sich über andere Methoden – aufsammeln und Transformation der Definitionstabelle (mittels Muster 21) – erreichen.

Weitere Anwendungen im Rahmen der Namensanalyse, bspw. ein Modulsystem mit komplexem Importverhalten, unter Ausnutzung der Forsetzung oder der Transformation der Definitionstabelle, werden in dieser Arbeit nicht präsentiert. Einerseits reichen die bereits verwendeten Muster für die Domänenspezifischen Sprachen aus Kapitel 2 aus, andererseits würde die Vorstellung komplexer Modulsysteme den Rahmen dieser Arbeit überschreiten.

6.2. Anwendungen von auf Beiträgen basierenden Mustern

Eine mögliche semantische Prüfung, die im Rahmen der Beschreibung von intelligenten Gebäuden (siehe dazu die Grammatik in Abbildung 6.1b) ist, ob Ereignisse „behandelt“ werden. Es soll also geprüft werden, ob für jedes Ereignis mindestens ein „Handler“ vorhanden ist. Ausgehend von den Informationen der Namensanalyse, d.h. welche Ereignisse existieren und verwendet werden (Vgl. Beispiel 6.1b), und der Referenz eines Ereignisses, kann für jedes Ereignis geprüft werden ob ein Handler dafür existiert.

Folgendes Beispiel zeigt eine mögliche Variante nach Auflösung der Namensanalyse zur Bestimmung ob ein Ereignis auch behandelt wird.

In Beispiel 6.5 wurden drei Muster angewendet. Kann der Fall ignoriert werden, dass ein Ereignis auch außerhalb einer Ereignisbehandlung referenziert wird, so ist die Lösung kürzer. Anstatt dem Kopieren

```
1  symbol VarDef attr ↑sym ← constituent VarDefId.sym
2
3  def_before_use VarDef.bind of VarDef.sym
4                          VarRef.bind of VarRef.sym
5  in Program.vars
6  via VarDef.bind:val ← VarDef.value
7  with error unknown with error predef
```

a) Namensanalyse mittels Mustern für den Ausschnitt aus der Grammatik in Abbildung 6.2a der Sprache aus Beispiel 2.2

```
1  use_before_def MachineDef.bind of MachineDef.sym
2                          MachineRef.bind of MachineRef.sym
3  in Program.machines
4  with error unknown with error predef
5  use_before_def StateDef.bind of StateDef.sym
6                          StateRef.bind of StateRef.sym
7  in Program.states
8  with error unknown with error predef
9  use_before_def EventDef.bind of EventDef.sym
10                         EventRef.bind of EventRef.sym
11 in Program.events
12 with error unknown with error predef
13 use_before_def TransitionDef.bind of TransitionDef.sym
14                         TransitionRef.bind of TransitionRef.sym
15 in Program.transitions
16 with error unkown with error predef
17 store_load EventDefId.bind:idnum ← EventDefId.idnum
18   through EventRef.id
```

b) Namensanalyse für die abstrakte Syntax aus Abbildung 6.2b der Sprache zu Beispiel 2.2 ohne Möglichkeit der Verschachtelung (somit nicht korrekte Semantik abbildend)

Beispiel 6.2 – Umsetzung der Namensanalysen mittels Musterinstanzen für die Beispiele aus Kapitel 1 und Kapitel 2.

nach unten kann an dieser Stelle in Beispiel 6.5 nur eine 1 stehen. Ebenso würden die ersten zwei Zeilen des Beispiels entfallen.

Eine weitere semantische Prüfung für das einführende Beispiel, Beispiel 1.2 ist die Bildung von Abhängigkeiten. Diese, recht einfache, Musteranwendung ist in folgendem Quelltext zu sehen:

```
1  symbol Declaration attr ↑bind ← constituent RqDefId.bind
2  deptype (Declaration, RqUseId.bind) in Description.rq_deps
```

Die Liste von Paaren mit Definition und Benutzung von Anforderungen, muss dann in der Wurzel des Programms analysiert werden. Da viele mögliche Algorithmen und deren Darstellung existieren, wird auf eine Präsentation dieser verzichtet. Darüber hinaus ist nicht Ziel dieser Arbeit eine Bibliothek mit nützlichen Algorithmen zur Bearbeitung von, in der Definitionstabelle abgelegten, Daten vorzustellen. Für die praktische Anwendbarkeit von Mustern sind solche Bibliotheken jedoch erforderlich.

```
1  use_before_def MachineDef.bind of MachineDef.sym,
2                          MachineRef.bind of MachineRef.sym
3    in Program.machines scope on StateDef.bind:machines
4  with error undef with error predef
5
6  use_before_def StateDefId.bind of StateDefID.sym
7                          StateRef.bind of StateRef.sym
8    in Program.states scope on StateMachine.bind:states
9  with error undef with error predef
10
11 symbol StateDef attr ↑bind ← constituent StateDefId.bind without StateStat
12 symbol StateMachine attr ↑bind ← constituent MachineDef.bind without MachineStat
```

Beispiel 6.3 – Korrigierte Attributierung mit Mustern zur Herstellung der Semantik wie in [29] beschrieben; zugehörige abstrakte Syntax wieder in Abbildung 6.2b aufgeführt

⟨*Description*⟩ ::= ⟨*Declaration*⟩*

⟨*Declaration*⟩ ::= ⟨*RootStat*⟩
| ⟨*RqDef*⟩

⟨*RootStat*⟩ ::= *'root'* ⟨*RqDefId*⟩
⟨*RqStat*⟩+

⟨*RqDef*⟩ ::= *'rq'* ⟨*RqDefId*⟩
⟨*RqStat*⟩+

⟨*RqStat*⟩ ::= *'require'*
⟨*Dependency*⟩+

⟨*RqStat*⟩ ::= *'aut'* ⟨*Author*⟩

⟨*RqStat*⟩ ::= *'desc'* ⟨*Text*⟩

⟨*Author*⟩ ::= *ID*

⟨*Text*⟩ ::= *STRING*

⟨*Dependency*⟩ ::= ⟨*RqUseId*⟩

⟨*RqUseId*⟩ ::= *ID*

a) Abstrakten Syntax zu der Sprache aus Beispiel 1.2

```
1  use_before_def RqDefId.bind of RqDefId.sym
2                 RqUseId.bind of RqUseId.sym
3  in Description.requirements
4  via RqDefId.bind:is_root ← RqDefId.is_root
5  with error predef with error undef
6
7  rule RootStat ::= RqDefId RqStats
8  attr RqDefId.is_root ← true
9  symbol RqDefId attr ↓is_root ← false
```

b) Benutzung vor Definition durch Muster für die abstrakte Syntax aus 6.4a

Beispiel 6.4 – Attributierung der Namensanalyse mittels Mustern für die Sprache aus Beispiel 1.2 und dazugehörige abstrakte Syntax

Eine Alternative unter Ausnutzung des Aufsammelns (Muster 10) besteht darin die Referenzen in einer Spalte von `RqDefinition.bind` abzulegen. Diese können dann, mittels Transformation (Muster 21) analysiert werden. Jedoch ist dies nicht für alle Algorithmen möglich, auch dann nicht, wenn dieser Algorithmus jedes Element der Definitionstabelle analysieren muss.

Ebenfalls häufig verwendet, ist das Durchnummieren von Vorkommen von Definitionen. Mit diesen Werten können automatische Namen für die Codegenerierung hergeleitet werden. Insbesondere in DSLs, bei denen eine Model-to-Text-Transformation in eine allgemeine Programmiersprache statt findet, kann dies erforderlich sein. Selbst bei Sprachen, die miteinander verwandt sind, können solche Umbenennungen zwingend erforderlich sein. Beispiel 6.6 zeigt einen Ausschnitt aus dem in dieser Arbeit herangezogenen Werkzeug eli. Dieser Ausschnitt ist übersetzbar mit `C`, jedoch, aufgrund der Verwendung von Schlüsselworten als Bezeichner, nicht mit `C++`. Der Ausschnitt entstammt der Implementierung des in eli verwendeten Parsergenerators. Hervorzuheben ist die fünfte Zeile von Beispiel 6.6, in der der Bezeichner „`new`“ verwendet wird. Ähnliche Effekte könnten auch zutage treten, wenn Anwender über die Namen von Modellelementen entscheiden. Durchnummerierte, automatisch generierte, Namen erlauben die Verwendung beliebiger Bezeichner für Modellelemente – insbesondere auch jene, die in der Zielsprache ungültig sind.

Die Erstellung eines Index auf Basis von Präfixsummen, wie es bei dem Muster 12 geschieht, kann benutzt werden, um Bezeichner durchzunummerieren. In einer Codegenerierung kann dies ebenfalls benutzt werden, um Menüeinträge automatisch durchzunummerieren, oder Address-Indizes zu setzen.

Beispiel 6.7 zeigt die Verwendung der Indexerstellung für die Sprachen aus dem einleitendem Beispiel, Beispiel 1.2, sowie die Sprache aus [29].

Beispiele über die Anwendung von Präfixsummen, in dieser Arbeit Muster 11, und deren Anwendung zur Implementierung paralleler Algorithmen zeigt [23].

```
1  symbol Program attr ↑c_handle = 0
2  symbol EventHandle attr ↑c_handle = 1
3
4  store_load ERef.bind:handled ← ERef.bind:handled + including (EventHandle.c_handle, Program.c_handle)
5    through EDef.bind in all_handles from Program
6
7  symbol EDef
8  cond this.all_handles > 0 ⇒ error "not handled: " ++ EDef.sym
```

*In den ersten zwei Zeilen wird festgelegt, dass die Behanldung eines Ereignisses nur unterhalb eines Nichtterminals ⟨EventHandle⟩ stattfindet. In allen anderen Fällen wird das Kopieren nach unten (***including***, Muster 4) bis zur Wurzel (⟨Program⟩) vordringen. Durch Verwendung des Musters* **store_and_load** *wird diese Information in der Defintionstabelle gespeichert, und, nachdem alle Referenzen besucht wurden, in den definierenden Symbolen aus der Definitionstabelle in das Attribut* `all_handles` *gespeichert.*
In den letzten zweil Zeilen dieses Beispiels wird die semantische Bedingung geprüft und ein entsprechender Fehler ausgegeben, wenn dies nicht erfolgreich war.

Beispiel 6.5 – Überprüfung ob jedes Ereignis behandelt wird

```
1  elemtype *mkelem(tag,ruleno,rulepos)
2  unsigned short tag, ruleno;
3  SEQunit rulepos;
4  {
5    elemtype *new;
6
7    if ( (new = (elemtype *)malloc(sizeof(elemtype)))  ==  (elemtype *)NULL )
8      {
9        INT_ALLOC_ERR("mkelem()");
10       exit(1);
11     }
12
13   /* more code ... */
14
15   return(new);
16 } /* end of mkelem() */
```

Beispiel 6.6 – Motivation zur Erstellung von Namen mittels Durchnummieren von Bezeichnern anhand eines Ausschnitts aus dem Quelltext von eli, der bereits nicht mehr mit C++ Übersetzern übersetzt werden kann.

```
1  count RqDefId from Description in idnum start 0
```

a) Indexbestimmung für Beispiel die Sprache aus Beispiel 1.2

```
1  count StateDef, EventDefId, TransitionDef
2  from Program in idnum start 0
```

b) Bestimmung von Indizes für die Sprache aus [29] mit Mustern

Beispiel 6.7 – Erstellung von Indizes zur automatischen Umbenennung von Variablennamen unter Verwendung typischer Muster.

6.3. Filterung, Berechnungen und Codegenerierung

In den bisherigen Beispielen dieser Arbeit wurde im wesentlichen die Definitionstabelle bzw. verschiedene Formen der Namensanalyse verwendet, um für die Beispiele eine Attributierung vorzunehmen. Auch bei der Codegenerierung und weiteren Berechnungen können diese Techniken eingesetzt werden. Für das Ergebnis dieser Codegenerierung auf Basis so einer Namensanalyse, zeigt Beispiel 6.8 eine mögliche Attributierung für eines der motivierenden Beispiele.

```
1  csv_name is "rq_" ++ idnum
2  store_load RqDefId.bind:csv_name ← RqDefId.csv_name
3    through RqUseId.csv_name
4
5  symbol Declaration
6  attr ↑bind:csv_code ← ↑csv_name ++ " " ++
7                                (constituent RqUseId.csv_name
8                                 with append_with_comma, id, "") ++ " " ++
9                                (constituent Author.sym with append_with_comma,
10                                id, "")
11 contribute Declaration.bind:csv_code to Description.code ← "Id Deps Author \n"
12 using \a,b ⇒ a ++ "\n" ++ b
```

In der letzten Zeile dieses Beispiels wird mittels \eine anonyme Funktion mit zwei Argumenten lokal definiert. Dies ist ähnlich wie in Haskell umsetzbar bzw. umgesetzt, nach -> erfolgt die Definition des Rückgabewerts.

Beispiel 6.8 – Codegenerierung als Namensanalyse bzgl. Beispiel 1.2

Weitere Musteranwendungen zur Bestimmung von Ergebnissen unter Ausnutzung der Fortsetzung und Filterung, bietet Beispiel 6.9 für die Ausdruckssprache aus Beispiel 1.1. Bei Beispiel 1.1 ist wichtig, dass Ausdrücke der Form **let** x = x **in** x nicht erlaubt sind – das zweite Vorkommen von x ist nicht definiert – und Namensbereiche sind ebenfalls nicht verwendet. Damit ergibt sich für die Berechnung des Werts eines Ausdrucks die Musteranwendung wie in Beispiel 6.9 gegeben.

```
1  store_load VarDef.bind:value ← VarDef.value
2  through VarRef.value
3
4  symbol Factor attr ↑value ← constituent VarRef.value, Int.value
5  symbol Term attr ↑value ← constituent Factor.value without Factor
6  symbol Expr attr ↑value ← constituent Term.value without Expr, VarDef
7
8  rule Expr ::= VarDef Expr Expr
9  attr VarDef.value ← Expr₂.value
10     Expr₁.value ←Expr₃.value
11 rule Expr ::= Expr Term attr Expr₁.value ←Expr₂.value + Term.value
12 rule Term ::= Term Factor attr Term₁.value ←Term₂.value * Factor.value
```

In diesem Beispiel werden in den Zeilen 4 – 6 Muster 5 mit Filterung verknüpft – einem nicht explizit vorgestellten Muster, da die Semantik intuitiv ersichtlich ist. Durch Filterung wird die Möglichkeit erreicht, dass Muster 5 verwendet werden kann. Darüber hinaus sind die wesentlichen semantischen Berechnungen in den letzten, darauf folgenden Zeilen gegeben.

Beispiel 6.9 – Berechnung des Wertes eines Ausdrucks für die Sprache aus Beispiel 1.1

Das Muster der Filterung über Attribute für komplexe Beiträge (Muster 15), kann gut in der Codegenerierung eingesetzt werden. Wenn redundante Informationen vorhanden sind und nur eine Information herausgeneriert werden muss, ist dieses Muster bspw. verwendbar. In den Beispielen dieser Arbeit kann dies vorkommen, wenn in der Sprache der aus [29] – Beispiel 2.3 – keine Überprüfung auf Einmaligkeit der Ereignisübergänge vorhanden ist. In [29] wird Smalltalk Code generiert. In dieser Arbeit wird stattdessen, soweit notwendig, C++ für die Beispiele generiert.

Beispiel 6.10 stellt eine Codegenerierung für die Behandlung von Ereignissen vor.

Bei der Codegenerierung in Beispiel 6.10 wird für die Zustandsübergänge ein zweidimensionales Feld angelegt und in diesem Feld, mittels Initialisierungsausdruck, das Ziel des Zustandsübergangs angelegt. Weitere Abschnitte der Codegenerierung für diese (und die anderen Beispielsprachen) können Anhang E entnommen werden.

```
1  collect EventRef.sym in StateMachine.eventsNames
2  autocopy eventsNames
3  symbol Transition
4  attr ↑is_unique ← count_elem(constituent EventRef.sym, ↓eventsNames) ≤ 1
5       ↑def_code ← "this ⇒ transitions[" ++ consituent TransitionDef.idnum ++ "][“
6                       ++ constituent EventRef.bind:id ++ "] = { "
7                       ++ ↑tostate_id ++ "} \n"
8
9  contribute Transition.def_code on Transition.is_unique
10   to StateMachine.init_code ← statemachine_init_code() using ++
```

Hinzufügen der initialen Zustandsübergangsregeln für die Initialisierung des Zustandsautomaten im Konstruktor durch Initialisierungsausdruck mittels Muster 15.

Beispiel 6.10 – Ausschnitt aus der Codegenerierung einer Ereignisbehandlung nach C++(Zeile 5) für die Sprache aus [29] unter Nutzung typischer Muster und insbesondere der Filterung über Attribute bei komplexen Beiträgen

```
1  symbol VarDef
2  attr ↑bind:type ← constituent TypeReference.sym
3       ↑bind:init_code ← constituent Value.sym
4       ↑var_init_code ← type_to_code(↑bind:type) ++ " "
5                          ++ ↑sym ++ " = " ++ ↑bind:init_code ++
6                          " "
7
8  symbol Stat
9  attr ↑in_sequence ← including Program.in_sequence, Sequence.in_sequence
10
11 symbol Program attr ↑in_sequence ← false
12 symbol Sequence attr ↑in_sequence ← true
13
14 contribute VarDef.var_init_code to Program.var_code using INL
15 contribute Stat.sequence_code on Stat.in_sequence to Sequence.code using INL
16 collect Sequence.code in Program.sequences
```

Beispiel 6.11 – Generierung von Code zur Definition und Initialisierung von Variablen für die Sprache aus [106] mit Hilfsfunktionen `type_to_code`, `sequence_to_code` und `INL`

Für die Sprache Ziria zeigt Beispiel 6.11 einen Teil der Codegenerierung zur Initialisierung von Variablen und der Transformation von Sequenzen.

Die in Beispiel 6.11 verwendeten Hilfsfunktionen, wie `type_to_code` und `sequence_to_code`, wandeln einen algebraischen Datentyp zur Repräsentation der, im darunter liegenden Teilbaum, vorhandenen Informationen in eine Zeichenkette um. Für die Sprache Ziria übernehmen solche Hilfsfunktionen unter anderem die Berechnung einer Umsetzungstabelle[4].

Damit schließt an dieser Stelle die Präsentation der Codegenerierung. Weite Teile der Codegenerierung und vor allem der resultierenden Attributgrammatiken können Anhang E entnommen werden.

6.4. Ausgewählte Resultate der Musteranwendungen

Die, in den vorherigen Abschnitten beschriebenen, Musteranwendungen führen nach Umformung in klassische (geordnete) Attributgrammatiken zu sehr umfangreichen Spezifikationen. In diesem Abschnitt werden diese Attributgrammatiken den Varianten unter Musteranwendung gegenüber gestellt. Auf Basis der abstrakten Syntax aus Abbildungen 6.1 und 6.2, werden für die Beispiele die daraus resultierenden Attributgrammatiken nach Musteranwendung präsentiert.

Wie bereits in Kapitel 4 und Kapitel 5 bei der Definition der Muster und verschiedener Beispiele gezeigt, werden nach Anwendung der Substitution die entsprechenden Attributierungsregeln und Attribute der ursprünglichen Attributgrammatik hinzugefügt. In diesem Kapitel ist die ursprüngliche, geordnete, Attributgrammatik immer die triviale, leere, Attributgrammatik.

[4] engl. Lookup-Table (LUT)

Quelle, Sprache	Zeilen mit Mustern	Zeilen resultierende AG	Bemerkungen
Ziria [106]	7	65	keine Codegenerierung
Anforderungsanalyse	26	282	Generierung CSV Tabelle
Smart Buildings [5]	28	368	keine Codegenerierung
Live Robots [29]	40	1149	Codegenerierung für Transitionen

Tabelle 6.1. – Gegenüberstellung des Spezifikationsumfangs mittels Mustern und resultierender Attributgrammatik

In den hier vorgestellten Beispielen, wie auch in den umfangreicheren Varianten von Anhang E, werden etwaige Zwischenschritte der Komposition eines Musters nicht mit aufgeführt, sondern dieses „Zwischenmuster“ direkt auch zurückgeführt. Für die Namensanalyse nach Beispiel 6.1b ergibt sich somit die in Beispiel 6.12 vorgestellte Attributierung für die ersten 4 Zeilen von Beispiel 6.1b.

Beispiel 6.12 enthält insgesamt annähernd 70 Zeilen an Spezifikationen für eine Attributgrammatik und wird aus 4 Zeilen von Beispiel 6.1b erzeugt.

Durch den Aufbau der Namensanalyse, sowie dem Aufbau von Mustern mittels Kettenattributierung, ist zwar die resultierende Attributgrammatik sehr umfangreich, dies hat dennoch keine längere Laufzeit zur Folge verglichen mit einer manuell entwickelten Attributgrammatik mit Kettenattributierung nur in den relevanten Teilbäumen. Wie bereits beschrieben sind reine Kopieranweisungen ohne semantische Relevanz in darunter liegenden Teilbäumen bei geordneten Attributgrammatiken nur ein Hinweis bzgl. der Besuchsreihenfolge im abstrakten Syntaxbaum. Für diese Argumentation siehe auch [77].

Einen Ausschnitt aus der Konstruktion von Tabelleneinträgen für die Sprache zur Anforderungsanalyse, aus dem einleitenden Beispiel 1.2, zeigt Beispiel 6.13. Im Gegensatz zu Beispiel 6.12 werden die reinen Kopieranweisungen (auch in Beispiel 1.2 präsentiert), hier nicht gezeigt. Die Grundlage der abstrakten Syntax ist dabei jedoch die in Beispiel 6.4 gezeigte, nicht jene aus Abbildung 3.1.

Es ist anzumerken, dass in Beispiel 6.13 nicht nur viele Kopieranweisungen nicht enthalten sind, ebenso auch nicht die initialen Wertsetzungen in den Produktionen von der Wurzel ausgehend.

Weitere Beispiele an dieser Stelle würden ähnlich aufgebaut sein. Auf die Präsentation weiterer resultierender Attributgrammatiken oder Ausschnitten aus diesen wird an dieser Stelle verzichtet, entsprechende Quellen finden sich in Anhang E.

6.5. Auswertung und Zusammenfassung

Die vorhergehenden Abschnitte haben für eine Reihe von Sprachen (siehe Kapitel 2) Musteranwendungen zur Spezifikation der Sprachsemantik präsentiert. Der Umfang der resultierenden Grammatik ist selbst für kleine Ausschnitte der Sprachspezifikation sehr umfangreich. So wird in Abschnitt 6.4 gezeigt, dass für 4 Zeilen aus Beispiel 6.1b die resultierende Attributgrammatik nahezu 70 Zeilen enthält. Umfangreichere abstrakte Syntaxen führen zu mehr Attributierungsregeln.

In Anhang E werden für die präsentierten Beispiele der Musteranwendung die vollständigen, resultierenden Attributgrammatiken präsentiert. Diese in diesem Kapitel zu präsentieren, hätte den Rahmen dieser Arbeit überschritten. In Anhang E sind die vollständigen Beispiele angegeben. Diese vollständigen Beispiele wurden für diesen Vergleich herangezogen. Tabelle 6.1 stellt den Umfang der Definition mittels Musteranwendung den Werten der resultierenden Attributgrammatiken gegenüber.

In der zweiten Spalte (Zeilen mit Mustern) wurden die, in diesem und vorherigen Kapiteln vorgestellten, Beispiele für die jeweilige Sprache aufsummiert.

```
1   rule Program ::= Process
2   attr Process.proc_namesIn ← emptyset
3        Program.proc_names ← Process.proc_namesOut
4
5   rule Process ::= PDef PStats
6   attr PDef.proc_namesIn ← Process.proc_namesIn
7        PStats.proc_namesIn ← PDef.proc_namesOut
8        Process.proc_namesOut ← PStats.proc_namesOut
9
10  rule PStats ::= PStats PStat
11  attr PStats2.proc_namesIn ←PStats1.proc_namesIn
12       PStat.proc_namesIn ← PStats2.proc_namesOut
13       PStats1.proc_namesOut ←PStat.proc_namesOut
14
15  rule PStats ::= ε
16  attr PStats.proc_namesOut ← PStats.proc_namesIn
17
18  rule PDef ::= PDefId
19  attr PDef.bind ← bindKey(PDef.proc_namesIn, PDef.sym
20       PDef.proc_namesOut ← PDef.proc_namesIn > =  PDef.bind
21  cond PDef.sym ∉ PDef.proc_namesIn ⇒ error "already defined " ++ PDef.sym
22
23  rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
24  attr SEntities.proc_namesIn ← PStat.proc_namesIn
25       PVars.proc_namesIn ← SEntities.proc_namesOut
26       PEvents.proc_namesIn ← PVars.proc_namesOut
27       Prefs.proc_namesIn ← PEvents.proc_namesOut
28       Main.proc_namesIn ← Prefs.proc_namesOut
29       EventHandles.proc_namesIn ← Main.proc_namesOut
30       PStat.proc_namesOut ← EventHandles.proc_namesOut
31
32  rule SEntities ::= SEntities SEntity
33  attr SEntities2.proc_namesIn ←SEntities1.proc_namesIn
34       SEntity.proc_namesIn ← SEntities2.proc_namesOut
35       SEntities1.proc_namesOut ←SEntity.proc_namesOut
36
37  rule SEntities ::= ε
38  attr SEntities.proc_namesOut ← SEntities.proc_namesIn
39
40  rule SEntity ::= EType EDef EType ExRef
41  attr EType1.proc_namesIn ←SEntity.proc_namesIn
42       EDef.proc_namesIn ← EType1.proc_namesOut
43       EType2.proc_namesIn ←EDef.proc_namesOut
44       ExRef.proc_namesIn ← EType2.proc_namesOut
45       SEntity.proc_namesOut ← ExRef.proc_namesOut
46
47  rule LEntity ::= EDef EType
48  attr EDef.proc_namesIn ← LEntity.proc_namesIn
49       EType.proc_namesIn ← EDef.proc_namesOut
50       LEntity.proc_namesOut ← EType.proc_namesOut
51
52  rule PVars ::= PVars PVar
53  attr PVars2.proc_namesIn ←PVars1.proc_namesIn
54       PVar.proc_namesIn ← PVars2.proc_namesOut
55       PVars1.proc_namesOut ←PVar.proc_namesOut
56
57  rule PVars ::= ε
58  attr PVars.proc_namesOut ← PVars.proc_namesIn
59
60  rule PEvent ::= SubscribeRef
61  attr SubscribeRef.proc_namesIn ← PEvent.proc_namesIn
62       PEvent.proc_namesOut ← SubscribeRef.proc_namesOut
63
64  rule EventHandle ::= ERef VDef CStats
65  attr ERef.proc_namesIn ← EventHandle.proc_namesIn
66       VDef.proc_namesIn ← ERef.proc_namesOut
67       CStats.proc_namesIn ← VDef.proc_namesOut
68       EventHandle.proc_namesOut ← CStats.proc_namesOut
69
70  rule EDef ::= ID
71  attr EDef.proc_namesOut ← EDef.proc_namesIn
72
73  rule ERef ::= ID
74  attr ERef.proc_namesOut ← ERef.proc_namesIn
```

Beispiel 6.12 – Namensanalyse nach Rückführung auf allgemeine, geordnete Attributgrammatiken für Beispiel 6.1b.

```
1   rule RqDef ::= RqDefId RqStats
2   attr RqDefId.idnumIn ← RqDef.idnumIn
3        RqStats.idnumIn ← RqDefId.idnumOut
4        RqDef.idnumOut ← RqStats.idnumOut
5        RqDefId.requirementsIn ← RqDef.requirementsIn ↞ RqDef.bind
6        RqStats.requirementsIn ← RqDefId.requirementsOut
7        RqDef.requirementsOut ← RqStats.requirementsOut
8        RqDefId.env ← RqDef.env
9        RqStats.env ← RqDef.env
10       RqDef.sym ← RqDefId.sym
11       RqDef.bind:csv_name ← RqDefId.csv_name
12       RqDef.csv_code ← RqDefId.csv_name ++ " "
13                              ++ RqStats.csv_name_const ++ " "
14                              ++ RqStats.author_sym_const
15       RqDef.codeOut ← RqDef.codeIn ++ "\n" ++ RqDef.bind:csv_code
16
17  rule RqDefId ::= ID
18  attr RqDefId.idnum ← RqDefId.idnumIn + 1
19       RqDefId.idnumOut ← RqDefId.idnumIn + 1
20       RqDefId.csv_name ← "rq_" ++ RqDefId.idnum
21       RqDef.bind ← bindKey(RqDefId.requirementsIn, RqDef.sym)
22       RqDefId.requirementsOut ← RqDefId.requirementsIn ↞ RqDefId.bind
```

Beispiel 6.13 – Semantisch relevanter Teil der Indexgenerierung und Generierung von Tabelleneinträgen in CSV für die Sprache der Anforderungsanalyse aus Beispiel 1.2

Die Werte aus Tabelle 6.1 zeigen, dass, je nach gewählten Mustern und gewählter abstrakter Syntax, die resultierende Attributgrammatik fast immer mindestens eine Größenordnung größer ist. Dabei ist zu beachten, dass die Sprachen, in der hier präsentierten Variante unterschiedlichen Umfang haben. Weiterhin sind die resultierenden Attributgrammatiken kaum optimiert: die verschiedenen resultierenden Attributgrammatiken für die Sprachen aus [29] und [5] werden, je Musteranwendung, aneinander gehangen. Wenngleich es kürzere Varianten gibt, ist dies einer manuellen Entwicklung, mit Dokumentation der unterschiedlichen semantischen Sprachteile, ähnlich. Alternativen, die bspw. nicht alle Nichtterminale attributieren, wurden nicht herangezogen. Dass dies in dieser Variante nicht vorkommt liegt an der Definition der Kettenattributierung als Abschluss eben einer solchen Kettenattributierung als Muster. Die mit so einer Variante einhergehenden, zusätzlichen, Beweisverpflichtungen für den Erhalt der Zerlegbarkeit und den Nachweis, dass dieses Muster weiterhin ordnungserhaltend wäre, würde den Rahmen dieser Arbeit überschreiten.

In diesem Kapitel wurden ein Teil der Muster auf einen Teil der Semantik der Beispiele angewendet. Weitere Variationen, Varianten und Möglichkeiten sind umsetzbar und wurden in dieser Arbeit jedoch nicht explizit aufgeführt. Ausgehend von einer Namensanalyse in verschiedenen Varianten wurden mittels Transfer weitere Informationen bzgl. der Definition von Bezeichnern an deren Referenzierungsstellen bekannt gemacht. Weiterhin wurden Marken bereichnet und Abhängigkeiten bestimmt. Letztendlich wurde für einen Teil der Beispiele auch eine Codegenerierung gezeigt.

Weitere Alternativen und Musteranwendungen für die Sprachen der Beispiele dieser Arbeit sind möglich. Auf eine Vorstellung dieser, dennoch ähnlichen, Umsetzungen wurde verzichtet – der Gewinn an Varianz in den Quelltexten wäre vernachlässigbar. In Anhang E werden ausgewählte, vollständige, Implementierungen der Beispiele unter Verwendung von Mustern vorgestellt. Dabei wird die Implementierung einer Erweiterung der Sprache(n) des Werkzeugs eli verwendet, die in Anhang F vorgestellt wird. Die Erweiterung aus Anhang F implementiert selbst noch keine Muster – diese werden in einem aufgesetzten Werkzeug implementiert.

Kapitel 7.

Schlussfolgerungen

Ziel dieser Arbeit war einen formalen, abstrakteren Mechanismus auf Basis von Attributgrammatiken zu entwickeln. Dieser Mechanismuss sollte Wiederverwendung durch Komposition erlauben, und, der generierte Übersetzer genauso performant sein wie bei Attributgrammatiken. Bereits in früheren Arbeiten wurde in den Implementierungen von Attributgrammatiken Konstrukte eingeführt, die die Spezifikation von Sprachsemantiken erleichtern. Dabei blieben jedoch immer wichtige Eigenschaften, wie die Berechenbarkeit unter Ausnutzung dieser Konstrukte entweder unbeachtet (siehe z. B. [76, 101]) oder führten zur Unentscheidbarkeit der Berechenbarkeit [26].

In dieser Arbeit wurden Muster bzw. typische Muster formal definiert. Ausgehend von Änderungsmengen und den Eigenschaften, die diese einhalten müssen zum Erhalt der Zerlegbarkeit der Attributgrammatik, auf die diese Änderungsmengen anggewendet wurden, wurden eine Reihe von Beweisen zum Aufbau und Struktur von Mustern sowie deren Komposition präsentiert. Weiterhin wurde eine Sprache auf Basis von Termen und Variablen zweiter Ordnung präsentiert und gezeigt unter welchen Umständen (siehe z. B. Satz 4.2) solch eine Attributgrammatik-unabhängige Musterdefinition zu Änderungsmengen führt, die eben die Eigenschaft der Zerlegbarkeit erhalten. Darüber hinaus wurde gezeigt, welche zusätzlichen Eigenschaften diese Muster einhalten müssen um die Ordnungseigenschaft einer Attributgrammatik zu erhalten. Letztendlich wurden eine Reihe aus der Literatur bekannter Attributierungsformen als Basismuster definiert. Mittels der Komposition von Mustern wurden weitere, neue Muster definiert. Durch Beweise wurde gezeigt, dass (und wie) diese Muster die wesentlichen, geforderten, Eigenschaften – Zerlegbarkeit und Ordnungserhalt – einhalten. Durch Beweis dieses Einhaltens und Rückführung auf geordnete Attributgrammatiken sind die Performanzeigenschaften geordneter Attributgrammatiken gezeigt: die resultierenden Attributgrammatiken sind geordnet.

Durch Implementierung von DSLs aus den verwandten Arbeiten, oder Teilen dieser, unter Verwendung von Mustern wurde gezeigt, dass diese Abstraktion kompakter ist, als klassiche Attributgrammatiken. Je nach Anzahl der Produktionen der abstrakten Syntax ist in den Beispielimplementierungen die Variante auf Basis klassischer Attributgrammatiken mindestens eine Größenordnung größer.

In den Fallstudien aus Kapitel 6 wurde für die Semantiken unterschiedlicher Sprachen gezeigt, dass Muster im Vergleich mit resultierenden Attributgrammatiken eine Größenordnung kleiner sind. Für die Umsetzung der Sprache aus [29] ist die resultierende Attributgrammatik um den Faktor 28 größer als die auf Mustern basierende Variante. Wenngleich Varianten mit den „Paradigmen“, die in [101] vorgestellt wurden, ebenfalls kompakter sind, so fehlen diesen doch die entscheidenden formalen Eigenschaften von Mustern. Nach der These dieser Arbeit existiert ein abstrakterer Mechanismus als Attributgrammatiken, welcher gleichzeitig ähnliche Performanz bietet. Für den Abstraktionsgrad wurde der Quellumfang herangezogen und zwischen Mustervariante und resultierender Variante verglichen. Aufgrund der Unschärfe und Subjektivität des Begriffs Abstraktionsgrad wurde der Spezifikationsumfang als Vergleichskriterium gewählt.

Durch die Eigenschaft, dass Muster zerlegungserhaltend sind und es für jedes zerlegungserhaltende Muster eine ordnungserhaltende Musteranwendung gibt[1] ist ebenso gezeigt, dass Muster performant sind. Es ist bekannt, dass für geordnete Attributgrammatiken effiziente Evaluatoren generiert werden können.

[1] Die dafür notwendigen Eigenschaften wurden in Abschnitt 4.3 entwickelt und sind in einer Implementierung der in dieser Arbeit vorgestellten Muster prüfbar bzw. „generierbar“.

Dies entbindet jedoch einen Entwickler nicht davon durchdachte Entwurfsentscheidungen zu treffen. Die abstrakte Syntax, angewendete Muster, die Verwendung der Definitionstabelle, wie diese verwendet wird und die Implementierung der Hilfsfunktionen sind nur einige Eigenschaften, die Einfluss auf die Performanz des generierten Evaluators haben. Gleichwohl kann statisch entschieden werden ob die gegebene Attributgrammatik geordnet ist.

Muster sind formal definiert als Funktion auf Attributgrammatiken mit konstanter abstrakter Syntax. Durch Definition von Eigenschaften dieser Funktionen konnte gezeigt werden, dass die resultierende Attributgrammatik ebenso günstige Eigenschaften - Berechenbarkeit, Zerlegbarkeit, Zerlegung und Ordnung - einhält.

Die These dieser Arbeit auf Seite 7 postulierte, dass es einen formal definierbaren Mechanismus auf Attributgrammatiken gibt, der abstrakter ist und geeignet ist die Semantik einer Sprache zu definieren. Für eine Reihe aktueller Modellierungssprachen aber auch für Beispiele aus der Implementierung von Programmiersprachen wurde gezeigt, wie diese unter Ausnutzung von Mustern auf Attributgrammatiken umsetzbar sind. Wie bereits beschrieben ist der Spezifikationsumfang mit Mustern wesentlich geringer als bei klassichen Attributgrammatiken. Somit wird die These dieser Arbeit als bestätigt angesehen.

Der Beitrag dieser Arbeit ist damit ein neuer Formalismus mit bewiesenen Performanzeigenschaften. Die vorliegende Dissertation erweitert damit die bisherigen, ingenieursmäßigen Lösungen, um eine wissenschaftliche Betrachtung dieser unter dem Augenmerk der Eigenschaften geordneter Attributgrammatiken. Die bisherigen ingenieursmäßigen Lösungen, wie in [101] vorgestellt, sind darüber hinaus erweitert wurden um neue, bisher nicht präsentierte Abstraktionen. Auch für diese neuen Lösungen erfolgte eine wissenschaftliche Betrachtung und der Nachweis, dass diese Lösungen abgeschlossen in der Anwendung in geordneten Attributgrammatiken ist.

Während bisherige Arbeiten keine Betrachtungen der formalen Eigenschaften bzgl. neuer Abstraktionsmechanismen auf Attributgrammatiken gemacht haben, schließt diese Arbeit diese Lücke für bekannte Abstraktionen – Symbolattributierung, Kettenattributierungen – und erweitert die bekannten Abstraktionen. Darüber hinaus ist wesentlicher Beitrag dieser Arbeit der Nachweis, dass diese Abstraktionen nicht zur Unentscheidbarkeit wichtiger Eigenschaften der Attributgrammatik führt, sondern innerhalb der Klasse geordneter Attributgrammatiken ist.

7.1. Alternative Ansätze

Während in frühen Arbeiten zu Attributgrammatiken eine formale Betrachtung zur Berechenbarkeit und Laufzeitverhalten erfolgt, siehe z. B. [68, 82], erfolgt in späteren Arbeiten nur eine Vorstellung der Konstrukte, nicht jedoch eine formale Betrachtung dieser – siehe bspw. [76, 86, 101]. Eine Ausnahme bildet Boylands Betrachtung von Referenzattributen bzw. entfernten Attributen referrierten Berechnungen in [26], wobei die Erkenntnis ist, dass für diese Attributgrammatiken die Berechenbarkeit unentscheidbar ist.

Neuere Arbeiten wie [21, 27] betrachten ebenfalls die Berechenbarkeit der Attributgrammatik und verwenden zur Generierung der Zerlegung andere Ansätze, als bspw. [75]. So werden in [21] die Abhängigkeiten so modelliert, dass ein externer Löser diese als Randbedingungen interpretiert. Die Ergebnisse der vorliegenden Arbeit können unmittelbar auch mit dem Verfahren von Binsbergen, Bransen und Dijkstra verwendet werden.

Syntaktische Konstrukte und Termersetzung sind unter anderem Gegenstand von [60, 112]. Über das Laufzeitverhalten dieser Konstrukte wird nicht direkt etwas berichtet. Eine Arbeit, die über das Laufzeitverhalten solcher Methoden berichtet ist [80], wobei das Termersetzungssystem Stratego zum Einsatz kommt und mindestens eine dreimal höhere Laufzeit hat, als das Vergleichswerkzeug JastAdd. Für JastAdd wiederum gelten die Aussagen aus [26] bzgl. der Berechenbarkeit. Gleichwohl können in JastAdd und Silver Attributgrammatiken spezifiziert werden, die ohne Referenzattribute und Termersetzung auskommen, sodass der Ansatz dieser Arbeit auch dort anwendbar ist.

Grundsätzlich sind Muster ein Ansatz, der zur Komposition von Attributgrammatiken genutzt werden kann. Im Gegensatz bspw. zu den Arbeiten Boylands [24, 25] wurden Muster nicht im Hinblick auf die Komposition unterschiedlicher Attributgrammatiken entwickelt. Auch ist nicht das Ziel die Erweiterung der abstrakten Syntax gewesen, wie dies bspw. in [50] der Fall war. Muster ändern nicht die abstrakte Syntax. Allerdings kann bei vielen Mustern deren Anwendung auf die Wurzel eines Teilbaums beschränkt werden, sodass ähnliche Ergebnisse erreichbar sein können.

7.2. Offene Fragestellungen und Ausblick

In zukünftiger Forschung ist nun zu klären, wie sich aus diesen Informationen statisch eine Berechnungsstrategie für die Kombination bestehender geordneter Attributgrammatiken herleiten lässt ohne den Zwischenschritt eine vollständige und gemeinsame „Kombinationsattributgrammatik“ zu erstellen und ob dies überhaupt möglich ist. Ziel sollte es sein, ausgehend von einem typischen Muster und einer abstrakten Syntax direkt die Abhängigkeitsgraphen aufzustellen und diese dann direkt für die Kombination zweier Attributgrammatiken zu verwenden. Der Vorteil eines solchen Verfahrens wäre, dass Geschäftsgeheimnisse ausreichend geschützt werden können, Plugin-basierte Übersetzer aus geordneten Attributgrammatiken generiert werden können und, dass die Generierung für interaktive Umgebungen einfacher wird.

Ein praktisches Beispiel solcher Übersetzer für interkative Umgebunden mit Plugins wäre ein guter Nachweis – analog des Nachweises, dass bidirektionale Transformationen mit typischen Mustern realisierbar sind. Darüber hinaus können bestehende Generatoren für geordnete Attributgrammatiken mit Hinblick auf die mögliche Parallelisierung für disjunkte Attributgrammatiken reimplementiert werden.

Darüber hinaus bestehen Optimierungsmöglichkeiten hinsichtlich des Speicherverbrauchs bei seiteneffektfreien Attributen unter der Verwendung der aus [76] und [101] bekannten Paradigmen des entfernten Attributzugriffs. Bei diesen Paradigmen wird immer wieder ein neues Attribut erzeugt und kopiert, da Seiteneffekte nach dem Kopiervorgang bei nochmaliger Verwendung zu Inkonsistenzen führen könnten. Außerdem sind bei mehrfachem Zugriff auf dasselbe entfernte Attribut diese Zugriffe unabhängig voneinander, sodass lokale Abhängigkeiten seltener vorkommen. Momentan integriert kein bestehendes System für geordnete Attributgrammatiken Typinformationen außer es handelt sich um den sogenannten `VOID`-Typ, den Typ der nur für zusätzliche Abhängigkeiten in geordneten Attributgrammatiken verwendet wird. In der Zwischensprache zwischen den Mustern und der Übersetzerbauwerkzeugsammlung eli werden bereits ausgiebige Typinformationen auch zur Inferenz der Operationen und Typen beim Aufsammeln und bei Faltungen verwendet. Diese Informationen sowie die Definition der Hilfsfunktionen können analysiert und das Resultat dieser Analysen weiter gegeben werden, sodass für garantiert seiteneffektfreie Funktionen bei Attributen mit den genannten Basismustern keine zusätzlichen Attribute generiert werden müssen.

Bei der Vorstellung der Muster offen geblieben sind bisher Muster höherer Ordnung, die Muster bzw. Musteranwendungen in andere Muster bzw. Musteranwendungen transfomieren. Einzig betrachtete Anwendung in dieser Arbeit war bisher die Komposition von Mustern. Für weitere Arten der Transformation, auch unter Berücksichtigung der Darstellungsform aus Abschnitt 4.5, erfordert umfangreich geführte Korrektheitsnachweise bzgl. der Eigenschaften aus Abschnitt 4.1 und Abschnitt 4.2.

Darüber hinaus kann davon ausgegangen werden, dass weitere komplexe Muster existieren. Das Finden und Beschreiben dieser sowie die Rückführung auf die gewählten Basismuster dieser Arbeit ist ein weiteres offenes Thema.

Eine weitere offene Frage ist, ob einerseits aus Spezifikationen in einer noch abstrakteren Sprache, wie OCL, geordnete Attributgrammatiken, auch unter Ausnutzung von Mustern, generiert werden können oder ob es möglich ist, typische Anfragen von OCL wie sie in [36] beschrieben werden, als Muster formuliert werden können.

Darüber hinaus sind Muster auch in Attributgrammatiken anwendbar, für die die Ordnungseigenschaft unerheblich ist, wie dies bspw. bei Referenzattributgrammatiken der Fall ist. Eine vergleichende Untersuchung ob Muster in diesen Systemen zu besserer Performanz führen ist eine offene Fragestellung. Allgemein ist in dieser Arbeit offen geblieben ob Muster in anderen Systemen anwendbar sind und welche Auswirkungen dies hat.

Der bereits in Abschnitt 5.3.2 angedeutete Zusammenhang funktionaler Programmierung und Mustern konnte im Rahmen dieser Arbeit nicht weiter untersucht werden. Weitere Forschung zu diesem Zusammenhang ist nötig. Aufgrund der bisher erkannten Zusammenhänge ist davon auszugehen, dass weitere Muster, die an funktionale Programme angelehnt sind, gefunden werden können. Aufgrund von Eigenschaften geordneter Attributgrammatiken ist davon auszugehen, dass Monaden und die Eigenschaften von Monaden durch entsprechende Muster darstellbar sind. Weiterhin ist zu untersuchen, ob komplexe Beiträge bereits Monaden darstellen oder das dazugehörige **`via`**-Konstrukt monadische Eigenschaften erfüllt bzw. erfüllen kann.

Literatur

[1] Aalst, W.M.P. van d. ; Hofstede, A.H.M. ter: YAWL: yet another workflow language. In: *Information Systems* 30 (2005), Nr. 4, 245 - 275. http://dx.doi.org/http://dx.doi.org/10.1016/j.is.2004.02.002. – DOI http://dx.doi.org/10.1016/j.is.2004.02.002. – ISSN 0306–4379

[2] Aho, Alfred V. ; Lam, Monica S. ; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006. – ISBN 0321486811

[3] Albreshne, Abdaladhem ; Lahcen, Ayoub A. ; Pasquier, Jacques ; Abdaladhem, A: A framework and its associated process-oriented domain specific language for managing smart residential environments. In: *International Journal of Smart Home* 7 (2013), Nr. 6, S. 377–392. http://dx.doi.org/http://dx.doi.org/10.14257/ijsh.2013.7.6.37. – DOI http://dx.doi.org/10.14257/ijsh.2013.7.6.37

[4] Albreshne, Abdaladhem ; Lahcen, Ayoub Ait L. ; Pasquier, Jacques: Using a Residential Environment Domain Ontology for Discovering and Integrating Smart Objects in Complex Scenarios. In: *Procedia Computer Science* 32 (2014), 997 - 1002. http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2014.05.524. – DOI http://dx.doi.org/10.1016/j.procs.2014.05.524. – ISSN 1877–0509

[5] Albreshne, Abdaladhem ; Pasquier, Jacques: A Domain Specific Language for High-level Process Control Programming in Smart Buildings. In: *Procedia Computer Science* 63 (2015), 65 - 73. http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2015.08.313. – DOI http://dx.doi.org/10.1016/j.procs.2015.08.313. – ISSN 1877–0509

[6] Alexander, Christopher: *The timeless way of building*. Bd. 1. New York: Oxford University Press, 1979

[7] Andrews, Tony ; Curbera, Francisco ; Dholakia, Hitesh ; Goland, Yaron ; Klein, Johannes ; Leymann, Frank ; Liu, Kevin ; Roller, Dieter ; Smith, Doug ; Thatte, Satish u. a.: *Business process execution language for web services version 1.1*. http://xml.coverpages.org/BPELv11-20030505-20030331-Diffs.pdf, 2003. – zuletzt aufgerufen am 04.03.2017

[8] Arne, Nordmann ; Nico, Hochgeschwender ; Dennis, Wigand ; Sebastian, Wrede: A survey on domain-specific modeling and languages in robotics. In: *Journal of Software Engineering in Robotics* 7 (2016), Nr. 1, S. 75–99

[9] Asperti, Andrea ; Guerrini, Stefano: *The optimal implementation of functional programming languages*. Bd. 45. Cambridge University Press, 1998

[10] Atkinson, Colin ; Gerbig, Ralph: Harmonizing Textual and Graphical Visualizations of Domain Specific Models. In: *Second Workshop on Graphical Modeling Language Development (GMLD 2013)*, 2013, 32

[11] Backhouse, Kevin: A functional semantics of attribute grammars. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2002, S. 142–157

[12] Backus, J. W. ; Bauer, F. L. ; Green, J. ; Katz, C. ; McCarthy, J. ; Naur, P. ; Perlis, A. J. ; Rutishauser, H. ; Samelson, K. ; Vauquois, B. ; Wegstein, J. H. ; Wijngaarden, A. van ; Woodger, M. ; Poel, W. L. d.: Revised report on the algorithmic language Algol 60. In: *Numerische Mathematik* 4 (1962), Nr. 1, 420–453. http://dx.doi.org/10.1007/BF01386340. – DOI 10.1007/BF01386340. – ISSN 0945–3245

[13] Badouel, Eric ; Tchougong, Rodrigue ; Nkuimi-Jugnia, CÉLESTIN ; Fotsing, Bernard: Attribute grammars as tree transducers over cyclic representations of infinite trees and their descriptional composition. In: *Theoretical Computer Science* (2013)

[14] Bancilhon, F. ; Spyratos, N.: Update Semantics of Relational Views. In: *ACM Trans. Database Syst.* 6 (1981), Dezember, Nr. 4, 557–575. http://dx.doi.org/10.1145/319628.319634. – DOI 10.1145/319628.319634. – ISSN 0362–5915

[15] Bell, Stoughton ; Gilbert, Edgar J.: Learning Recursion with Syntax Diagrams. In: *SIGCSE Bull.* 6 (1974), September, Nr. 3, 44–45. http://dx.doi.org/10.1145/988881.988890. – DOI 10.1145/988881.988890. – ISSN 0097–8418

[16] Berg, Christian ; Zimmermann, Wolf: DSL implementation for model-based development of pumps. In: *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*. Springer, 2014, S. 391–406

[17] Berg, Christian ; Zimmermann, Wolf: Evaluierung von Möglichkeiten zur Implementierung von Semantischen Analysen für Domänenspezifische Sprachen. (2014)

[18] Berg, Christian ; Zimmermann, Wolf: Typische Muster bei der Entwicklung Domänen-spezifischer Sprachen mit Attributgrammatiken. In: Knoop, Jens (Hrsg.) ; Ertl, M. A. (Hrsg.): *18. Kolloquium Programmiersprachen und Grundlagen der Programmierung*. 2015 (Schriftenreihe des Instituts für Computersprachen Bericht 2015-IX-1), S. 28 – 43

[19] Berg, Christian ; Zimmermann, Wolf: Eigenschaften typischer Muster auf geordneten Attributgrammatiken. In: Amme, Wolfram (Hrsg.) ; Heinze, Thomas S. (Hrsg.): *Programmiersprachen und Grundlagen der Programmierung 19. Kolloquium, KPS 2017*. 2017 (Jenaer Schriften zur Mathematik und Informatik Bericht Math/Inf/02/2017), S. 29 – 44

[20] Bergmann, Gábor: Translating OCL to graph patterns. In: *International Conference on Model Driven Engineering Languages and Systems* Springer, 2014, S. 670–686

[21] Binsbergen, L. T. ; Bransen, Jeroen ; Dijkstra, Atze: Linearly Ordered Attribute Grammars: With Automatic Augmenting Dependency Selection. In: *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*. New York, NY, USA : ACM, 2015 (PEPM '15). – ISBN 978–1–4503–3297–2, 49–60

[22] Bird, R. S.: Using circular programs to eliminate multiple traversals of data. In: *Acta Informatica* 21 (1984), Nr. 3, 239–250. http://dx.doi.org/10.1007/BF00264249. – DOI 10.1007/BF00264249. – ISSN 1432–0525

[23] Blelloch, Guy E.: Prefix sums and their applications. (1990)

[24] Boyland, John ; Graham, Susan L.: Composing Tree Attributions. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1994 (POPL '94). – ISBN 0–89791–636–0, 375–388

[25] Boyland, John T.: Descriptional Composition of Compiler Components. Berkeley, CA, USA : University of California at Berkeley, 1996 (UCB//CSD-96-916). – Forschungsbericht. – http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oai

[26] Boyland, John T.: Remote Attribute Grammars. In: *Journal of the ACM* 52 (2005), Juli, Nr. 4, 627–687. http://dx.doi.org/10.1145/1082036.1082042. – DOI 10.1145/1082036.1082042. – ISSN 0004–5411

[27] Bransen, Jeroen ; Middelkoop, Arie ; Dijkstra, Atze ; Swierstra, S. D.: The Kennedy-Warren Algorithm Revisited: Ordering Attribute Grammars. Version: 2012. http://dx.doi.org/10.1007/978-3-642-27694-1_14. In: Russo, Claudio (Hrsg.) ; Zhou, Neng-Fa (Hrsg.): *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – DOI 10.1007/978–3–642–27694–1_14. – ISBN 978–3–642–27694–1, 183–197

[28] Bürger, Christoff ; Karol, Sven ; Wende, Christian ; Assmann, Uwe: Reference Attribute Grammars for Metamodel Semantics. Version: 2011. http://dx.doi.org/10.1007/978-3-642-19440-5_3. In: Malloy, Brian (Hrsg.) ; Staab, Steffen (Hrsg.) ; Brand, Mark (Hrsg.): *Software Language Engineering* Bd. 6563. Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–19440–5_3. – ISBN 978–3–642–19439–9, 22–41

[29] Campusano, Miguel ; Fabry, Johan: Live Robot Programming: The language, its implementation, and robot {API} independence. In: *Science of Computer Programming* 133, Part 1 (2017), 1 - 19. http://dx.doi.org/http://dx.doi.org/10.1016/j.scico.2016.06.002. – DOI http://dx.doi.org/10.1016/j.scico.2016.06.002. – ISSN 0167–6423

[30] Cantor, Georg: Beiträge zur Begründung der transfiniten Mengenlehre. In: *Mathematische Annalen* 46 (1895), Nr. 4, S. 481–512

[31] Clark, Tony ; Frank, Ulrich ; Kulkarni, Vinay ; Barn, Balbir ; Turk, Dan: Domain specific languages for the model driven organization. In: *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*. New York, NY, USA : ACM, 2013 (GlobalDSL '13). – ISBN 978–1–4503–2043–6, 22–27

[32] Classen, Andreas ; Heymans, Patrick ; Schobbens, Pierre-Yves: What's in a feature: A requirements engineering perspective. Version: 2008. http://dx.doi.org/10.1007/978-3-540-78743-3_2. In: *Fundamental Approaches to Software Engineering*. Springer, 2008. – DOI 10.1007/978–3–540–78743–3_2, S. 16–30

[33] Clemm, Geoffrey ; Osterweil, Leon: A Mechanism for Environment Integration. In: *ACM Trans. Program. Lang. Syst.* 12 (1990), Januar, Nr. 1, 1–25. http://dx.doi.org/10.1145/77606.77607. – DOI 10.1145/77606.77607. – ISSN 0164–0925

[34] Cordy, James R.: The TXL source transformation language. In: *Science of Computer Programming* 61 (2006), Nr. 3, 190 - 210. http://dx.doi.org/http://dx.doi.org/10.1016/j.scico.2006.04.002. – DOI http://dx.doi.org/10.1016/j.scico.2006.04.002. – ISSN 0167–6423

[35] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald. ; Stein, Clifford: *Algorithmen-Eine Einführung*. Oldenbourg Verlag, 2004

[36] Costal, Dolors ; Gómez, Cristina ; Queralt, Anna ; Raventós, Ruth ; Teniente, Ernest: Facilitating the definition of general constraints in UML. Version: 2006. http://dx.doi.org/10.1007/11880240_19. In: *Model Driven Engineering Languages and Systems*. Springer, 2006. – DOI 10.1007/11880240_19, S. 260–274

[37] Czarnecki, Krzysztof ; Foster, J. N. ; Hu, Zhenjiang ; Lämmel, Ralf ; Schürr, Andy ; Terwilliger, James F.: Bidirectional Transformations: A Cross-Discipline Perspective. Version: 2009. http://dx.doi.org/10.1007/978-3-642-02408-5_19. In: Paige, Richard F. (Hrsg.): *Theory and Practice of Model Transformations: Second International Conference, ICMT 2009, Zu-*

rich, Switzerland, June 29-30, 2009. Proceedings. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – DOI 10.1007/978–3–642–02408–5_19. – ISBN 978–3–642–02408–5, 260–283

[38] Deransart, Pierre ; Jourdan, Martin ; Lorho, Bernard: *Attribute Grammars: Definitions, Systems and Bibliography*. Springer-Verlag New York, 1988. – ISBN 0–387–50056–1

[39] Deursen, Arie van ; Klint, Paul ; Visser, Joost: Domain-specific Languages: An Annotated Bibliography. In: *SIGPLAN Not.* 35 (2000), Juni, Nr. 6, 26–36. http://dx.doi.org/10.1145/352029.352035. – DOI 10.1145/352029.352035. – ISSN 0362–1340

[40] Diestel, Reinhard: *Graphentheorie*. Springer, 2017 (Springer Lehrbuch). – ISBN 9783961340040

[41] Ducournau, Roland: Implementing Statically Typed Object-oriented Programming Languages. In: *ACM Comput. Surv.* 43 (2011), April, Nr. 3, 18:1–18:48. http://dx.doi.org/10.1145/1922649.1922655. – DOI 10.1145/1922649.1922655. – ISSN 0360–0300

[42] Duris, Étienne ; Parigot, Didier ; Roussel, Gilles ; Jourdan, Martin: Structure-directed Genericity in Functional Programming and Attribute Grammars / INRIA. Version: 1997. https://hal.inria.fr/inria-00073586. 1997 (RR-3105). – Research Report. – Projet OSCAR

[43] Ebert, Christof ; Jastram, Michael: ReqIF: Seamless requirements interchange format between business partners. In: *Software, IEEE* 29 (2012), Nr. 5, S. 82–87. http://dx.doi.org/10.1109/MS.2012.121. – DOI 10.1109/MS.2012.121

[44] Ekman, Torbjörn ; Hedin, Görel: Rewritable Reference Attributed Grammars. Version: 2004. http://dx.doi.org/10.1007/978-3-540-24851-4_7. In: Odersky, Martin (Hrsg.): *ECOOP 2004 – Object-Oriented Programming* Bd. 3086. Springer Berlin Heidelberg, 2004. – DOI 10.1007/978–3–540–24851–4_7. – ISBN 978–3–540–22159–3, 147–171

[45] Eysholdt, Moritz ; Behrens, Heiko: Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. New York, NY, USA : ACM, 2010 (OOPSLA '10). – ISBN 978–1–4503–0240–1, 307–309

[46] Farnum, Charles: Pattern-based Tree Attribution. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1992 (POPL '92). – ISBN 0–89791–453–8, 211–222

[47] Farrow, R. ; Marlowe, T. J. ; Yellin, D. M.: Composable Attribute Grammars: Support for Modularity in Translator Design and Implementation. In: *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1992 (POPL '92). – ISBN 0–89791–453–8, 223–234

[48] Feldman, Stuart I.: Make—A program for maintaining computer programs. In: *Software: Practice and experience* 9 (1979), Nr. 4, S. 255–265

[49] Florence, Spencer P. ; Fetscher, Bruke ; Flatt, Matthew ; Temps, William H. ; Kiguradze, Tina ; West, Dennis P. ; Niznik, Charlotte ; Yarnold, Paul R. ; Findler, Robert B. ; Belknap, Steven M.: POP-PL: A Patient-oriented Prescription Programming Language. In: *SIGPLAN Not.* 51 (2015), Oktober, Nr. 3, 131–140. http://dx.doi.org/10.1145/2936314.2814221. – DOI 10.1145/2936314.2814221. – ISSN 0362–1340

[50] Ganzinger, Harald ; Giegerich, Robert: Attribute Coupled Grammars. In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*. New York, NY, USA : ACM, 1984 (SIGPLAN '84). – ISBN 0–89791–139–3, 157–170

[51] Goos, Gerhard: *Vorlesungen über Informatik: Band 1: Grundlagen und funktionales Programmieren*. Springer-Verlag, 2013

[52] Gray, Robert W. ; Levi, Steven P. ; Heuring, Vincent P. ; Sloane, Anthony M. ; Waite, William M.: Eli: A complete, flexible compiler construction system. In: *Communications of the ACM* 35 (1992), Nr. 2, S. 121–130

[53] Griswold, Ralph E. ; Griswold, Madge T.: *The Icon programming language*. Bd. 55. Prentice-Hall Englewood Cliffs, NJ, 1983

[54] Grosch, Josef: AG-an attribute evaluator generator / GMD Forschungsstelle an der Universität Karlsruhe. 1989 (16). – Forschungsbericht. – http://www.cocolab.com/products/cocktail/doc.pdf/toolbox.pdf

[55] Halmos, Paul R.: *Naive set theory*. Courier Dover Publications, 2017. – ISBN 9780486814872

[56] Hedin, Görel: Reference attributed grammars. In: *Informatica (Slovenia)* 24 (2000), Nr. 3, S. 301–317

[57] Hedin, Görel ; Magnusson, Eva: JastAdd—an aspect-oriented compiler construction system. In: *Science of Computer Programming* 47 (2003), Nr. 1, 37 - 58. http://dx.doi.org/http://dx.doi.org/10.1016/S0167-6423(02)00109-0. – DOI http://dx.doi.org/10.1016/S0167–6423(02)00109–0. – ISSN 0167–6423

[58] *Kapitel* An overview of door attribute grammars. In: Hedin, Görel: *Compiler Construction: 5th International Conference, CC '94 Edinburgh, U.K., April 7–9, 1994 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. – ISBN 978–3–540–48371–7, 31–51

[59] Hedin, Görel: Attribute Extension - a Technique for Enforcing Programming Conventions. In: *Nordic J. of Computing* 4 (1997), Mârz, Nr. 1, 93–122. http://dl.acm.org/citation.cfm?id=640141.640146. – ISSN 1236–6064

[60] Hedin, Görel: An Introductory Tutorial on JastAdd Attribute Grammars. Version: 2011. http://dx.doi.org/10.1007/978-3-642-18023-1_4. In: Fernandes, Joã. M. (Hrsg.) ; Lämmel, Ralf (Hrsg.) ; Visser, Joost (Hrsg.) ; Saraiva, Joã. (Hrsg.): *Generative and Transformational Techniques in Software Engineering III* Bd. 6491. Springer Berlin Heidelberg, 2011. – DOI 10.1007/978–3–642–18023–1_4. – ISBN 978–3–642–18022–4, 166–200

[61] Heidenreich, Florian ; Johannes, Jendrik ; Karol, Sven ; Seifert, Mirko ; Thiele, Michael ; Wende, Christian ; Wilke, Claas: Integrating OCL and textual modelling languages. In: *Proceedings of the 2010 international conference on Models in software engineering*. Berlin, Heidelberg : Springer-Verlag, 2011 (MODELS'10). – ISBN 978–3–642–21209–3, 349–363

[62] Hess, Steffen ; Gross, Anne ; Maier, Andreas ; Orfgen, Marius ; Meixner, Gerrit: Standardizing model-based in-vehicle infotainment development in the German automotive industry. In: *Proceedings of the 4th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*. New York, NY, USA : ACM, 2012 (AutomotiveUI '12). – ISBN 978–1–4503–1751–1, 59–66

[63] Hindley, J R. ; Seldin, Jonathan P.: *Lambda-calculus and combinators: an introduction*. Bd. 13. Cambridge University Press Cambridge, 2008

[64] Hopcroft, John E. ; Motwani, Rajeev ; Ullman, Jeffrey D.: *Introduction to Automata Theory, Languages and Computation*. 3rd. Pearson, 2007. – ISBN 9780321455369

[65] Hulette, G. C. ; Sottile, M. J. ; Malony, A. D.: WOOL: A Workflow Programming Language. In: *2008 IEEE Fourth International Conference on eScience*, 2008, S. 71–78

[66] Ichikawa, Kazuhiro ; Chiba, Shigeru: Composable User-defined Operators That Can Express User-defined Literals. In: *Proceedings of the 13th International Conference on Modularity*. New York, NY, USA : ACM, 2014 (MODULARITY '14). – ISBN 978–1–4503–2772–5, 13–24

[67] Irons, Edgar T.: A Syntax Directed Compiler for ALGOL 60. In: *Commun. ACM* 4 (1961), Januar, Nr. 1, 51–55. http://dx.doi.org/10.1145/366062.366083. – DOI 10.1145/366062.366083. – ISSN 0001–0782

[68] Jazayeri, Mehdi ; Ogden, William F. ; Rounds, William C.: The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars. In: *Commun. ACM* 18 (1975), Dezember, Nr. 12, 697–706. http://dx.doi.org/10.1145/361227.361231. – DOI 10.1145/361227.361231. – ISSN 0001–0782

[69] Jazayeri, Mehdi ; Walter, Kenneth G.: Alternating Semantic Evaluator. In: *Proceedings of the 1975 Annual Conference*. New York, NY, USA : ACM, 1975 (ACM '75), 230–234

[70] Johnson, Ralph ; Helm, Richard ; Vlissides, John ; Gamma, Erich: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995

[71] Johnsson, Thomas: Attribute grammars as a functional programming paradigm. Version: 1987. http://dx.doi.org/10.1007/3-540-18317-5_10. In: Kahn, Gilles (Hrsg.): *Functional Programming Languages and Computer Architecture* Bd. 274. Springer Berlin Heidelberg, 1987. – DOI 10.1007/3–540–18317–5_10. – ISBN 978–3–540–18317–4, 154–173

[72] Kadhim, BasimM. ; Waite, WilliamM.: Maptool — supporting modular syntax development. Version: 1996. http://dx.doi.org/10.1007/3-540-61053-7_67. In: Gyimóthy, Tibor (Hrsg.): *Compiler Construction* Bd. 1060. Springer Berlin Heidelberg, 1996. – DOI 10.1007/3–540–61053–7_67. – ISBN 978–3–540–61053–3, 268–280

[73] Kastens, U. ; Waite, W. M.: Modularity and reusability in attribute grammars. In: *Acta Informatica* 31 (1994), Nr. 7, 601–627. http://dx.doi.org/10.1007/BF01177548. – DOI 10.1007/BF01177548. – ISSN 1432–0525

[74] Kastens, U. ; Waite, W.M.: An abstract data type for name analysis. In: *Acta Informatica* 28 (1991), Nr. 6, 539–558. http://dx.doi.org/10.1007/BF01463944. – DOI 10.1007/BF01463944. – ISSN 0001–5903

[75] Kastens, Uwe: Ordered attributed grammars. In: *Acta Informatica* 13 (1980), Nr. 3, S. 229–256. http://dx.doi.org/10.1007/BF00288644. – DOI 10.1007/BF00288644

[76] Kastens, Uwe: Attribute Grammars as a specification method. Version: 1991. http://dx.doi.org/10.1007/3-540-54572-7_2. In: Alblas, Henk (Hrsg.) ; Melichar, Bořivoj (Hrsg.): *Attribute Grammars, Applications and Systems* Bd. 545. Springer Berlin Heidelberg, 1991. – DOI 10.1007/3–540–54572–7_2. – ISBN 978–3–540–54572–9, 16–47

[77] Kastens, Uwe: Implementation of visit-oriented attribute evaluators. In: *Attribute Grammars, Applications and Systems* Springer, 1991, S. 114–139

[78] Kastens, Uwe ; Hutt, Brigitte ; Zimmermann, Erich: *Lecture Notes in Computer Science*. Bd. 141: *GAG, a practical compiler generator*. Springer-Verlag, 1982. http://dx.doi.org/10.1007/BFb0034297. http://dx.doi.org/10.1007/BFb0034297

[79] Kastens, Uwe ; Waite, William M.: Reusable specification modules for type analysis. In: *Software: Practice and Experience* 39 (2009), Nr. 9, 833–864. http://dx.doi.org/10.1002/spe.917. – DOI 10.1002/spe.917. – ISSN 1097–024X

[80] Kats, Lennart C. L. ; Sloane, Anthony M. ; Visser, Eelco: Decorated Attribute Grammars: Attribute Evaluation Meets Strategic Programming. Version: 2009. http://dx.doi.org/10.1007/978-3-642-00722-4_11. In: Moor, Oege de (Hrsg.) ; Schwartzbach, MichaelI. (Hrsg.): *Compiler Construction* Bd. 5501. Springer Berlin Heidelberg, 2009. – DOI 10.1007/978–3–642–00722–4_11. – ISBN 978–3–642–00721–7, 142-157

[81] Kennedy, Ken ; Warren, Scott K.: Automatic Generation of Efficient Evaluators for Attribute Grammars. In: *Proceedings of the 3rd ACM SIGACT-SIGPLAN Symposium on Principles on Programming Languages*. New York, NY, USA : ACM, 1976 (POPL '76), 32–49

[82] Knuth, Donald E.: Semantics of context-free languages. In: *Mathematical systems theory* 2 (1968), Nr. 2, 127–145. http://dx.doi.org/10.1007/BF01692511. – DOI 10.1007/BF01692511. – ISSN 1433–0490

[83] Kokash, Natallia ; Moodie, Stuart L. ; Smith, Mike K. ; Holford, Nick: Implementing a Domain-specific Language for Model-based Drug Development. In: *Procedia Computer Science* 63 (2015), 308 - 316. http://dx.doi.org/http://dx.doi.org/10.1016/j.procs.2015.08.348. – DOI http://dx.doi.org/10.1016/j.procs.2015.08.348. – ISSN 1877–0509

[84] *Kapitel* Declarative Name Binding and Scope Rules. In: Konat, Gabriël ; Kats, Lennart ; Wachsmuth, Guido ; Visser, Eelco: *Software Language Engineering: 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–36089–3, 311–331

[85] Kosar, Tomaž ; Bohra, Sudev ; Mernik, Marjan: Domain-Specific Languages: A Systematic Mapping Study. In: *Information and Software Technology* 71 (2016), 77 - 91. http://dx.doi.org/https://doi.org/10.1016/j.infsof.2015.11.001. – DOI https://doi.org/10.1016/j.infsof.2015.11.001. – ISSN 0950–5849

[86] Koskimies, Kai: Object-orientation in attribute grammars. Version: 1991. http://dx.doi.org/10.1007/3-540-54572-7_11. In: Alblas, Henk (Hrsg.) ; Melichar, Bořivoj (Hrsg.): *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1991. – DOI 10.1007/3–540–54572–7_11. – ISBN 978–3–540–38490–8, 297–329

[87] Křikava, Filip ; Collet, Philippe: On the use of an internal DSL for enriching EMF models. In: *Proceedings of the 12th Workshop on OCL and Textual Modelling* ACM, 2012, S. 25–30

[88] Kusel, A. ; Schönböck, M. J.and W. J.and Wimmer ; Kappel, G. ; Retschitzegger, W. ; Schwinger, W.: Reuse in model-to-model transformation languages: are we there yet? In: *Software & Systems Modeling* 14 (2013), Nr. 2, 537–572. http://dx.doi.org/10.1007/s10270-013-0343-7. – DOI 10.1007/s10270–013–0343–7. – ISSN 1619–1374

[89] Lakatos, D. ; Poruban, J. ; Bacikova, M.: Declarative specification of references in DSLs. In: *Computer Science and Information Systems (FedCSIS), 2013 Federated Conference on*, 2013, S. 1527–1534

[90] Lämmel, Ralf: Typed generic traversal with term rewriting strategies. In: *The Journal of Logic and Algebraic Programming* 54 (2003), Nr. 1, 1 – 64. http://dx.doi.org/https://doi.org/10.1016/S1567-8326(02)00028-0. – DOI https://doi.org/10.1016/S1567–8326(02)00028–0. – ISSN 1567–8326

[91] Lauenroth, Kim ; Pohl, Klaus: Towards Automated Consistency Checks of Product Line Requirements Specifications. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA : ACM, 2007 (ASE '07). – ISBN 978–1–59593–882–4, 373–376

[92] Levy, Azriel: *Basic set theory*. Bd. 13. Courier Corporation, 2002. – ISBN 9780486420790

[93] Lewis, P.M. ; Rosenkrantz, D.J. ; Stearns, R.E.: Attributed translations. In: *Journal of Computer and System Sciences* 9 (1974), Nr. 3, 279 - 307. http://dx.doi.org/http://dx.doi.org/10.1016/S0022-0000(74)80045-0. – DOI http://dx.doi.org/10.1016/S0022–0000(74)80045–0. – ISSN 0022–0000

[94] Limoncelli, Thomas A.: 10 Optimizations on Linear Search. In: *Queue* 14 (2016), August, Nr. 4, 10:20–10:33. http://dx.doi.org/10.1145/2984629.2984631. – DOI 10.1145/2984629.2984631. – ISSN 1542–7730

[95] Magnusson, Eva ; Ekman, Torbjörn ; Hedin, Görel: Extending Attribute Grammars with Collection Attributes–Evaluation and Applications. In: *Source Code Analysis and Manipulation, 2007. SCAM 2007. Seventh IEEE International Working Conference on*, 2007, S. 69–80

[96] Matsuda, Kazutaka ; Hu, Zhenjiang ; Nakano, Keisuke ; Hamana, Makoto ; Takeichi, Masato: Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*. New York, NY, USA : ACM, 2007 (ICFP '07). – ISBN 978–1–59593–815–2, 47–58

[97] Meer, Arjan P. d.: *Domain specific languages and their type systems*, PhD Thesis, Eindhoven University of Technology, Diss., 2014

[98] Merkle, Bernhard: Textual modeling tools: overview and comparison of language workbenches. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. New York, NY, USA : ACM, 2010 (SPLASH '10). – ISBN 978–1–4503–0240–1, 139–148

[99] Mernik, Marjan ; Heering, Jan ; Sloane, Anthony M.: When and how to develop domain-specific languages. In: *ACM Comput. Surv.* 37 (2005), Dezember, Nr. 4, 316–344. http://dx.doi.org/10.1145/1118890.1118892. – DOI 10.1145/1118890.1118892. – ISSN 0360–0300

[100] Mernik, Marjan ; Mitja, Lenič ; Avdičaušević, Enis ; Žumer, Viljem: Multiple attribute grammar inheritance. In: *Informatika* (2000), January

[101] Paakki, Jukka: Attribute Grammar Paradigms – a High-level Methodology in Language Implementation. In: *ACM Comput. Surv.* 27 (1995), Juni, Nr. 2, 196–255. http://dx.doi.org/10.1145/210376.197409. – DOI 10.1145/210376.197409. – ISSN 0360–0300

[102] Pereira, Maria João V. ; Fonseca, João ; Henriques, Pedro R.: Ontological approach for DSL development. In: *Computer Languages, Systems & Structures* 45 (2016), 35 - 52. http://dx.doi.org/https://doi.org/10.1016/j.cl.2015.12.004. – DOI https://doi.org/10.1016/j.cl.2015.12.004. – ISSN 1477–8424

[103] Peyton Jones, Simon L.: *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1987. – ISBN 013453333X

[104] Saraiva, João ; Swierstra, Doaitse: Generic Attribute Grammars. (1999)

[105] Saraiva, Joao ; Van Wyk, Eric: Realizing Bidirectional Transformations in Attribute Grammars. In: *INForum 2010 - II Simpósio de Informática*, 2010, S. 213–216

[106] Stewart, Gordon ; Gowda, Mahanth ; Mainland, Geoffrey ; Radunovic, Bozidar ; Vytiniotis, Dimitrios ; Agullo, Cristina L.: Ziria: A DSL for Wireless Systems Programming. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and*

Operating Systems. New York, NY, USA : ACM, 2015 (ASPLOS '15). – ISBN 978–1–4503–2835–7, 415–428

[107] Sun, Yu ; Gray, Jeff ; Bulheller, Karlheinz ; Baillou, Nicolaus von: A model-driven approach to support engineering changes in industrial robotics software. In: *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg : Springer-Verlag, 2012 (MODELS'12). – ISBN 978–3–642–33665–2, 368–382

[108] Swierstra, Doaitse ; Vogt, Harald: Higher Order Attribute Grammars. Version: 1991. http://dx.doi.org/10.1007/3-540-54572-7_10. In: Alblas, Henk (Hrsg.) ; Melichar, Bořivoj (Hrsg.): *Attribute Grammars, Applications and Systems: International Summer School SAGA Prague, Czechoslovakia, June 4–13, 1991 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1991. – DOI 10.1007/3–540–54572–7_10. – ISBN 978–3–540–38490–8, 256–296

[109] Swierstra, S D. ; Alcocer, Pablo R A. ; Saraiva, Joao: Designing and implementing combinator languages. In: *International School on Advanced Functional Programming* Springer, 1998, S. 150–206

[110] Szárnyas, Gábor ; Izsó, Benedek ; Ráth, István ; Varró, Dániel: The Train Benchmark: cross-technology performance evaluation of continuous model queries. In: *Software & Systems Modeling* (2017), 1–29. http://dx.doi.org/10.1007/s10270-016-0571-8. – DOI 10.1007/s10270–016–0571–8. – ISSN 1619–1374

[111] Tienari, Martti: On the definition of an attribute grammar. Version: 1980. http://dx.doi.org/10.1007/3-540-10250-7_31. In: Jones, Neil D. (Hrsg.): *Semantics-Directed Compiler Generation: Proceedings of a Workshop Aarhus, Denmark, January 1980*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1980. – DOI 10.1007/3–540–10250–7_31. – ISBN 978–3–540–38339–0, 408–414

[112] Van Wyk, Eric ; Bodin, Derek ; Gao, Jimin ; Krishnan, Lijesh: Silver: An Extensible Attribute Grammar System. In: *Sci. Comput. Program.* 75 (2010), Januar, Nr. 1-2, 39–54. http://dx.doi.org/10.1016/j.scico.2009.07.004. – DOI 10.1016/j.scico.2009.07.004. – ISSN 0167–6423

[113] Van Wyk, Eric ; Moor, Oege de ; Backhouse, Kevin ; Kwiatkowski, Paul: Forwarding in Attribute Grammars for Modular Language Design. Version: 2002. http://dx.doi.org/10.1007/3-540-45937-5_11. In: Horspool, R.Nigel (Hrsg.): *Compiler Construction* Bd. 2304. Springer Berlin Heidelberg, 2002. – DOI 10.1007/3–540–45937–5_11. – ISBN 978–3–540–43369–9, 128-142

[114] Viera, Marcos ; Swierstra, S. D.: Attribute grammar macros. In: *Science of Computer Programming* 96, Part 2 (2014), 211 - 229. http://dx.doi.org/http://dx.doi.org/10.1016/j.scico.2014.01.014. – DOI http://dx.doi.org/10.1016/j.scico.2014.01.014. – ISSN 0167–6423. – Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012)

[115] Visser, Eelco: Stratego: A Language for Program Transformation Based on Rewriting Strategies System Description of Stratego 0.5. In: Middeldorp, Aart (Hrsg.): *Rewriting Techniques and Applications: 12th International Conference, RTA 2001 Utrecht, The Netherlands, May 22–24, 2001 Proceedings*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2001. – ISBN 978–3–540–45127–3, 357–361

[116] 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*. New York, NY, USA : ACM, 1989 (PLDI '89). – ISBN 0–89791–306–X, 131–145

[117] Waite, William M. ; Goos, Gerhard: *Compiler construction*. New York : Springer-Verlag, 1984. – ISBN 13:978–1–4612–9731–4

[118] Wan, Zhanyong: *Functional Reactive Programming for Real-Time Reactive Systems*, Department of Computer Science, Yale University, Diss., December 2002

[119] Weise, Daniel ; Crew, Roger: Programmable Syntax Macros. In: *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 1993 (PLDI '93). – ISBN 0–89791–598–4, 156–165

[120] Wirth, Niklaus: What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions? In: *Commun. ACM* 20 (1977), November, Nr. 11, 822–823. http://dx.doi.org/10.1145/359863.359883. – DOI 10.1145/359863.359883. – ISSN 0001–0782

[121] Zimmermann, Wolf: Modell-basierte Programmgenerierung und Methoden des Übersetzerbaus-Zwei Seiten derselben Medaille? In: *Software Engineering (Workshops)* Bd. 215 Citeseer, 2013, S. 23–25

Anhang A.

Notationen

$e \in M_\varepsilon$ Elemente e, die einer Menge $M \cup \{\varepsilon\}$ entnommen werden können, wobei $M \neq \varepsilon$

Σ Für eine kontextfreie Grammatik (N, T, P, Z): $\Sigma = (N \cup T)$

$\mathtt{Y}_i$ Zur Unterscheidung des i-ten Vorkommens des Symbols $\mathtt{Y} \in \Sigma$ in einer Produktion $p \in P$ einer Kontextfreien Grammatik (N, T, P, Z)

$\mathtt{X}_0 \Rightarrow \mathtt{X}_i$ Nichtterminal $\mathtt{X}_0 \in N$ und Symbole $\mathtt{X}_i \in \Sigma$ für eine kontextfreie Grammatik (N, T, P, Z) mit herleitbar (siehe Def. 3.2).

$\overset{*}{\rightsquigarrow}$ reflexiv-transitiver Abschluss der Ableitungsrelation kontextfreier Grammatiken

$\overset{+}{\rightsquigarrow}$ transitiver Abschluss der Ableitungsrelation kontextfreier Grammatiken

Wurzel $\mathtt{X}_0$ eines Teilbaums $\mathtt{X}_0$ bei $\mathtt{X}_0 \overset{*}{\rightsquigarrow} \mathtt{X}_i$

$\mathtt{X}.a$ Kurzschreibweise für ein Attribut $a \in A_\mathtt{X}$; $\mathtt{X}$ wird als zu a zugeordnetes Symbol bezeichnet.

$\mathtt{X}.b : s$ Attribut $\mathtt{X}.b \in A$ vom Typ **`Binding`** für ein Symbol $\mathtt{X} \in \Sigma$ einer Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ mit Spalte s der Definitionstabelle

definiertes Attribut Für $\mathtt{X}.a \leftarrow e \in R_p$ einer Attributgrammatik $AG \triangleq (G, A, R, B)$ mit abstrakter Syntax $G \triangleq (N, T, P, Z)$ und beliebigem Ausdruck e ist $\mathtt{X}.a$ das in p definierte Attribut.

Klasse des Attributs ererbt oder synthetisiert.

Zugriffe auf Tupelelemente Für beliebige Tupel $T_n \triangleq (E_1, \cdots, E_n)$ entspricht die Funktion $E_{i,T_n} : T_n \to E_i$ dem Zugriff auf das Tupelelement E_i.

Anwendung der Attributgrammatik-unabhängigen Musterdefinition $AG' = \mathcal{M}_u(AG, S)$ wird gebildet durch Bestimmung der Mengen nach Definition 4.16 und Anwendung dieser gefundenen Mengen nach Lemma 4.9.

resultierende Attributgrammatik AG' nach Anwendung eines oder mehrerer Muster.

Potenzmenge Sei U ein Universum, dann ist $2^U \triangleq \{X : X \subseteq U\}$ die Potenzmenge zu U.

Anhang B.

Mathematische Definitionen

B.1. Mengen, Listen, Sorten, Terme und Variablen

Mengen können das Fundament der Mathematik bilden

Definition B.1. (nach [30]) Eine **Menge** ist jede Zusammenfassung von bestimmten, wohlunterschiedenen Objekten m zu einem Ganzen M. Diese Objekte heißen **Elemente** der Menge M, geschrieben $m \in M$.

Weitere Eigenschaften von Mengen ausgehend vom intuitiven Verständnis von Mengen wird in [55] präsentiert. Die axiomatische Mengenlehre, die ebenfalls als Fundament der Mathematik angesehen werden kann, ist mit der Einführung der Zermelo-Fraenkel-Mengenlehre mit dem Auswahlaxiom in [92]. Soweit nicht anders angegeben sind die vorliegenden Definitionen bzgl. Mengen [55] entnommen und sonst [51].

Wie in anderen Arbeiten wird davon ausgegangen, dass die leere Menge das neutrale Element bzgl. Mengenvereinigung ist, dass zwei Mengen gleich sind, wenn für beide Mengen die Elemente übereinstimmen. Darüber hinaus, dass der Schnitt zweier Mengen nur die Elemente enhält, die in beiden Mengen enthalten sind, dass die Kardinalität einer Menge eine Funktion von Mengen in die natürlichen Zahlen ist, sodass diese genau der Anzahl der Elemente in dieser Menge entspricht. Darüber hinaus ist der Schnitt einer Menge mit der leeren Menge leer, mit sich selbst wieder diese Menge und ist der Schnitt zweier Mengen leer, dann ist die dijsunkte Vereinigung dieser Mengen definiert. Eine Teilmenge N einer Menge M bedeutet, dass jedes Element von N auch ein Element in M ist, echt ist diese Teilmenge, wenn es mindestens in M noch ein Element gibt, was nicht in N enthalten ist. An dieser Stelle werden die dafür üblich geführten Beweise nicht weiter aufgeführt. Folgende Definitionen sollen im Rahmen dieser Arbeit für Mengen ausreichend sein:

Definition B.2. Die Menge ohne Elemente heißt **leere Menge**, geschrieben $\emptyset$.

Definition B.3. Seien A und B Mengen, diese heißen **gleich** genau dann wenn sie dieselben Elemente enthalten, geschrieben $A = B$.

Definition B.4. Seien A und B Mengen, dann heißt A **Teilmenge** von B genau dann, wenn jedes Element aus A auch Element von B ist, geschrieben als $A \subseteq B$.

Anhand von Def. B.4 gilt, dass wenn $A \subseteq B$ und $B \subseteq A$ somit $A = B$ gilt, nach Def. B.3. Eine Menge M kann durch Aufzählung der Elemente, wie in $M = \{a, b, c\}$, spezifiziert werden oder über die Eigenschaften, die solch eine Menge ausmacht: $M' = \{x \colon x \in \mathbb{N}, x > 5, x < 10\}$.

Die folgenden Eigenschaften werden üblicherweise ebenso definiert oder hergeleitet, an dieser Stelle wird auf solche Definitionen und Herleitungen verzichtet. Im Folgenden sind $a, b, \cdots, z$ Elemente von Mengen und groß geschriebene Buchstaben $A, B, \cdots, Z$ Mengen.

$$
\begin{aligned}
A &\subseteq A \\
A &\notin A \text{ für alle Mengen } A \\
\emptyset &\subseteq A \text{ für alle Mengen } A \\
A \cup B &= \{x \colon x \in A \text{ oder } x \in B\} \\
A \cap B &= \{x \colon x \in A \text{ und } x \in B\} \\
A \uplus B &= \{x \colon x \in A \text{ und } x \in B\} \text{ falls } A \cap B = \emptyset \\
\{a, a\} &= \{a\}
\end{aligned}
$$

Darüber hinaus ist wichtig, dass Mengen aus Mengen aufgebaut werden können. Eine Darstellung der Axiome von Peano lässt sich über Mengen erreichen, wobei die leere Menge eine natürliche Zahl (0) ist und jeder Nachfolger einer natürlichen Zahl durch Vereinigung mit der Menge, die nur die leere Mengen enthält erreicht wird, d. h. $s(a) = a \cup \{a\}$.

Weitere Eigenschaften werden an dieser Stelle nicht vorgestellt. Aus Mengen lassen sich Relationen ableiten:

Definition B.5. Seien M und N Mengen, eine Menge $M \times N \triangleq \{(x, y) \colon x \in M \text{ und } y \in N\}$ heißt **karthesisches Produkt** der Mengen $MundN$.

Für Relationen über zwei Mengen wird das Element $(x, y) \in M \times N$ mit *Paar* bezeichnet, für Produkte $M_1 \times \cdots \times M_n$ endlich vieler Mengen heißen die Elemente $(x_1, \cdots, x_n)$ n-Tupel. Ist $x \neg y$ dann gilt $(x, y) \neg (y, x)$ – die Paare sind geordnet[51]. Ebenfalls aus [51] stammt die Notation für $M^1 = M$, $M^0 = \{\emptyset\}$ und $M^n = M^{n-1} \times M$ für dieselbe Menge M und die Definition für M^+ mit

$$M^+ = \bigcup_{i \in \mathbb{N} \setminus \{0\}} M^i = \{x \colon \text{ es gibt } i \geq 1 \text{ und } x \in M^i\}$$

und $M^* = M^0 \cup M^+$.

Relationen und, die im folgenden Abschnitt beschriebenen Graphen, erlauben die Definition von Funktionen. Für diese Arbeit ist ein Universum eine Menge, welche Objekte möglicher Mengen enthält. Eine Relation ist eine Teilmenge eines karthesischen Produkts. Jede Relation $\rho \subseteq U \times V$ über Universen U und V ist eine Funktion (geschrieben als $\rho \colon U \to V$.

Definition B.6. ([51]) Sei A eine Menge und $\oplus \colon A \times A \to A$ eine Funktion, dann heißt $(A, \oplus)$ **Monoid** genau dann, wenn

1. für alle $a, b, c \in A$ gilt $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ und
2. ein $e \in A$ mit $e \oplus a = a \oplus e$ existiert

Für `[]` als e und dem „Anhängen“ als Operation $\oplus$ wobei Anhängen von y an x dann $x\dot{y} = xy$ ist, so bildet dieser Monoid eine **Liste** oder **Folge** über einer Grundmenge U (auch Universum). Dargestellt wird eine Liste über $x_i \in U$ als $[x_1, x_2, \cdots, x_n]$ wobei diese Liste dann die Länge n hat. Statt U^* wird auch $[U]$ verwendet[51].

Ein Term zu einer Signatur ist wie folgt definiert:

Definition B.7. Ein **Term** der Sorte s über $\Sigma = (S, F)$ und $(X_s)_{s \in S}$ (Variablen der Sorte s) ist induktiv wie folgt definiert:

- Jede Variable $x \in X_s$ ist Term der Sorte s
- Jede Konstante $f \colon \to s$ ist Term der Sorte s

- Wenn $f\colon s_1 \times \cdots \times s_n \to s \in F$ und $t_1, \cdots, t_n$ Terme der Sorten $s_1, \cdots s_n$, dann ist $f(t_1, \cdots, t_n)$ Term der Sorte s,

wobei für Variablen der Sorte s gilt: $X_s \cap F_{\varepsilon,s} = \emptyset$ gilt. Ein **Grundterm** ist ein Term ohne Variablen, auch variablenfreier Term genannt. Wobei

B.2. Graphen und Bäume

Soweit nicht anders angegeben sind die hier vorgestellten Definitionen weitestgehend [51] und [117] entnommen. Jedoch wird an dieser Stelle nur bis zur Definition von (geordneten) Bäumen hingearbeitet und daher an einigen Stellen von den klassischen Definitionen abgewichen.

Weitere Informationen zu Graphen, Graphalgorithmen und weitere Definitionen finden sich bspw. in [35] und [40]. Ebenfalls in [51] wird eine Einführung über die diesen Definitionen zugrunde liegenden Mengenbegriffen und darauf aufbauendem Relationsbegriff gegeben.

Definition B.8. Das Paar $G \triangleq (E, K)$ heißt **gerichteter Graph** wobei E eine Menge von **Ecken** ist und $K \subseteq E \times E$ eine Realtion ist. K heißt Menge von **Kanten**. Ist E endlich, so heißt der Graph G endlich.

Ausgehend von dieser Definition lässt sich ein Teilgraph wie folgt definieren:

Definition B.9. Sei $G \triangleq (E, K)$ ein gerichteter Graph, der Graph $G' \triangleq (E', K')$ heißt **gerichteter Teilgraph** zu G genau dann, wenn

1. $E' \subseteq E$ und
2. $K' = \{(e, e')\colon e, e' \in E' \text{ und } (e, e') \in K\}$

Über die Definition des gerichteten Teilgraphen lassen sich die Eckengrade eines gerichteten Graphen bestimmen:

Definition B.10. Sei $G \triangleq (E, K)$ ein gerichteter Graph. Für die gerichteten Teilgraphen

- $G_{in,e} = (E_{in,e}, K_{in,e})$ mit $E_{in,e} = \{e'\colon (e', e) \in K\} \cup \{e\}$ und $K_{in,e} = \{(e', e) \in K\}$;
- $G_{out,e} = (E_{out,e}, K_{out,e})$ mit $E_{out,e} = \{e'\colon (e, e') \in K\} \cup \{e\}$ und $K_{out,e} = \{(e, e') \in K\}$

für eine Ecke $e \in E$ heißt $K_{in,e}$ die Menge der **Eingangskanten** von e, $K_{out,e}$ heißt Menge der **Ausgangskanten** von e; jedes $e' \in (E_{in,e} \cup E_{out,e})$ mit $e \neq e'$ heißt **Nachbar** von e. Der **Ein- bzw. Ausgangsgrad einer Ecke** $e \in E$ ist definiert als $indeg_e \triangleq \left|K_{in,e}\right|$ und $outdeg_e \triangleq \left|K_{out,e}\right|$. Für $e \in E$ und alle $e' \in E_{out,e}$ mit $e' \neq e$ ist e **Quelle** und e' **Ziel** für die Kanten $(e, e') \in K_{out,e}$

Definition B.10 weicht von der Definition in [51] ab, sodass Nachbar, Quelle, Ziel und Ein- und Ausgangsgrad über Teilgraphen definiert sind und nicht eine gesonderte Definition benötigen.

Zur Unterscheidung zwischen allgemeinen gerichteten Graphen und Bäumen ist folgende Definition wesentlich:

Definition B.11. Sei $G \triangleq (E, K)$ ein gerichteter Graph. Eine Folge von Ecken $\langle e_1, \cdots, e_n\rangle$ heißt **gerichteter Kantenzug von** e_1 **nach** e_n **der Länge** $n-1$ genau dann, wenn ein Teilgraph $G' = (E', K')$ zu G existiert, sodass $e_i \in E'$ für $1 \leq i \leq n$ und für alle i mit $1 \leq i \leq n-1$ gilt $(e_i, e_{i+1}) \in K'$. Ist $e_n = e_1$ und $n > 1$, dann heißt der gerichtete Kantenzug **Zyklus**. Der gerichtete Graph G heißt **azyklisch** wenn kein gerichteter Kantenzug existiert der ein Zyklus ist.

Zur Definition von Bäumen ist weiterhin der Begriff des markierten gerichteten Graphen notwendig:

Definition B.12. Das Tripel $G \triangleq (E, K, \mu)$ heißt **gerichteter kantenmarkierter Graph** genau dann, wenn

1. (E, K) ein gerichteter Graph,
2. $\mu\colon K \to M$ eine **Markierungsfunktion** und
3. M eine Menge von Marken

ist. Ist $\mu\colon E \to M$ die Markierungsfunktion, so heißt G **gerichteter eckenmarkierter Graph**.

Letztendlich folgen nun die Definitionen für Bäume und geordnete Bäume, die in Kapitel 3 für die Definition abstrakter Syntaxbäume notwendig sind.

Definition B.13. Ein gerichteter azyklischer Graph $G \triangleq (E, K)$ heißt **Baum** mit **Wurzel** e_0 genau dann, wenn für alle $e \in E$ der Eingangsgrad $indeg_e \leq 1$ ist und für alle $e \in E, e \neq e_0$ ein gerichteter Kantenzug von e_0 nach e existiert und der Eingangsgrad $index_{e_0} = 0$ ist. Eine Ecke $e \in E$ mit $outdeg_e = 0$ heißt **Blatt**. Für eine Kante $(e_1, e_2) \in K$ heißt e_2 **Kind** von e_1 und e_1 heißt **Elter** von e_2. Existiert für $e_1', e_2' \in E$ ein Pfad von e_1' nach e_2', so heißt e_2' **Nachfahre** von e_1' und e_1' wiederum **Vorfahr** von e_2'. Falls für eine Ecke e mit Kanten $(e, e_1) \in K$ und $(e, e_2) \in K$ existieren, mit $e_1 \neq e_2$, dann heißen e_1 und e_2 **Geschwister**. Ein gerichteter kantenmarkierter Graph $G' \triangleq (E', K', \mu)$ mit $\mu\colon K \to \mathbb{N}$ heißt **geordneter Baum** genau dann, wenn (E', K') ein Baum ist und für jede Ecke $e \in E'$ und alle $k \in K_{out,e}$ gilt: $\mu(k) \in \{0, \cdots, outdeg_e - 1\}$ und $\mu(k) = \mu(k')$ impliziert $k = k'$ und μ ist bijektiv. Falls $\mu((e, e')) = i$, dann heißt e' das i-te **Kind** von e.

Anhang C.

Substitution und Semantik von Prädikaten

Auf Basis der Definition von Attributwerttermen (Def. 4.7), Prädikattermen (Def. 4.8) und der Substitution von Prädikattermen (Def. 4.9) werden an dieser Stelle die Semantik und weitere Eigenschaften der Substitution sowie der Prädikate vorgestellt. Weiterhin sei auf die Definitionen aus Abschnitt B.1 verwiesen.

Grundsätzlich ist die Substitution wie folgt definiert:

Definition C.1. (u. a. angelehnt an [51]) Sei $\Sigma = (S, F)$ eine Signatur und X eine Menge von Variablen. Eine Abbildung $\sigma\colon X \to \mathcal{T}(\Sigma, X)$ heißt **Substitution** wobei $\sigma(x)$ ein Term der Sorte $s \in S$ ist, falls $x : s \in X$ und $\sigma(x) \neq x$ nur für endlich viele Variablen ist. Die Menge $\mathcal{T}(\Sigma, X)$ heißt Menge der Σ-Terme mit Variablen X.

Substitutionen werden als $\sigma = \left[t_1/x_1, \cdots, t_n/x_n\right]$ dargestellt. Folgende Definition zeigt die Anwendung einer Substitution auf einen Term:

Definition C.2. Sei $\Sigma = (S, F)$ eine Signatur und t Term der Sorte s über Σ und σ eine Substitution, $\sigma\colon X \to \mathcal{T}(\Sigma, X)$, die **Anwendung der Substitution** $t\sigma$ ist induktiv wie folgt definiert:

1. ist $t = x : s$ eine Variable $x \in X_s$ der Sorte s, dann ist $t\sigma = \sigma(x : s)$;
2. ist $t = c$ eine Konstante $c :\to s \in F$, dann ist $t\sigma = t$ und
3. $t = f(t_1, \cdots, t_n)$ für $f\colon s_1 \times \cdots \times s_n \to s \in F$ und $t_1, \cdots, t_n$ Terme der Sorten $s_1, \cdots, s_n$ dann ist $t\sigma = f(t_1\sigma, \cdots, t_n\sigma)$

Die Substitution von Termen ersetzt Variablen durch Terme. Im Rahmen dieser Arbeit haben Terme, wie auch Variablen, eine Sorte:

Definition C.3. Sei v eine Variable einer Sorte $\mathfrak{A}$ und $\sigma = \left[t/v\right]$ eine Substitution der Variablen v durch einen Grundterm t. Die Substitution σ heißt **sortentreue Substitution** genau dann, wenn t von der Sorte $\mathfrak{A}$ ist.

Im Folgenden wird die Sorte eines Terms oder einer Variablen innerhalb der Substitution dargestellt durch eine nach einem Doppelpunkt angegebene Sorte, wie bspw. $\left[t : \mathfrak{A}/v : \mathfrak{F}\right]$. Bei Termen die Unterterme unterschiedlicher Sorten enthalten, werden diese bis zur möglichen Anwendbarkeit herunter angewendet. Beispielsweise sind Prädikatterme aus Termen verschiedener Sorten aufgebaut und enthalten Variablen unterschiedlicher Ordnung.

Hervorzuheben an der sortentreuen Substitution ist, dass Grundterme der entsprechenden Sorte zu als Ergebnis der Substitution anzuwenden sind. Dies unterscheidet die sortentreue Substitution dieser Arbeit von der üblichen Substitution.

Anhang D.

Details der Implementierungen

D.1. Löschoperationen

Der Algorithmus, der ausgehend von der Löschanweisung eines (oder mehrerer) Attribute, alle zu entfernenden Attribute und Attributierungsregeln bestimmt, kann als transitiver Abschluss bestimmt werden. Die zu entfernenden Attribute sind, wie in Abschnitt 4.5.1 über

$\langle PatStat \rangle$ ::= $\langle DeleteStat \rangle$

$\langle DeleteStat \rangle$ ::= '`delete`' $\langle AttributeReference \rangle \leftarrow \langle Attribution \rangle$ '`in`' $\langle Production \rangle$

Eine Alternative Implementierung der Löschoperation kann durch transitiven Abschluss der Löschoperation erreicht werden: sei a das zu löschende Attribut und die dazugehörige Attributierungsregel $r = a \leftarrow f(b_1, \cdots, b_n)$ in R_p der Produtkion p. Die Anweisung **`delete`** $a \leftarrow f(b_1, \cdots, b_n)$ **`in`** p entspricht dann rekursiv den Anweisungen:

$$\begin{array}{l} \textbf{delete}\ a \leftarrow f(b_1, \cdots, b_n)\ \textbf{in}\ p \\ \quad \textbf{delete}\ b_1 \leftarrow \cdots\ \textbf{in}\ q_1 \\ \qquad \cdots \\ \quad \textbf{delete}\ b_n \leftarrow \cdots\ \textbf{in}\ q_n \end{array}$$

für alle b_i mit $b_i \in AD_{q_i}$ für $1 \leq i \leq n$, $i, n \in \mathbb{N}$. Dabei ist dann die Abbruchbedingung, bzw. das Ende der zu löschenden Attribute erreicht wenn $b_i \leftarrow c$ für eine Konstante c gefunden ist. Weiterhin sind alle Regel r, r_i mit $r = a \leftarrow f(b_1, \cdots, b_n)$ und $r_i = b_i \leftarrow \cdots$ ebenfalls in die zu löschenden Mengen aufzunehmen.

D.2. Herleitung Substitution für Symbolattributierungen

Das Verfahren zur Herleitung der einzelnen Mengen verläuft mehrstufig, dabei bezeichnet das in Abbildung 4.1 als $\langle SubstitutedSymbolVar \rangle$ angegebene Nichtterminal die Substitution $[X/S]$, falls X das vom Entwickler angegeben $\langle SubstitutedSymbolVar \rangle$ ist, formaler somit $[\langle\ SubstitutedSymbolVar\ \rangle/S]$. Analoges gilt für $\langle SubstitutedFunctionVariable \rangle$, $\langle SubstitutedDefAttribute \rangle$ und $\langle SubstitutedAttributeReference \rangle$. Im folgenden wird die Funktion `GetDependentAttrs` verwendet. Diese Funktion liefert zu einem Attribut, einer Attributgrammatik und einer Produktion alle abhängigen Attribute als Liste. Sei *prod* die angegebene Produktion der Attributgrammatik AG mit $a \in AD_{prod}$, dann existiert eine Regel $r \in R(AG)$ mit $r = a \leftarrow f(b_1, \cdots, b_n)$, so liefert die Funktion `GetDependentAttrs`(AG, a, *prod*) als Ergebnis $[b_1, \cdots, b_n]$ als Liste zurück. Analog liefert `GetProdSymbols` die Symoble einer Produktion als Liste zurück, d. h. `GetProdSymbols`(AG, *prod*) liefert für eine Produktion $p = \mathtt{X}_0 ::= \mathtt{X}_1\ \cdots\ \mathtt{X}_n$ die Liste $[\mathtt{X}_0, \mathtt{X}_1, \cdots, \mathtt{X}_n]$.

```
1: function CompleteSubstForSymbolcomp(AG, symb, defattrs, refattrs, function)   ▷
   Attributgrammatik und bereits festgelegte Terme für Variablen
2:     for p ∈ P(G(AG)) ∧ ⌈symb⌉_p > 0 do
3:         if attr ∈ defattrs ∧ attr ∈ AD_p then
4:             depAttrs ← GetDependentAttrs(AG, attr, p)
5:             prodSymbs ← GetProdSymbols(AG, p)
6:             i ← 1
7:             j ← 0
8:             for j < length(prodSymbs) do
9:                 S_j ← prodSymbs[j]
10:                j ← j + 1
11:            end for
12:            for j < n do
13:                S_j ← ε
14:                j ← j + 1
15:            end for
16:            if attr ∈ AI(symb) ∧|symb|_p > 0 then
17:                for i < length(depAttrs) do
18:                    c_{d_i} ← depAttrs[i]
19:                    i ← i + 1
20:                end for
21:                for i < d_l do
22:                    c_{d_i} ← ε
23:                    i ← i + 1
24:                end for
25:            else                                          ▷ Attribut ist synthetisiert
26:                S_0 ← symb
27:                for i < length(depAttrs) do
28:                    b_{d_i} ← depAttrs[i]
29:                    i ← i + 1
30:                end for
31:                for i < b_m do
32:                    b_{d_i} ← ε
33:                    i ← i + 1
34:                end for
35:            end if
36:        end if
37:    end for
       return c_{d_1}, ··· , c_{d_l}, b_{d_1}, ··· , b_{d_j}, S_0, ··· , S_n     ▷ Gebe noch fehlende Substitutionen zurück
38: end function
```

Algorithmus D.1 – Substitutionsherleitung für Symbolattributierungen ausgehend von einer Teilmenge der passenden Substitution

Nachdem alle Substitutionen gefunden sind, ist die Auswertung der Prädikate und damit einhergehend die zu erweiternde Teilmenge der ursprünglichen Attributgrammatik zu finden. Dazu müssen die Prädikate nach Definition 4.10 auszuwerten. Auf diese Auswertung wird hier verzichtet. Ebenso wird nicht die nach Definition 4.11 vorgegebene Einordnung der Produktionen und Attributierungsregeln in AG_+ und AG_- algorithmisch nochmals vorgestellt. Eine nochmalige, minimal genauere, Vorstellung von Definition 4.11 würde keine weitere Erkenntnis liefern.

Ebenso wird Definition 4.16 nicht nochmals algorithmisch dargestellt. Die Mengen ergeben sich direkt aus der Definition heraus.

D.3. Substituion für Kopieranweisungen

Bei Muster 3 wird von einem Attribut a ausgegangen, welches von dem Punkt der Definition herunter kopiert wird. Sei p eine Produktion in der a definiert wird und `X` das Symbol für das dieses Attribut definiert wurde. Dann wähle für alle Produktionen q mit linker Seite `X` alle Symbole `Y` für die es keine (definierende) Attributierungsregel für das Attribut a dieses Symbols gibt. Für jedes dieser Symbole füge in jeder Regel q eine Kopierregel $\texttt{Y}.a \leftarrow \texttt{X}.a$ ein. Letztendlich ist diese Rekursion bis zu jedem Blatt von dem Teilbaum mit der Wurzel `X` durchzuführen. Dieser Prozess fängt bei jeder solchen Produktion mit Definition für a „neu“ an. Eine Implementierung als Arbeitslisten[1] Algorithmus kann hier genutzt werden.

D.4. Substitutionsbeispiele

Ein Beispiel der praktischen Anwendung von Muster 7 im Sinne der wiederholten Anwendung der Symbolattributierung für mehrere Symbole. In Beispiel 6.2a wurde die Namensanalyse für die Sprache Ziria vorgestellt. Ohne Verwendung des Musters 9 zur Herleitung der Namensanalyse zu verwenden, wird eine Namensanalyse für dieses Beispiel in Beispiel D.1 unter Verwendung von Muster 1 und Musterkomposition sowie Muster 7 realisiert.

```
1  bind is newKey({VarDef, VarRef}.vars, {VarDef, VarRef}.sym)
2  chain vars head Program ← ∅
3  symbol VarDef
4  attr ↑bind:val ← VarDef.value
5       ↑vars ← tail.vars ← ↑bind:val
6  cond ↑sym ∉ ↓vars  ⇒ error "Already defined: " ++ ↑sym
7
8  symbol VarRef
9  cond ↑sym ∈ ↓vars  ⇒ error "Unknown Reference: " ++ ↑sym
```

a) Muster-basierte Attributgrammatik zur Beschreibung der Namensanalyse mittels Symbolberechnungen, Kettenattributen und Musterabbildung

$$Subs_1 = \left[\texttt{VarDef}/S, \texttt{VarDef.bind}/a_1, newKey/f, VarDef.vars/b_1, VarDef.sym/b_2\right]$$
$$Subs_2 = \left[\texttt{VarRef}/S, \texttt{VarRef.bind}/a_1, newKey/f, VarRef.vars/b_1, VarRef.sym/b_2\right]$$

b) Substitutionsmengen für das Muster der Attributabbildung in der ersten Zeile von D.1a durch Angabe der einzelnen Substitutionen der Symbolattributierung (Basismuster 2 und Basismuster 3) ohne Angabe der Substitutionen zu ε.

.

Beispiel D.1 – Alternative Rückführung der Namensanalyse für Beispiel 6.2a unter Ausnutzung von Muster 7 mit Angabe der Substitutionen in der Rückführung von Muster 7 auf Symbolattributierungen

[1] engl. Worklist Algorithm

Anhang E.

Ausgewählte Implementierungen mit Mustern und die resultierenden Attributgrammatiken

Neben der vollständigen Darstellung von nur in Ausschnitten präsentierten Attributgrammatiken, werden in diesem Kapitel die resultierenden Attributgrammatiken nach Anwendung der Muster der jeweiligen Beispiele vorgestellt.

E.1. Komplettierung der Beispiele

Quelltext E.1 zeigt die vollständige Variante von Beispiel 1.1 aus Kapitel 1 mit allen Kopieranweisungen.

```
1  -- Namensanalyse
2  rule Root ::= Expr
3  attr Expr.envIn = []
4
5  rule Expr ::= VarDef Expr Expr
6  attr Expr2.envIn = Expr1.envIn
7       Expr3.envIn = ((VarDef.name, Expr2.value):Expr1.envIn)
8  cond VarDef.name ∉ Expr1.envIn  ⇒ error "Already defined " ++ VarDef.name
9
10 rule Factor ::= VarRef
11 attr Factor.value = Factor.envIn[VarRef.name]
12 cond VarRef.name ∉ Factor.envIn  ⇒ error "Unknown Variable " ++ VarRef.name
13
14 -- Konstantenberechnung
15 rule Root ::= Expr
16 attr report "Output value = " ++ Expr.value
17
18 rule Term ::= Term Factor
19 attr Term1.value = Term2.value * Factor.value
20
21 rule Expr ::= Expr Term
22 attr Expr1.value = Expr2.value + Term.value
23
24 rule Expr ::= VarDef Expr Expr
25 attr Expr1.value = Expr3.value
26
27 -- Identitätsausgabe
28 rule Root ::= Expr
29 attr report "Identity Code = " ++ Expr.output
30
31 rule Expr ::= VarDef Expr Expr
32 attr Expr1.output = "let " ++ VarDef.name ++ " = " ++ Expr2.output ++
33                          " in " ++ Expr3.output
34
35 rule Term ::= Term Factor
36 attr Term1.output = Term2.output ++ " * " ++ Factor.output
37
38 rule Expr ::= Expr Term
39 attr Expr1.output = Expr2.output ++ " + " ++ Term.output
40
41 -- Kopieranweisungen Namensanalyse
42 rule Expr ::= Expr Term
43 attr Expr2.envIn = Expr1.envIn
44      Term.envIn = Expr1.envIn
45
46 rule Expr ::= Term
```

```
47  attr Term.envIn = Expr.envIn
48
49  rule Term ::= Factor
50  attr Factor.envIn = Term.envIn
51
52  rule Term ::= Term Factor
53  attr Factor.envIn = Term1.envIn
54       Term2.envIn = Term1.envIn
55
56  -- Kopieranweisungen Wertberechnung
57  rule Factor ::= Int
58  attr Factor.value = Int
59
60  rule Expr ::= Term
61  attr Expr.value = Term.value
62
63  rule Term ::= Factor
64  attr Term.value = Factor.value
65
66  -- Kopieranweisungen Ausgabegenerierung
67  rule Expr ::= Term
68  attr Expr.output = Term.output
69
70  rule Factor ::= Int
71  attr Factor.output = "" ++ Int
72
73  rule Factor ::= VarRef
74  attr Factor.output = "" ++ VarRef.name
75
76  rule Term ::= Factor
77  attr Term.output = Factor.output
78
79  rule VarDef ::= Id
80  attr VarDef.name = Id
81
82  rule VarRef ::= Id
83  attr VarRef.name = Id
```

Quelltext E.1 – Vollständige Attributgrammatik zu Beispiel 1.1

Die Vervollständigung von Beispiel 1.2 zeigt Beispiel E.2.

```
1   rule Program ::= Decls
2   attr Decls.declsIn ← ∅
3        Decls.env ← Decls.declsOut
4
5   rule Decls ::= Decls Decl
6   attr Decls2.declsIn ←Decls1.declsIn
7        Decl.declsIn ← Decls2.declsOut
8        Decls1.declsOut ←Decl.declsOut
9        Decls2.env ←Decls1.env
10       Decl.env ← Decls1.env
11
12  rule Decl ::= RqDecl
13  attr RqDecl.declsIn ← Decl.declsIn
14       Decl.declsOut ← RqDecl.declsOut
15       RqDecl.env ← Decl.env
16
17  rule RqDecl ::= RqDefId RqReferences
18  attr RqDefId.declsIn ← RqDecl.declsIn
19       RqDecl.declsOut ← RqDefId.declsOut
20       RqReferences.env ← RqDecl.env
21       RqDefId.env ← RqDecl.env
22       RqDefId.is_root ← false
23
24  rule RqDefId ::= ID
25  attr RqDefId.bind ← bindKey(RqDefId.declsIn, RqDefId.sym)
26       RqDefId.bind:is_root ← RqDefId.is_root
27       RqDefId.declsOut ← RqDefId.declsIn ↞ RqDefId.bind:is_root
28  cond RqDefId.sym ∉ RqDefId.declsIn  ⇒ error "Already Defined "
29                                                  ++ RqDefId.sym
30
31  rule RqReference ::= ID
32  attr RqReference.declsOut ← RqReference.declsIn
33       RqReference.bind ← bindInEnv(RqReference.env, RqReference.sym)
34  cond RqReference.sym ∈ RqReference.env  ⇒ error "Unknown Requirement "
35                                                   ++ RqReference.sym
36
37  rule Decls ::= ε
38  attr Decls.declsOut ← Decls.declsIn
39
40  rule RqReferences ::= RqReferences RqReference
41  attr RqReferences2.env ←RqReferences1.env
```

```
42        RqReference.env ← RqReferences1.env
43
44  rule RootDecl ::= RqDefId RqReferences
45  attr RqDefId.declsIn ← RootDecl.declsIn
46       RootDecl.declsOut ← RqDefId.declsOut
47       RqReferences.env ← RootDecl.env
48       RqDefId.env ← RootDecl.env
49       RqDefId.is_root ← true
50
51  rule Decl ::= RootDecl
52  attr RootDecl.declsIn ← Decl.declsIn
53       Decl.declsOut ← RootDecl.declsOut
54       RootDecl.env ← Decl.env
```

Quelltext E.2 – Ausschnitt aus einer Attributgrammatik zur Namensanalyse von Anforderungen bestehend nur aus Kopieranweisungen.

E.2. Resultierende Attributgrammatiken nach Musteranwendung

Ausgehend von den Beispielen aus Kapitel 6 präsentiert dieser Abschnitt eben diese vollständig bzgl. der in Kapitel 6 gezeigten abstrakten Syntaxen. Weitere semantische Analysen oder weitere Teile der Codegenerierung werden nicht präsentiert. So wird auch darauf verzichtet an dieser Stelle weitere Inhalte der Codegenerierung bspw. für Beispiel 2.3 zu zeigen. Dies kann über eine Hilfsfunktion realisiert werden – für die Semantik der Attributgrammatik müssen nicht 50 zusätzliche Zeilen statischen C++ Quelltextes angeführt werden.

Quelltext E.3 – Resultierende Attributgrammatik zu Beispiel 2.1 unter Verwendung der Muster aus Beispiel 6.1b und Beispiel 6.5.

```
 1  rule Program ::= Process
 2  attr Process.proc_namesIn ← emptyset
 3
 4  rule Process ::= PDef PStats
 5  attr PDef.proc_namesIn ← Process.proc_namesIn
 6       PStats.proc_namesIn ← PDef.proc_namesOut
 7       Process.proc_namesOut ← PStats.proc_namesOut
 8
 9  rule PStats ::= PStats PStat
10  attr PStats2.proc_namesIn ←PStats1.proc_namesIn
11       PStat.proc_namesIn ← PStats2.proc_namesOut
12       PStats1.proc_namesOut ←PStat.proc_namesOut
13
14  rule PStats ::= ε
15  attr PStats.proc_namesOut ← PStats.proc_namesIn
16
17  rule PDef ::= PDefId
18  attr PDef.bind ← bindKey(PDef.proc_namesIn, PDef.sym
19       PDef.proc_namesOut ← PDef.proc_namesIn > =  PDef.bind
20  cond PDef.sym ∉ PDef.proc_namesIn  ⇒ error "already defined " ++ PDef.sym
21
22  rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
23  attr SEntities.proc_namesIn ← PStat.proc_namesIn
24       PVars.proc_namesIn ← SEntities.proc_namesOut
25       PEvents.proc_namesIn ← PVars.proc_namesOut
26       Prefs.proc_namesIn ← PEvents.proc_namesOut
27       Main.proc_namesIn ← Prefs.proc_namesOut
28       EventHandles.proc_namesIn ← Main.proc_namesOut
29       PStat.proc_namesOut ← EventHandles.proc_namesOut
30
31  rule SEntities ::= SEntities SEntity
32  attr SEntities2.proc_namesIn ←SEntities1.proc_namesIn
33       SEntity.proc_namesIn ← SEntities2.proc_namesOut
34       SEntities1.proc_namesOut ←SEntity.proc_namesOut
35
36  rule SEntities ::= ε
37  attr SEntities.proc_namesOut ← SEntities.proc_namesIn
38
39  rule SEntity ::= EType EDef EType ExRef
40  attr EType1.proc_namesIn ←SEntity.proc_namesIn
41       EDef.proc_namesIn ← EType1.proc_namesOut
42       EType2.proc_namesIn ←EDef.proc_namesOut
43       ExRef.proc_namesIn ← EType2.proc_namesOut
44       SEntity.proc_namesOut ← ExRef.proc_namesOut
45
46  rule LEntity ::= EDef EType
```

```
47  attr EDef.proc_namesIn ← LEntity.proc_namesIn
48       EType.proc_namesIn ← EDef.proc_namesOut
49       LEntity.proc_namesOut ← EType.proc_namesOut
50
51  rule PVars ::= PVars PVar
52  attr PVars2.proc_namesIn ←PVars1.proc_namesIn
53       PVar.proc_namesIn ← PVars2.proc_namesOut
54       PVars1.proc_namesOut ←PVar.proc_namesOut
55
56  rule PVars ::= ε
57  attr PVars.proc_namesOut ← PVars.proc_namesIn
58
59  rule PEvent ::= SubscribeRef
60  attr SubscribeRef.proc_namesIn ← PEvent.proc_namesIn
61       PEvent.proc_namesOut ← SubscribeRef.proc_namesOut
62
63  rule EventHandle ::= ERef VDef CStats
64  attr ERef.proc_namesIn ← EventHandle.proc_namesIn
65       VDef.proc_namesIn ← ERef.proc_namesOut
66       CStats.proc_namesIn ← VDef.proc_namesOut
67       EventHandle.proc_namesOut ← CStats.proc_namesOut
68
69  rule EDef ::= ID
70  attr EDef.proc_namesOut ← EDef.proc_namesIn
71
72  rule ERef ::= ID
73  attr ERef.proc_namesOut ← ERef.proc_namesIn
74
75  rule Program ::= Process
76  attr Process.event_namesIn ← emptyset
77       Program.event_names ← Process.event_namesOut
78
79  rule Process ::= PDef PStats
80  attr PDef.event_namesIn ← Process.event_namesIn
81       PStats.event_namesIn ← PDef.event_namesOut
82       Process.event_namesOut ← PStats.event_namesOut
83
84  rule PStats ::= PStats PStat
85  attr PStats2.event_namesIn ←PStats1.event_namesIn
86       PStat.event_namesIn ← PStats2.event_namesOut
87       PStats1.event_namesOut ←PStat.event_namesOut
88
89  rule PStats ::= ε
90  attr PStats.event_namesOut ← PStats.event_namesIn
91
92  rule PDef ::= PDefId
93  attr PDef.event_namesOut ← PDef.event_namesIn
94
95
96  rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
97  attr SEntities.event_namesIn ← PStat.event_namesIn
98       PVars.event_namesIn ← SEntities.event_namesOut
99       PEvents.event_namesIn ← PVars.event_namesOut
100      Prefs.event_namesIn ← PEvents.event_namesOut
101      Main.event_namesIn ← Prefs.event_namesOut
102      EventHandles.event_namesIn ← Main.event_namesOut
103      PStat.event_namesOut ← EventHandles.event_namesOut
104
105 rule SEntities ::= SEntities SEntity
106 attr SEntities2.event_namesIn ←SEntities1.event_namesIn
107      SEntity.event_namesIn ← SEntities2.event_namesOut
108      SEntities1.event_namesOut ←SEntity.event_namesOut
109
110 rule SEntities ::= ε
111 attr SEntities.event_namesOut ← SEntities.event_namesIn
112
113 rule SEntity ::= EType EDef EType ExRef
114 attr EType1.event_namesIn ←SEntity.event_namesIn
115      EDef.event_namesIn ← EType1.event_namesOut
116      EType2.event_namesIn ←EDef.event_namesOut
117      ExRef.event_namesIn ← EType2.event_namesOut
118      SEntity.event_namesOut ← ExRef.event_namesOut
119
120 rule LEntity ::= EDef EType
121 attr EDef.event_namesIn ← LEntity.event_namesIn
122      EType.event_namesIn ← EDef.event_namesOut
123      LEntity.event_namesOut ← EType.event_namesOut
124
125 rule PVars ::= PVars PVar
126 attr PVars2.event_namesIn ←PVars1.event_namesIn
127      PVar.event_namesIn ← PVars2.event_namesOut
128      PVars1.event_namesOut ←PVar.event_namesOut
129
130 rule PVars ::= ε
131 attr PVars.event_namesOut ← PVars.event_namesIn
```

```
132
133  rule PEvent ::= SubscribeRef
134  attr SubscribeRef.event_namesIn ← PEvent.event_namesIn
135       PEvent.event_namesOut ← SubscribeRef.event_namesOut
136
137  rule EventHandle ::= ERef VDef CStats
138  attr ERef.event_namesIn ← EventHandle.event_namesIn
139       VDef.event_namesIn ← ERef.event_namesOut
140       CStats.event_namesIn ← VDef.event_namesOut
141       EventHandle.event_namesOut ← CStats.event_namesOut
142
143  rule EDef ::= ID
144  attr EDef.bind ← bindKey(EDef.event_namesIn, EDef.sym)
145       EDef.event_namesOut ← EDef.event_namesIn ↞ EDef.bind
146  cond EDef.sym ∉ EDef.event_namesIn ⇒ error "Already Defined " ++ EDef.sym
147
148  rule ERef ::= ID
149  attr ERef.bind ← bindInEnv(ERef.event_namesIn, ERef.sym)
150       ERef.event_namesOut ← ERef.event_namesIn ↞ ERef.bind
151  cond ERef.sym ∈ ERef.event_namesIn ⇒ error "Unknown Reference " ++ ERef.sym
152
153  rule Program ::= Process
154  attr Process.var_namesIn ← emptyset
155       Program.var_names ← Process.var_namesOut
156
157  rule Process ::= PDef PStats
158  attr PDef.var_namesIn ← Process.var_namesIn
159       PStats.var_namesIn ← PDef.var_namesOut
160       Process.var_namesOut ← PStats.var_namesOut
161
162  rule PStats ::= PStats PStat
163  attr PStats2.var_namesIn ←PStats1.var_namesIn
164       PStat.var_namesIn ← PStats2.var_namesOut
165       PStats1.var_namesOut ←PStat.var_namesOut
166
167  rule PStats ::= ε
168  attr PStats.var_namesOut ← PStats.var_namesIn
169
170  rule PDef ::= PDefId
171  attr PDef.var_namesOut ← PDef.var_namesIn
172
173
174  rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
175  attr SEntities.var_namesIn ← PStat.var_namesIn
176       PVars.var_namesIn ← SEntities.var_namesOut
177       PEvents.var_namesIn ← PVars.var_namesOut
178       Prefs.var_namesIn ← PEvents.var_namesOut
179       Main.var_namesIn ← Prefs.var_namesOut
180       EventHandles.var_namesIn ← Main.var_namesOut
181       PStat.var_namesOut ← EventHandles.var_namesOut
182
183  rule SEntities ::= SEntities SEntity
184  attr SEntities2.var_namesIn ←SEntities1.var_namesIn
185       SEntity.var_namesIn ← SEntities2.var_namesOut
186       SEntities1.var_namesOut ←SEntity.var_namesOut
187
188  rule SEntities ::= ε
189  attr SEntities.var_namesOut ← SEntities.var_namesIn
190
191  rule SEntity ::= EType EDef EType ExRef
192  attr EType1.var_namesIn ←SEntity.var_namesIn
193       EDef.var_namesIn ← EType1.var_namesOut
194       EType2.var_namesIn ←EDef.var_namesOut
195       ExRef.var_namesIn ← EType2.var_namesOut
196       SEntity.var_namesOut ← ExRef.var_namesOut
197
198  rule LEntity ::= EDef EType
199  attr EDef.var_namesIn ← LEntity.var_namesIn
200       EType.var_namesIn ← EDef.var_namesOut
201       LEntity.var_namesOut ← EType.var_namesOut
202
203  rule PVars ::= PVars PVar
204  attr PVars2.var_namesIn ←PVars1.var_namesIn
205       PVar.var_namesIn ← PVars2.var_namesOut
206       PVars1.var_namesOut ←PVar.var_namesOut
207
208  rule PVars ::= ε
209  attr PVars.var_namesOut ← PVars.var_namesIn
210
211  rule PEvent ::= SubscribeRef
212  attr SubscribeRef.var_namesIn ← PEvent.var_namesIn
213       PEvent.var_namesOut ← SubscribeRef.var_namesOut
214
215  rule EventHandle ::= ERef VDef CStats
216  attr ERef.var_namesIn ← EventHandle.var_namesIn
```

```
217       VDef.var_namesIn ← ERef.var_namesOut
218       CStats.var_namesIn ← VDef.var_namesOut
219       EventHandle.var_namesOut ← CStats.var_namesOut
220
221   rule EDef ::= ID
222   attr EDef.var_namesOut ← EDef.var_namesIn
223
224   rule ERef ::= ID
225   attr ERef.var_namesOut ← ERef.var_namesIn
226
227   rule VDef ::= ID
228   attr VDef.bind ← bindKey(VDef.var_namesIn, VDef.sym)
229        VDef.var_namesOut ← VDef.var_namesIn ↞ VDef.bind
230   cond VDef.sym ∉ VDef.var_namesIn ⇒ error "Already Defined " ++ VDef.sym
231
232   rule Program ::= Process
233   attr Program.c_handle ← 0
234        Process.incl_c_handle ← Program.c_handle
235
236   rule EventHandle ::= ERef VDef CStats
237   attr EventHandle.c_handle ← 1
238        ERef.incl_c_handle ← EventHandle.c_handle
239        VDef.incl_c_handle ← EventHandle.c_handle
240        CStats.incl_c_handle ← EventHandle.c_handle
241
242   rule Process ::= PDef PStats
243   attr PDef.incl_c_handle ← Process.incl_c_handle
244        PStats.incl_c_handle ← Process.incl_c_handle
245
246   rule PStats ::= PStats PStat
247   attr PStats2.incl_c_handle ←PStats1.incl_c_handle
248        PStat.incl_c_handle ← PStats1.incl_c_handle
249
250   rule PDef ::= PDefId
251   attr PDefId.incl_c_handle ← PDef.incl_c_handle
252
253   rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
254   attr SEntities.incl_c_handle ← PStat.incl_c_handle
255        PVars.incl_c_handle ← PStat.incl_c_handle
256        PEvents.incl_c_handle ← PStat.incl_c_handle
257        Prefs.incl_c_handle ← PStat.incl_c_handle
258        Main.incl_c_handle ← PStat.incl_c_handle
259        EventHandles.incl_c_handle ← PStat.incl_c_handle
260
261   rule SEntities ::= SEntities SEntity
262   attr SEntities2.incl_c_handle ←SEntities1.incl_c_handle
263        SEntity.incl_c_handle ← SEntities1.incl_c_handle
264
265   rule SEntity ::= EType EDef EType ExRef
266   attr EType1.incl_c_handle ←SEntity.incl_c_handle
267        EDef.incl_c_handle ← SEntity.incl_c_handle
268        EType2.incl_c_handle ←SEntity.incl_c_handle
269
270   rule LEntity ::= EDef EType
271   attr EDef.incl_c_handle ← LEntity.incl_c_handle
272        EType.incl_c_handle ← LEntity.incl_c_handle
273
274   rule PVars ::= PVars PVar
275   attr PVars2.incl_c_handle ←PVars1.incl_c_handle
276        PVar.incl_c_handle ← PVars1.incl_c_handle
277
278   rule PEvent ::= SubscribeRef
279   attr SubscribeRef.incl_c_handle ← PEvent.incl_c_handle
280
281   rule ERef ::= ID
282   attr ERef.bind:handled ← ERef.bind:handled + ERef.incl_c_handle ↞ ERef.gotk_handledIn
283       ERef.gotk_handledOut ←ERef.gotk_handledIn
284
285
286   rule Program ::= Process
287   attr Process.gotk_handledIn ←()
288        Program.gotk_handled ←Process.gotk_handledOut
289        Process.incl_gotk_handled ←Program.gotk_handled
290
291   rule Process ::= PDef PStats
292   attr PDef.gotk_handledIn ←Process.gotk_handledIn
293        PStats.gotk_handledIn ←PDef.gotk_handledOut
294        Process.gotk_handledOut ←PStats.gotk_handledOut
295        PDef.incl_gotk_handled ←Process.incl_gotk_handled
296        PStats.incl_gotk_handled ←Process.incl_gotk_handledIn
297
298   rule PStats ::= PStats PStat
299   attr PStats2.gotk_handledIn ←PStats1.gotk_handledIn
300        PStat.gotk_handledIn ←Pstats2.gotk_handledOut
301        PStats1.gotk_handledOut ←PStat.gotk_handledOut
```

```
302       PStats2.incl_gotk_handled ←PStats1.incl_gotk_handled
303       PStat.incl_gotk_handled ←PStats1.incl_gotk_handled
304
305   rule PStats ::= ε
306   attr PStats.gotk_handledOut ←PStats.gotk_handledIn
307
308   rule PDef ::= PDefId
309   attr PDefId.gotk_handledIn ←PDef.gotk_handledIn
310       PDef.gotk_handledOut ←PDefId.gotk_handledOut
311
312   rule PStat ::= SEntities PVars PEvents Prefs Main EventHandles
313   attr SEntities.gotk_handledIn ←PStat.gotk_handledIn
314       PVars.gotk_handledIn ←SEntities.gotk_handledOut
315       PEvents.gotk_handledIn ←PVars.gotk_handledOut
316       Prefs.gotk_handledIn ←PEvents.gotk_handledOut
317       Main.gotk_handledIn ←Prefs.gotk_handledOut
318       EventHandles.gotk_handledIn ←Main.gotk_handledOut
319       PStat.gotk_handledOut ←EventHandles.gotk_handledOut
320       SEntities.incl_gotk_handled ←PStat.gotk_handled
321       PVars.incl_gotk_handled ←PStat.gotk_handled
322       PEvents.incl_gotk_handled ←PStat.gotk_handled
323       Prefs.incl_gotk_handled ←PStat.gotk_handled
324       Main.incl_gotk_handled ←PStat.gotk_handled
325       EventHandles.incl_gotk_handled ←PStat.gotk_handled
326
327   rule SEntities ::= SEntities SEntity
328   attr SEntities2.gotk_handledIn ←SEntities1.gotk_handledIn
329       SEntity.gotk_handledIn ←SEntities2.gotk_handledOut
330       SEntities1.gotk_handledOut ←SEntity.gotk_handledOut
331       SEntities2.incl_gotk_handled ←SEntities1.incl_gotk_handled
332       SEntity.incl_gotk_handled ←SEntities1.incl_gotk_handled
333
334   rule SEntities ::= ε
335   attr SEntities.gotk_handledOut ←SEntities.gotk_handledIn
336
337   rule SEntity ::= EType EDef EType ExRef
338   attr EType1.gotk_handledIn ←SEntity.gotk_handledIn
339       EDef.gotk_handledIn ←EType1.gotk_handledOut
340       EType2.gotk_handledIn ←EDef.gotk_handledOut
341       ExRef.gotk_handledIn ←EType2.gotk_handledOut
342       SEntity.gotk_handledOut ←ExRef.gotk_handledOut
343       EType1.incl_gotk_handled ←SEntity.incl_gotk_handled
344       EType2.incl_gotk_handled ←SEntity.incl_gotk_handled
345       EDef.incl_gotk_handled ←SEntity.incl_gotk_handled
346       ExRef.incl_gotk_handled ←SEntity.incl_gotk_handled
347
348   rule LEntity ::= EDef EType
349   attr Edef.gotk_handledIn ←LEntity.gotk_handledIn
350       EType.gotk_handledIn ←EDef.gotk_handledOut
351       EDef.incl_gotk_handled ←LEntity.incl_gotk_handled
352       EType.incl_gotk_handled ←LEntity.incl_gotk_handled
353
354   rule PVars ::= PVars PVar
355   attr PVars2.gotk_handledIn ←PVars1.gotk_handledIn
356       PVar.gotk_handledIn ←PVars2.gotk_handledOut
357       PVars1.gotk_handledOut ←PVars2.gotk_handledOut
358       PVars2.incl_gotk_handled ←PVars1.incl_gotk_handled
359       PVar.incl_gotk_handled ←PVars1.incl_gotk_handled
360
361   rule PVars ::= ε
362   attr PVars.gotk_handledOut ←PVars.gotk_handledIn
363
364   rule PEvent ::= SubscribeRef
365   attr SubscribeRef.gotk_handledIn ←PEvent.gotk_handledIn
366       PEvent.gotk_handledOut ←SubscribeRef.gotk_handledOut
367       SubscribeRef.incl_gotk_handled ←PEvent.incl_gotk_handled
368
369   rule EventHandle ::= ERef VDef CStats
370   attr ERef.gotk_handledIn ←EventHandle.gotk_handledIn
371       VDef.gotk_handledIn ←ERef.gotk_handledOut
372       CStats.gotk_handledIn ←VDef.gotk_handledOut
373       EventHandle.gotk_handledOut ←CStats.gotk_handledOut
374       ERef.incl_gotk_handled ←EventHandle.incl_gotk_handled
375       VDef.incl_gotk_handled ←EventHandle.incl_gotk_handled
376       CStats.incl_gotk_handled ←EventHandle.incl_gotk_handled
377
378   rule EDef ::= ID
379   attr EDef.all_handles ← EDef.bind:handled ↞ EDef.incl_gotk_handled
380       EDef.gotk_handledOut ←EDef.gotk_handledIn
381   cond EDef.all_handles > 0 ⇒ error "not handled: " ++ EDef.sym
382
383   rule ERef ::= ID
384   attr ERef.gotk_handledOut ←ERef.gotk_handledIn
385
386   rule VDef ::= ID
```

```
387  attr VDef.gotk_handledOut ←VDef.gotk_handledIn
388
389  rule LEntity ::= EDef EType
390  attr EDef.type ← EType.sym
391
392  rule SEntity ::= EType EDef EType ExRef
393  attr EDef.type ← EType_1.sym
394  cond EType_1.sym = EType_2.sym
395     ⇒ error "type mismatch " ++ EType_1.sym ++ " vs. " ++ EType_2.sym
396
397  attr EType.proc_namesIn ← ForLoop.proc_namesIn
398       EDef.proc_namesIn ← EType.proc_namesOut
399       ERef.proc_namesIn ← EDef.proc_namesOut
400       ForStats.proc_namesIn ← ERef.proc_namesOut
401       ForLoop.proc_namesOut ← ForStats.proc_namesOut
402
403
404  rule ForLoop ::= EType EDef ERef ForStats
405  attr EType.event_namesIn ← ForLoop.event_namesIn
406       EDef.event_namesIn ← EType.event_namesOut
407       ERef.event_namesIn ← EDef.event_namesOut
408       ForStats.event_namesIn ← ERef.event_namesOut
409       ForLoop.event_namesOut ← ForStats.event_namesOut
410
411  rule ForLoop ::= EType EDef ERef ForStats
412  attr EType.var_namesIn ← ForLoop.var_namesIn
413       EDef.var_namesIn ← EType.var_namesOut
414       ERef.var_namesIn ← EDef.var_namesOut
415       ForStats.var_namesIn ← ERef.var_namesOut
416       ForLoop.var_namesOut ← ForStats.var_namesOut
417
418  rule ForLoop ::= EType EDef ERef ForStats
419  attr EType.incl_c_handle ← ForLoop.incl_c_handle
420       EDef.incl_c_handle ← ForLoop.incl_c_handle
421       ERef.incl_c_handle ← ForLoop.incl_c_handle
422       ForStats.incl_c_handle ← ForLoop.incl_c_handle
423
424  rule ForLoop ::= EType EDef ERef ForStats
425  attr EType.gotk_handledIn ←ForLoop.gotk_handledIn
426       EDef.gotk_handledIn ←EType.gotk_handledOut
427       ERef.gotk_handledIn ←EDef.gotk_handledOut
428       ForStats.gotk_handledIn ←ERef.gotk_handledOut
429       ForLoop.gotk_handledOut ←ForStats.gotk_handledOut
430       EType.incl_gotk_handled ←ForLoop.incl_gotk_handled
431       ERef.incl_gotk_handled ←ForLoop.incl_gotk_handled
432       EDef.incl_gotk_handled ←ForLoop.incl_gotk_handled
433       ForStats.incl_gotk_handled ←ForLoop.incl_gotk_handled
```

Zusätzlich wird in [4] beschrieben wie Webservices aus solchen Spezifikationen erzeugt werden. In der vorliegenden Arbeit wird dieser Code hier nicht aufgeführt. Die Codegenerierung wird in [4, 5] und [3] nicht im Detail beschrieben. Anhand der in den Arbeiten aufgeführten Beispielen, kann jedoch davon ausgegangen werden, dass eine vergleichbar einfache Texttransformation mit Verwendung von Typinformationen stattfindet. Eine Transformation an dieser Stelle wäre somit rein spekulativ und würde ggf. die Transformation der eigentlichen Arbeit in [5] nicht ausreichend würdigen.

In Quelltext E.3 wird das Kopieren nach Unten (Muster 4) angewendet, dass auf nahezu allen Pfaden von der Wurzel (bzw. dem Symbol mit dem zu kopierenden Attribut) ein Weg zum Ziel existiert. Für den Vergleich in Kapitel 6 wurde jedoch der Wert gewählt für ausschließlich in der gegebenen abstrakten Syntax vorhandenen Pfaden. Die Anzahl an Zeilen dieser „optimierten“ Variante beträgt immernoch 333.

```
1   rule Program ::= Stats
2   attr Stats.varsIn ← ∅
3        Program.vars ← Stats.varsOut
4   rule Stats ::= Stats Stat
5   attr Stats_2.varsIn ←Stats_1.varsIn
6        Stat.varsIn ← Stats_2.varsOut
7        Stats_1.varsOut ←Stat.varsOut
8
9   rule Stats ::= ε
10  attr Stats.varsOut ← Stats.varsIn
11
12  rule Stat ::= Sequence
13  attr Sequence.varsIn ← Stat.varsIn
14       Stat.varsOut ← Sequence.varsOut
15
16  rule Stat ::= Composition
17  attr Composition.varsIn ← Stat.varsIn
```

```
18        Stat.varsOut ← Composition.varsOut
19
20    rule Stat ::= Conditional
21    attr Conditional.varsIn ← Stat.varsIn
22         Stat.varsOut ← Conditional.varsOut
23
24    rule Stat ::= VarDef
25    attr VarDef.varsIn ← Stat.varsIn
26         Stat.varsOut ← VarDef.varsOut
27
28    rule Sequence ::= VarRef ComputeRef Stat Stats
29    attr VarRef.varsIn ← Sequence.varsIn
30         ComputeRef.varsIn ← VarRef.varsOut
31         Stat.varsIn ← ComputeRef.varsOut
32         Stats.varsIn ← Stat.varsIn
33         Sequence.varsOut ← Stats.varsOut
34
35    rule Composition ::= Stat Stat
36    attr Stat1.varsIn ←Composition.varsIn
37         Stat2.varsIn ←Stat1.varsOut
38         Composition.varsOut ← Stat2.varsOut
39
40    rule Conditional ::= Expression Stat Stat
41    attr Expression.varsIn ← Conditional.varsIn
42         Stat1.varsIn ←Expression.varsOut
43         Stat2.varsIn ←Stat1.varsOut
44         Conditional.varsOut ← Stat2.varsOut
45
46    rule VarDef ::= VarDefId TypeReference Value Stat
47    attr VarDef.sym ← VarDefId.sym
48         VarDef.bind ← bindKey(VarDef.varsIn, VarDef.sym)
49         VarDefId.varsIn ← VarDef.varsIn ↞ VarDef.bind
50         TypeReference.varsIn ← VarDefId.varsOut
51         Value.varsIn ← TypeReference.varsOut
52         Stat.varsIn ← Value.varsOut
53         VarDef.bind:val ← Value.value
54         VarDef.varsOut ← Stat.varsOut ↞ VarDef.bind:val
55    cond VarDef.sym ∉ VarDef.varsIn  ⇒ error "already defined " ++ VarDef.sym
56
57    rule Value ::= VALUE
58    attr Value.varsOut ← Value.varsIn
59
60    rule VarDefId ::= ID
61    attr VarDefId.varsOut  ← VarDefId.varsIn
62
63    rule VarRef ::= ID
64    attr VarRef.bind ← bindInEnv(VarRef.varsIn, VarRef.sym)
65         VarRef.varsOut ← VarRef.varsIn ↞ VarRef.bind
```

Die Vervollständigung von Beispiel 6.2a ist in Quelltext E.2 dargestellt. Selbst ohne (die umfangreiche) Codegenerierung ist sichtbar, dass Muster gut geeignet sind, Spezifikationen von Attributgrammatiken zu verringern.

Für die abstrakte Syntax aus Abbildung 6.4a zeigt Quelltext E.4.

```
 1    rule Description ::= Declarations
 2    attr Declarations.requirementsIn ← ∅
 3         Declarations.env ← Declarations.requirementsOut
 4         Declarations.idnumIn ← 0
 5         Declarations.rq_depsIn ← []
 6         Description.rq_deps ← Declarations.rq_depsOut
 7         Declarations.gotResk_csv_name ←Declarations.gotResk_csv_nameOut
 8         Declarations.gotResk_csv_nameIn ←()
 9         Declarations.codeIn ← "Id  Deps  Author \n"
10         Description.code ← Declarations.codeOut
11
12    rule Declarations ::= Declarations Declaration
13    attr Declarations2.requirementsIn ←Declarations1.requirementsIn
14         Declaration.requirementsIn ← Declarations2.requirementsOut
15         Declarations1.requirementsOut ←Declaration.requirementsOut
16         Declarations2.env ←Declarations1.env
17         Declaration.env ← Declarations1.env
18         Declarations2.idnumIn ←Declarations1.idnumIn
19         Declaration.idnumIn ← Declarations2.idnumOut
20         Declarations1.idnumOut ←Declaration.idnumOut
21         Declarations2.rq_depsIn ←Declarations1.rq_depsIn
22         Declaration.rq_depsIn ← Declarations2.rq_depsOut
23         Declarations1.rq_depsOut ←Declaration.rq_depsOut
24         Declarations2.gotResk_csv_nameIn ←Declarations1.gotResk_csv_nameIn
25         Declaration.gotResk_csv_nameIn ←Declarations2.gotResk_csv_nameOut
26         Declarations1.gotResk_csv_nameOut ←Declaration.gotResk_csv_nameOut
27         Declarations2.gotResk_csv_name ←Declarations1.gotResk_csv_name
```

```
28        Declaration.gotResk_csv_name ←Declarations1.gotResk_csv_name
29        Declarations2.codeIn ←Declarations1.codeIn
30        Declaration.codeIn ← Declarations2.codeOut
31        Declarations1.codeOut ←Declaration.codeOut
32
33  rule Declarations ::= ε
34  attr Declarations.requirementsOut ← Declarations.requirementsIn
35       Declarations.idnumOut ← Declarations.idnumIn
36       Declarations.rq_depsOut ← Declaration.rq_depsIn
37       Declarations.gotResk_csv_nameOut ←Declarations.gotResk_csv_nameIn
38       Declarations.codeOut ← Declarations.codeIn
39
40  rule Declaration ::= RootStat
41  attr RootStat.requirementsIn ← Declaration.requirementsIn
42       Declaration.requirementsOut ← RootStat.requirementsOut
43       RootStat.env ← Declaration.env
44       RootStat.idnumIn ← Declaration.idnumIn
45       Declaration.idnumOut ← RootStat.idnumOut
46       Declaration.bind ← RootStat.const_bind
47       RootStat.incl_bind ← Declaration.bind
48       RootStat.rq_depsIn ← Declaration.rq_depsIn
49       Declaration.rq_depsOut ← RootStat.rq_depsOut
50       RootStat.gotResk_csv_nameIn ←Declaration.gotResk_csv_nameIn
51       Declaration.gotResk_csv_nameOut ←RootStat.gotResk_csv_nameOut
52       RootStat.gotResk_csv_name ←Declaration.gotResk_csv_name
53       Declaration.const_csv_name ← RootStat.const_csv_name
54       Declaration.const_sym ← RootStat.const_sym
55       Declaration.bind:csv_code
56           ← Declaration.bind:csv_name ++ " " ++ Declaration.const_csv_name ++ " "
57           ++ Declaration.const_sym ++ " " ↞ Declaration.gotResk_csv_name
58       Declaration.codeOut ← (λ a,b ⇒ a ++ "\n" ++ b)(Declaration.codeIn, Declaration.bind:csv_code)
59
60  rule Declaration ::= RqDef
61  attr RqDef.requirementsIn ← Declaration.requirementsIn
62       Declaration.requirementsOut ← RqDef.requirementsOut
63       RqDef.env ← Declaration.env
64       RqDef.idnumIn ← Declaration.idnumIn
65       Declaration.idnumOut ← RqDef.idnumOut
66       Declaration.bind ← RqDef.const_bind
67       RqDef.incl_bind ← Declaration.bind
68       RqDef.rq_depsIn ← Declaration.rq_depsIn
69       Declaration.rq_depsOut ← RqDef.rq_depsOut
70       RqDef.gotResk_csv_nameIn ←Declaration.gotResk_csv_nameIn
71       Declaration.gotResk_csv_nameOut ←RqDef.gotResk_csv_nameOut
72       RqDef.gotResk_csv_name ←Declaration.gotResk_csv_name
73       Declaration.const_csv_name ← RqDef.const_csv_name
74       Declaration.const_sym ← RqDef.const_sym
75       Declaration.bind:csv_code
76           ← Declaration.bind:csv_name ++ " " ++ Declaration.const_csv_name ++ " "
77           ++ Declaration.const_sym ++ " " ↞ Declaration.gotResk_csv_name
78       Declaration.codeOut ← (λ a,b ⇒ a ++ "\n" ++ b)(Declaration.codeIn, Declaration.bind:csv_code)
79
80  rule RootStat ::= RqDefId RqStats
81  attr RqDefId.requirementsIn ← RootStat.requirementsIn
82       RqStats.requirementsIn ← RqDefId.requirementsOut
83       RootStat.requirementsOut ← RqStats.requirementsOut
84       RqStats.env ← RootStat.env
85       RqDefId.idnumIn ← RootStat.idnumIn
86       RqStats.idnumIn ← RqDefId.idnumOut
87       RootStat.idnumOut ← RqStats.idnumOut
88       RqDefId.is_root ← true
89       RootStat.const_bind ← RqDefId.const_bind
90       RqStats.incl_bind ← RootStat.incl_bind
91       RqDefId.rq_depsIn ← RootStat.rq_depsIn
92       RqStats.rq_depsIn ← RqDefId.rq_depsOut
93       RootStat.rq_depsOut ← RqStats.rq_depsOut
94       RqDefId.gotResk_csv_nameIn ←RootStat.gotResk_csv_nameIn
95       RqStats.gotResk_csv_nameIn ←RqDefId.gotResk_csv_nameOut
96       RootStat.gotResk_csv_nameOut ←RqStats.gotResk_csv_nameOut
97       RqStats.gotResk_csv_name ←RootStat.gotResk_csv_name
98       RootStat.const_csv_name ← RqStats.const_csv_name
99       RootStat.const_sym ← RqStats.const_sym
100
101 rule RqDef ::= RqDefId RqStats
102 attr RqDefId.requirementsIn ← RqDef.requirementsIn
103      RqStats.requirementsIn ← RqDefId.requirementsOut
104      RqDef.requirementsOut ← RqStats.requirementsOut
105      RqStats.env ← RqDef.env
106      RqDefId.idnumIn ← RqDef.idnumIn
107      RqStats.idnumIn ← RqDefId.idnumOut
108      RqDef.idnumOut ← RqStats.idnumOut
109      RqDefId.is_root ← false
110      RqDef.const_bind ← RqDefId.const_bind
111      RqStats.incl_bind ← RqDef.incl_bind
112      RqDefId.rq_depsIn ← RqDef.rq_depsIn
```

```
113       RqStats.rq_depsIn ← RqDefId.rq_depsOut
114       RqDef.rq_depsOut ← RqStats.rq_depsOut
115       RqDefId.gotResk_csv_nameIn ←RqDef.gotResk_csv_nameIn
116       RqStats.gotResk_csv_nameIn ←RqDefId.gotResk_csv_nameOut
117       RqDef.gotResk_csv_nameOut ←RqStats.gotResk_csv_nameOut
118       RqStats.gotResk_csv_name ←RqDef.gotResk_csv_name
119       RqDef.const_csv_name ← RqStats.const_csv_name
120       RqDef.const_sym ← RqStats.const_sym
121
122  rule RqStats ::= RqStats RqStat
123  attr RqStats_2.requirementsIn ←RqStats_1.requirementsIn
124       RqStat.requirementsIn ← RqStats_2.requirementsOut
125       RqStats_1.requirementsOut ←RqStat.requirementsOut
126       RqStats_2.env ←RqStats_1.env
127       RqStat.env ← RqStats_1.env
128       RqStats_2.idnumIn ←RqStats_1.idnumIn
129       RqStat.idnumIn ← RqStats_2.idnumOut
130       RqStat_1.idnumOut ←RqStat.idnumOut
131       RqStats_2.incl_bind ←RqStats_1.incl_bind
132       RqStat.incl_bind ← RqStats_1.incl_bind
133       RqStats_2.rq_depsIn ←RqStats_1.rq_depsIn
134       RqStat.rq_depsIn ← RqStats_2.rq_depsOut
135       RqStat_1.rq_depsOut ←RqStat.rq_depsOut
136       RqStats_2.gotResk_csv_nameIn ←RqStats_1.gotResk_csv_nameIn
137       RqStat.gotResk_csv_nameIn ←RqStats_2.gotResk_csv_nameOut
138       RqStats_1.gotResk_csv_nameOut ←RqStat.gotResk_csv_nameOut
139       RqStats_2.gotResk_csv_name ←RqStats_1.gotResk_csv_name
140       RqStat.gotResk_csv_name ←RqStats_1.gotResk_csv_name
141       RqStats_1.const_csv_name ←append_with_comma(RqStats_2.const_csv_name, RqStat.const_csv_name)
142       RqStats_1.const_sym← append_with_comma(RqStats_2.const_sym, RqStat.const_sym)
143
144  rule RqStats ::= RqStat
145  attr RqStat.requirementsIn ← RqStats.requirementsIn
146       RqStats.requirementsOut ← RqStat.requirementsOut
147       RqStat.env ← RqStats.env
148       RqStat.idnumIn ← RqStats.idnumIn
149       RqStats.idnumOut ← RqStat.idnumOut
150       RqStat.incl_bind ← RqStats.incl_bind
151       RqStat.rq_depsIn ← RqStats.rq_depsIn
152       RqStats.rq_depsOut ← RqStat.rq_depsOut
153       RqStat.gotResk_csv_nameIn ←RqStats.gotResk_csv_nameIn
154       RqStats.gotResk_csv_nameOut ←RqStat.gotResk_csv_nameOut
155       RqStat.gotResk_csv_name ←RqStats.gotResk_csv_name
156       RqStats.const_csv_name ← RqStat.const_csv_name
157       RqStats.const_sym ← RqStat.const_sym
158
159  rule RqStat ::= Dependencies
160  attr Dependencies.requirementsIn ← RqStat.requirementsIn
161       RqStat.requirementsOut ← Dependencies.requirementsOut
162       Dependencies.env ← RqStat.env
163       Dependencies.idnumIn ← RqStat.idnumIn
164       RqStat.idnumOut ← Dependencies.idnumOut
165       Dependencies.incl_bind ← RqStat.incl_bind
166       Dependencies.rq_depsIn ← RqStat.rq_depsIn
167       RqStat.rq_depsOut ← Dependencies.rq_depsOut
168       Dependencies.gotResk_csv_nameIn ←RqStat.gotResk_csv_nameIn
169       RqStat.gotResk_csv_nameOut ←Dependencies.gotResk_csv_nameOut
170       Dependencies.gotResk_csv_name ←RqStat.gotResk_csv_name
171       RqStat.const_csv_name ← Dependencies.const_csv_name
172       RqStat.const_sym ← ""
173
174  rule RqStat ::= Author
175  attr Author.requirementsIn ← RqStat.requirementsIn
176       RqStat.requirementsOut ← Author.requirementsOut
177       Author.idnumIn ← RqStat.idnumIn
178       RqStat.idnumOut ← Author.idnumOut
179       Author.rq_depsIn ← RqStat.rq_depsIn
180       RqStat.rq_depsOut ← Author.rq_depsOut
181       Author.gotResk_csv_nameIn ←RqStat.gotResk_csv_nameIn
182       RqStat.gotResk_csv_nameOut ←Author.gotResk_csv_nameOut
183       RqStat.const_csv_name ← ""
184       RqStat.const_sym ← Author.const_sym
185
186  rule RqStat ::= Text
187  attr Text.requirementsIn ← RqStat.requirementsIn
188       RqStat.requirementsOut ← Text.requirementsOut
189       Text.idnumIn ← RqStat.idnumIn
190       RqStat.idnumOut ← Text.idnumOut
191       Text.rq_depsIn ← RqStat.rq_depsIn
192       RqStat.rq_depsOut ← Text.rq_depsOut
193       Text.gotResk_csv_nameIn ←RqStat.gotResk_csv_nameIn
194       RqStat.gotResk_csv_nameOut ←Text.gotResk_csv_nameOut
195       RqStat.const_csv_name ← ""
196       RqStat.const_sym ← ""
197
```

```
198  rule Dependencies ::= Dependencies Dependency
199  attr Dependencies_2.requirementsIn ←Dependencies_1.requirementsIn
200       Dependency.requirementsIn ← Dependencies_2.requirementsOut
201       Dependencies_1.requirementsOut ←Dependency.requirementsOut
202       Dependencies_2.env ←Dependencies_1.env
203       Dependency.env ← Dependencies_1.env
204       Dependencies_2.idnumIn ←Dependencies_1.idnumIn
205       Dependency.idnumIn ← Dependencies_2.idnumOut
206       Dependencies_1.idnumOut ←Dependency.idnumOut
207       Dependencies_2.incl_bind ←Dependencies_1.incl_bind
208       Dependency.incl_bind ← Dependencies_1.incl_bind
209       Dependencies_2.rq_depsIn ←Dependencies_1.rq_depsIn
210       Dependency.rq_depsIn ← Dependencies_2.rq_depsOut
211       Dependencies_1.rq_depsOut ←Dependency.rq_depsOut
212       Dependencies_2.gotResk_csv_nameIn ←Dependencies_1.gotResk_csv_nameIn
213       Dependency.gotResk_csv_nameIn ←Dependencies_2.gotResk_csv_nameOut
214       Dependencies_1.gotResk_csv_nameOut ←Dependency.gotResk_csv_nameOut
215       Dependencies_2.gotResk_csv_name ←Dependencies_1.gotResk_csv_name
216       Dependency.gotResk_csv_name ←Dependencies_1.gotResk_csv_name
217       Dependencies_1.const_csv_name
218           ← append_with_comma(Dependencies_2.const_csv_name, Dependency.const_csv_name)
219
220  rule Dependencies ::= Dependency
221  attr Dependency.requirementsIn ← Dependencies.requirementsIn
222       Dependencies.requirementsOut ← Dependency.requirementsOut
223       Dependency.env ← Dependencies.env
224       Dependency.idnumIn ← Dependencies.idnumIn
225       Dependencies.idnumOut ← Dependency.idnumOut
226       Dependency.incl_bind ← Dependencies.incl_bind
227       Dependency.rq_depsIn ← Dependencies.rq_depsIn
228       Dependencies.rq_depsOut ← Dependency.rq_depsOut
229       Dependency.gotResk_csv_nameIn ←Dependencies.gotResk_csv_nameIn
230       Dependencies.gotResk_csv_nameOut ←Dependency.gotResk_csv_nameOut
231       Dependency.gotResk_csv_name ←Dependencies.gotResk_csv_name
232       Dependencies.const_csv_name ← Dependency.const_csv_name
233
234  rule Dependency ::= RqUseId
235  attr RqUseId.requirementsIn ← Dependency.requirementsIn
236       Dependency.requirementsOut ← RqUseId.requirementsOut
237       RqUseId.env ← Dependency.env
238       RqUseId.idnumIn ← Dependency.idnumIn
239       Dependency.idnumOut ← RqUseId.idnumOut
240       RqUseId.incl_bind ← Dependency.incl_bind
241       RqUseId.rq_depsIn ← Dependency.rq_depsIn
242       Dependency.rq_depsOut ← RqUseId.rq_depsOut
243       RqUseId.gotResk_csv_nameIn ←Dependency.gotResk_csv_nameIn
244       Dependency.gotResk_csv_nameOut ←RqUseId.gotResk_csv_nameOut
245       RqUseId.gotResk_csv_name ←Dependency.gotResk_csv_name
246       Dependency.const_csv_name ← RqUseId.const_csv_name
247
248  rule Text ::= STRING
249  attr Text.requirementsOut ← Text.requirementsIn
250       Text.idnumOut ← Text.idnumIn
251       Text.rq_depsOut ← Text.rq_depsIn
252       Text.gotResk_csv_nameOut ←Text.gotResk_csv_nameIn
253
254  rule RqDefId ::= ID
255  attr RqDefId.bind ← bindKey(RqDefId.requirementsIn, RqDefId.sym)
256       RqDefId.bind:is_root ← RqDefId.is_root ↞ RqDefId.bind
257       RqDefId.requirementsOut ← RqDefId.requirementsIn ↞ RqDefId.bind:is_root
258       RqDefId.idnum ← RqDefId.idnumIn + 1
259       RqDefId.idnumOut ← RqDefId.idnumIn + 1
260       RqDefId.const_bind ← RqDefId.bind
261       RqDefId.rq_depsOut ← RqDefId.rq_depsIn
262       RqDefId.csv_name ← "rq_" ++ RqDefId.idnum
263       RqDefId.bind:csv_name ← RqDefId.csv_name
264       RqDefId.gotResk_csv_nameOut ←RqDefId.gotResk_csv_nameIn ↞RqDefId.bind:csv_name
265  cond RqDefId.sym ∉ RqDefId.requirementsIn
266           ⇒ error "already defined: " ++ RqDefId.sym
267
268  rule RqUseId ::= ID
269  attr RqUseId.requirementsOut ← RqUseId.requirementsIn
270       RqUseId.bind ← bindingInEnv(RqUseId.env, RqUseId.sym)
271       RqUseId.idnumOut ← RqUseId.idnumIn
272       RqUseId.t_1 ←RqUseId.incl_bind
273       RqUseId.t_2 ←(RqUseId.t_1, RqUseId.bind)
274       RqUseId.rq_depsOut ← RqUseId.rq_depsIn ++ [RqUseId.t_2]
275       RqUseId.csv_name ← RqUseId.bind:csv_name ↞ RqUseId.gotResk_csv_name
276       RqUseId.gotResk_csv_nameOut ←gotResk_csv_nameIn
277       RqUseId.const_csv_name ← id(RqUseId.csv_name)
278  cond RqUseId.sym ∈ RqUseId.env ⇒error "unknown Reference " ++ RqUseId.sym
279
280  rule Author ::= ID
281  attr Author.requirementsOut ← Author.requirementsIn
282       Author.idnumOut ← Author.idnumIn
```

```
283       Author.rq_depsOut ← Author.rq_depsIn
284       Author.gotResk_csv_nameOut ←Author.gotResk_csv_nameIn
285       Author.const_sym ← id(Author.sym)
```

Quelltext E.4 – Semantik der Anforderungsanalysesprache mit Generierung von CSV - Ausgabe

In Quelltext E.4 wird nicht für jeden Aspekt der Generierung jedes Mal eine neue Attributgrammatik erstellt. Wäre dies der Fall, dann würde für jeden Aspekt, d. h. für jedes der Beispiele 6.4, 6.7a, Beispiel 6.8 und die Konstruktion der Äbhängigkeiten aus Abschnitt 6.2 jeweils eine Attributgrammatik erstellt und diese dann aneinander gereiht werden. Somit wären viele Produktionen wiederholt. Unter dieser Betrachtung sind die resultierenden Attributgrammatiken somit bereits kompakter. Bei der manuellen Entwicklung von Attributgrammatiken, kann jedoch so vorgegangen werden um diese Aspekte, d. h. die geprüften semantischen Eigenschaften, herauszustellen.

Ebenso werden in den gewählten Beispielen nicht rein maschinell alle möglichen Kettenattributierungen bis zu den Terminalen durchgeführt, sondern nur bis zu der tiefsten notwendigen Ebene. Wenngleich bspw. in der Regel `Text` ::= `STRING` das Attribut $\texttt{gotResk}_{\texttt{csv\_name}}$ in ererbter und synthetisierter Variante zur Kettenattributierung keine Relevanz hat, so ist jedoch auf selber Ebene, d. h. Nichtterminale, die direkt ableitbar aus demselben Nichtterminal wie `Text` sind, dieses Attribut semantisch relevant.

Nicht aufgeführt in den Beispielen der Arbeit ist folgender Quelltext, der für die Codegenerierung der Sprache aus [29] in dieser Arbeit wesentlich ist:

```
1  rule Transition ::= EventRef StateRef StateRef TransitionDef
2  attr Transition.fromstate_id ← StateRef1.id
3       Transition.tostate_id ← StateRef2.id
4  store_load StateDef.bind:idnum ← StateDef.idnum
5       through StateRef.id
```

Die resultierende Attributgrammatik ohne etwas Optimierungen zur Vereinfachung der Spezifikation, wie dies in Quelltext E.4 geschehen ist, zeigt Quelltext E.5.

```
 1  ex:liverobots
 2
 3
 4  rule Program ::= StateMachine
 5  attr StateMachine.machinesIn ← ∅
 6       StateMachines.machines ← StateMachine.machinesOut
 7
 8  rule StateMachine ::= MachineDef MachineStats
 9  attr MachineDef.machinesIn ← StateMachines.machinesIn
10       MachineStats.machinesIn ← MachineDef.machinesOut
11       StateMachine.machinesOut ← MachineStats.machinesOut
12       MachineDef.machines ← StateMachine.machines
13       MachineStats.machines ← StateMachine.machines
14       StateMachine.bind ← MachineDef.bind
15
16  rule MachineStats ::= MachineStats MachineStat
17  attr MachineStats2.machinesIn ←MachineStats1.machinesIn
18       MachineStat.machinesIn ← MachineStats2.machinesOut
19       MachineStats1.machinesOut ←MachineStat.machinesOut
20       MachineStats2.machines ←MachineStats1.machines
21       MachineStat.machines ← MachineStats1.machines
22
23  rule MachineStats ::= ε
24  attr MachineStats.machinesOut ← MachineStats.machinesIn
25
26  rule MachineStat ::= StateDef
27  attr StateDef.machinesIn ← MachineStat.machinesIn
28       MachineStat.machinesOut ← StateDef.machinesOut
29       StateDef.machines ← MachineStat.machines
30
31  rule MachineStat ::= Transition
32  attr Transition.machinesIn ← MachineStat.machinesIn
33       MachineStat.machinesOut ← Transition.machinesOut
34       Transition.machines ← MachineStat.machines
35
36  rule MachineStat ::= EventDef
37  attr EventDef.machinesIn ← MachineStat.machinesIn
38       MachineStat.machinesOut ← EventDef.machinesOut
39       EventDef.machines ← MachineStat.machines
40
41  rule MachineStat ::= Timeout
```

```
42  attr Timeout.machinesIn ← MachineStat.machinesIn
43       MachineStat.machinesOut ← Timeout.machinesOut
44       Timeout.machines ← MachineStat.machines
45
46  rule MachineStat ::= VarDef
47  attr VarDef.machinesIn ← MachineStat.machinesIn
48       MachineStat.machinesOut ← VarDef.machinesOut
49       VarDef.machines ← MachineStat.machines
50
51  rule StateDef ::= StateDefId StateStats
52  attr StateDefId.machinesIn ← StateDef.machinesIn
53       StateStats.machinesIn ← newscope(StateDefId.machinesOut)
54       StateStats.machines ← StateStats.machinesOut
55       StateDef.bind ← StateDefId.bind
56       StateDef.bind:machines ← StateStats.machinesOut
57       StateDef.machinesOut ← StateDefId.machinesOut
58
59  rule StateStats ::= StateStats StateStat
60  attr StateStats_2.machinesIn ←StateStats_1.machinesIn
61       StatStat.machinesIn ← StatStats_2.machinesOut
62       StateStats_1.machinesOut ←StatStat.machinesOut
63       StateStats_2.machines ←StatStats_1.machines
64       StatStat.machines ← StatStats_1.machines
65
66  rule StateStats ::= ε
67  attr StateStats.machinesOut ← StateStats.machinesIn
68
69  rule StateStat ::= Entry
70  attr Entry.machinesIn ← StateStat.machinesIn
71       StateStat.machinesOut ← Entry.machinesOut
72       Entry.machines ← StateStat.machines
73
74  rule StateStat ::= Running
75  attr Running.machinesIn ← StateStat.machinesIn
76       StateStat.machinesOut ← Running.machinesOut
77       Running.machines ← StateStat.machines
78
79  rule StateStat ::= StateMachine
80  attr StateMachine.machinesIn ← StateStat.machinesIn
81       StateStat.machinesOut ← StateMachine.machinesOut
82       StateMachine.machines ← StateStat.machines
83
84  rule Transition ::= EventRef StateRef StateRef TransitionDef
85  attr EventRef.machinesIn ← Transition.machinesIn
86       StateRef_1.machinesIn ←EventRef.machinesOut
87       StateRef_2.machinesIn ←StateRef_1.machinesOut
88       TransitionDef.machinesIn ← StateRef_2.machinesOut
89       Transition.machinesOut ← TransitionDef.machinesOut
90
91  rule Entry ::= SmallCode
92  attr Entry.machinesOut ← Entry.machinesIn
93
94  rule Running ::= SmallCode
95  attr Running.machinesOut ← Running.machinesIn
96
97  rule EventDef ::= EventDefId SmallCode
98  attr EventDefId.machinesIn ← EventDef.machinesIn
99       EventDef.machinesOut ← EventDefId.machinesIn
100
101 rule Timeout ::= Number StateRef StateRef
102 attr StateRef_1.machinesIn ←Timeout.machinesIn
103      StateRef_2.machinesIn ←StateRef_1.machinesOut
104      TimeOut.machinesOut ← StateRefe_2.machinesOut
105
106 rule VarDef ::= VarDefId SmallCode
107 attr VarDefId.machinesIn ← VarDef.machinesIn
108      VarDef.machinesOut ← VarDefId.machinesOut
109
110 rule MachineDef ::= ID
111 attr MachineDef.bind ← bindKey(MachineDef.machinesIn, MachineDef.sym)
112      MachineDef.machinesOut ← MachineDef.machinesIn ↞ MachineDef.bind
113 cond MachineDef.sym ∉ MachineDef.machinesIn  ⇒ error "already defined " ++ MachineDef.sym
114
115 rule StateDefId ::= ID
116 attr StateDefId.machinesOut ← StateDefId.machinesIn
117
118 rule EventDefId ::= ID
119 attr EventDefId.machinesOut ← EventDefId.machinesIn
120
121 rule TransitionDef ::= ID
122 attr TransitionDef.machinesOut ← TransitionDef.machinesIn
123
124 rule StateRef ::= ID
125 attr StateRef.machinesOut ← StateRef.machinesIn
126
```

```
127  rule EventRef ::= ID
128  attr EventRef.machinesOut ← EventRef.machinesIn
129
130
131  rule Program ::= StateMachine
132  attr StateMachine.statesIn ← ∅
133       StateMachines.states ← StateMachine.statesOut
134
135  rule StateMachine ::= MachineDef MachineStats
136  attr MachineDef.statesIn ← StateMachines.statesIn
137       MachineStats.statesIn ← newscope(MachineDef.statesOut)
138       StateMachine.statesOut ← MachineStats.statesOut
139       MachineDef.states ← StateMachine.states
140       MachineStats.states ← MachineStats.statesOut
141       MachineDef.bind:states ← MachineStats.statesOut
142
143  rule MachineStats ::= MachineStats MachineStat
144  attr MachineStats₂.statesIn ←MachineStats₁.statesIn
145       MachineStat.statesIn ← MachineStats₂.statesOut
146       MachineStats₁.statesOut ←MachineStat.statesOut
147       MachineStats₂.states ←MachineStats₁.states
148       MachineStat.states ← MachineStats₁.states
149
150  rule MachineStats ::= ε
151  attr MachineStats.statesOut ← MachineStats.statesIn
152
153  rule MachineStat ::= StateDef
154  attr StateDef.statesIn ← MachineStat.statesIn
155       MachineStat.statesOut ← StateDef.statesOut
156       StateDef.states ← MachineStat.states
157
158  rule MachineStat ::= Transition
159  attr Transition.statesIn ← MachineStat.statesIn
160       MachineStat.statesOut ← Transition.statesOut
161       Transition.states ← MachineStat.states
162
163  rule MachineStat ::= EventDef
164  attr EventDef.statesIn ← MachineStat.statesIn
165       MachineStat.statesOut ← EventDef.statesOut
166       EventDef.states ← MachineStat.states
167
168  rule MachineStat ::= Timeout
169  attr Timeout.statesIn ← MachineStat.statesIn
170       MachineStat.statesOut ← Timeout.statesOut
171       Timeout.states ← MachineStat.states
172
173  rule MachineStat ::= VarDef
174  attr VarDef.statesIn ← MachineStat.statesIn
175       MachineStat.statesOut ← VarDef.statesOut
176       VarDef.states ← MachineStat.states
177
178  rule StateDef ::= StateDefId StateStats
179  attr StateDefId.statesIn ← StateDef.statesIn
180       StateStats.statesIn ← StateDefId.statesOut
181       StateStats.states ← StateDef.states
182       StateDef.statesOut ← StateStats.statesOut
183
184  rule StateStats ::= StateStats StateStat
185  attr StateStats₂.statesIn ←StateStats₁.statesIn
186       StatStat.statesIn ← StatStats₂.statesOut
187       StateStats₁.statesOut ←StatStat.statesOut
188       StateStats₂.states ←StatStats₁.states
189       StatStat.states ← StatStats₁.states
190
191  rule StateStats ::= ε
192  attr StateStats.statesOut ← StateStats.statesIn
193
194  rule StateStat ::= Entry
195  attr Entry.statesIn ← StateStat.statesIn
196       StateStat.statesOut ← Entry.statesOut
197       Entry.states ← StateStat.states
198
199  rule StateStat ::= Running
200  attr Running.statesIn ← StateStat.statesIn
201       StateStat.statesOut ← Running.statesOut
202       Running.states ← StateStat.states
203
204  rule StateStat ::= StateMachine
205  attr StateMachine.statesIn ← StateStat.statesIn
206       StateStat.statesOut ← StateMachine.statesOut
207       StateMachine.states ← StateStat.states
208
209  rule Transition ::= EventRef StateRef StateRef TransitionDef
210  attr EventRef.statesIn ← Transition.statesIn
211       StateRef₁.statesIn ←EventRef.statesOut
```

```
212       StateRef2.statesIn ←StateRef1.statesOut
213       TransitionDef.statesIn ← StateRef2.statesOut
214       Transition.statesOut ← TransitionDef.statesOut
215       StateRef1.states ←Transition.states
216       StateRef2.states ←Transition.states
217
218  rule Entry ::= SmallCode
219  attr Entry.statesOut ← Entry.statesIn
220
221  rule Running ::= SmallCode
222  attr Running.statesOut ← Running.statesIn
223
224  rule EventDef ::= EventDefId SmallCode
225  attr EventDefId.statesIn ← EventDef.statesIn
226       EventDef.statesOut ← EventDefId.statesIn
227
228  rule Timeout ::= Number StateRef StateRef
229  attr StateRef1.statesIn ←Timeout.statesIn
230       StateRef2.statesIn ←StateRef1.statesOut
231       Timeout.statesOut ← StateRefe2.statesOut
232       StateRef1.states ←Timeout.states
233       StateRef2.states ←Timeout.states
234
235  rule VarDef ::= VarDefId SmallCode
236  attr VarDefId.statesIn ← VarDef.statesIn
237       VarDef.statesOut ← VarDefId.statesOut
238
239  rule MachineDef ::= ID
240  attr MachineDef.statesOut ← MachineDef.statesIn
241
242  rule StateDefId ::= ID
243  attr StateDefId.bind ← bindKey(StateDefId.statesIn, StateDefId.sym)
244       StateDefId.statesOut ← StateDefId.statesIn ↞ StateDefId.bind
245
246  rule EventDefId ::= ID
247  attr EventDefId.statesOut ← EventDefId.statesIn
248
249  rule TransitionDef ::= ID
250  attr TransitionDef.statesOut ← TransitionDef.statesIn
251
252  rule StateRef ::= ID
253  attr StateRef.statesOut ← StateRef.statesIn
254       StateRef.bind ← bindingInEnv(StateRef.states, StateRef.sym)
255  cond StateRef.sym ∈ StateRef.states  ⇒ error "unknown Reference " ++ StatRef.sym
256
257  rule EventRef ::= ID
258  attr EventRef.statesOut ← EventRef.statesIn
259
260
261  rule Program ::= StateMachine
262  attr StateMachine.eventsIn ← ∅
263       StateMachines.events ← StateMachine.eventsOut
264
265  rule StateMachine ::= MachineDef MachineStats
266  attr MachineDef.eventsIn ← StateMachines.eventsIn
267       MachineStats.eventsIn ← newscope(MachineDef.eventsOut)
268       StateMachine.eventsOut ← MachineStats.eventsOut
269       MachineDef.events ← StateMachine.events
270       MachineStats.events ← MachineStats.eventsOut
271       MachineDef.bind:events ← MachineStats.eventsOut
272
273  rule MachineStats ::= MachineStats MachineStat
274  attr MachineStats2.eventsIn ←MachineStats1.eventsIn
275       MachineStat.eventsIn ← MachineStats2.eventsOut
276       MachineStats1.eventsOut ←MachineStat.eventsOut
277       MachineStats2.events ←MachineStats1.events
278       MachineStat.events ← MachineStats1.events
279
280  rule MachineStats ::= ε
281  attr MachineStats.eventsOut ← MachineStats.eventsIn
282
283  rule MachineStat ::= StateDef
284  attr StateDef.eventsIn ← MachineStat.eventsIn
285       MachineStat.eventsOut ← StateDef.eventsOut
286       StateDef.events ← MachineStat.events
287
288  rule MachineStat ::= Transition
289  attr Transition.eventsIn ← MachineStat.eventsIn
290       MachineStat.eventsOut ← Transition.eventsOut
291       Transition.events ← MachineStat.events
292
293  rule MachineStat ::= EventDef
294  attr EventDef.eventsIn ← MachineStat.eventsIn
295       MachineStat.eventsOut ← EventDef.eventsOut
296       EventDef.events ← MachineStat.events
```

```
297
298  rule MachineStat ::= Timeout
299  attr Timeout.eventsIn ← MachineStat.eventsIn
300       MachineStat.eventsOut ← Timeout.eventsOut
301       Timeout.events ← MachineStat.events
302
303  rule MachineStat ::= VarDef
304  attr VarDef.eventsIn ← MachineStat.eventsIn
305       MachineStat.eventsOut ← VarDef.eventsOut
306       VarDef.events ← MachineStat.events
307
308  rule StateDef ::= StateDefId StateStats
309  attr StateDefId.eventsIn ← StateDef.eventsIn
310       StateStats.eventsIn ← StateDefId.eventsOut
311       StateStats.events ← StateDef.events
312       StateDef.eventsOut ← StateStats.eventsOut
313
314  rule StateStats ::= StateStats StateStat
315  attr StateStats_2.eventsIn ←StateStats_1.eventsIn
316       StatStat.eventsIn ← StatStats_2.eventsOut
317       StateStats_1.eventsOut ←StatStat.eventsOut
318       StateStats_2.events ←StatStats_1.events
319       StatStat.events ← StatStats_1.events
320
321  rule StateStats ::= ε
322  attr StateStats.eventsOut ← StateStats.eventsIn
323
324  rule StateStat ::= Entry
325  attr Entry.eventsIn ← StateStat.eventsIn
326       StateStat.eventsOut ← Entry.eventsOut
327       Entry.events ← StateStat.events
328
329  rule StateStat ::= Running
330  attr Running.eventsIn ← StateStat.eventsIn
331       StateStat.eventsOut ← Running.eventsOut
332       Running.events ← StateStat.events
333
334  rule StateStat ::= StateMachine
335  attr StateMachine.eventsIn ← StateStat.eventsIn
336       StateStat.eventsOut ← StateMachine.eventsOut
337       StateMachine.events ← StateStat.events
338
339  rule Transition ::= EventRef StateRef StateRef TransitionDef
340  attr EventRef.eventsIn ← Transition.eventsIn
341       StateRef_1.eventsIn ←EventRef.eventsOut
342       StateRef_2.eventsIn ←StateRef_1.eventsOut
343       TransitionDef.eventsIn ← StateRef_2.eventsOut
344       Transition.eventsOut ← TransitionDef.eventsOut
345       StateRef_1.events ←Transition.events
346       StateRef_2.events ←Transition.events
347
348  rule Entry ::= SmallCode
349  attr Entry.eventsOut ← Entry.eventsIn
350
351  rule Running ::= SmallCode
352  attr Running.eventsOut ← Running.eventsIn
353
354  rule EventDef ::= EventDefId SmallCode
355  attr EventDefId.eventsIn ← EventDef.eventsIn
356       EventDef.eventsOut ← EventDefId.eventsIn
357
358  rule Timeout ::= Number StateRef StateRef
359  attr StateRef_1.eventsIn ←Timeout.eventsIn
360       StateRef_2.eventsIn ←StateRef_1.eventsOut
361       Timeout.eventsOut ← StateRefe_2.eventsOut
362       StateRef_1.events ←Timeout.events
363       StateRef_2.events ←Timeout.events
364
365  rule VarDef ::= VarDefId SmallCode
366  attr VarDefId.eventsIn ← VarDef.eventsIn
367       VarDef.eventsOut ← VarDefId.eventsOut
368
369  rule MachineDef ::= ID
370  attr MachineDef.eventsOut ← MachineDef.eventsIn
371
372  rule StateDefId ::= ID
373  attr StateDefId.eventsOut ← StateDefId.eventsIn
374
375  rule EventDefId ::= ID
376  attr EventDefId.bind ← bindKey(EventDefId.eventsIn, EventDefId.sym)
377       EventDefId.eventsOut ← EventDefId.eventsIn ↞ EventDefId.bind
378  cond EventDefId.sym ∉ EventDefId.eventsIn  ⇒ error "Already Defined " ++ EventDefId.sym
379
380  rule TransitionDef ::= ID
381  attr TransitionDef.eventsOut ← TransitionDef.eventsIn
```

```
382
383  rule EventRef ::= ID
384  attr EventRef.eventsOut ← EventRef.eventsIn
385       EventRef.bind ← bindingInEnv(EventRef.events, EventRef.sym)
386  cond EventRef.sym ∈ EventRef.events  ⇒ error "unknown Reference " ++ EventRef.sym
387
388  rule StateRef ::= ID
389  attr StateRef.eventsOut ← StateRef.eventsIn
390
391
392  rule Program ::= StateMachine
393  attr StateMachine.transitionsIn ← ∅
394       StateMachines.transitions ← StateMachine.transitionsOut
395
396  rule StateMachine ::= MachineDef MachineStats
397  attr MachineDef.transitionsIn ← StateMachines.transitionsIn
398       MachineStats.transitionsIn ← newscope(MachineDef.transitionsOut)
399       StateMachine.transitionsOut ← MachineStats.transitionsOut
400       MachineDef.transitions ← StateMachine.transitions
401       MachineStats.transitions ← MachineStats.transitionsOut
402
403  rule MachineStats ::= MachineStats MachineStat
404  attr MachineStats2.transitionsIn ←MachineStats1.transitionsIn
405       MachineStat.transitionsIn ← MachineStats2.transitionsOut
406       MachineStats1.transitionsOut ←MachineStat.transitionsOut
407       MachineStats2.transitions ←MachineStats1.transitions
408       MachineStat.transitions ← MachineStats1.transitions
409
410  rule MachineStats ::= ε
411  attr MachineStats.transitionsOut ← MachineStats.transitionsIn
412
413  rule MachineStat ::= StateDef
414  attr StateDef.transitionsIn ← MachineStat.transitionsIn
415       MachineStat.transitionsOut ← StateDef.transitionsOut
416       StateDef.transitions ← MachineStat.transitions
417
418  rule MachineStat ::= Transition
419  attr Transition.transitionsIn ← MachineStat.transitionsIn
420       MachineStat.transitionsOut ← Transition.transitionsOut
421       Transition.transitions ← MachineStat.transitions
422
423  rule MachineStat ::= EventDef
424  attr EventDef.transitionsIn ← MachineStat.transitionsIn
425       MachineStat.transitionsOut ← EventDef.transitionsOut
426       EventDef.transitions ← MachineStat.transitions
427
428  rule MachineStat ::= Timeout
429  attr Timeout.transitionsIn ← MachineStat.transitionsIn
430       MachineStat.transitionsOut ← Timeout.transitionsOut
431       Timeout.transitions ← MachineStat.transitions
432
433  rule MachineStat ::= VarDef
434  attr VarDef.transitionsIn ← MachineStat.transitionsIn
435       MachineStat.transitionsOut ← VarDef.transitionsOut
436       VarDef.transitions ← MachineStat.transitions
437
438  rule StateDef ::= StateDefId StateStats
439  attr StateDefId.transitionsIn ← StateDef.transitionsIn
440       StateStats.transitionsIn ← StateDefId.transitionsOut
441       StateStats.transitions ← StateDef.transitions
442       StateDef.transitionsOut ← StateStats.transitionsOut
443
444  rule StateStats ::= StateStats StateStat
445  attr StateStats2.transitionsIn ←StateStats1.transitionsIn
446       StatStat.transitionsIn ← StatStats2.transitionsOut
447       StateStats1.transitionsOut ←StatStat.transitionsOut
448       StateStats2.transitions ←StatStats1.transitions
449       StatStat.transitions ← StatStats1.transitions
450
451  rule StateStats ::= ε
452  attr StateStats.transitionsOut ← StateStats.transitionsIn
453
454  rule StateStat ::= Entry
455  attr Entry.transitionsIn ← StateStat.transitionsIn
456       StateStat.transitionsOut ← Entry.transitionsOut
457       Entry.transitions ← StateStat.transitions
458
459  rule StateStat ::= Running
460  attr Running.transitionsIn ← StateStat.transitionsIn
461       StateStat.transitionsOut ← Running.transitionsOut
462       Running.transitions ← StateStat.transitions
463
464  rule StateStat ::= StateMachine
465  attr StateMachine.transitionsIn ← StateStat.transitionsIn
466       StateStat.transitionsOut ← StateMachine.transitionsOut
```

```
467       StateMachine.transitions ← StateStat.transitions
468
469   rule Transition ::= EventRef StateRef StateRef TransitionDef
470   attr EventRef.transitionsIn ← Transition.transitionsIn
471        StateRef1.transitionsIn ←EventRef.transitionsOut
472        StateRef2.transitionsIn ←StateRef1.transitionsOut
473        TransitionDef.transitionsIn ← StateRef2.transitionsOut
474        Transition.transitionsOut ← TransitionDef.transitionsOut
475        StateRef1.transitions ←Transition.transitions
476        StateRef2.transitions ←Transition.transitions
477
478   rule Entry ::= SmallCode
479   attr Entry.transitionsOut ← Entry.transitionsIn
480
481   rule Running ::= SmallCode
482   attr Running.transitionsOut ← Running.transitionsIn
483
484   rule EventDef ::= EventDefId SmallCode
485   attr EventDefId.transitionsIn ← EventDef.transitionsIn
486        EventDef.transitionsOut ← EventDefId.transitionsIn
487
488   rule Timeout ::= Number StateRef StateRef
489   attr StateRef1.transitionsIn ←Timeout.transitionsIn
490        StateRef2.transitionsIn ←StateRef1.transitionsOut
491        Timeout.transitionsOut ← StateRefe2.transitionsOut
492        StateRef1.transitions ←Timeout.transitions
493        StateRef2.transitions ←Timeout.transitions
494
495   rule VarDef ::= VarDefId SmallCode
496   attr VarDefId.transitionsIn ← VarDef.transitionsIn
497        VarDef.transitionsOut ← VarDefId.transitionsOut
498
499   rule MachineDef ::= ID
500   attr MachineDef.transitionsOut ← MachineDef.transitionsIn
501
502   rule StateDefId ::= ID
503   attr StateDefId.transitionsOut ← StateDefId.transitionsIn
504
505   rule EventDefId ::= ID
506   attr EventDefId.transitionsOut ← EventDefId.transitionsIn
507
508   rule TransitionDef ::= ID
509   attr TransitionDef.bind ← bindKey(TransitionDef.transitionsIn, TransitionDef.sym)
510        TransitionDef.transitionsOut ← TransitionDef.transitionsIn ↞ TransitionDef.bind
511   cond TransitionDef.sym ∉ TransitionDef.transitionsIn
512          ⇒ error "Already Defined " ++ TransitionDef.sym
513
514   rule EventRef ::= ID
515   attr EventRef.transitionsOut ← EventRef.transitionsIn
516
517   rule StateRef ::= ID
518   attr StateRef.transitionsOut ← StateRef.transitionsIn
519
520
521   rule Program ::= StateMachine
522   attr StateMachine.eventsNamesIn ← []
523        StateMachines.eventsNames ← StateMachine.eventsNamesOut
524
525   rule StateMachine ::= MachineDef MachineStats
526   attr MachineDef.eventsNamesIn ← StateMachines.eventsNamesIn
527        MachineStats.eventsNamesIn ← newscope(MachineDef.eventsNamesOut)
528        StateMachine.eventsNamesOut ← MachineStats.eventsNamesOut
529        MachineDef.eventsNames ← StateMachine.eventsNames
530        MachineStats.eventsNames ← MachineStats.eventsNamesOut
531
532
533   rule MachineStats ::= MachineStats MachineStat
534   attr MachineStats2.eventsNamesIn ←MachineStats1.eventsNamesIn
535        MachineStat.eventsNamesIn ← MachineStats2.eventsNamesOut
536        MachineStats1.eventsNamesOut ←MachineStat.eventsNamesOut
537        MachineStats2.eventsNames ←MachineStats1.eventsNames
538        MachineStat.eventsNames ← MachineStats1.eventsNames
539
540   rule MachineStats ::= ε
541   attr MachineStats.eventsNamesOut ← MachineStats.eventsNamesIn
542
543   rule MachineStat ::= StateDef
544   attr StateDef.eventsNamesIn ← MachineStat.eventsNamesIn
545        MachineStat.eventsNamesOut ← StateDef.eventsNamesOut
546        StateDef.eventsNames ← MachineStat.eventsNames
547
548   rule MachineStat ::= Transition
549   attr Transition.eventsNamesIn ← MachineStat.eventsNamesIn
550        MachineStat.eventsNamesOut ← Transition.eventsNamesOut
551        Transition.eventsNames ← MachineStat.eventsNames
```

```
552
553  rule MachineStat ::= EventDef
554  attr EventDef.eventsNamesIn ← MachineStat.eventsNamesIn
555       MachineStat.eventsNamesOut ← EventDef.eventsNamesOut
556       EventDef.eventsNames ← MachineStat.eventsNames
557
558  rule MachineStat ::= Timeout
559  attr Timeout.eventsNamesIn ← MachineStat.eventsNamesIn
560       MachineStat.eventsNamesOut ← Timeout.eventsNamesOut
561       Timeout.eventsNames ← MachineStat.eventsNames
562
563  rule MachineStat ::= VarDef
564  attr VarDef.eventsNamesIn ← MachineStat.eventsNamesIn
565       MachineStat.eventsNamesOut ← VarDef.eventsNamesOut
566       VarDef.eventsNames ← MachineStat.eventsNames
567
568  rule StateDef ::= StateDefId StateStats
569  attr StateDefId.eventsNamesIn ← StateDef.eventsNamesIn
570       StateStats.eventsNamesIn ← StateDefId.eventsNamesOut
571       StateStats.eventsNames ← StateDef.eventsNames
572       StateDef.eventsNamesOut ← StateStats.eventsNamesOut
573
574  rule StateStats ::= StateStats StateStat
575  attr StateStats2.eventsNamesIn ←StateStats1.eventsNamesIn
576       StatStat.eventsNamesIn ← StatStats2.eventsNamesOut
577       StateStats1.eventsNamesOut ←StatStat.eventsNamesOut
578       StateStats2.eventsNames ←StatStats1.eventsNames
579       StatStat.eventsNames ← StatStats1.eventsNames
580
581  rule StateStats ::= ε
582  attr StateStats.eventsNamesOut ← StateStats.eventsNamesIn
583
584  rule StateStat ::= Entry
585  attr Entry.eventsNamesIn ← StateStat.eventsNamesIn
586       StateStat.eventsNamesOut ← Entry.eventsNamesOut
587       Entry.eventsNames ← StateStat.eventsNames
588
589  rule StateStat ::= Running
590  attr Running.eventsNamesIn ← StateStat.eventsNamesIn
591       StateStat.eventsNamesOut ← Running.eventsNamesOut
592       Running.eventsNames ← StateStat.eventsNames
593
594  rule StateStat ::= StateMachine
595  attr StateMachine.eventsNamesIn ← StateStat.eventsNamesIn
596       StateStat.eventsNamesOut ← StateMachine.eventsNamesOut
597       StateMachine.eventsNames ← StateStat.eventsNames
598
599  rule Transition ::= EventRef StateRef StateRef TransitionDef
600  attr EventRef.eventsNamesIn ← Transition.eventsNamesIn
601       StateRef1.eventsNamesIn ←EventRef.eventsNamesOut
602       StateRef2.eventsNamesIn ←StateRef1.eventsNamesOut
603       TransitionDef.eventsNamesIn ← StateRef2.eventsNamesOut
604       Transition.eventsNamesOut ← TransitionDef.eventsNamesOut
605       StateRef1.eventsNames ←Transition.eventsNames
606       StateRef2.eventsNames ←Transition.eventsNames
607
608  rule Entry ::= SmallCode
609  attr Entry.eventsNamesOut ← Entry.eventsNamesIn
610
611  rule Running ::= SmallCode
612  attr Running.eventsNamesOut ← Running.eventsNamesIn
613
614  rule EventDef ::= EventDefId SmallCode
615  attr EventDefId.eventsNamesIn ← EventDef.eventsNamesIn
616       EventDef.eventsNamesOut ← EventDefId.eventsNamesIn
617       EventDefId.eventsNames ← EventDef.eventsNames
618
619  rule Timeout ::= Number StateRef StateRef
620  attr StateRef1.eventsNamesIn ←Timeout.eventsNamesIn
621       StateRef2.eventsNamesIn ←StateRef1.eventsNamesOut
622       Timeout.eventsNamesOut ← StateRefe2.eventsNamesOut
623       StateRef1.eventsNames ←Timeout.eventsNames
624       StateRef2.eventsNames ←Timeout.eventsNames
625
626  rule VarDef ::= VarDefId SmallCode
627  attr VarDefId.eventsNamesIn ← VarDef.eventsNamesIn
628       VarDef.eventsNamesOut ← VarDefId.eventsNamesOut
629       VarDefId.eventsNames ← VarDef.eventsNames
630
631  rule MachineDef ::= ID
632  attr MachineDef.eventsNamesOut ← MachineDef.eventsNamesIn
633
634  rule StateDefId ::= ID
635  attr StateDefId.eventsNamesOut ← StateDefId.eventsNamesIn
636
```

```
637  rule EventDefId ::= ID
638  attr EventDefId.eventsNamesOut ← EventDefId.eventsNamesIn
639
640  rule TransitionDef ::= ID
641  attr TransitionDef.eventsNamesOut ← TransitionDef.eventsNamesIn
642
643  rule EventRef ::= ID
644  attr EventRef.eventsNamesOut ← EventRef.eventsNamesIn ++ [EventRef.sym]
645
646  rule StateRef ::= ID
647  attr StateRef.eventsNamesOut ← StateRef.eventsNamesIn
648
649
650  rule Program ::= StateMachine
651  attr StateMachine.idnumIn ← []
652
653  rule StateMachine ::= MachineDef MachineStats
654  attr MachineDef.idnumIn ← StateMachines.idnumIn
655       MachineStats.idnumIn ← newscope(MachineDef.idnumOut)
656       StateMachine.idnumOut ← MachineStats.idnumOut
657
658  rule MachineStats ::= MachineStats MachineStat
659  attr MachineStats2.idnumIn ←MachineStats1.idnumIn
660       MachineStat.idnumIn ← MachineStats2.idnumOut
661       MachineStats1.idnumOut ←MachineStat.idnumOut
662
663  rule MachineStats ::= ε
664  attr MachineStats.idnumOut ← MachineStats.idnumIn
665
666  rule MachineStat ::= StateDef
667  attr StateDef.idnumIn ← MachineStat.idnumIn
668       MachineStat.idnumOut ← StateDef.idnumOut
669
670  rule MachineStat ::= Transition
671  attr Transition.idnumIn ← MachineStat.idnumIn
672       MachineStat.idnumOut ← Transition.idnumOut
673
674  rule MachineStat ::= EventDef
675  attr EventDef.idnumIn ← MachineStat.idnumIn
676       MachineStat.idnumOut ← EventDef.idnumOut
677
678  rule MachineStat ::= Timeout
679  attr Timeout.idnumIn ← MachineStat.idnumIn
680       MachineStat.idnumOut ← Timeout.idnumOut
681
682  rule MachineStat ::= VarDef
683  attr VarDef.idnumIn ← MachineStat.idnumIn
684       MachineStat.idnumOut ← VarDef.idnumOut
685
686  rule StateDef ::= StateDefId StateStats
687  attr StateDefId.idnumIn ← StateDef.idnumIn
688       StateStats.idnumIn ← StateDefId.idnumOut
689       StateDef.idnumOut ← StateStats.idnumOut + 1
690       StateDef.idnum ← StateStats.idnumOut + 1
691
692  rule StateStats ::= StateStats StateStat
693  attr StateStats2.idnumIn ←StateStats1.idnumIn
694       StatStat.idnumIn ← StatStats2.idnumOut
695       StateStats1.idnumOut ←StatStat.idnumOut
696       StateStats2.idnum ←StatStats1.idnum
697
698  rule StateStats ::= ε
699  attr StateStats.idnumOut ← StateStats.idnumIn
700
701  rule StateStat ::= Entry
702  attr Entry.idnumIn ← StateStat.idnumIn
703       StateStat.idnumOut ← Entry.idnumOut
704       Entry.idnum ← StateStat.idnum
705
706  rule StateStat ::= Running
707  attr Running.idnumIn ← StateStat.idnumIn
708       StateStat.idnumOut ← Running.idnumOut
709       Running.idnum ← StateStat.idnum
710
711  rule StateStat ::= StateMachine
712  attr StateMachine.idnumIn ← StateStat.idnumIn
713       StateStat.idnumOut ← StateMachine.idnumOut
714       StateMachine.idnum ← StateStat.idnum
715
716  rule Transition ::= EventRef StateRef StateRef TransitionDef
717  attr EventRef.idnumIn ← Transition.idnumIn
718       StateRef1.idnumIn ←EventRef.idnumOut
719       StateRef2.idnumIn ←StateRef1.idnumOut
720       TransitionDef.idnumIn ← StateRef2.idnumOut
721       Transition.idnumOut ← TransitionDef.idnumOut
```

```
722       StateRef1.idnum ←Transition.idnum
723       StateRef2.idnum ←Transition.idnum
724
725   rule Entry ::= SmallCode
726   attr Entry.idnumOut ← Entry.idnumIn
727
728   rule Running ::= SmallCode
729   attr Running.idnumOut ← Running.idnumIn
730
731   rule EventDef ::= EventDefId SmallCode
732   attr EventDefId.idnumIn ← EventDef.idnumIn
733       EventDef.idnumOut ← EventDefId.idnumIn
734       EventDefId.idnum ← EventDef.idnum
735
736   rule Timeout ::= Number StateRef StateRef
737   attr StateRef1.idnumIn ←Timeout.idnumIn
738       StateRef2.idnumIn ←StateRef1.idnumOut
739       Timeout.idnumOut ← StateRefe2.idnumOut
740       StateRef1.idnum ←Timeout.idnum
741       StateRef2.idnum ←Timeout.idnum
742
743   rule VarDef ::= VarDefId SmallCode
744   attr VarDefId.idnumIn ← VarDef.idnumIn
745       VarDef.idnumOut ← VarDefId.idnumOut
746       VarDefId.idnum ← VarDef.idnum
747
748   rule MachineDef ::= ID
749   attr MachineDef.idnumOut ← MachineDef.idnumIn
750
751   rule StateDefId ::= ID
752   attr StateDefId.idnumOut ← StateDefId.idnumIn
753
754   rule EventDefId ::= ID
755   attr EventDefId.idnumOut ← EventDefId.idnumIn + 1
756       EventDefId.idnum ← EventDefId.idnum + 1
757
758   rule TransitionDef ::= ID
759   attr TransitionDef.idnumOut ← TransitionDef.idnumIn + 1
760       TransitionDef.idnum ← TransitionDef.idnumIn + 1
761
762   rule EventRef ::= ID
763   attr EventRef.idnumOut ← EventRef.idnumIn
764
765   rule StateRef ::= ID
766   attr StateRef.idnumOut ← StateRef.idnumIn
767
768
769   rule Program ::= StateMachine
770   attr StateMachine.gotk_idnum In ←[]
771       StateMachines.gotk_idnum ←StateMachine.gotk_idnum Out
772
773   rule StateMachine ::= MachineDef MachineStats
774   attr MachineDef.gotk_idnum In ←StateMachines.gotk_idnum In
775       MachineStats.gotk_idnum In ←newscope(MachineDef.gotk_idnum Out)
776       StateMachine.gotk_idnum Out ←MachineStats.gotk_idnum Out
777       MachineDef.gotk_idnum ←StateMachine.gotk_idnum
778       MachineStats.gotk_idnum ←MachineStats.gotk_idnum Out
779
780
781   rule MachineStats ::= MachineStats MachineStat
782   attr MachineStats2.gotk_idnum In ←MachineStats1.gotk_idnum In
783       MachineStat.gotk_idnum In ←MachineStats2.gotk_idnum Out
784       MachineStats1.gotk_idnum Out ←MachineStat.gotk_idnum Out
785       MachineStats2.gotk_idnum ←MachineStats1.gotk_idnum
786       MachineStat.gotk_idnum ←MachineStats1.gotk_idnum
787
788   rule MachineStats ::= ε
789   attr MachineStats.gotk_idnum Out ←MachineStats.gotk_idnum In
790
791   rule MachineStat ::= StateDef
792   attr StateDef.gotk_idnum In ←MachineStat.gotk_idnum In
793       MachineStat.gotk_idnum Out ←StateDef.gotk_idnum Out
794       StateDef.gotk_idnum ←MachineStat.gotk_idnum
795
796   rule MachineStat ::= Transition
797   attr Transition.gotk_idnum In ←MachineStat.gotk_idnum In
798       MachineStat.gotk_idnum Out ←Transition.gotk_idnum Out
799       Transition.gotk_idnum ←MachineStat.gotk_idnum
800
801   rule MachineStat ::= EventDef
802   attr EventDef.gotk_idnum In ←MachineStat.gotk_idnum In
803       MachineStat.gotk_idnum Out ←EventDef.gotk_idnum Out
804       EventDef.gotk_idnum ←MachineStat.gotk_idnum
805
806   rule MachineStat ::= Timeout
```

```
807  attr Timeout.gotk_idnum In ←MachineStat.gotk_idnum In
808       MachineStat.gotk_idnum Out ←Timeout.gotk_idnum Out
809       Timeout.gotk_idnum ←MachineStat.gotk_idnum
810
811  rule MachineStat ::= VarDef
812  attr VarDef.gotk_idnum In ←MachineStat.gotk_idnum In
813       MachineStat.gotk_idnum Out ←VarDef.gotk_idnum Out
814       VarDef.gotk_idnum ←MachineStat.gotk_idnum
815
816  rule StateDef ::= StateDefId StateStats
817  attr StateDefId.gotk_idnum In ←StateDef.gotk_idnum In
818       StateStats.gotk_idnum In ←StateDefId.gotk_idnum Out
819       StateStats.gotk_idnum ←StateDef.gotk_idnum
820       StateDef.gotk_idnum Out ←StateStats.gotk_idnum Out
821
822  rule StateStats ::= StateStats StateStat
823  attr StateStats_2.gotk_idnum In ←StateStats_1.gotk_idnum In
824       StatStat.gotk_idnum In ←StatStats_2.gotk_idnum Out
825       StateStats_1.gotk_idnum Out ←StatStat.gotk_idnum Out
826       StateStats_2.gotk_idnum ←StatStats_1.gotk_idnum
827       StatStat.gotk_idnum ←StatStats_1.gotk_idnum
828
829  rule StateStats ::= ε
830  attr StateStats.gotk_idnum Out ←StateStats.gotk_idnum In
831
832  rule StateStat ::= Entry
833  attr Entry.gotk_idnum In ←StateStat.gotk_idnum In
834       StateStat.gotk_idnum Out ←Entry.gotk_idnum Out
835       Entry.gotk_idnum ←StateStat.gotk_idnum
836
837  rule StateStat ::= Running
838  attr Running.gotk_idnum In ←StateStat.gotk_idnum In
839       StateStat.gotk_idnum Out ←Running.gotk_idnum Out
840       Running.gotk_idnum ←StateStat.gotk_idnum
841
842  rule StateStat ::= StateMachine
843  attr StateMachine.gotk_idnum In ←StateStat.gotk_idnum In
844       StateStat.gotk_idnum Out ←StateMachine.gotk_idnum Out
845       StateMachine.gotk_idnum ←StateStat.gotk_idnum
846
847  rule Transition ::= EventRef StateRef StateRef TransitionDef
848  attr EventRef.gotk_idnum In ←Transition.gotk_idnum In
849       StateRef_1.gotk_idnum In ←EventRef.gotk_idnum Out
850       StateRef_2.gotk_idnum In ←StateRef_1.gotk_idnum Out
851       TransitionDef.gotk_idnum In ←StateRef_2.gotk_idnum Out
852       Transition.gotk_idnum Out ←TransitionDef.gotk_idnum Out
853       StateRef_1.gotk_idnum ←Transition.gotk_idnum
854       StateRef_2.gotk_idnum ←Transition.gotk_idnum
855       EventRef.gotk_idnum ←Transition.gotk_idnum
856
857  rule Entry ::= SmallCode
858  attr Entry.gotk_idnum Out ←Entry.gotk_idnum In
859
860  rule Running ::= SmallCode
861  attr Running.gotk_idnum Out ←Running.gotk_idnum In
862
863  rule EventDef ::= EventDefId SmallCode
864  attr EventDefId.gotk_idnum In ←EventDef.gotk_idnum In
865       EventDef.gotk_idnum Out ←EventDefId.gotk_idnum In
866       EventDefId.gotk_idnum ←EventDef.gotk_idnum
867
868  rule Timeout ::= Number StateRef StateRef
869  attr StateRef_1.gotk_idnum In ←Timeout.gotk_idnum In
870       StateRef_2.gotk_idnum In ←StateRef_1.gotk_idnum Out
871       Timeout.gotk_idnum Out ←StateRefe_2.gotk_idnum Out
872       StateRef_1.gotk_idnum ←Timeout.gotk_idnum
873       StateRef_2.gotk_idnum ←Timeout.gotk_idnum
874
875  rule VarDef ::= VarDefId SmallCode
876  attr VarDefId.gotk_idnum In ←VarDef.gotk_idnum In
877       VarDef.gotk_idnum Out ←VarDefId.gotk_idnum Out
878       VarDefId.gotk_idnum ←VarDef.gotk_idnum
879
880  rule MachineDef ::= ID
881  attr MachineDef.gotk_idnum Out ←MachineDef.gotk_idnum In
882
883  rule StateDefId ::= ID
884  attr StateDefId.gotk_idnum Out ←StateDefId.gotk_idnum In
885
886  rule EventDefId ::= ID
887  attr EventDefId.gotk_idnum Out ←EventDefId.gotk_idnum In ←EventDefId.bind:idnum
888       EventDefId.bind:idnum ← EventDefId.idnum
889
890  rule TransitionDef ::= ID
891  attr TransitionDef.gotk_idnum Out ←TransitionDef.gotk_idnum In
```

```
892
893  rule EventRef ::= ID
894  attr EventRef.gotk_idnumOut ←EventRef.gotk_idnumIn
895       EventRef.id ← EventRef.bind:idnum ↞ EventRef.gotk_idnum
896
897  rule StateRef ::= ID
898  attr StateRef.gotk_idnumOut ←StateRef.gotk_idnumIn
899
900
901  rule Program ::= StateMachine
902  attr StateMachine.gotk_2idnumIn ←[]
903       StateMachines.gotk_2idnum ←StateMachine.gotk_2idnumOut
904
905  rule StateMachine ::= MachineDef MachineStats
906  attr MachineDef.gotk_2idnumIn ←StateMachines.gotk_2idnumIn
907       MachineStats.gotk_2idnumIn ←newscope(MachineDef.gotk_2idnumOut)
908       StateMachine.gotk_2idnumOut ←MachineStats.gotk_2idnumOut
909       MachineDef.gotk_2idnum ←StateMachine.gotk_2idnum
910       MachineStats.gotk_2idnum ←MachineStats.gotk_2idnumOut
911
912
913  rule MachineStats ::= MachineStats MachineStat
914  attr MachineStats_2.gotk_2idnumIn ←MachineStats_1.gotk_2idnumIn
915       MachineStat.gotk_2idnumIn ←MachineStats_2.gotk_2idnumOut
916       MachineStats_1.gotk_2idnumOut ←MachineStat.gotk_2idnumOut
917       MachineStats_2.gotk_2idnum ←MachineStats_1.gotk_2idnum
918       MachineStat.gotk_2idnum ←MachineStats_1.gotk_2idnum
919
920  rule MachineStats ::= ε
921  attr MachineStats.gotk_2idnumOut ←MachineStats.gotk_2idnumIn
922
923  rule MachineStat ::= StateDef
924  attr StateDef.gotk_2idnumIn ←MachineStat.gotk_2idnumIn
925       MachineStat.gotk_2idnumOut ←StateDef.gotk_2idnumOut
926       StateDef.gotk_2idnum ←MachineStat.gotk_2idnum
927
928  rule MachineStat ::= Transition
929  attr Transition.gotk_2idnumIn ←MachineStat.gotk_2idnumIn
930       MachineStat.gotk_2idnumOut ←Transition.gotk_2idnumOut
931       Transition.gotk_2idnum ←MachineStat.gotk_2idnum
932
933  rule MachineStat ::= EventDef
934  attr EventDef.gotk_2idnumIn ←MachineStat.gotk_2idnumIn
935       MachineStat.gotk_2idnumOut ←EventDef.gotk_2idnumOut
936       EventDef.gotk_2idnum ←MachineStat.gotk_2idnum
937
938  rule MachineStat ::= Timeout
939  attr Timeout.gotk_2idnumIn ←MachineStat.gotk_2idnumIn
940       MachineStat.gotk_2idnumOut ←Timeout.gotk_2idnumOut
941       Timeout.gotk_2idnum ←MachineStat.gotk_2idnum
942
943  rule MachineStat ::= VarDef
944  attr VarDef.gotk_2idnumIn ←MachineStat.gotk_2idnumIn
945       MachineStat.gotk_2idnumOut ←VarDef.gotk_2idnumOut
946       VarDef.gotk_2idnum ←MachineStat.gotk_2idnum
947
948  rule StateDef ::= StateDefId StateStats
949  attr StateDefId.gotk_2idnumIn ←StateDef.gotk_2idnumIn
950       StateStats.gotk_2idnumIn ←StateDefId.gotk_2idnumOut
951       StateStats.gotk_2idnum ←StateDef.gotk_2idnum
952       StateDef.gotk_2idnumOut ←StateStats.gotk_2idnumOut ↞StateDef.bind:idnum
953       StateDef.bind:idnum ← StateDef.idnum
954
955  rule StateStats ::= StateStats StateStat
956  attr StateStats_2.gotk_2idnumIn ←StateStats_1.gotk_2idnumIn
957       StatStat.gotk_2idnumIn ←StatStats_2.gotk_2idnumOut
958       StateStats_1.gotk_2idnumOut ←StatStat.gotk_2idnumOut
959       StateStats_2.gotk_2idnum ←StatStats_1.gotk_2idnum
960       StatStat.gotk_2idnum ←StatStats_1.gotk_2idnum
961
962  rule StateStats ::= ε
963  attr StateStats.gotk_2idnumOut ←StateStats.gotk_2idnumIn
964
965  rule StateStat ::= Entry
966  attr Entry.gotk_2idnumIn ←StateStat.gotk_2idnumIn
967       StateStat.gotk_2idnumOut ←Entry.gotk_2idnumOut
968       Entry.gotk_2idnum ←StateStat.gotk_2idnum
969
970  rule StateStat ::= Running
971  attr Running.gotk_2idnumIn ←StateStat.gotk_2idnumIn
972       StateStat.gotk_2idnumOut ←Running.gotk_2idnumOut
973       Running.gotk_2idnum ←StateStat.gotk_2idnum
974
975  rule StateStat ::= StateMachine
976  attr StateMachine.gotk_2idnumIn ←StateStat.gotk_2idnumIn
```

```
977       StateStat.gotk2idnumOut ←StateMachine.gotk2idnumOut
978       StateMachine.gotk2idnum ←StateStat.gotk2idnum
979
980   rule Transition ::= EventRef StateRef StateRef TransitionDef
981   attr EventRef.gotk2idnumIn ←Transition.gotk2idnumIn
982        StateRef1.gotk2idnumIn ←EventRef.gotk2idnumOut
983        StateRef2.gotk2idnumIn ←StateRef1.gotk2idnumOut
984        TransitionDef.gotk2idnumIn ←StateRef2.gotk2idnumOut
985        Transition.gotk2idnumOut ←TransitionDef.gotk2idnumOut
986        StateRef1.gotk2idnum ←Transition.gotk2idnum
987        StateRef2.gotk2idnum ←Transition.gotk2idnum
988        EventRef.gotk2idnum ←Transition.gotk2idnum
989
990   rule Entry ::= SmallCode
991   attr Entry.gotk2idnumOut ←Entry.gotk2idnumIn
992
993   rule Running ::= SmallCode
994   attr Running.gotk2idnumOut ←Running.gotk2idnumIn
995
996   rule EventDef ::= EventDefId SmallCode
997   attr EventDefId.gotk2idnumIn ←EventDef.gotk2idnumIn
998        EventDef.gotk2idnumOut ←EventDefId.gotk2idnumIn
999        EventDefId.gotk2idnum ←EventDef.gotk2idnum
1000
1001  rule Timeout ::= Number StateRef StateRef
1002  attr StateRef1.gotk2idnumIn ←Timeout.gotk2idnumIn
1003       StateRef2.gotk2idnumIn ←StateRef1.gotk2idnumOut
1004       Timeout.gotk2idnumOut ←StateRefe2.gotk2idnumOut
1005       StateRef1.gotk2idnum ←Timeout.gotk2idnum
1006       StateRef2.gotk2idnum ←Timeout.gotk2idnum
1007
1008  rule VarDef ::= VarDefId SmallCode
1009  attr VarDefId.gotk2idnumIn ←VarDef.gotk2idnumIn
1010       VarDef.gotk2idnumOut ←VarDefId.gotk2idnumOut
1011       VarDefId.gotk2idnum ←VarDef.gotk2idnum
1012
1013  rule MachineDef ::= ID
1014  attr MachineDef.gotk2idnumOut ←MachineDef.gotk2idnumIn
1015
1016  rule StateDefId ::= ID
1017  attr StateDefId.gotk2idnumOut ←StateDefId.gotk2idnumIn
1018
1019  rule EventDefId ::= ID
1020  attr EventDefId.gotk2idnumOut ←EventDefId.gotk2idnumIn
1021
1022  rule TransitionDef ::= ID
1023  attr TransitionDef.gotk2idnumOut ←TransitionDef.gotk2idnumIn
1024
1025  rule EventRef ::= ID
1026  attr EventRef.gotk2idnumOut ←EventRef.gotk2idnumIn
1027
1028  rule StateRef ::= ID
1029  attr StateRef.gotk2idnumOut ←StateRef.gotk2idnumIn
1030       StateRef.id ← StateRef.bind:idnum ↞ StateRef.gotk2idnum
1031
1032
1033  rule Program ::= StateMachine
1034  attr StateMachine.init_codeIn ← statemachine_init_code()
1035       StateMachines.init_code ← StateMachine.init_codeOut ++ "} "
1036
1037  rule StateMachine ::= MachineDef MachineStats
1038  attr MachineDef.init_codeIn ← StateMachines.init_codeIn
1039       MachineStats.init_codeIn ← newscope(MachineDef.init_codeOut)
1040       StateMachine.init_codeOut ← MachineStats.init_codeOut
1041
1042  rule MachineStats ::= MachineStats MachineStat
1043  attr MachineStats2.init_codeIn ←MachineStats1.init_codeIn
1044       MachineStat.init_codeIn ← MachineStats2.init_codeOut
1045       MachineStats1.init_codeOut ←MachineStat.init_codeOut
1046
1047  rule MachineStats ::= ε
1048  attr MachineStats.init_codeOut ← MachineStats.init_codeIn
1049
1050  rule MachineStat ::= StateDef
1051  attr StateDef.init_codeIn ← MachineStat.init_codeIn
1052       MachineStat.init_codeOut ← StateDef.init_codeOut
1053
1054  rule MachineStat ::= Transition
1055  attr Transition.init_codeIn ← MachineStat.init_codeIn
1056       MachineStat.init_codeOut ← Transition.init_codeOut
1057
1058  rule MachineStat ::= EventDef
1059  attr EventDef.init_codeIn ← MachineStat.init_codeIn
1060       MachineStat.init_codeOut ← EventDef.init_codeOut
1061
```

```
1062  rule MachineStat ::= Timeout
1063  attr Timeout.init_codeIn ← MachineStat.init_codeIn
1064       MachineStat.init_codeOut ← Timeout.init_codeOut
1065
1066  rule MachineStat ::= VarDef
1067  attr VarDef.init_codeIn ← MachineStat.init_codeIn
1068       MachineStat.init_codeOut ← VarDef.init_codeOut
1069
1070  rule StateDef ::= StateDefId StateStats
1071  attr StateDefId.init_codeIn ← StateDef.init_codeIn
1072       StateStats.init_codeIn ← StateDefId.init_codeOut
1073       StateDef.init_codeOut ← StateStats.init_codeOut
1074
1075  rule StateStats ::= StateStats StateStat
1076  attr StateStats₂.init_codeIn ←StateStats₁.init_codeIn
1077       StatStat.init_codeIn ← StatStats₂.init_codeOut
1078       StateStats₁.init_codeOut ←StatStat.init_codeOut
1079
1080  rule StateStats ::= ε
1081  attr StateStats.init_codeOut ← StateStats.init_codeIn
1082
1083  rule StateStat ::= Entry
1084  attr Entry.init_codeIn ← StateStat.init_codeIn
1085       StateStat.init_codeOut ← Entry.init_codeOut
1086
1087  rule StateStat ::= Running
1088  attr Running.init_codeIn ← StateStat.init_codeIn
1089       StateStat.init_codeOut ← Running.init_codeOut
1090
1091  rule StateStat ::= StateMachine
1092  attr StateMachine.init_codeIn ← StateStat.init_codeIn
1093       StateStat.init_codeOut ← StateMachine.init_codeOut
1094
1095  rule Transition ::= EventRef StateRef StateRef TransitionDef
1096  attr EventRef.init_codeIn ← Transition.init_codeIn
1097       StateRef₁.init_codeIn ←EventRef.init_codeOut
1098       StateRef₂.init_codeIn ←StateRef₁.init_codeOut
1099       TransitionDef.init_codeIn ← StateRef₂.init_codeOut
1100       Transition.is_unique ← count_elem(Transition.const_sym, Transition.eventsNames) ≤ 1
1101       Transition.def_code ← "this ⇒ transitions[" ++ Transition.fromstate_id ++ "][“
1102                             ++ Transition.const_id ++ "] = { " ++ Transition.tostate_id
1103                             ++ "} \n"
1104       Transition.init_codeOut ← if Transition.is_unique
1105                                 then TransitionDef.init_codeOut ++ Transition.def_code
1106                                 else TransitionDef.init_codeOut
1107       Transition.const_sym ← EventRef.const_sym
1108       Transition.const_id ← EventRef.const_id
1109       Transition.fromstate_id ← StateRef₁.id
1110       Transition.tostate_id ← StateRef₂.id
1111
1112  rule Entry ::= SmallCode
1113  attr Entry.init_codeOut ← Entry.init_codeIn
1114
1115  rule Running ::= SmallCode
1116  attr Running.init_codeOut ← Running.init_codeIn
1117
1118  rule EventDef ::= EventDefId SmallCode
1119  attr EventDefId.init_codeIn ← EventDef.init_codeIn
1120       EventDef.init_codeOut ← EventDefId.init_codeIn
1121
1122  rule Timeout ::= Number StateRef StateRef
1123  attr StateRef₁.init_codeIn ←Timeout.init_codeIn
1124       StateRef₂.init_codeIn ←StateRef₁.init_codeOut
1125       Timeout.init_codeOut ← StateRefe₂.init_codeOut
1126
1127  rule VarDef ::= VarDefId SmallCode
1128  attr VarDefId.init_codeIn ← VarDef.init_codeIn
1129       VarDef.init_codeOut ← VarDefId.init_codeOut
1130
1131  rule MachineDef ::= ID
1132  attr MachineDef.init_codeOut ← MachineDef.init_codeIn
1133
1134  rule StateDefId ::= ID
1135  attr StateDefId.init_codeOut ← StateDefId.init_codeIn
1136
1137  rule EventDefId ::= ID
1138  attr EventDefId.init_codeOut ← EventDefId.init_codeIn
1139
1140  rule TransitionDef ::= ID
1141  attr TransitionDef.init_codeOut ← TransitionDef.init_codeIn
1142
1143  rule EventRef ::= ID
1144  attr EventRef.init_codeOut ← EventRef.init_codeIn
1145       EventRef.const_id ← EventRef.id
1146       EventRef.const_sym ← EventRef.sym
```

```
1147
1148  rule StateRef ::= ID
1149  attr StateRef.init_codeOut ← StateRef.init_codeIn
```

Quelltext E.5 – Resultierende Attributgrammatik der Beispiele für die Sprache aus [29] mit Codegenerierung für Zustandsübergänge.

E.3. Hilfsfunktionen der Beispiele

Die in dieser Arbeit verwendeten Hilfsfunktion sind, in Haskell-angelehnter Syntax, wie folgt definiert:

```
1  append_with_comma :: String ⇒ String ⇒ String
2  append_with_comma a b = if a = "" then b
3                          else if b = "" then a
4                          else a ++ ", " ++ b
5
6  count_elem :: [a] ⇒ a ⇒ Int
7  count_elem [] _ = 0
8  count_elem (x:xs) y = if x = y then 1 + count_elem(xs, y)
9                        else count_elem(xs, y)
10
11 bindKey :: Environment ⇒ StringTableKey ⇒ Binding
12 bindingInEnv :: Environment ⇒ StringTableKey ⇒ Binding
13 newscope :: Environment ⇒ Environment
```

Für die Funktion `count_elem` wäre eine Alternative Implementierung mittels `filter` und `length` der Haskell Standardbibliothek realisierbar.

Die Funktionen `bindKey` und `bindInEnv` entsprechen den groß geschriebenen Varianten der Namensanalyse aus [74]. Im Unterschied zu [74] wird zwischen Bindung und Eintrag in die Definitionstabelle in dieser Arbeit nicht unterschieden. Folgende Übersicht zeigt die üblichen verwendeten Typen:

`Environment` Eine Definitionstabelle.

`Binding` Ein Zeiger auf einen Eintrag in die Definitionstabelle.

`StringTableKey` Ein Index in die Symboltabelle.

`String` Eine Zeichenfolge.

Sev Eine Meldungsart, $Sev = \{FATAL, ERROR, WARNING, NOTE\}$.

`Int` Ein Ganzzahltyp.

[a] Eine mit der Typvariablen `a` instanziierte Liste.

Mittels `bindKey` wird ein neuer Eintrag in der Definitionstabelle angelegt; das erste Argument von `bindKey` ist die Definitionstabelle zum Ablegen und das zweite Argument der Bezeichner für den ein Eintrag angelegt werden soll. `bindKey` liefert als Resultat einen Eintrag in die Definitionstabelle oder `NoBinding` falls solch ein Eintrag bereits existiert.

Analog sucht `bindInEnv` nach einem Eintrag im ersten Argument für einen Bezeichner, der als zweites Argument übergeben werden kann. `bindInEnv` liefert ebenfalls `NoBinding`, jedoch falls kein solcher Eintrag existiert. `newscope` erzeugt einen neuen Namensraum unterhalb des übergebenen Arguments. Für weitere Details siehe [74].

Anhang F.

Erweiterung der Sprache zur Beschreibung geordneter Attributgrammatiken

F.1. Einleitung

Die in diesem Kapitel vorgestellte Sprache erweitert geordnete Attributgrammatiken um eine Haskell-artige Syntax zur Definition von Datentypen. Diese Datentypen und darauf arbeitende Funktionen, deren Ausgabe, Funktionen zum Debuggen sowie eine Typprüfung werden darin implementiert. Darüber hinaus kann eine Typinferenz genutzt werden. Auf Basis dieser Sprache, die mittelfristig Teil des bekannten Übersetzerbau-Werkzeugkastens eli (siehe [52]) sein soll, wurden die Muster dieser Arbeit implementiert.

Neben den bereits benannten Sprachteilen wurden ebenfalls Möglichkeiten der Definition von Pretty-Printing und der Definitionstabelle in dieser Sprache hinzugefügt. Aufgrund diesen Hinzufügens kann innerhalb der Attributgrammatik geprüft werden ob der Aufruf solch einer Hilfsfunktion typsicher ist. Darüber hinaus wurden die üblichen Namensanalysen implementiert.

Die Sprache selbst ist in sich selbst definiert und ist somit Bootstrapping-fähig. Bootstrapping bezeichnet den Prozess, dass ein Übersetzer für eine Sprache in dieser Sprache selbst implementiert ist.

Da der Umfang allein der semantischen Spezifikation mit ca. 25500 Zeilen sehr hoch ist, wird darauf verzichtet diesen darzustellen. Einige Implementierungsdetails und Entscheidungen wurden getroffen um hohe Kompatibilität mit eli und `C++11` zu erreichen. In eli werden konkrete und abstrakte Syntax getrennt und mittels des in [72] vorgestellten Werkzeugs verbunden.

Das in diesem Kapitel vorgestellte Werkzeug wurde schrittweise um sprachliche Konzepte erweitert, sodass in Teilen der Implementierung nur eine Teilmenge der sprachlichen Möglichkeiten genutzt werden. Weiterhin sind bestimmte Funktionen und Attributierungen in der Implementierung genau so umgesetzt, da bestimmte Fähigkeiten noch nicht oder fehlerhaft umgesetzt waren.

Im folgenden wird nur die konkrete Syntax präsentiert und die Implementierung nur intuitiv besprochen.

F.2. Allgemeines, Kopieranweisungen und Laden

Wie auch in Haskell werden Bezeichner mit kleinen Buchstaben beginnend als Funktionen, Konstanten und Variablen[1] interpretiert. Typnamen sowie Terminale und Nichtterminale beginnen dagegen mit einem Großbuchstaben. Ausnahmen von dieser Regel sind eine Reihe vordefinierter Bezeichner und Typen wie **`int`** oder `NoBinding`. Die zugrunde liegende Implementierung hat hierbei das Sprachdesign beeinflusst.

[1] Im Gegensatz zu Haskell sind Variablen in L2, aufgrund der Abbildung nach `C++11`, zuweisbar.

```
1  ADA_COMMENT
2  C_COMMENT
3
4  glaString: C_STRING_LIT [mkidn]
5  glaIdentifier : $[a-z][a-zA-Z0-9_]* [mkidn]
6  glaTypename : $[A-Z][a-zA-Z0-9_]* [mkidn]
7  glaNumber: C_INT_DENOTATION [mkidn]
8
9  glaCText: $\{ [CTextOrNot]
10 glaBackslash:$(\\){1,2} [mkidn]
11 glaLiteral: PASCAL_STRING [mkidn]
```

Quelltext F.1 – Lexikalische Analyse als Spezifikation in eli für L2 (l2.gla)

Quelltext F.2 zeigt die konkrete Syntax um reinen C++ Quelltext in Header-Datei und C++ Implementierungs-Datei zu kopieren. Die unterschiedlichen Varianten (**post** und ohne Voranstellen dieses Schlüsselworts) dienen dem Kopieren vor bzw. nach dem von dieser Sprache – im Folgenden L2 genannt – generierten Quelltexts. Ebenfalls lassen sich mit **lido** Blöcke reiner Attributierungsanweisungen hinzufügen.

```
1  AAARoot → program
2  program → decls
3  decls → decls decl
4  decls → ε
5
6  decl → copyto_head_decl
7  decl → copyto_source_decl
8  decl → copyto_lido_decl
9  decl → load_decl
10
11 copyto_head_decl → &'switch_ctext();' 'head' ctext &'switch_ctext();'
12 copyto_source_decl → &'switch_ctext();' 'source' ctext &'switch_ctext();'
13 copyto_head_decl → 'post' &'switch_ctext();' 'head' ctext &'switch_ctext();'
14 copyto_source_decl → 'post' &'switch_ctext();' 'source' ctext &'switch_ctext();'
15
16 copyto_lido_decl → &'switch_ctext();' 'lido' ctext &'switch_ctext();'
17 load_decl → 'load' &'CheckNewInput(GetCurrTok());' string
18
19 ctext → glaCText
```

Quelltext F.2 – Konkrete Syntax zum Kopieren und Laden (l2.con)

Das Symbol $\langle AAARoot \rangle$ entspricht der Wurzel Z einer kontextfreien Grammatik. Mit `switch_ctext` wird die lexikalische Analyse umgeschalten[2].

Das Laden (Schlüsselworts **load**) dient dem importieren von Quelldateien[3] und der korrekten Bestimmung von Positionsdaten bei mehreren Eingabedateien.

F.3. Definitionstabellendefinition und Pretty Printing

Das System aus [52] enthält bereits Möglichkeiten der Definition einer Definitionstabelle und der kompakten Spezifikation von Pretty Printing. Allerdings werden diese Definitionen nicht auf Typkorrektheit und Existenz in der Attributgrammatik untersucht. In L2 wurde daher ein Mechanismus zur Spezifikation dieser Sprachteile umgesetzt und um eine Typanalyse sowie verschiedene Namensanalysen und weitere Bestandteile erweitert. So wird ebenfalls Quelltext generiert, der es erlaubt selbstdefinierte Datentypen

[2] Siehe dazu auch die eli-Dokumentation zur lexikalischen Analyse unter http://eli-project.sourceforge.net/elionline/-lex_1.html.
[3] Siehe zu diesem Mechanismus http://eli-project.sourceforge.net/elionline/input_1.html.

automatisch beim Debugging zu verwenden[4] und diese ebenfalls im Pretty Printing ohne Anpassung zu verwenden.

Quelltext F.3 zeigt die konkrete Syntax zur Umsetzung von Definitionstabellen und Pretty Printing in L2.

```
20  decl → ptg_decl
21
22  ptg_decl → 'ptg' '{' ptg_definitions '}'
23
24  ptg_definitions → ptg_definitions ptg_definition
25  ptg_definitions → ε
26
27  ptg_definition → ptg_def_id ':' ptg_patterns
28
29  ptg_def_id → identifier
30  ptg_def_id → typename
31
32  ptg_patterns → ptg_patterns ptg_pattern
33  ptg_patterns → ptg_pattern
34
35  ptg_pattern → string
36  ptg_pattern → ptg_insertion
37  ptg_pattern → ptg_call
38  ptg_pattern → ptg_optional
39
40  ptg_insertion → '$' ptg_optional_number ptg_optional_type
41  ptg_optional_number → ε
42  ptg_optional_number → ptg_number
43  ptg_optional_type → ε
44  ptg_optional_type → eli_type_reference
45  ptg_optional_type → 'as' simple_type
46
47  eli_type_reference → 'int'
48  eli_type_reference → 'string'
49  eli_type_reference → 'pointer'
50  eli_type_reference → 'char'
51  eli_type_reference → 'double'
52  eli_type_reference → 'float'
53  eli_type_reference → 'long'
54  eli_type_reference → 'short'
55
56
57  typename → glaTypename
58
59  ptg_call → '[' ptg_function_reference ptg_call_params ']'
60
61  ptg_function_reference → typename
62  ptg_function_reference → identifier
63
64
65  ptg_call_params → ε
66  ptg_call_params → ptg_call_params ptg_call_param
67  ptg_call_param → ptg_insertion
68
69  ptg_optional → '{' ptg_patterns '}'
70
71  ptg_number → number
72
73  number → glaNumber
74  string → glaString
75  identifier → glaIdentifier
76
77
78  decl → pdl_decl
79  pdl_decl → 'pdl' '{' pdl_defs '}'
80
```

[4] Hierfür wurde ein angepasstes Ausgabemodul auf Basis von `C++11` implementiert bei dem überladene Funktionen und Meta-Programmierung zum Einsatz kommen. Eine Variante um die grafische Oberfläche mit Tcl/TK von eli zu verwenden wurde bisher nicht umgesetzt.

```
81  pdl_defs → pdl_defs pdl_def
82  pdl_defs → ε
83  pdl_def → load_decl
84
85  pdl_def → pdl_include_decl
86  pdl_include_decl → string
87
88  pdl_def → pdl_def_names
89  pdl_def_names → pdl_property_names '::' pdl_type ';'
90  pdl_def_names → pdl_property_names ':' pdl_type ';'
91  pdl_property_names → pdl_property_names ',' pdl_property_def_id
92  pdl_property_names → pdl_property_def_id
93
94  pdl_property_def_id → typename
95  pdl_property_def_id → identifier
96
97  pdl_type → simple_type
```

Quelltext F.3 – Konkrete Syntax zur Spezifikation und Implementierung von Definitionstabelle und Pretty Printing(l2.con)

F.4. Datentypen

Datentypen werden wie in Haskell mit dem Schlüsselwort **`data`** deklariert. Siehe dazu auch die konkrete Syntax aus Quelltext F.4. Diese werden auf Enumeratoren oder eine Klassenhierarchie in `C++11` abgebildet. Neben einer automatischen Speicherverwaltung und der Erzeugung von Ausgabefunktionen werden ebenfalls Hilfsfunktionen generiert, sodass diese Datentypen in der Attributgrammatik verwendbar sind.

```
102 decl → newtype_decl
103
104 newtype_decl → 'newtype' type_def_id '=' simple_type ';'
105
106 simple_type → type_reference
107 simple_type → list_of_type
108 simple_type → eli_type_reference
109
110 list_of_type → '[' type_construction ']'
111
112 simple_type → tuple_of_type
113 tuple_of_type → '(' type_construction ',' type_constructions ')'
114 type_constructions → type_constructions ',' type_construction
115 type_constructions → type_construction
116
117 simple_type → map_type
118
119 map_type → '(' type_construction '=>' type_construction ')'
120
121 type_construction → simple_type
122
123 decl → data_decl
124
125 data_decl → 'data' data_def_id ';'
126 data_decl → 'data' data_def_id '=' data_constructors ';'
127 data_decl → 'data' data_def_id '=' '{' data_record_arguments '}'
128
129 data_record_arguments → data_record_arguments ',' data_record_argument
130 data_record_arguments → data_record_argument
131
132 data_record_argument → data_record_arg_def_id '::' simple_type
133 data_record_arg_def_id → identifier
134
135 data_def_id → typename
136
137 data_constructors → data_constructors '|' data_constructor
138 data_constructors → data_constructor
```

```
139  data_constructor → data_constructor_simple
140  data_constructor → data_constructor_complex
141
142  data_constructor_simple → data_constructor_def_id
143  data_constructor_def_id → typename
144
145  data_constructor_complex → data_constructor_def_id data_constructor_arguments
146
147  data_constructor_arguments → data_constructor_arguments data_constructor_argument
148  data_constructor_arguments → data_constructor_argument
149
150  data_constructor_argument → simple_type
151
152  type_def_id → typename
153  type_reference → typename
```

Quelltext F.4 – Konkrete Syntax zur Spezifikation von Datentypen (l2.con)

F.5. Funktionen, Funktionssignaturen, Konstanten

Quelltext F.5 zeigt die konkrete Syntax zur Spezifikation von Funktionen und Funktionssignaturen.

```
156  decl → function_impl
157  function_impl → function_def_id function_patterns function_rhs
158
159  function_rhs → '=' function_expression
160  function_rhs → function_guards
161  function_rhs → '=' 'do' '{' function_do_statements '}'
162
163  function_do_statements → function_do_statements function_do_statement
164  function_do_statements → ε
165
166  function_do_statement → function_expression ';'
167  function_do_statement → function_cond_statement ';'
168  function_do_statement → function_var_def_id '<-' function_expression ';'
169  function_do_statement → function_return_statement ';'
170  function_do_statement → function_cond_msg ';'
171
172  function_do_statement → function_stat_assign ';'
173  function_do_statement → function_each_stat
174  function_do_statement → function_noreturn
175
176  function_noreturn → 'noreturn'
177  function_noreturn → 'noreturn' ';'
178  function_noreturn → 'fallthrough'
179  function_noreturn → 'fallthrough' ';'
180
181  function_each_stat → 'each' function_var_def_id 'of' function_primary_expression ':'
182                         '{' function_do_statements '}'
183  function_each_stat → 'each' function_var_def_id 'of' function_primary_expression ':'
184                          function_do_statement
185
186  function_stat_assign → function_postfix_expression '=' function_expression
187
188  function_cond_msg → 'when' function_expression 'then' 'msg' function_expression
189
190  function_return_statement → 'return' function_expression
191
192  function_cond_statement → 'cond' function_expression '=>' function_error_expr
193                            'with' function_expression
194  function_cond_statement → 'cond' function_expression '=>' function_error_expr
195                            'with' '(' ')'
196
197  function_guards → function_guards function_guard
198  function_guards → function_guard
199
200  function_guard → '|' function_expression function_rhs ';'
```

```
201
202  function_patterns → function_patterns function_pattern
203  function_patterns → ε
204
205  function_pattern → function_var_def_id
206  function_pattern → function_list_pattern
207  function_pattern → function_tuple_pattern
208  function_pattern → function_pattern_ignore
209  function_pattern → function_pattern_tcon
210  function_pattern → function_pattern_tconsimple
211  function_pattern → function_constant_pattern
212
213  function_constant_pattern → general_constant
214
215  function_pattern_tconsimple → type_reference
216  function_pattern_tcon → '(' type_reference type_constructor_argument_binds ')'
217
218  type_constructor_argument_binds → type_constructor_argument_binds
219                                      type_constructor_argument_bind
220  type_constructor_argument_binds → type_constructor_argument_bind
221  type_constructor_argument_bind → function_var_def_id
222
223  function_pattern_ignore → '_'
224  function_tuple_pattern → '(' function_var_def_id ',' function_tuple_pattern_elements ')'
225  function_tuple_pattern_elements → function_tuple_pattern_elements ','
226                                      function_tuple_pattern_element
227  function_tuple_pattern_elements → function_tuple_pattern_element
228  function_tuple_pattern_element → function_var_def_id
229
230  function_list_pattern → '(' function_var_def_id ':' function_var_def_id ')'
231  function_list_pattern → '(' function_var_def_id ':' '[' ']' ')'
232
233  function_expression → function_c_like_expr
234  function_expression → function_hs_like_expr
235
236  function_hs_like_expr → function_error_expr
237  function_hs_like_expr → function_if_expr
238  function_hs_like_expr → function_let_expr
239  function_hs_like_expr → function_lambda
240
241  function_lambda → lambda function_var_def_id opt_typing '->' function_expression
242  opt_typing → ε
243  opt_typing → '::' simple_type
244  lambda → glaBackslash
245
246  function_let_expr → 'let' function_let_var_defs 'in' function_expression
247
248  function_let_var_defs → function_let_var_defs ',' function_let_var_def
249  function_let_var_defs → function_let_var_def
250
251  function_let_var_def → function_var_def_id opt_typing '=' function_expression
252
253  function_if_expr → 'if' function_expression 'then' function_expression
254                       'else' function_expression
255  function_error_expr → 'error' function_expression
256
257  function_c_like_expr → function_logical_or_expression
258  function_logical_or_expression → function_logical_and_expression
259  function_logical_or_expression → function_logical_or_expression '||'
260                                     function_logical_and_expression
261
262  function_logical_and_expression → function_equality_expression
263  function_logical_and_expression → function_logical_and_expression '&&'
264                                      function_equality_expression
265
266  function_equality_expression → function_relational_expression
267  function_equality_expression → function_equality_expression '=='
268                                   function_relational_expression
269  function_equality_expression → function_equality_expression '!='
270                                   function_relational_expression
271
272  function_relational_expression → function_relational_expression '<'
```

```
273                                     function_concat_expression
274  function_relational_expression → function_relational_expression '>'
275                                     function_concat_expression
276  function_relational_expression → function_relational_expression '<='
277                                     function_concat_expression
278  function_relational_expression → function_relational_expression '>='
279                                     function_concat_expression
280  function_relational_expression → function_concat_expression
281
282  function_concat_expression → function_additive_expression
283  function_concat_expression → function_concat_expression '++'
284                                 function_additive_expression
285
286  function_additive_expression → function_multiplicative_expression
287  function_additive_expression → function_additive_expression '+'
288                                   function_multiplicative_expression
289  function_additive_expression → function_additive_expression '-'
290                                   function_multiplicative_expression
291
292  function_multiplicative_expression → function_unary_expression
293  function_multiplicative_expression → function_multiplicative_expression '*'
294                                       function_unary_expression
295  function_multiplicative_expression → function_multiplicative_expression '/'
296                                       function_unary_expression
297  function_multiplicative_expression → function_multiplicative_expression '%'
298                                       function_unary_expression
299
300  function_unary_expression → function_postfix_expression
301  function_unary_expression → '++' function_postfix_expression
302  function_unary_expression → '-' function_postfix_expression
303  function_unary_expression → '&' function_postfix_expression
304  function_unary_expression → '!' function_postfix_expression
305  function_unary_expression → '*' function_postfix_expression
306
307  function_postfix_expression → function_primary_expression
308  function_postfix_expression → function_postfix_expression
309                                  '[' function_expression ']'
310  function_postfix_expression → function_postfix_expression
311                                  '->' function_primary_expression
312  function_postfix_expression → function_postfix_expression
313                                  ':' function_primary_expression
314
315  function_primary_expression → function_constant
316
317  function_constant → general_constant
318  general_constant → string
319  general_constant → number
320  general_constant → 'True'
321  general_constant → 'true'
322  general_constant → 'False'
323  general_constant → 'false'
324  general_constant → 'coordref'
325  general_constant → '[' ']'
326
327  function_primary_expression → function_call
328
329  function_primary_expression → '(' function_tuple_construction ')'
330  function_tuple_construction → function_tuple_arguments ',' function_tuple_argument
331  function_tuple_arguments → function_tuple_arguments ',' function_tuple_argument
332  function_tuple_arguments → function_tuple_argument
333  function_tuple_argument → function_expression
334
335  function_call → callable_reference '(' function_call_params ')'
336  function_call → callable_reference '(' ')'
337  function_call → callable_reference
338
339  callable_reference → identifier
340  callable_reference → typename
341  callable_reference → 'end'
342
343  function_call_params → function_call_params ',' function_call_param
344  function_call_params → function_call_param
```

```
345  function_call_param → function_expression
346
347  function_var_def_id → identifier
348
349  function_primary_expression → '(' function_expression ')'
350
351  decl → constant_decl
```

Quelltext F.5 – Konkrete Syntax zur Beschreibung von Funktionssignaturen und Funktionsimplementierungen (l2.con)

Die Funktionen unterstützen Typpolymorphie, welche auch automatisch inferriert werden kann. In **do**-Blöcken werden die Variablen automatisch typisiert. Wie in Haskell können die Datentypen (siehe Quelltext F.4) dekonstruiert werden. Dabei wird ebenfalls geprüft, dass alle Konstruktoren eines Datentyps, d.h. alle Untertypen in einer Klassenhierarchie, als dekonstruiertes Argument vorkommen.

F.6. Teilsystem: Attributgrammatik

Ursprüngliche Absicht bei der Beschreibung der abstrakten Syntax war die Entwicklung eines Transformationsmechansimus zur Unterstützung von Attributen höherer Ordnung bei gleichzeitiger Sicherstellung einer Ordnungsrelation auf den erstellten abstrakten Syntaxbäumen. Dies wurde jedoch so nicht umgesetzt, sodass die Namensanalyse für abstrakte Syntaxbäume und deren Beschreibung keine Relevanz im Sinne dieser Arbeit haben. Die konkrete Syntax dieses Teilsystems sowie der Systeme zur Beschreibung von Symbolberechnungen, Klassensymbolen und Attributdeklarationen ist in Quelltext F.6 abgebildet.

```
352
353  constant_decl → 'constant' constant_def_id '::' simple_type ';'
354  constant_decl → 'constant' constant_def_id '::' simple_type '=' function_expression ';'
355
356  constant_def_id → identifier
357  constant_def_id → typename
358
359  decl → function_type_decl
360
361  function_type_decl → function_def_id '::' function_type ';'
362  simple_type → '(' function_type ')'
363
364  function_type → function_types '->' return_type
365  function_types → function_types '->' simple_type
366  function_types → simple_type
367
368  return_type → simple_type
369
370  function_def_id → identifier
371
372  simple_type → '(' ')'
373  simple_type → type_variable_def_id
374  type_variable_def_id → identifier
375
376  decl → class_abstree_decl
377  decl → abstree_decl
378
379  class_abstree_decl → 'class' 'abstree' abstree_def_id abstree_changes abstree_opt_rules
380  abstree_decl → 'abstree' abstree_def_id abstree_changes abstree_opt_rules
381
382  abstree_def_id → typename
383  abstree_def_id → identifier
384
385  abstree_changes → ε
386  abstree_changes → abstree_changes abstree_change
387
388  abstree_change → abstree_prefixes
389  abstree_change → abstree_is
390  abstree_change → abstree_inherits
391  abstree_change → abstree_trafo
```

```
392  abstree_change → abstree_depends
393
394  abstree_prefixes → 'prefixes' abstree_reference_list 'with' typename ',' new_ruleprefix
395  new_ruleprefix → typename
396  new_ruleprefix → identifier
397
398  abstree_is → 'is' abstree_reference_list
399  abstree_reference_list → abstree_reference_list ',' abstree_reference
400  abstree_reference_list → abstree_reference
401
402  abstree_reference → identifier
403  abstree_reference → typename
404
405  abstree_inherits → 'inherits' abstree_reference_list
406
407  abstree_trafo → 'trafo' identifier 'from' abstree_reference
408  abstree_trafo → 'trafo' identifier 'to' abstree_reference
409  abstree_depends → '<-' abstree_reference_list
410
411  abstree_opt_rules → '=>' abstree_rules 'end'
412  abstree_opt_rules → ';'
413
414  abstree_rules → abstree_rules abstree_rule
415  abstree_rules → abstree_rule
416
417  abstree_rule → abstree_term ';'
418  abstree_rule → abstree_production
419
420  abstree_term → 'term' tree_symbol_def_id
421  abstree_term → 'term' tree_symbol_def_id 'as' simple_type opt_conversion_ref
422  abstree_term → 'term' tree_symbol_def_id 'is' simple_type
423
424  tree_symbol_def_id → typename
425
426  opt_conversion_ref → 'via' typename
427  opt_conversion_ref → 'via' identifier
428  opt_conversion_ref → ε
429
430  symbol_def_id → typename
431
432  abstree_production → tree_symbol_def_id '→' symbols opt_rulename ';'
433
434  symbols → symbols symbol
435  symbols → ε
436
437  symbol → literal
438  symbol → '$' symbol_ref_tree
439  symbol → symbol_ref_tree
440
441  symbol_ref_tree → symbol_reference opt_tree_hint
442
443  symbol_reference → typename
444
445  literal → glaLiteral
446
447  opt_rulename → '<' rule_def_id '>'
448  opt_rulename → ε
449
450  rule_def_id → typename
451  rule_def_id → identifier
452
453
454  decl → global_attribute_decl
455  global_attribute_decl → attr_class global_attr_def_ids opt_typing ';'
456
457  attr_class → 'inh'
458  attr_class → 'synt'
459  attr_class → 'thread'
460  attr_class → 'chain'
461  attr_class → 'infer'
462
463  global_attr_def_ids → global_attr_def_ids ',' global_attr_def_id
```

```
464  global_attr_def_ids → global_attr_def_id
465
466  global_attr_def_id → identifier
467  decl → rule_specification
468  rule_specification → rule_keyword rule_pattern rule_guard_rhs
469
470  rule_guard → '|' rule_expression rule_guard_rhs
471  rule_guard_rhs → rule_guard
472  rule_guard_rhs → '=>' rule_computations 'end'
473
474
475  rule_keyword → 'rule'
476  rule_keyword → 'rules'
477  rule_keyword → 'class' 'rule'
478  rule_keyword → 'class' 'rules'
479
480  opt_tree_hint → ε
481  opt_tree_hint → '@' abstree_reference
482
483  rule_pattern → '<' rule_references ',' rule_reference '>' opt_tree_hint opt_production
484  rule_pattern → '<' rule_reference '>' opt_tree_hint opt_production
485  rule_pattern → rule_lhs_pattern '↦' rule_rhs_pattern
486
487  rule_reference → typename
488  rule_reference → identifier
489
490  rule_references → rule_references ',' rule_reference
491  rule_references → rule_reference
492
493  rule_lhs_pattern → opt_rule_var_def symbol_reference opt_tree_hint
494  rule_lhs_pattern → opt_rule_var_def '_' opt_tree_hint
495  opt_rule_var_def → ε
496  opt_rule_var_def → rule_var_def_id ':'
497
498  opt_production → rule_lhs_pattern '↦' rule_rhs_pattern
499  opt_production → ε
500
501  rule_rhs_pattern → rule_rhs_pattern_symbols
502  rule_rhs_pattern_symbols → rule_rhs_pattern_symbols rule_rhs_pattern_symbol
503  rule_rhs_pattern_symbols → ε
504  rule_rhs_pattern_symbol → opt_rule_var_def '_'
505  rule_rhs_pattern_symbol → '__'
506  rule_rhs_pattern_symbol → opt_rule_var_def symbol opt_deeper_rule_pattern
507  rule_rhs_pattern_symbol → string
508  rule_rhs_pattern_symbol → number
509
510  opt_deeper_rule_pattern → ε
511  opt_deeper_rule_pattern → '(' deeper_rule_pattern_symbols ')'
512
513  deeper_rule_pattern_symbols → deeper_rule_pattern_symbols deeper_rule_pattern_symbol
514  deeper_rule_pattern_symbols → ε
515
516  deeper_rule_pattern_symbol → '_'
517  deeper_rule_pattern_symbol → '__'
518  deeper_rule_pattern_symbol → opt_rule_var_def symbol_reference
519  deeper_rule_pattern_symbol → string
520  deeper_rule_pattern_symbol → number
521
522  rule_computations → rule_computations rule_computation
523  rule_computations → ε
524
525  rule_computation → rule_assign_stat opt_rule_dependency ';'
526  rule_computation → rule_expression opt_rule_dependency ';'
527  rule_computation → rule_ordered_computation
528  rule_computation → rule_chainstart opt_rule_dependency ';'
529
530  rule_chainstart → 'chainstart' rule_symbol_attr_reference '=' rule_expression
531
532  rule_assign_stat → rule_symbol_attr_reference rule_rhs_expr
533  opt_rule_dependency → ε
534  opt_rule_dependency → '<-' rule_attribute_references
535
```

```
536  rule_attribute_references → rule_attribute_reference
537  rule_attribute_references → '(' rule_attribute_reference_list ')'
538
539  rule_attribute_reference → rule_symbol_attr_reference
540  rule_attribute_reference → remote_attribute
541
542  rule_attribute_reference_list → rule_attribute_reference_list ','
543                                    rule_attribute_reference
544  rule_attribute_reference_list → rule_attribute_reference
545
546  rule_symbol_attr_reference → symbol_reference opt_symbol_index '.'
547                                 symbol_attribute_reference
548  rule_symbol_attr_reference → rule_var_reference ' ' symbol_attribute_reference
549  rule_symbol_attr_reference → rule_attr_class ' ' symbol_attribute_reference
550
551  rule_attr_class → 'tail'
552  rule_attr_class → 'head'
553
554
555  rule_var_reference → identifier
556
557  opt_symbol_index → ε
558  opt_symbol_index → '!' number
559
560  symbol_attribute_reference → identifier
561  symbol_attribute_reference → 'tree'
562
563  rule_ordered_computation → 'do' '{' rule_ordered_stats '}'
564  rule_ordered_stats → rule_ordered_stats rule_ordered_stat
565  rule_ordered_stats → rule_ordered_stat
566
567  rule_ordered_stat → rule_expression ';'
568  rule_ordered_stat → rule_symbol_attr_reference rule_rhs_expr ';'
569  rule_ordered_stat → 'return' rule_expression ';'
570  rule_rhs_expr → '=' rule_expression
571  rule_rhs_expr → rule_guard_exprs
572
573  rule_guard_exprs → rule_guard_exprs rule_guard_expr
574  rule_guard_exprs → rule_guard_expr
575
576  rule_guard_expr → '|' rule_expression rule_guard_exprs ';'
577  rule_guard_expr → '|' rule_expression '=>' rule_expression ';'
578  rule_guard_expr → '|' '_' '=>' rule_expression
579
580  remote_attribute → remote_include
581  remote_attribute → remote_constituent
582
583  remote_include → 'including' symbol_attr_references
584
585  symbol_attr_references → symbol_attr_reference
586  symbol_attr_references → '(' symbol_attr_reference_list ',' symbol_attr_reference ')'
587
588  symbol_attr_reference → symbol_reference ' ' symbol_attribute_reference
589
590  symbol_attr_reference_list → symbol_attr_reference_list ',' symbol_attr_reference
591  symbol_attr_reference_list → symbol_attr_reference
592
593  remote_constituent → opt_symbol_reference
594                         'constituent' symbol_attr_references remote_options
595  remote_constituent → opt_symbol_reference
596                         'constituents' symbol_attr_references remote_options
597
598  opt_symbol_reference → symbol_reference opt_symbol_index
599  opt_symbol_reference → ε
600
601  remote_options → opt_remote_shield remote_with opt_remote_shield
602  remote_options → ε
603  remote_options → remote_shield
604  opt_remote_shield → ε
605  opt_remote_shield → remote_shield
606
607  remote_shield → 'shield' '(' symbol_references ',' symbol_reference ')'
```

```
608  remote_shield → 'shield' symbol_reference
609  remote_shield → 'shield' '(' ')'
610
611  symbol_references → symbol_references ',' symbol_reference
612  symbol_references → symbol_reference
613
614  remote_append_function → callable_reference
615  remote_append_function → rule_lambda
616  remote_append_function → remote_binary_operator
617
618  remote_binary_operator → '+'
619  remote_binary_operator → '-'
620  remote_binary_operator → '/'
621  remote_binary_operator → '*'
622  remote_binary_operator → '++'
623  remote_binary_operator → '=='
624  remote_binary_operator → '!='
625  remote_binary_operator → '<'
626  remote_binary_operator → '>'
627  remote_binary_operator → '<='
628  remote_binary_operator → '>='
629  remote_binary_operator → '||'
630  remote_binary_operator → '&&'
631  remote_binary_operator → '%'
632
633  remote_single_function → callable_reference
634  remote_single_function → '?'
635  remote_single_function → ':' '[' ']'
636  remote_single_function → rule_lambda
637  remote_empty_function → callable_reference
638  remote_empty_function → general_constant
639  remote_empty_function → '?'
640
641  remote_with → 'with' 'infer'
642  remote_with → 'with' '(' simple_type ',' remote_append_function ','
643                           remote_single_function ',' remote_empty_function ')'
644  remote_with → 'with' remote_append_function ',' remote_single_function ','
645                        remote_empty_function
646
647  rule_expression → rule_lido_like_expr
648  rule_expression → rule_hs_like_expr
649
650  rule_hs_like_expr → rule_when_expression
651  rule_hs_like_expr → rule_error_expr
652  rule_hs_like_expr → rule_if_expr
653  rule_hs_like_expr → rule_let_expr
654  rule_hs_like_expr → rule_lambda
655  rule_hs_like_expr → rule_ordered_computation
656
657  rule_lambda → lambda rule_lambda_var_defs '->' rule_expression
658  opt_typing → ε
659  opt_typing → '::' simple_type
660  lambda → glaBackslash
661
662  rule_lambda_var_defs → rule_lambda_var_defs ',' rule_lambda_var_def
663  rule_lambda_var_defs → rule_lambda_var_def
664  rule_lambda_var_def → rule_var_def_id opt_typing
665
666  rule_var_def_id → identifier
667
668  rule_let_expr → 'let' rule_let_var_defs 'in' rule_expression
669
670  rule_let_var_defs → rule_let_var_defs ',' rule_let_var_def
671  rule_let_var_defs → rule_let_var_def
672
673  rule_let_var_def → rule_var_def_id opt_typing '=' rule_expression
674
675  rule_if_expr → 'if' rule_expression 'then' rule_expression 'else' rule_expression
676  rule_error_expr → 'error' rule_expression
677
678  rule_when_expression → 'cond' rule_expression '=>' rule_expression
679  rule_when_expression → 'when' rule_expression 'then' rule_expression
```

```
680
681  rule_lido_like_expr → rule_logical_or_expression
682  rule_logical_or_expression → rule_logical_and_expression
683  rule_logical_or_expression → rule_logical_or_expression '||' rule_logical_and_expression
684
685  rule_logical_and_expression → rule_equality_expression
686  rule_logical_and_expression → rule_logical_and_expression '&&' rule_equality_expression
687
688  rule_equality_expression → rule_relational_expression
689  rule_equality_expression → rule_equality_expression '==' rule_relational_expression
690  rule_equality_expression → rule_equality_expression '!=' rule_relational_expression
691
692  rule_relational_expression → rule_relational_expression '<'  rule_concat_expression
693  rule_relational_expression → rule_relational_expression '>'  rule_concat_expression
694  rule_relational_expression → rule_relational_expression '<=' rule_concat_expression
695  rule_relational_expression → rule_relational_expression '>=' rule_concat_expression
696  rule_relational_expression → rule_concat_expression
697
698  rule_concat_expression → rule_additive_expression
699  rule_concat_expression → rule_concat_expression '++' rule_additive_expression
700
701  rule_additive_expression → rule_multiplicative_expression
702  rule_additive_expression → rule_additive_expression '+' rule_multiplicative_expression
703  rule_additive_expression → rule_additive_expression '-' rule_multiplicative_expression
704
705  rule_multiplicative_expression → rule_unary_expression
706  rule_multiplicative_expression → rule_multiplicative_expression
707                                   '*' rule_unary_expression
708  rule_multiplicative_expression → rule_multiplicative_expression
709                                   '/' rule_unary_expression
710  rule_multiplicative_expression → rule_multiplicative_expression
711                                   '%' rule_unary_expression
712
713  rule_unary_expression → rule_postfix_expression
714  rule_unary_expression → '++' rule_postfix_expression
715  rule_unary_expression → '-' rule_postfix_expression
716  rule_unary_expression → '&' rule_postfix_expression
717  rule_unary_expression → '!' rule_postfix_expression
718  rule_unary_expression → '*' rule_postfix_expression
719
720  rule_postfix_expression → rule_primary_expression
721  rule_postfix_expression → rule_postfix_expression '[' rule_expression ']'
722  rule_postfix_expression → rule_postfix_expression '->' rule_primary_expression
723  rule_postfix_expression → rule_postfix_expression ':' rule_primary_expression
724
725  rule_primary_expression → rule_constant
726
727  rule_constant → general_constant
728
729  rule_primary_expression → rule_calling_expr
730  rule_primary_expression → '(' rule_tuple_construction ')'
731  rule_tuple_construction → rule_tuple_arguments ',' rule_tuple_argument
732  rule_tuple_arguments → rule_tuple_arguments ',' rule_tuple_argument
733  rule_tuple_arguments → rule_tuple_argument
734  rule_tuple_argument → rule_expression
735
736
737  rule_calling_expr → rule_callable_reference '(' rule_call_params ')'
738  rule_calling_expr → rule_callable_reference '(' ')'
739  rule_calling_expr → rule_callable_reference
740
741  rule_callable_reference → identifier
742  rule_callable_reference → typename
743  rule_callable_reference → 'end'
744  rule_callable_reference → rule_symbol_attr_reference
745
746  rule_callable_reference → '<' rule_reference '>' opt_tree_hint
747  rule_primary_expression → remote_attribute
748
749  rule_call_params → rule_call_params ',' rule_call_param
750  rule_call_params → rule_call_param
751  rule_call_param → rule_expression
```

```
752
753 rule_primary_expression → '(' rule_expression ')'
754
755 decl → symbol_decl
756 symbol_decl → symbol_keyword symbol_def_id opt_tree_hint symbol_options
757                opt_symbol_computations
758
759 symbol_keyword → 'symbol'
760 symbol_keyword → 'class' 'symbol'
761
762 symbol_def_id → typename
763
764 symbol_options → ε
765 symbol_options → symbol_options symbol_option
766
767 symbol_option → symbol_local_attrs
768 symbol_option → symbol_inheritance
769
770 symbol_option → symbol_local_using
771
772 symbol_local_using → 'using' local_using_reference_decls
773 local_using_reference_decls → local_using_reference_decls ',' local_using_reference_decl
774 local_using_reference_decls → local_using_reference_decl
775
776 local_using_reference_decl → attr_class local_using_references '::' simple_type
777 local_using_reference_decl → attr_class '(' local_using_references_no_class ')'
778 local_using_references → local_using_references ',' local_using_reference
779 local_using_references → local_using_reference
780
781 local_using_references_no_class → local_using_references_no_class
782                                   ',' local_using_reference_no_class
783 local_using_references_no_class → local_using_reference_no_class
784
785 local_using_reference_no_class → local_using_reference '::' simple_type
786
787 local_using_reference → identifier
788
789 symbol_ref_tree_list → symbol_ref_tree_list ',' symbol_ref_tree
790 symbol_ref_tree_list → symbol_ref_tree
791
792 symbol_inheritance → '<-' symbol_ref_tree_list
793
794 symbol_local_attrs → 'having' symbol_local_attribute_decls
795 symbol_local_attribute_decls → symbol_local_attribute_decls
796                                ',' symbol_local_attribute_decl
797 symbol_local_attribute_decls → symbol_local_attribute_decl
798
799 symbol_local_attribute_decl → attr_class symbol_local_attr_defs '::' simple_type
800 symbol_local_attribute_decl → attr_class '(' symbol_attr_decls_noclass ')'
801
802 symbol_attr_decls_noclass → symbol_attr_decls_noclass ',' symbol_attr_decl_noclass
803 symbol_attr_decls_noclass → symbol_attr_decl_noclass
804
805 symbol_attr_decl_noclass → symbol_local_attr_defs '::' simple_type
806
807 symbol_local_attr_defs → symbol_local_attr_defs ',' symbol_local_attr_def_id
808 symbol_local_attr_defs → symbol_local_attr_def_id
809
810 symbol_local_attr_def_id → identifier
811
812 opt_symbol_computations → ';'
813 opt_symbol_computations → '=>' symbol_computations 'end'
814
815 symbol_computations → symbol_computations symbol_computation
816 symbol_computations → ε
817
818 symbol_computation → symbol_assign_stat opt_symbol_dependency ';'
819 symbol_computation → symbol_expression opt_symbol_dependency ';'
820 symbol_computation → symbol_ordered_computation
821 symbol_computation → symbol_chainstart opt_symbol_dependency ';'
822
823 % statt chainstart geht auch nur 'head' attr = x
```

```
824  symbol_chainstart → 'chainstart' symbol_local_attribute_reference
825                        '=' symbol_expression
826
827
828  opt_symbol_dependency → ε
829  opt_symbol_dependency → '<-' symbol_attribute_references
830
831  symbol_attribute_references → symbol_local_attribute_ref
832  symbol_attribute_references → '(' symbol_attribute_reference_list ')'
833
834  symbol_local_attribute_ref → remote_attribute
835  symbol_local_attribute_ref → symbol_local_attribute_reference
836  symbol_local_attribute_reference →  local_attr_class ' ' local_attr_reference
837
838  symbol_attribute_reference_list → symbol_attribute_reference_list ','
839                                    symbol_local_attribute_ref
840  symbol_attribute_reference_list → symbol_local_attribute_ref
841
842  local_attr_reference → identifier
843  local_attr_class → 'this'
844  local_attr_class → 'inh'
845  local_attr_class → 'synt'
846  local_attr_class → 'tail'
847  local_attr_class → 'head'
848
849
850  symbol_assign_stat → symbol_local_attribute_reference symbol_rhs_expr
851
852  symbol_ordered_computation → 'do' '{' symbol_ordered_stats '}'
853  symbol_ordered_stats → symbol_ordered_stats symbol_ordered_stat
854  symbol_ordered_stats → symbol_ordered_stat
855
856  symbol_ordered_stat → symbol_expression ';'
857  symbol_ordered_stat → symbol_local_attribute_reference symbol_rhs_expr ';'
858  symbol_ordered_stat → 'return' symbol_expression ';'
859  symbol_rhs_expr → '=' symbol_expression
860  symbol_rhs_expr → symbol_guard_exprs
861
862  symbol_guard_exprs → symbol_guard_exprs symbol_guard_expr
863  symbol_guard_exprs → symbol_guard_expr
864
865  symbol_guard_expr → '|' symbol_expression symbol_guard_exprs ';'
866  symbol_guard_expr → '|' symbol_expression '=>' symbol_expression
867  symbol_guard_expr → '|' '_' '=>' symbol_expression
868
869  symbol_expression → symbol_lido_like_expr
870  symbol_expression → symbol_hs_like_expr
871
872  symbol_hs_like_expr → symbol_when_expression
873  symbol_hs_like_expr → symbol_error_expr
874  symbol_hs_like_expr → symbol_if_expr
875  symbol_hs_like_expr → symbol_let_expr
876  symbol_hs_like_expr → symbol_lambda
877  symbol_hs_like_expr → symbol_ordered_computation
878
879  symbol_lambda → lambda symbol_lambda_var_defs '->' symbol_expression
880  opt_typing → ε
881  opt_typing → '::' simple_type
882  lambda → glaBackslash
883
884  symbol_lambda_var_defs → symbol_lambda_var_defs ',' symbol_lambda_var_def
885  symbol_lambda_var_defs → symbol_lambda_var_def
886  symbol_lambda_var_def → symbol_var_def_id opt_typing
887
888  symbol_var_def_id → identifier
889
890  symbol_let_expr → 'let' symbol_let_var_defs 'in' symbol_expression
891
892  symbol_let_var_defs → symbol_let_var_defs ',' symbol_let_var_def
893  symbol_let_var_defs → symbol_let_var_def
894
895  symbol_let_var_def → symbol_var_def_id opt_typing '=' symbol_expression
```

```
896
897
898  symbol_if_expr → 'if' symbol_expression 'then' symbol_expression
899                   'else' symbol_expression
900  symbol_error_expr → 'error' symbol_expression
901
902  symbol_when_expression → 'cond' symbol_expression '=>' symbol_expression
903  symbol_when_expression → 'when' symbol_expression 'then' symbol_expression
904
905  symbol_lido_like_expr → symbol_logical_or_expression
906  symbol_logical_or_expression → symbol_logical_and_expression
907  symbol_logical_or_expression → symbol_logical_or_expression
908                                 '||' symbol_logical_and_expression
909
910  symbol_logical_and_expression → symbol_equality_expression
911  symbol_logical_and_expression → symbol_logical_and_expression
912                                  '&&' symbol_equality_expression
913
914  symbol_equality_expression → symbol_relational_expression
915  symbol_equality_expression → symbol_equality_expression
916                               '==' symbol_relational_expression
917  symbol_equality_expression → symbol_equality_expression
918                               '!=' symbol_relational_expression
919
920  symbol_relational_expression → symbol_relational_expression
921                                 '<'  symbol_concat_expression
922  symbol_relational_expression → symbol_relational_expression
923                                 '>'  symbol_concat_expression
924  symbol_relational_expression → symbol_relational_expression
925                                 '<=' symbol_concat_expression
926  symbol_relational_expression → symbol_relational_expression
927                                 '>=' symbol_concat_expression
928  symbol_relational_expression → symbol_concat_expression
929
930  symbol_concat_expression → symbol_additive_expression
931  symbol_concat_expression → symbol_concat_expression '++' symbol_additive_expression
932
933  symbol_additive_expression → symbol_multiplicative_expression
934  symbol_additive_expression → symbol_additive_expression
935                               '+' symbol_multiplicative_expression
936  symbol_additive_expression → symbol_additive_expression
937                               '-' symbol_multiplicative_expression
938
939  symbol_multiplicative_expression → symbol_unary_expression
940  symbol_multiplicative_expression → symbol_multiplicative_expression
941                                     '*' symbol_unary_expression
942  symbol_multiplicative_expression → symbol_multiplicative_expression
943                                     '/' symbol_unary_expression
944  symbol_multiplicative_expression → symbol_multiplicative_expression
945                                     '%' symbol_unary_expression
946
947  symbol_unary_expression → symbol_postfix_expression
948  symbol_unary_expression → '++' symbol_postfix_expression
949  symbol_unary_expression → '-' symbol_postfix_expression
950  symbol_unary_expression → '&' symbol_postfix_expression
951  symbol_unary_expression → '!' symbol_postfix_expression
952  symbol_unary_expression → '*' symbol_postfix_expression
953
954  symbol_postfix_expression → symbol_primary_expression
955  symbol_postfix_expression → symbol_postfix_expression
956                              '[' symbol_expression ']'
957  symbol_postfix_expression → symbol_postfix_expression
958                              '->' symbol_primary_expression
959  symbol_postfix_expression → symbol_postfix_expression
960                              ':' symbol_primary_expression
961
962  symbol_primary_expression → symbol_constant
963
964  symbol_constant → general_constant
965
966  symbol_primary_expression → symbol_calling_expr
967  symbol_primary_expression → '(' symbol_tuple_construction ')'
```

```
968  symbol_tuple_construction → symbol_tuple_arguments ',' symbol_tuple_argument
969  symbol_tuple_arguments → symbol_tuple_arguments ',' symbol_tuple_argument
970  symbol_tuple_arguments → symbol_tuple_argument
971  symbol_tuple_argument → symbol_expression
972
973
974  symbol_calling_expr → symbol_callable_reference '(' symbol_call_params ')'
975  symbol_calling_expr → symbol_callable_reference '(' ')'
976  symbol_calling_expr → symbol_callable_reference
977
978  symbol_callable_reference → identifier
979  symbol_callable_reference → typename
980  symbol_callable_reference → 'end'
981  symbol_callable_reference → symbol_local_attribute_ref
982  symbol_callable_reference → '<' rule_reference '>' opt_tree_hint
983
984  symbol_call_params → symbol_call_params ',' symbol_call_param
985  symbol_call_params → symbol_call_param
986  symbol_call_param → symbol_expression
987
988  symbol_primary_expression → '(' symbol_expression ')'
```

Quelltext F.6 – Konkrete Syntax zur Spezifikation der abstrakten Syntax und Attributberechnungen sowie Attributdeklarationen (l2.con)

Neben einem Teil der in dieser Arbeit vorgestellten Muster können darüber hinaus mehrere ähnlich aufgebaute Attributierungsregeln gleichzeitig über Pattern Matching[5] attributiert werden.

Weiterhin wurden zusätzliche Möglichkeiten zu deklaration Symbol-lokaler Attribute geschaffen, weitere Varianten der Beschreibung von Ausdrücken in Attributierungsregeln, anonyme Funktionen, zusätzliche Faltungsoperatoren und ein umfangreiches Typsystem.

F.7. Beispiel aus der Implementierung

Da die Implementierung einen großen Umfang hat, wird darauf verzichtet diese im Detail in dieser Arbeit aufzuführen. Stattdessen wird ein Ausschnitt aus der Implementierung gezeigt, der nahezu alle Möglichkeiten die L2 bietet, verwendet.

```
16863  abstree initial =>
16864    Decl ::= Global_AttributeDecl <rDeclGlobalAttribute>
16865    Global_AttributeDecl ::= AttributeClass Global_AttributeDefIds OptionalType
16866                             <rGlobalAttributeDecl>
16867    AttributeClass ::= <rAttributeClassInherited>
16868    AttributeClass ::= <rAttributeClassSynthesized>
16869    AttributeClass ::= <rAttributeClassThread>
16870    AttributeClass ::= <rAttributeClassChain>
16871    AttributeClass ::= <rAttributeClassInfer>
16872    Global_AttributeDefId ::= Identifier <rGlobalAttributeDefId>
16873    Global_AttributeDefIds ::= Global_AttributeDefIds Global_AttributeDefId
16874                               <rGlobalAttributeDefIdList2>
16875    Global_AttributeDefIds ::= Global_AttributeDefId <rGlobalAttributeDefIdList1>
16876
16877  symbol Global_AttributeDecl having synt reportable_sym :: StringTableKey attr
16878    this.bind ← constituent Global_AttributeDefId.bind with
16879      (Binding, \x::Binding, y::Binding -> if x = NoBinding then y else x, id,
16880       NoBinding)
16881    this.reportable_sym ← constituent Global_AttributeDefId.sym with
16882      (StringTableKey, \x::StringTableKey, y::StringTableKey ->
16883      if x = NoIdn then y else x, id, NoIdn)
16884    this.in_fn_typing ← false
16885    this.allow_fn_typing ← true
```

[5] Eigentlich wäre auch hier der Begriff „Muster über dem Aufbau von Produktionen“ ebenso richtig. Grundsätzlich entspricht dies der Beschreibung eines Prädikats (siehe Kapitel 4) über Produktionen, jedoch ist die ursprüngliche Idee die des Pattern Matching auf Produktionen gewesen.

```
16886    head.arg_type_indx_chn ← 0
16887    this.arg_type_indx_res ← -1
16888    head.fn_arg_chn ← NewEnv()
16889
16890
16891
16892  chain attr_names_chn :: Environment
16893  synt attr_names_res :: Environment
16894
16895  symbol Program attr
16896    head.attr_names_chn ← NewEnv()
16897    ↑attr_names_res ← tail.attr_names_chn >>= tail.attr_names_chn
16898
16899  data AttributeDir = AT_Synt | AT_Inh | AT_Chain | AT_Thread | AT_Infer | AT_Unknown
16900
16901  symbol Global_AttributeDefId attr this.sym ← constituent Identifier.sym
16902  symbol Global_AttributeDecl having synt attr_dir :: AttributeDir,
16903                                     synt attr_type :: D_type attr
16904    this.attr_dir ← constituent AttributeClass.attr_dir
16905    this.attr_type ← constituent OptionalType.result_type
16906
16907  symbol AttributeClass having synt attr_dir :: AttributeDir
16908  rule <rAttributeClassInherited> attr AttributeClass.attr_dir ← AT_Inh
16909  rule <rAttributeClassSynthesized> attr AttributeClass.attr_dir ← AT_Synt
16910  rule <rAttributeClassThread> attr AttributeClass.attr_dir ← AT_Thread
16911  rule <rAttributeClassChain> attr AttributeClass.attr_dir ← AT_Chain
16912  rule <rAttributeClassInfer> attr AttributeClass.attr_dir ← AT_Infer
16913
16914  pdl {
16915    AT_Direction : AttributeDir
16916    AT_Type : D_type
16917  }
16918
16919  symbol Global_AttributeDefId having inh dir :: AttributeDir, synt b :: Binding,
16920                                      inh type :: D_type attr
16921    this.type ← including Global_AttributeDecl.attr_type
16922    this.dir ← including Global_AttributeDecl.attr_dir
16923
16924    this.bind ← BindKey(↓attr_names_chn, this.sym, NewKey())
16925    this.b ← BindingInScope(↓attr_names_chn, this.sym)
16926    this.key ← KeyOf(this.bind)
16927
16928    cond this.bind ≠ NoBinding => do{
16929      Report(ERROR, coordref, concat_ind("Already defined as an attribute: ", this.sym))
16930      Report(NOTE, GetPos(KeyOf(this.b), NoPosition),
16931             "This is the place of the original definition.")
16932    }
16933
16934    this.gotActions ← do {
16935      ResetPos(this.key, coordref)
16936      ResetAT_Direction(this.key, this.dir)
16937      ResetAT_Type(this.key, this.type)
16938      when this.dir = AT_Thread then
16939        insert_thread_attributes(↓attr_names_chn, this.sym, this.type, coordref)
16940    }
16941
16942    ↑attr_names_chn ← tail.attr_names_chn >>= this.gotActions
16943
16944  abstree initial =>
16945    Remote_Attribute ::= Remote_Including <rRemoteUp>
16946    Remote_Attribute ::= Remote_Constituent <rRemoteDown>
16947    Remote_Including ::= Symbol_AttributeReferences <rRemoteIncluding>
16948    Symbol_AttributeReferences ::= SymbolAttribute <rSymbolAttributeReferencesSingle>
16949    Symbol_AttributeReferences ::= Symbol_AttributeReferenceList SymbolAttribute
16950                                   <rSymbolAttributeReferencesMultiple>
16951    Symbol_AttributeReferenceList ::= Symbol_AttributeReferenceList SymbolAttribute
16952                                      <rSymbolAttributeReferenceList2>
16953    Symbol_AttributeReferenceList ::= SymbolAttribute <rSymbolAttributeReferenceList1>
16954    SymbolAttribute ::= Symbol_Reference Symbol_AttributeReference <rSymbolAttributeRef>
16955
16956    Remote_Constituent ::= Opt_SymbolReference Symbol_AttributeReferences Remote_Options
16957                           <rRemoteConstituent>
```

```
16958    Opt_SymbolReference ::= Symbol_Reference Opt_SymbolIndex <rOptSymbolIsSymbol>
16959    Opt_SymbolReference ::= <rNoSymbolReference>
16960    Remote_Options ::= Opt_Remote_Shield Remote_With Opt_Remote_Shield
16961                       <rRemoteOptionsWith>
16962    Remote_Options ::= <rNoRemoteOptions>
16963    Remote_Options ::= Remote_Shield <rRemoteOptionsShield>
16964    Opt_Remote_Shield ::= <rNoRemoteShield>
16965    Opt_Remote_Shield ::= Remote_Shield <rOptRemoteShield>
16966
16967    Remote_Shield ::= Symbol_References Symbol_Reference <rRemoteShieldMultiple>
16968    Remote_Shield ::= Symbol_Reference <rRemoteShieldSingle>
16969    Remote_Shield ::= <rRemoteUnshieldSelf>
16970
16971    Symbol_References ::= Symbol_References Symbol_Reference <rSymbolReferenceList2>
16972    Symbol_References ::= Symbol_Reference <rSymbolReferenceList1>
16973
16974    Remote_App::= Callable_Reference <rRemoteAppendCall>
16975    Remote_App::= Rule_Lambda <rRemoteAppendLambda>
16976    Remote_App::= Remote_AppendBinary <rRemoteAppendIsBinary>
16977    Remote_AppendBinary ::= <rRemoteAppendADD>
16978    Remote_AppendBinary ::= <rRemoteAppendSUB>
16979    Remote_AppendBinary ::= <rRemoteAppendDIV>
16980    Remote_AppendBinary ::= <rRemoteAppendMUL>
16981    Remote_AppendBinary ::= <rRemoteAppendConcat>
16982    Remote_AppendBinary ::= <rRemoteAppendEQ>
16983    Remote_AppendBinary ::= <rRemoteAppendNE>
16984    Remote_AppendBinary ::= <rRemoteAppendLT>
16985    Remote_AppendBinary ::= <rRemoteAppendGT>
16986    Remote_AppendBinary ::= <rRemoteAppendLE>
16987    Remote_AppendBinary ::= <rRemoteAppendGE>
16988    Remote_AppendBinary ::= <rRemoteAppendOR>
16989    Remote_AppendBinary ::= <rRemoteAppendAND>
16990    Remote_AppendBinary ::= <rRemoteAppendMOD>
16991
16992    Remote_Single ::= Callable_Reference <rRemoteSingleCall>
16993    Remote_Single ::= <rRemoteSingleInfer>
16994    Remote_Single ::= <rRemoteSingleCreateList>
16995    Remote_Single ::= Rule_Lambda <rRemoteSingleLambda>
16996
16997    Remote_Empty ::= Callable_Reference <rRemoteEmptyCall>
16998    Remote_Empty ::= Constant <rRemoteEmptyConstant>
16999    Remote_Empty ::= <rRemoteEmptyInfer>
17000
17001    Remote_With ::= <rRemoteWithInfer>
17002    Remote_With ::= Simple_Typing Remote_AppRemote_Single Remote_Empty <rRemoteWithLido>
17003    Remote_With ::= Remote_AppRemote_Single Remote_Empty <rRemoteWithInferTypes>
17004
17005  data D_RemoteCombine = RC_BinOp D_BinOp
17006                       | RC_Call Binding
17007                       | RC_Infer
17008                       | RC_Lambda D_RuleExpression
17009
17010  data D_RemoteSingle = RSi_List
17011                      | RSi_Infer
17012                      | RSi_Lambda D_RuleExpression
17013                      | RSi_Call Binding
17014
17015  data D_RemoteEmpty = REm_Constant D_constant
17016                     | REm_Call Binding
17017                     | REm_Infer
17018
17019  data D_RemoteShield = RS_None
17020                      | RS_Unshield
17021                      | RS_Shield [StringTableKey]
17022
17023  data D_ConstituentOptions = CO_None
17024                            | CO_OnlyShield D_RemoteShield
17025                            | CO_Infer D_RemoteShield
17026                            | CO_Functions D_RemoteShield D_type D_RemoteCombine
17027                                           D_RemoteSingle D_RemoteEmpty
17028
17029  data D_RemoteAttribute = D_RemoteIncluding [D_SymbolLocalAttribute]
```

```
17030                         | D_RemoteConstituents [D_SymbolLocalAttribute]
17031                                                 D_ConstituentOptions
17032  newtype D_RemoteAttributes = [D_RemoteAttribute]
17033
17034  symbol Remote_Shield having synt shield_symbols :: D_RemoteShield attr
17035    this.shield_symbols ← RS_Shield(constituent Symbol_Reference.sym with infer)
17036
17037  rule <rRemoteUnshieldSelf> attr
17038    Remote_Shield.shield_symbols ← RS_Unshield()
17039
17040  symbol Opt_Remote_Shield having synt shield_symbols :: D_RemoteShield attr
17041   this.shield_symbols ← constituent Remote_Shield.shield_symbols
17042
17043  rule <rNoRemoteShield> =>
17044    Opt_Remote_Shield.shield_symbols ← RS_None()
17045
17046  symbol Remote_Options having synt shielded :: D_RemoteShield attr
17047    this.shielded ← RS_None()
17048
17049  combine_shields :: D_RemoteShield -> D_RemoteShield -> CoordPtr -> D_RemoteShield
17050  combine_shields (RS_Shield a) (RS_Shield b) pos = RS_Shield(append_list(a, b))
17051  combine_shields RS_Unshield RS_Unshield pos = RS_Unshield()
17052  combine_shields RS_None x pos = x
17053  combine_shields x RS_None pos = x
17054  combine_shields a b pos = report_invalid_shields(a, b, pos)
17055
17056  report_invalid_shields :: D_RemoteShield -> D_RemoteShield -> CoordPtr
17057                            -> D_RemoteShield
17058  report_invalid_shields a b pos = do {
17059    Report(ERROR, pos, "Cannot mix shield and unshield. Sorry.")
17060    return RS_Unshield()
17061  }
17062
17063  rule <rRemoteOptionsWith> attr
17064    Remote_Options.shielded ← combine_shields(Opt_Remote_Shield!1.shield_symbols,
17065                                              Opt_Remote_Shield!2.shield_symbols,
17066                                              coordref)
17067
17068  rule <rRemoteOptionsShield> attr
17069    Remote_Options.shielded ← Remote_Shield.shield_symbols
17070
17071
17072  symbol Remote_Empty having synt empty :: D_RemoteEmpty attr
17073    this.empty ← REm_Infer()
17074
17075  rule <rRemoteEmptyCall> attr
17076    Remote_Empty.empty ← REm_Call(Callable_Reference.bind)
17077      >>= Callable_Reference.result_type2
17078
17079  rule <rRemoteEmptyConstant> attr
17080    Remote_Empty.empty ← REm_Constant(Constant.dcon)
17081
17082  symbol Remote_Single having synt single :: D_RemoteSingle attr
17083    this.single ← RSi_Infer()
17084
17085  rule <rRemoteSingleCall> attr
17086    Remote_Single.single ← RSi_Call(Callable_Reference.bind)
17087      >>= Callable_Reference.result_type2
17088
17089  rule <rRemoteSingleCreateList> attr
17090    Remote_Single.single ← RSi_List()
17091
17092  rule <rRemoteSingleLambda> attr
17093    Remote_Single.single ← RSi_Lambda(Rule_Lambda.rexpr_code)
17094
17095  symbol Remote_AppendBinary having synt op :: D_BinOp attr this.op ← BO_Add
17096  rule <rRemoteAppendAND> attr Remote_AppendBinary.op ← BO_And
17097  rule <rRemoteAppendSUB> attr Remote_AppendBinary.op ← BO_Sub
17098  rule <rRemoteAppendDIV> attr Remote_AppendBinary.op ← BO_Div
17099  rule <rRemoteAppendMUL> attr Remote_AppendBinary.op ← BO_Mul
17100  rule <rRemoteAppendConcat> attr Remote_AppendBinary.op ← BO_Concat
17101  rule <rRemoteAppendEQ> attr Remote_AppendBinary.op ← BO_Eq
```

```
17102  rule <rRemoteAppendNE> attr Remote_AppendBinary.op ← BO_Ne
17103  rule <rRemoteAppendLT> attr Remote_AppendBinary.op ← BO_Lt
17104  rule <rRemoteAppendGT> attr Remote_AppendBinary.op ← BO_Gt
17105  rule <rRemoteAppendLE> attr Remote_AppendBinary.op ← BO_Le
17106  rule <rRemoteAppendGE> attr Remote_AppendBinary.op ← BO_Ge
17107  rule <rRemoteAppendOR> attr Remote_AppendBinary.op ← BO_Or
17108  rule <rRemoteAppendMOD> attr Remote_AppendBinary.op ← BO_Mod
17109
17110  symbol Remote_Apphaving synt app:: D_RemoteCombine attr
17111    this.app= RC_Infer()
17112
17113  rule <rRemoteAppendCall> attr
17114    Remote_Append.app= RC_Call(Callable_Reference.bind)
17115      >>= Callable_Reference.result_type2
17116
17117  rule <rRemoteAppendLambda> attr
17118    Remote_Append.app= RC_Lambda(Rule_Lambda.rexpr_code)
17119
17120  rule <rRemoteAppendIsBinary> attr
17121    Remote_Append.app= RC_BinOp(Remote_AppendBinary.op)
17122
17123
17124  symbol Remote_Options having synt remote_opt :: D_ConstituentOptions attr
17125    this.remote_opt ← CO_OnlyShield(this.shielded)
17126
17127  symbol Remote_With having synt remote_with :: D_ConstituentOptions attr
17128    this.remote_with ← CO_Infer(including Remote_Options.shielded)
17129
17130  rule <rRemoteWithLido> attr
17131    Remote_With.remote_with ←
17132      CO_Functions(including Remote_Options.shielded,
17133                   simple_type_to_tvar(Simple_Typing.sym,
17134                                       including Program.type_names_res2,
17135                                       coordref),
17136                   Remote_Append.append, Remote_Single.single, Remote_Empty.empty)
17137      >>= including Program.all_types
17138
17139  rule <rRemoteWithInferTypes> attr
17140    Remote_With.remote_with ←
17141      CO_Functions(including Remote_Options.shielded, newTyVar(),
17142                   Remote_Append.append, Remote_Single.single, Remote_Empty.empty)
17143
17144  rule <rNoRemoteOptions> =>
17145    Remote_Options.remote_opt ← CO_None()
17146
17147  rule <rRemoteOptionsWith> attr
17148    Remote_Options.remote_opt ← Remote_With.remote_with
17149
17150
17151
17152  symbol Remote_Attribute having synt is_including :: Bool attr
17153    this.is_including ← false
17154
17155  rule <rRemoteUp> attr Remote_Attribute.is_including ← true
17156
17157  symbol Remote_Attribute having synt remote_attr :: D_RemoteAttribute
17158  rule <rRemoteUp> attr
17159    Remote_Attribute.remote_attr ←
17160      D_RemoteIncluding(constituent SymbolAttribute.sattr_ref with infer)
17161
17162  rule <rRemoteDown> attr
17163    Remote_Attribute.remote_attr ←
17164      D_RemoteConstituents(constituent SymbolAttribute.sattr_ref with infer,
17165                           constituent Remote_Options.remote_opt)
17166
17167  -- ...
```

Quelltext F.7 – Ausschnitt aus der Implementierung der semantischen Analysen und Codegenerierung (impl.l2)

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig und ohne fremde Hilfe verfasst habe. Ich habe keine anderen, als die von mir angegebenen, Quellen und Hilfsmittel benutzt. Die, den benutzten Werken, wörtliche oder inhaltlihc entnommenen Stellen sind als solche kenntlich gemacht worden.

Ich habe mich bisher nicht um den Doktorgrad beworben.

Torgau, 28.11.2018
Ort, Datum

Christian Berg

Christian Berg

Persönliche Daten

Name	Christian Berg
Geburtsdaten	07. September 1984 in Halle / Saale
Staatsbürgerschaft	Deutsch
Familienstand	ledig

Berufserfahrung

01/2016 – 01/2019	**wissenschaftlicher Mitarbeiter**, Institut für Informatik, MLU, Prof. Dr. Zimmermann.
01/2013 – 12/2015	**wissenschaftlicher Mitarbeiter (Drittmittel)**, Institut für Informatik, MLU, Prof. Dr. Zimmermann. Compilerentwicklung Drittmittelprojekt „ELSY“
09/2012 – 12/2012	**freiberufl. Software-Entwickler**, Steinbeis Institut. Modell-basierte Entwicklung
02/2012 – 06/2012	**Softwareentwicklung und Beratung**, Manitz IT GmbH. J2EE-Entwicklung in Teilzeit
10/2011 – 01/2012	**Software-Entwickler**, Codewrights GmbH. Compilerentwicklung
05/2007 – 08/2008	**wissenschaftliche Hilfskraft**, Institut für Informatik, MLU. Tutor in C-Programmierung, Übungskontrolleur mehrerer Fächer
11/2007 – 07/2008	**wissenschaftliche Hilfskraft**, Drittmittelprojekt „mySputnik“. in Kooperation mit dem MDR

Bildungsweg

07/2010 – 03/2011	**Diplomarbeit**, Thema: Translation Validation für Übersetzer Backends mit Beweisprüfern.
04/2007 – 09/2007	**Projektarbeit „mySputnik“**, *in Kooperation mit dem MDR*, Projektarbeit im Rahmen des Studiums begonnen.
10/2004 – 03/2011	**Studium der Informatik mit Wahlpflichtfach Physik**, *Martin-Luther-Universität*, Halle-Wittenberg. *Vertiefungsrichtungen: „Softwaretechnik und Übersetzerbau“, „Datenbanken“*
09/2000 – 07/2004	**Gymnasium**, *Christian-Wolff-Gymnasium*, Halle, Abitur.

Sprachen

Deutsch	Muttersprache	Englisch	fließend in Wort und Schrift

Christian Berg

0171 831 53 97 • *jayfigh.2011+christian.berg.dissertation@gmail.com*